Sommario

Algoritmi di ordinamento lineari:

- **CountingSort**
- **BucketSort**
- **RadixSort**

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 opportune 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.

Un esempio banale

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

Quale complessità è necessaria per ottenere il risultato?

O(1)

perché il risultato non può che essere

1,2,…,n-1,n

E se ci sono duplicati?

CountingSort: le ipotesi

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

Per ordinare un array A[1..*n*] *CountingSort* richiede un secondo array B[1..*n*] (in cui mettere la sequenza ordinata) ed un array ausiliario C[0..*k*].

Il counting sort è anche chiamato integer sort. In generale integer sorting è il problema di ordinare array di interi sfruttandone opportune proprietà e senza usare confronti.

CountingSort: passo 1

Come primo passo nell'array ausiliario C[0..*k*] si memorizzano le frequenze degli elementi in A, cioè C[i]= numero di occorrenze di i in A

```
for j = 1 to n do
invariante: C[A[j]] è il numero di elementi uguali a A[j]
    C[A[j]] = C[A[j]] + 1
```

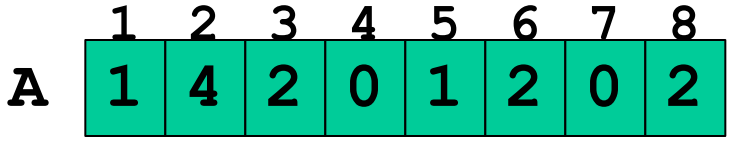
Esempio: A contiene elementi in [0,4]

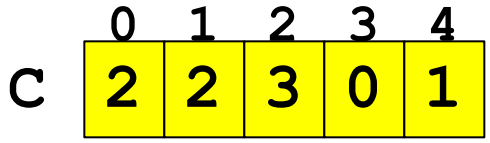
	1	2	3	4	5	6	7	8
A	1	4	2	0	1	2	0	2

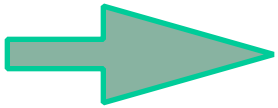
	0	1	2	3	4
C	2	2	3	0	1

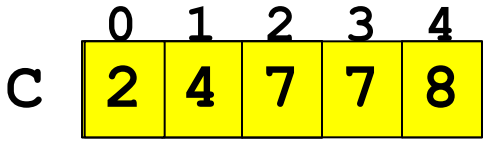
CountingSort: passo 2

Nel secondo passo nell'array ausiliario C[0..*k*] si calcola il rango di ogni elemento in A. Il rango, cioè il numero degli elementi minori o uguali a ciascuno, fornisce la posizione che ogni elementi di A deve assumere nell'array ordinato.
Poiché 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.









CountingSort: terzo passo

Nel terzo 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**

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

i = C[A[j]], B[i] = A[j]

C[A[j]] = C[A[j]] - 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 j=5 j=6 j=7 j=8

	0	1	2	3	4
C	0	2	4	7	7

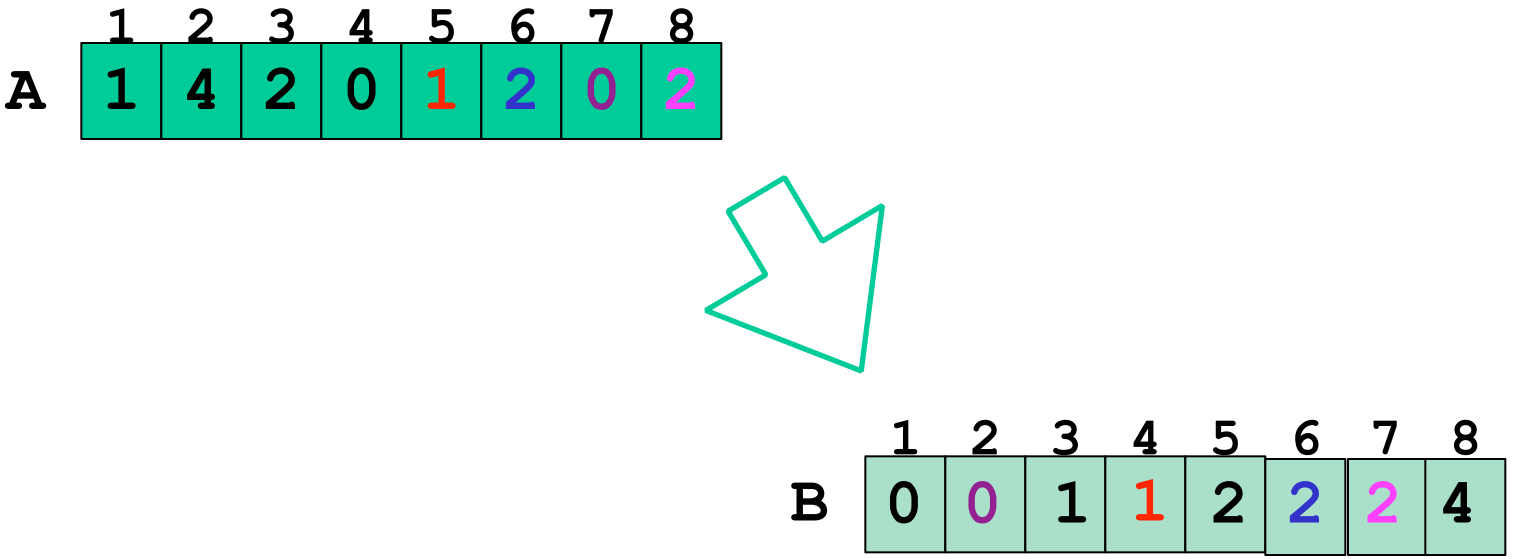
	1	2	3	4	5	6	7	8
B	0	0	1	1	2	2	2	4

Pseudocodice 1 CountingSort.

```
CountingSort1(A)
input: un array A di n elementi.
prec: gli elementi di A sono interi in [0,k] per una costante k > 0
postc: restituisce un array B in cui gli elementi di A sono ordinati
 for i = 0 to k do C[i] = 0
   for j = 1 to n do
invariante: C[A[j]] è il numero di elementi uguali a A[j]
      C[A[j]] = C[A[j]] + 1
   for i = 1 to k do
      C[i] = C[i] + C[i-1]
   invariante: C[i] è il numero di elementi minori o uguali a i
 for j = 1 to n do
     invariante: C[A[j]] è il numero di elementi minori o uguali ad A[j]
meno quelli uguali ad A[j] già copiati in B
      i = C[A[j]], B[i] = A[j]
      C[A[j]] = C[A[j]] - 1
```

Stabilità

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



Esempio di applicazione

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

Un algoritmo di ordinamento è stabile se preserva l'ordine relativo dei duplicati.

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

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

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

Gli elementi uguali non sono nello stesso ordine relativo!

	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 j=5

	0	1	2	3	4
C	1	3	7	7	7

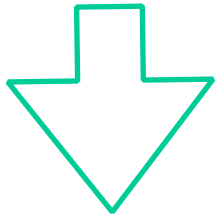
	1	2	3	4	5	6	7	8
B	0		1	1			2	4

Come modificare il ciclo per la stabilità?

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

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

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

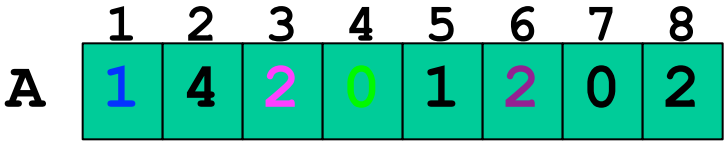


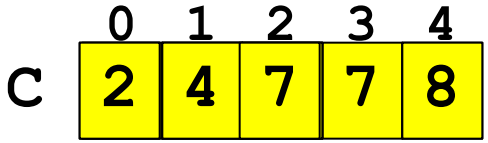
```
for j = n downto 1 do
```

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

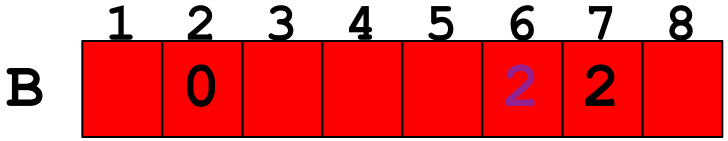
```
    i = C[A[j]], B[i] = A[j]
    C[A[j]] = C[A[j]] - 1
```

Esempio di esecuzione versione stabile





j=8 j=7 j=6



Algoritmo CountingSort.

CountingSort (A)

input: un array A di n elementi.

prec: gli elementi di A sono interi in [0,k] per una costante $k > 0$

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

```
 for i = 0 to k do C[i] = 0
    for j = 1 to n do
invariante: C[A[j]] è il numero di elementi uguali a A[j]
       C[A[j]] = C[A[j]] + 1
    for i = 1 to k do
        C[i] = C[i] + C[i-1]
    invariante: C[i] è il numero di elementi minori o uguali a i
  for j = n downto 1 do
      invariante: C[A[j]] è il numero di elementi minori o uguali ad A[j]
meno quelli uguali ad A[j] già copiati in B
        i = C[A[j]], B[i] = A[j]
        C[A[j]] = C[A[j]] - 1
```

Complessità CountingSort.

```
CountingSort (A)
for i = 0 to k do C[i] = 0                    Θ(k)
   for j = 1 to n do                          Θ(n)
     C[A[j]] = C[A[j]] + 1
   for i = 1 to k do                          Θ(k)
       C[i] = C[i] + C[i-1]
 for j = n downto 1 do                        Θ(n)
        i = C[A[j]], B[i] = A[j]
       C[A[j]] = C[A[j]] - 1
```

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

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

Algoritmo BucketSort

Supponiamo che le chiavi siano nell'intervallo [0,m], e di voler ordinare n elementi.

Con il bucket sort si individuano un certo numero di bucket e si distribuiscono gli n numeri nei bucket.

BucketSort

Bucket sort(L)

1. **Inizializza un array di k bucket inizialmente vuoti (liste concatenate, code,…)**
2. **Inserisci ogni elemento di L nel suo bucket.**
3. **ordina gli elementi in ogni bucket**
4. **copia gli elementi di ogni bucket nell'array originario scorrendo l'array dei bucket da sinistra a destra.**

Spesso il bucketsort è presentato come un modo per ordinare coppie chiave-elemento, dove molti elementi hanno la stessa chiave. In tal caso il bucket è individuato dalla chiave e il passo 3 diventa superfluo.

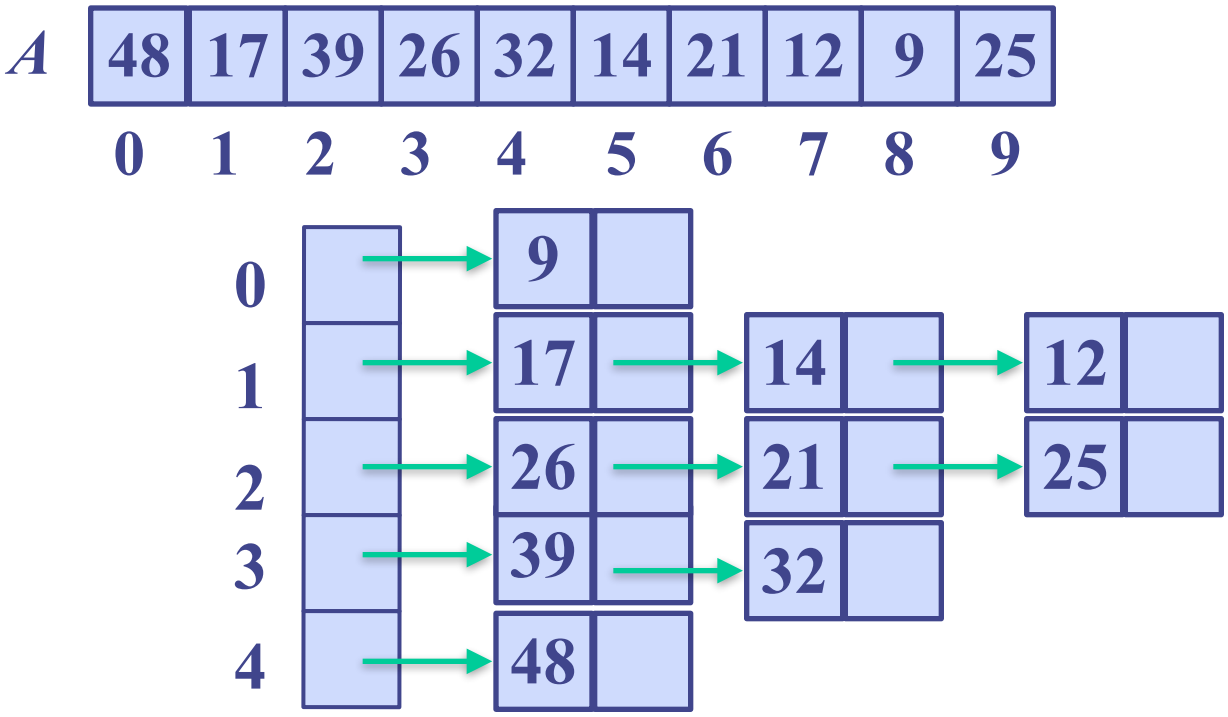
Esempio

Bucket sort(L)

1. **Inizializza un array di k bucket inizialmente vuoti (liste concatenate, code,…)**
2. **Inserisci ogni elemento di L di chiave x nel bucket di indice x dell'array.**
3. **ordina gli elementi in ogni bucket**
4. **copia gli elementi di ogni bucket nell'array originario scorrendo l'array dei buckets da sinistra a destra.**

I numeri sono nell'intervallo [0,49], k = 5 e nell'entrata i si mettono i numeri nell'intervallo [i*10,(i+1)*10-1], per 0≤i≤4

Passi 1 e 2

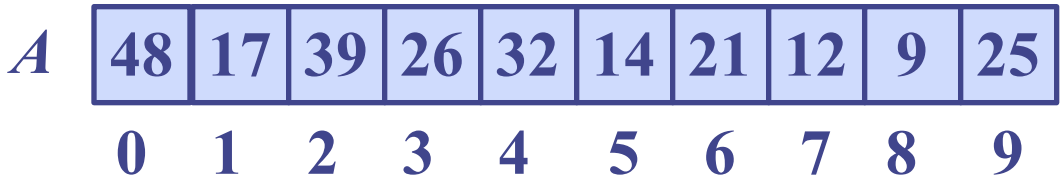


Esempio

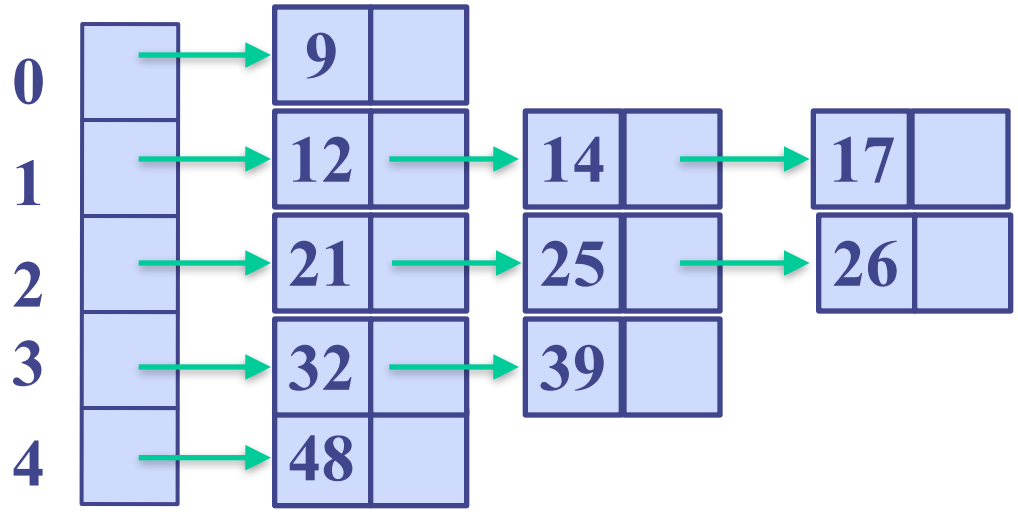
Bucket sort(L)

1. **Inizializza un array di k bucket inizialmente vuoti (liste concatenate, code,…)**
2. **Inserisci ogni elemento di L di chiave x nel bucket di indice x dell'array.**
3. **ordina gli elementi in ogni bucket**
4. **copia gli elementi di ogni bucket nell'array originario scorrendo l'array dei buckets da sinistra a destra.**

I numeri sono nell'intervallo [0,49], k = 5 e nell'entrata i si mettono i numeri nell'intervallo [i*10,(i+1)*10-1], per 0≤i≤4



Passo 3

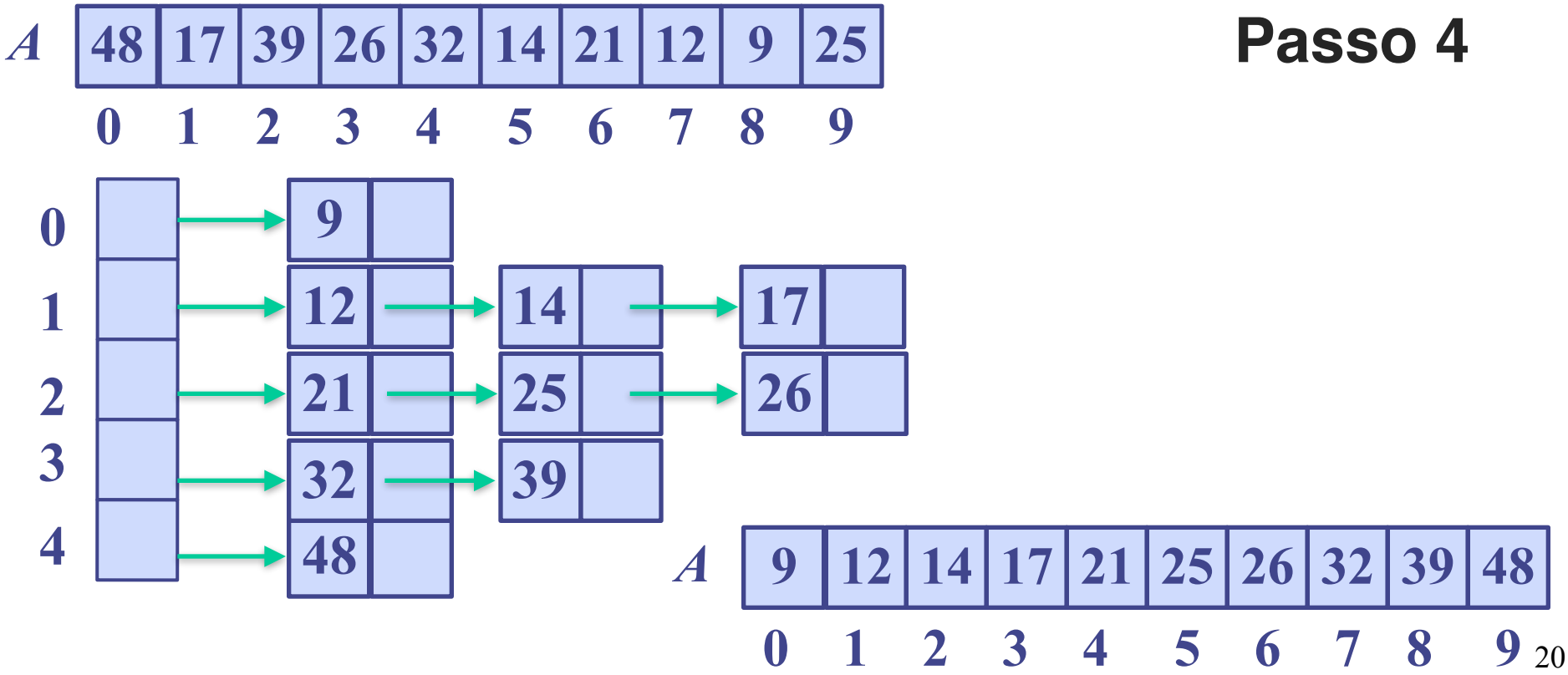


Esempio

Bucket sort(L)

1. **Inizializza un array di k bucket inizialmente vuoti (liste concatenate, code,…)**
2. **Inserisci ogni elemento di L di chiave x nel bucket di indice x dell'array.**
3. **ordina gli elementi in ogni bucket**
4. **copia gli elementi di ogni bucket nell'array originario scorrendo l'array dei buckets da sinistra a destra.**

I numeri sono nell'intervallo [0,49], k = 5 e nell'entrata i si mettono i numeri nell'intervallo [i*10,(i+1)*10-1], per 0≤i≤4



Complessità BucketSort

Bucket sort(L)

1. **Inizializza un array di k bucket inizialmente vuoti (liste concatenate, code,…)** — $\Theta(k)$
2. **Inserisci ogni elemento di L nel suo bucket.** — $\Theta(n)$
3. **ordina gli elementi in ogni bucket**
4. **copia gli elementi di ogni bucket nell'array originario scorrendo l'array dei buckets da sinistra a destra.** — $\Theta(n+k)$

Se si usa l'insertion sort per ordinare gli elementi in ogni bucket, e se dovessero finire tutti nello stesso bucket avremmo complessità $\Theta(n^2)$, nel caso peggiore.

Si dimostra che <u>in media</u> la complessità è $\Theta(n)$, sotto l'ipotesi che l'input si distribuisca uniformemente nei bucket..

Complessità BucketSort

Bucket sort(L)

1. **Inizializza un array di k bucket inizialmente vuoti (liste concatenate, code,…)** — $\Theta(k)$
2. **Inserisci ogni elemento di L nel suo bucket.** — $\Theta(n)$
3. **ordina gli elementi in ogni bucket**
4. **copia gli elementi di ogni bucket nell'array originario scorrendo l'array dei buckets da sinistra a destra.** — $\Theta(n+k)$

Se si usa il quicksort per ordinare gli elementi in ogni bucket, se in ogni bucket ci sono al più n/k elementi si avrà complessità k*O(n/klgn/k) = O(n lg n/k) in media, che diventa O(n) se k = O(n).

RadixSort

RadixSort (A)
input: A è un array di numeri in base *b*
prec: ogni numero di A ha *d* cifre, A[i] = $c_d \ldots c_2 c_1$
postc: A è ordinato
  for j = 1 to d do
invariante:A[1..n] è permutazione di $a_1,\ldots,a_n$
ordinata rispetto alle ultime j-1 cifre $c_{j-1}\ldots c_1$.
  "usa un algoritmo stabile per ordinare A[1..n] rispetto alla j-esima cifra"

Esecuzione su numeri di 4 cifre in base 10

	4	3	2	1	4	3	2	1	4	3	2	1	4	3	2	1	4	3	2	1
A[1]	1	4	2	7	9	8	9	0	8	2	2	3	0	0	3	9	0	0	3	9
A[2]	0	2	4	1	0	2	4	1	7	5	2	5	3	1	6	2	0	2	4	1
A[3]	7	5	2	5	3	1	6	2	1	4	2	7	8	2	2	3	1	2	3	9
A[4]	3	1	6	2	8	2	2	3	1	2	3	9	1	2	3	9	1	4	2	7
A[5]	9	8	9	0	7	5	2	5	0	0	3	9	0	2	4	1	3	1	6	2
A[6]	1	2	3	9	1	4	2	7	0	2	4	1	1	4	2	7	7	5	2	5
A[7]	8	2	2	3	1	2	3	9	3	1	6	2	7	5	2	5	8	2	2	3
A[8]	0	0	3	9	0	0	3	9	9	8	9	0	9	8	9	0	9	8	9	0

RadixSort: complessità

RadixSort (A,n)
input: A è un array di numeri in base *b*
prec: ogni numero di A ha *d* cifre, A[i] = $c_d \ldots c_2 c_1$
post: A è ordinato
for j = 1 to d do
"usa CountingSort per ordinare A[1..n] rispetto alla j-esima cifra" ——— $T_{CS}(n,b) = \Theta(n+b)$

Complessità: $T_{RS}(n,d,b) = \Theta(d(n+b))$
Quando d è costante e b = O(n), radixSort è in $\Theta(n)$

Confronti

Bucket sort e Radix sort sono generalizzazioni del counting sort. Se ogni bucket ha dimensione 1 si ottiene il Counting sort.

il Counting Sort assume che gli elementi siano in [0,k] e si basa sull'array dei ranghi, caso peggiore O(n + k). Utile se k è piccolo.

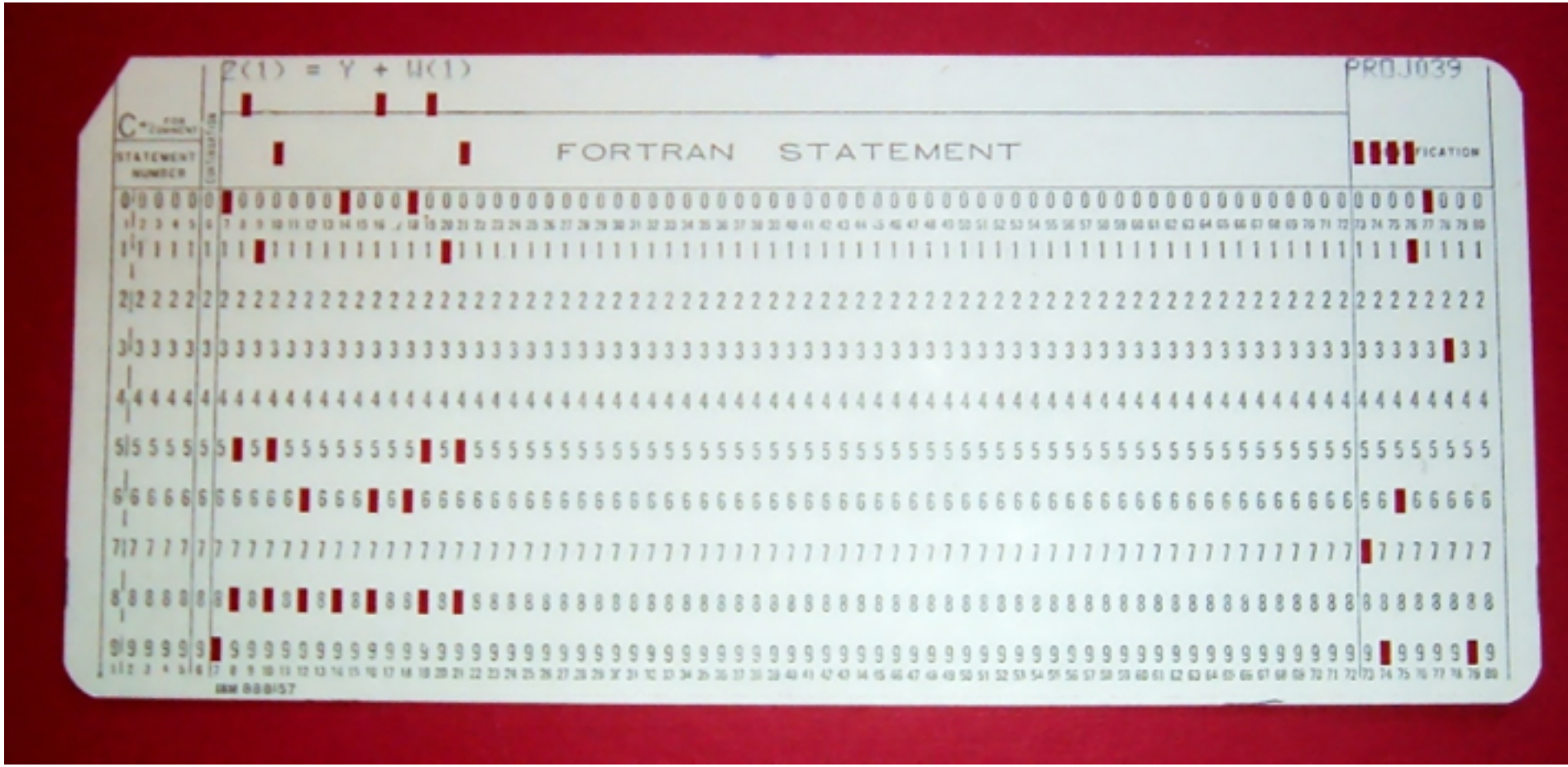
Il Bucket sort presuppone la conoscenza sulla distribuzione dell'input, caso peggiore $\Theta(n^2)$, caso medio $\Theta(n)$.

Il Radix sort assume che gli interi siano di d cifre, con ogni cifra in base b, caso peggiore O(d(n + b)). Utile quando i numeri da ordinare sono molti, d è costante e piccolo.

Nessuno ordina in loco

RadixSort: la preistoria

Radix sort è l'algoritmo che veniva usato per ordinare le schede perforate. Una scheda perforata è stata per molto tempo lo strumento per memorizzare e fornire in input sia dati che programmi, erano di cartoncino e organizzate in 80 colonne. In ogni colonna si poteva praticare un foro in una fra 12 posizioni.



From Wikipedia, the free encyclopedia: l'immagine di una scheda perforata di produzione IBM

RadixSort: la preistoria

La macchina ordinatrice era programmata per ordinare prima la colonna meno significativa, poi via via colonna per colonna fino alla più significativa.
Ogni scheda nel pacchetto veniva distribuita in uno dei 12 contenitori, a seconda delle posizioni dei fori.
Un operatore poteva allora recuperare le schede contenitore per contenitore così da avere le schede ordinate.

From Wikipedia, the free encyclopedia: IBM 082 Sorte