POLITECNICO DI MILANO

Complessità dell'Ordinamento

Algoritmi, Strutture Dati e Calcolo Parallelo

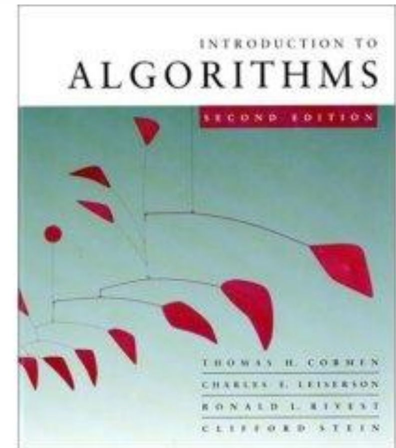
Prof. Pier Luca Lanzi

Riferimenti

- Questo materiale è tratto dalle trasparenze del corso Introduction to Algorithms (2005-fall-6046) tenuto dal Prof. Leiserson all'MIT (http://people.csail.mit.edu/cel/)

- T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein Introduction to Algorithms, Second Edition, The MIT Press, Cambridge, Massachusetts London, England McGraw-Hill Book Company

- Queste trasparenze sono disponibili sui siti http://www.pierlucalanzi.net http://www.slideshare.net/pierluca.lanzi



Algoritmi di ordinamento...

Algoritmi di Ordinamento: Conclusioni

- Diversi algoritmi di ordinamento:
  - Insertion Sort: ottimo θ(n), medio - pessimo θ(n2)
  - Merge Sort: ottimo-medio-pessimo θ(n log n)
  - Heapsort: pessimo O(n log n)
  - Quick Sort: ottimo – medio θ(n log n), pessimo θ(n2)

- Nota:
  - Tutti questi algoritmi sono basati su confronti le decisioni sull'ordinamento vengono prese in base al confronto (<,=,>) fra due valori

- Domanda: E' possibile fare meglio di O(n log n)?

Qual è la complessità dell’ordinamento?

Qual è la complessità dell'ordinamento?

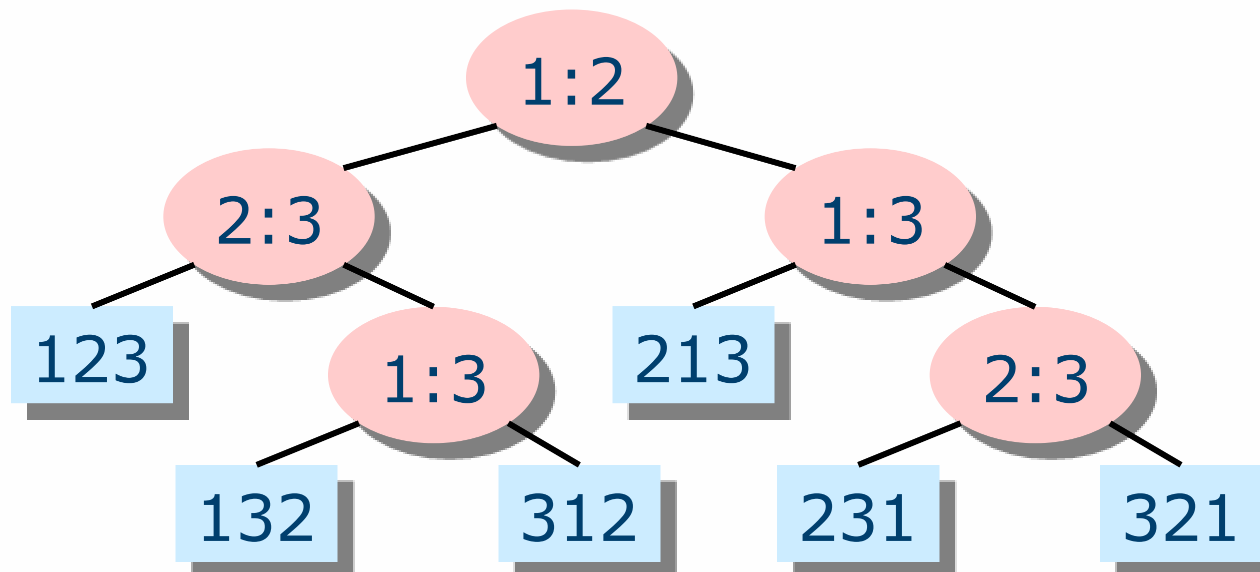
- Gli algoritmi di sort visti fino a ora sono sort comparativi
- Per ordinare un insieme di elementi usa solo la comparazione fra due elementi (insertion sort, merge sort, quick sort)
- La complessità migliore per il worst-case che abbiamo visto finora è $\Theta(n \lg n)$

È $\Theta(n \lg n)$ il meglio che possiamo fare?

Per rispondere usiamo gli alberi di decisione

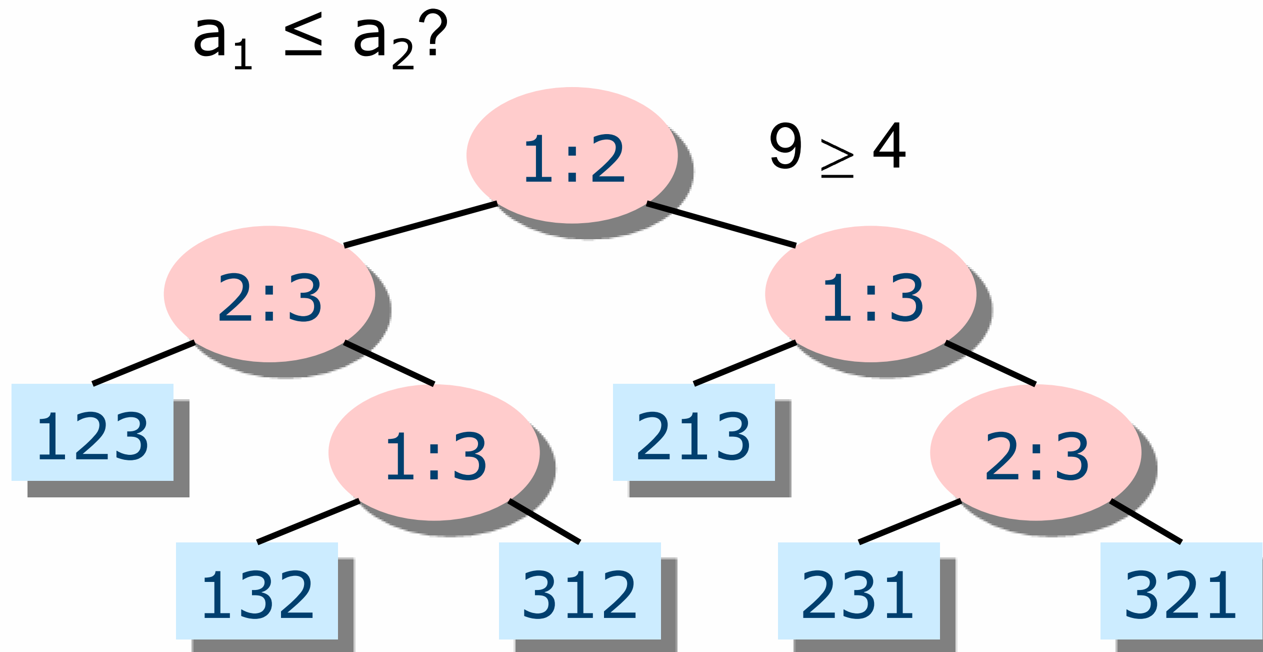
Quanti Confronti?

- Supponiamo di dover ordinare un vettore di tre elementi, a_1, a_2, e a_3, quanti/quali confronti dobbiamo effetuare?
- Utilizziamo gli alberi di decisione, con questa notazione
  - i:j indica il confronto fra a_i e a_j
  - il sottoalbero sinistro rappresenta i confronti successivi se $a_i \leq a_j$
  - quello destro I confronti se $a_i > a_j$
- Ogni foglia contiene una permutazione $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$ che rappresenta l'ordinamento $a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$

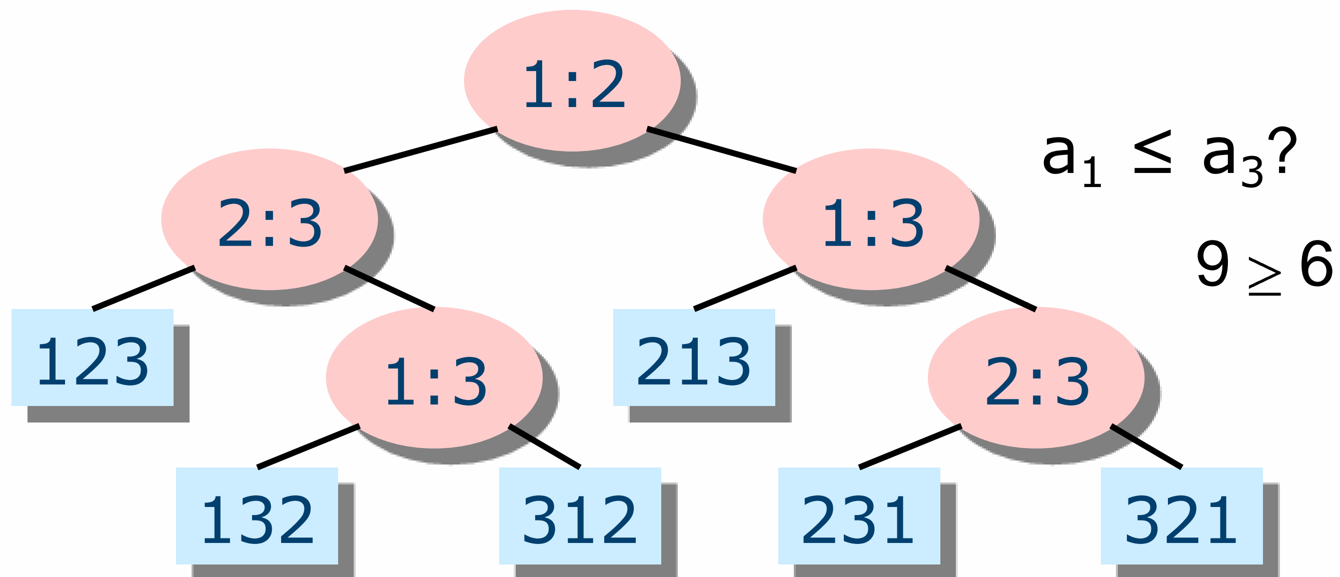


Esempio

- Come esempio, proviamo ad ordinare il vettore <9,4,6>



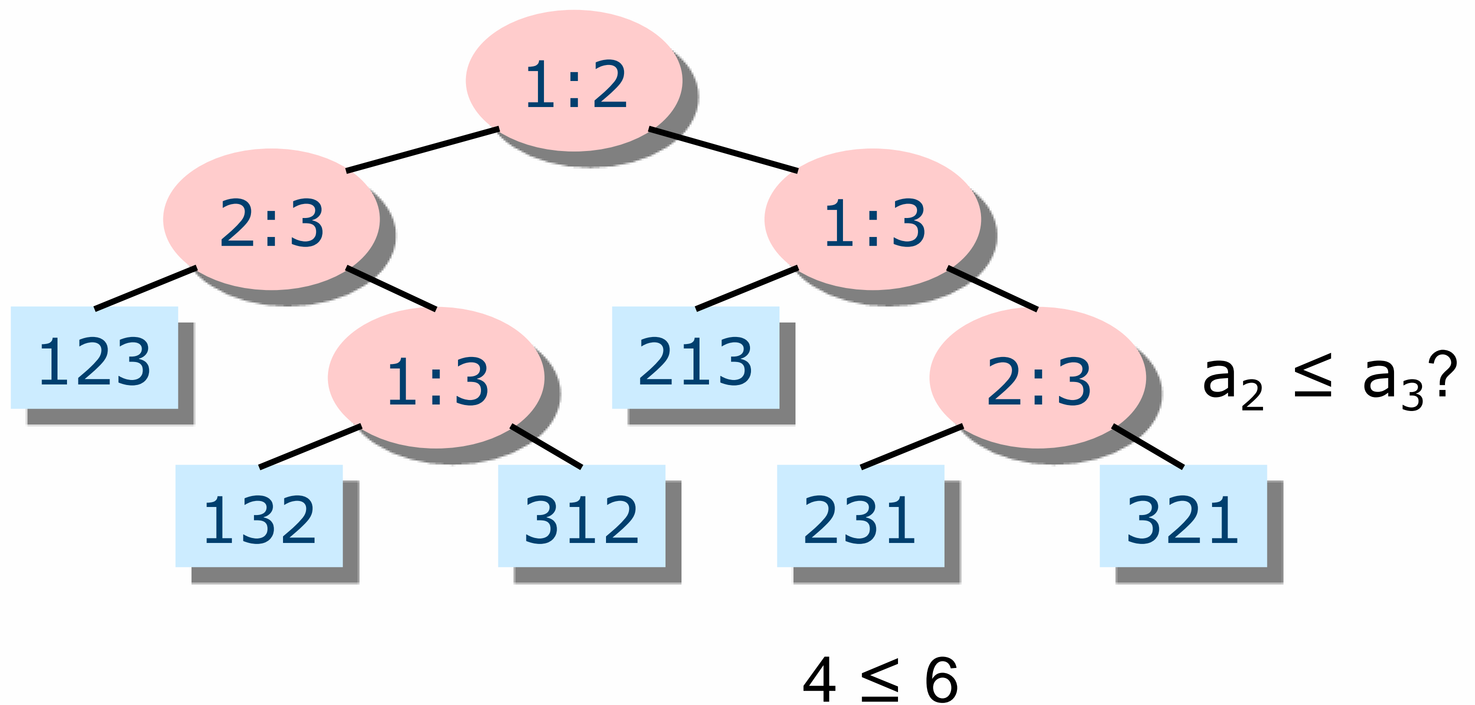
Esempio: Ordinamento di <9,4,6>



$a_1 \leq a_3$?

$9 \geq 6$

Esempio: Ordinamento di <9,4,6>



Alberi di Decisione come Modello dell'Ordinamento

Un albero di decisione modella l'esecuzione di un qualsiasi algoritmo di ordinamento per confronto

- Ogni algoritmo basato su confronto può essere sempre descritto tramite un albero di decisione
- Cammino radice-foglia in un albero di decisione:
  sequenza di confronti eseguiti dall'algoritmo corrispondente
- Altezza dell'albero di decisione:
  # confronti eseguiti dall'algoritmo corrispondente nel caso pessimo
- Altezza media dell'albero di decisione:
  # confronti eseguiti dall'algoritmo corrispondente nel caso medio

Limitazione Inferiore alla Complessità dell'Ordinamento per Confronto

- Lemma: Un albero di decisione per l'ordinamento di n elementi contiene almeno n! Foglie

- Lemma: Sia T un albero binario in cui ogni nodo interno ha esattamente 2 figli e sia k il numero delle sue foglie. L'altezza dell'albero è almeno log k

- Teorema: Un albero di decisione per l'ordinamento di n elemento ha altezza $\Omega(n \lg n)$.

  Dimostrazione: l'albero deve contenere almeno n! foglie quindi dato che un albero di altezza h contiene al massimo 2^h foglie si ha che $n! \leq 2^h$ e quindi:

  $h \geq \lg(n!)$

  $\geq \lg((n/e)^n)$ (approssimazione Stirling)

  $= n \lg n - n \lg e$

  $= \Omega(n \lg n)$

Limitazione Inferiore alla Complessità dell'Ordinamento per Confronto

- Corollario: Il merge sort è un algoritmo di ordinamento basato su confronti asintoticamente ottimo.

Ordinare contando...

Counting Sort

Ordina un insieme di numeri interi
compresi fra 0 e k

Conta quante volte compare ogni elemento
e poi genera il vettore ordinato

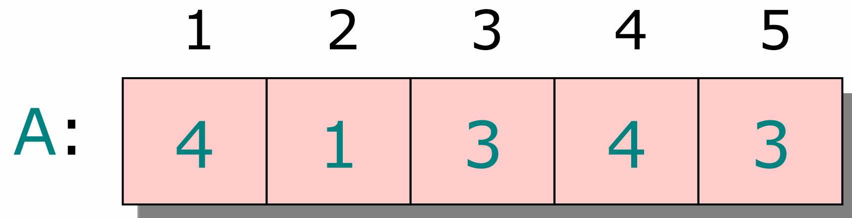
Counting Sort

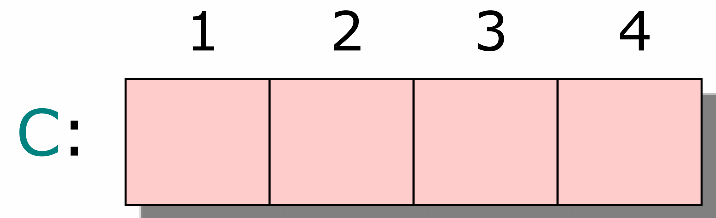
```
int A[n];     // input data 0≤A[i]<k
int C[k];     // auxiliary array

for(i=0; i<k; i++) C[i] = 0;

for(j=0; j<n; j++) C[A[ j]]++;

j=0;
for(i=0; i<k; i++)
   for(x=0; x<C[i]; x++)
   {
      A[j] = i;
      j++;
   }
```

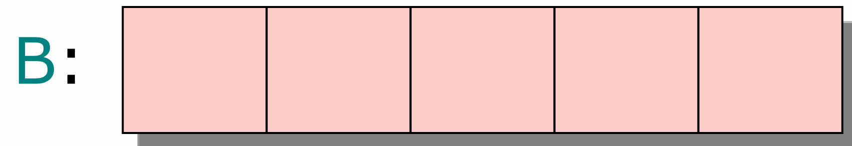
Counting Sort
(Versione del libro di testo)


```
for i ← 1 to k
    do C[i] ← 0

for j ← 1 to n
    do C[A[ j]] ← C[A[ j]] + 1

for i ← 2 to k
    do C[i] ← C[i] + C[i–1]

for j ← n downto 1
    do  B[C[A[ j]]] ← A[ j]
       C[A[ j]] ← C[A[ j]] – 1
```

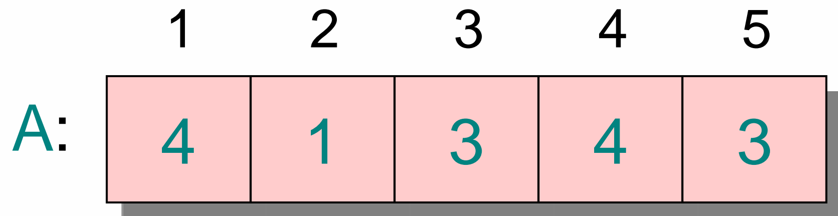
Counting Sort: Esempio

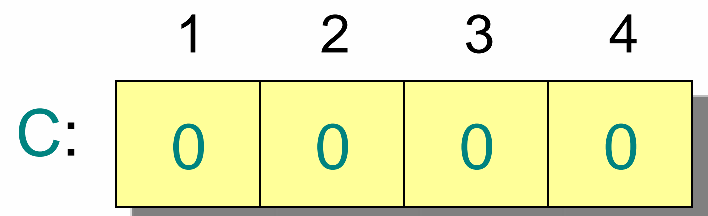


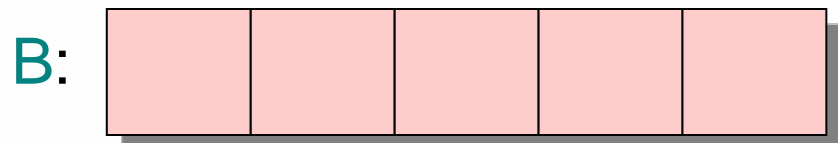




Primo Loop

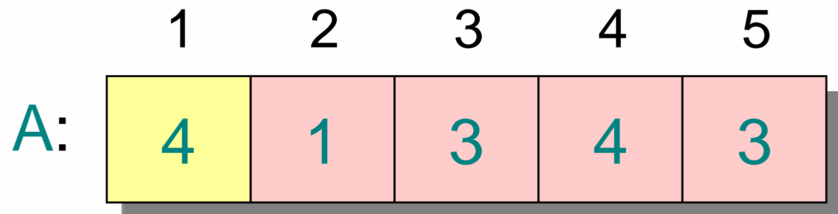


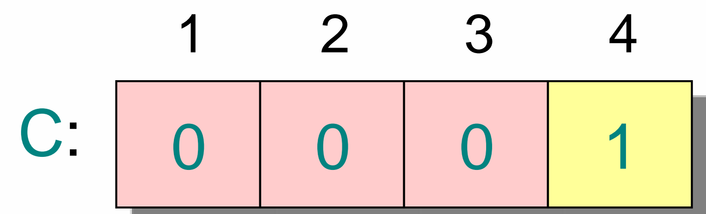


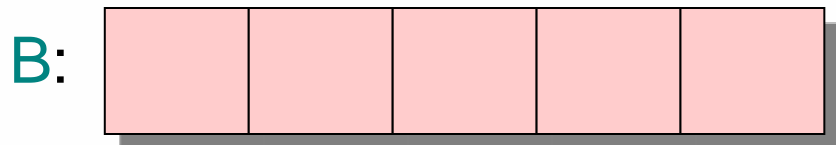



```
for i ← 1 to k
   do C[i] ← 0
```

Secondo Loop

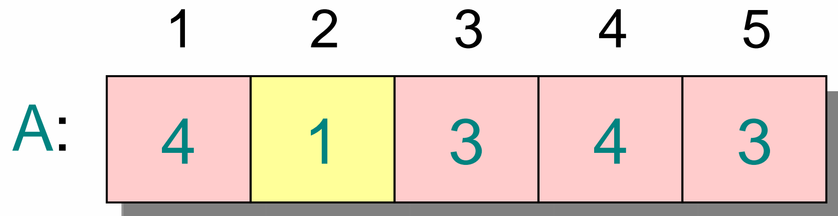


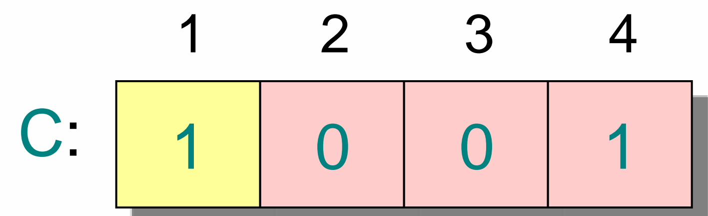


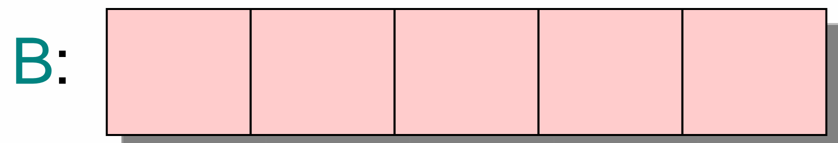



```
for j ← 1 to n
   do C[A[ j]] ← C[A[ j]] + 1
   // C[i]=|{key = i}|
```

Secondo Loop

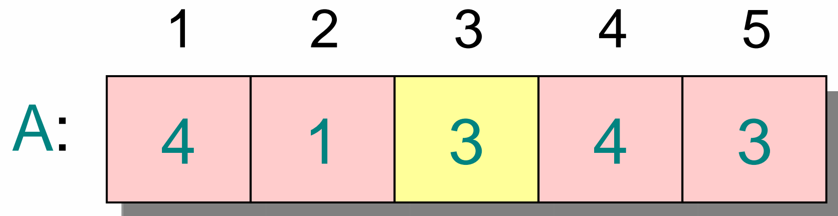


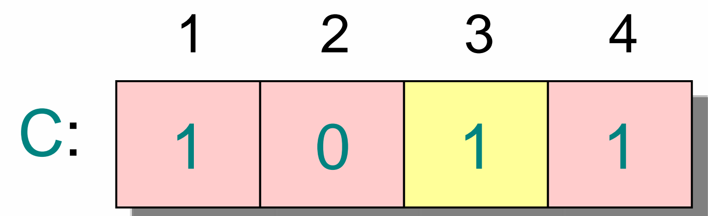





```
for j ← 1 to n
    do C[A[j]] ← C[A[j]] + 1
    // C[i]=|{key = i}|
```

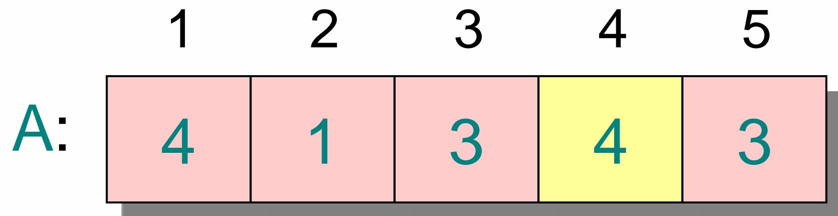
Secondo Loop

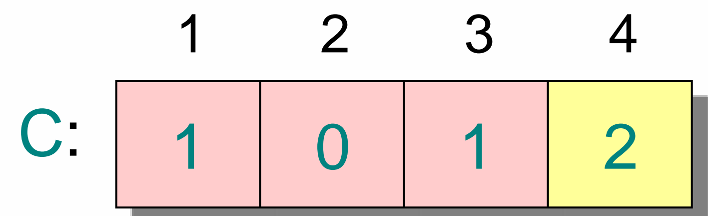


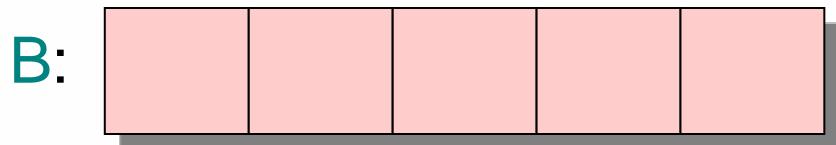


```
for j ← 1 to n
   do C[A[j]] ← C[A[j]] + 1
   // C[i]=|{key = i}|
```

Secondo Loop

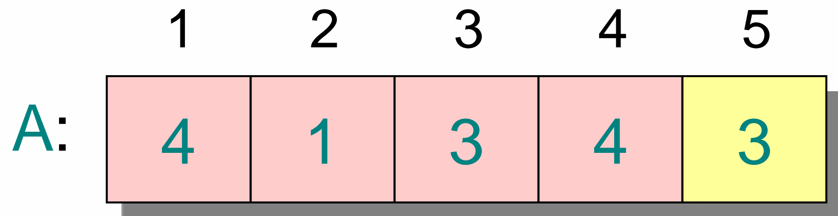


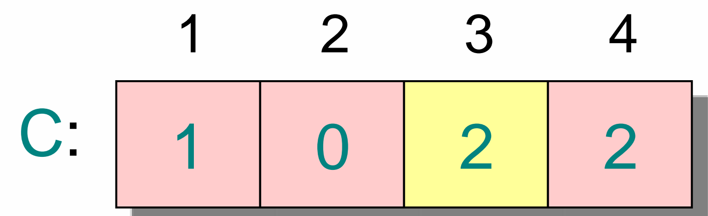


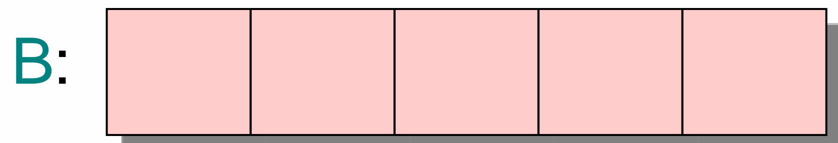



```
for j ← 1 to n
    do C[A[j]] ← C[A[j]] + 1
    // C[i]=|{key = i}|
```

Secondo Loop

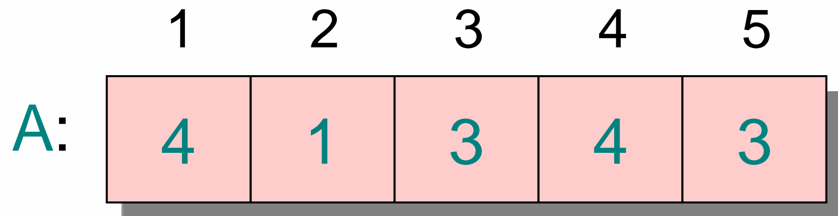


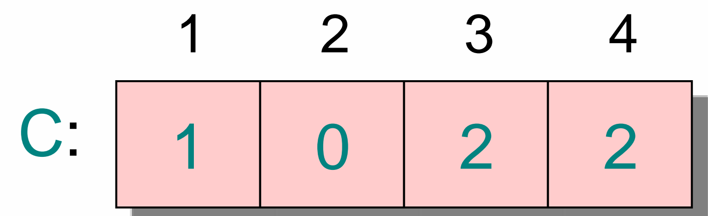


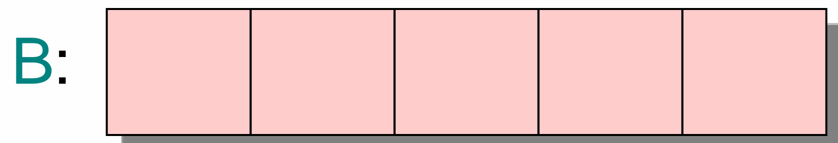


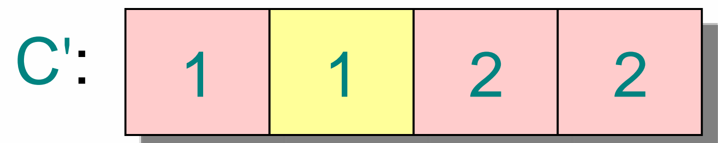

```
for j ← 1 to n
    do C[A[j]] ← C[A[j]] + 1
    // C[i]=|{key = i}|
```

Terzo Loop



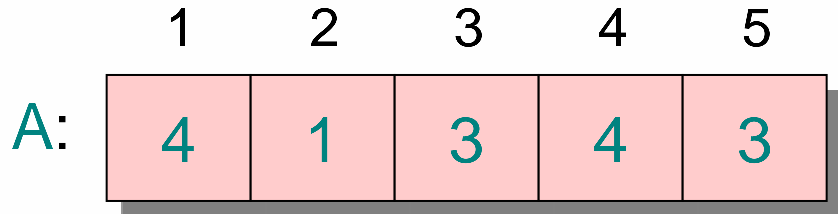


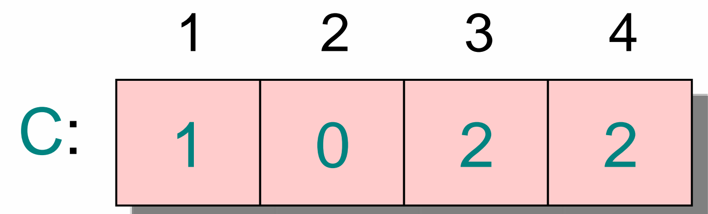


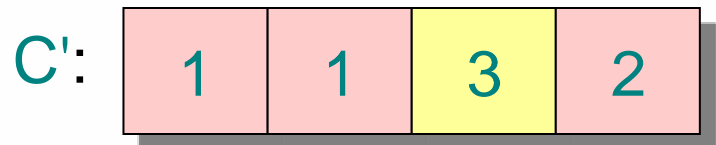



```
for i ← 2 to k
   do C[i] ← C[i] + C[i-1]
   // C[i] = |{key ≤ i}|
```

Terzo Loop

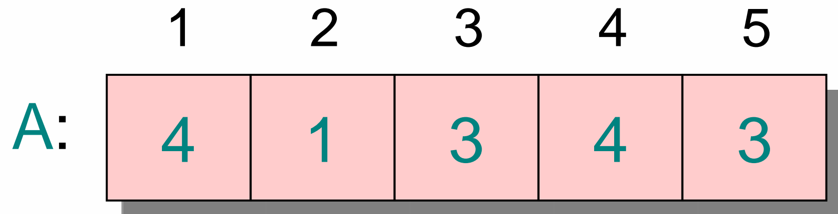


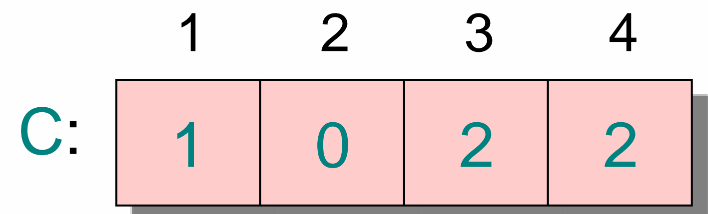


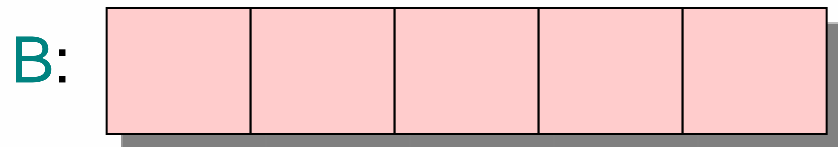


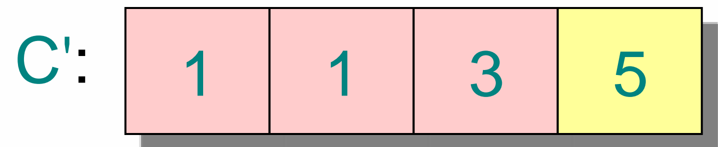

```
for i ← 2 to k
    do C[i] ← C[i] + C[i-1]
    // C[i] = |{key ≤ i}|
```

Terzo Loop



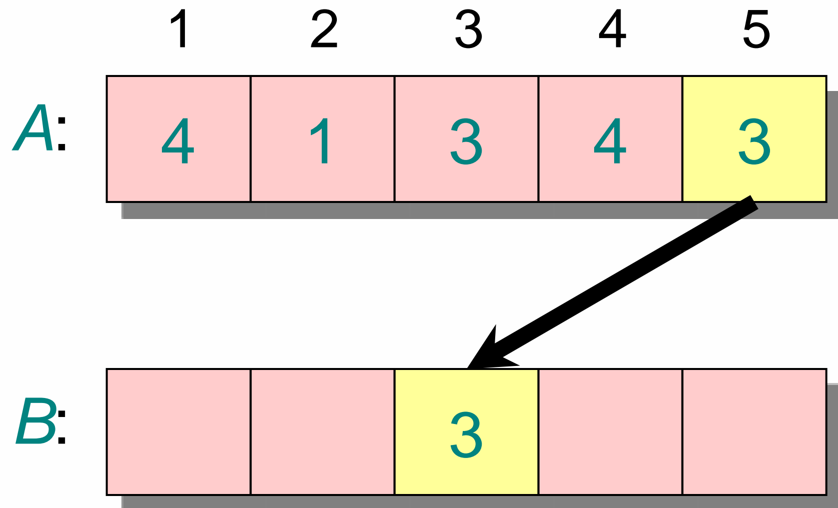


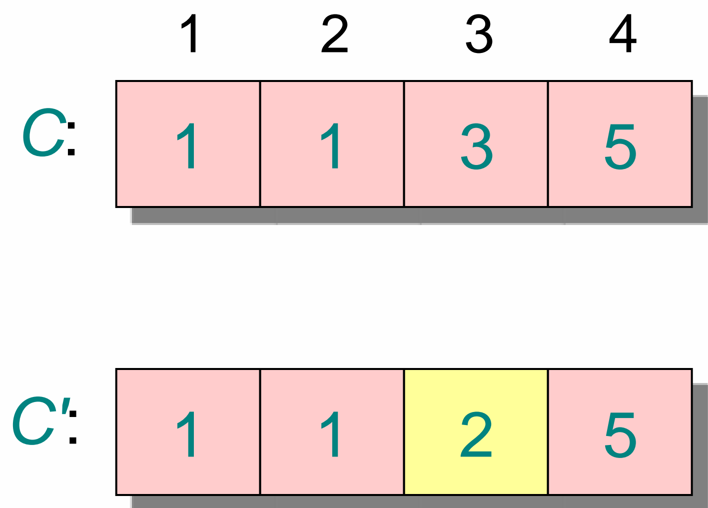





```
for i ← 2 to k
   do C[i] ← C[i] + C[i-1]
   // C[i] = |{key ≤ i}|
```

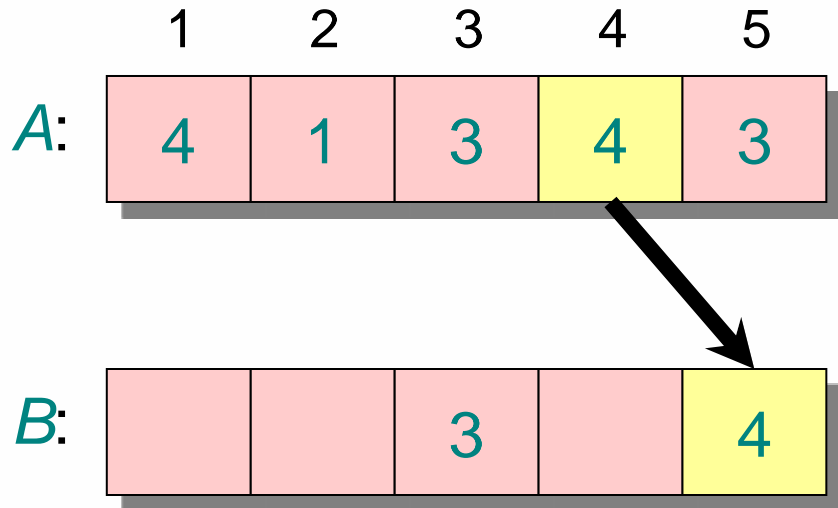
Quarto Loop

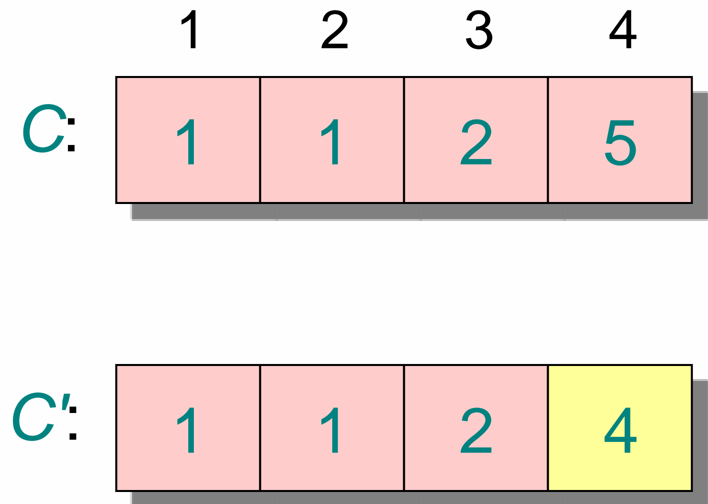





```
for j ← n downto 1
   do B[C[A[ j]]] ← A[ j]
      C[A[ j]] ← C[A[ j]] - 1
```

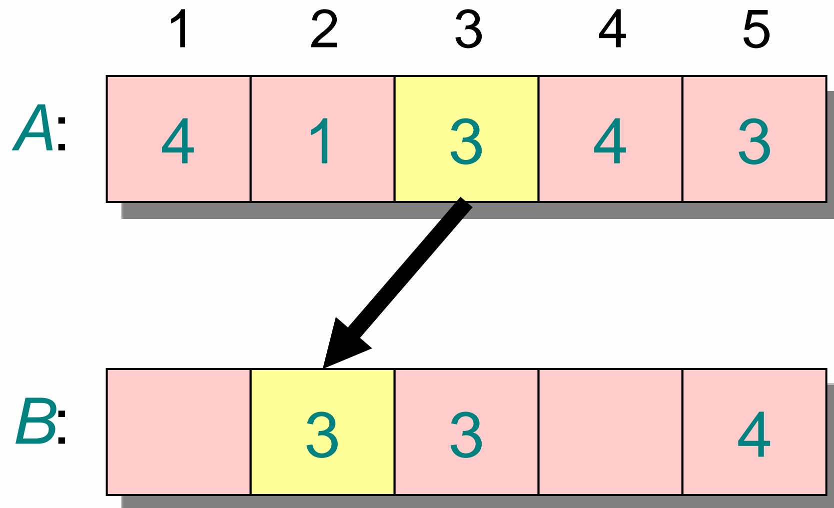
Quarto Loop

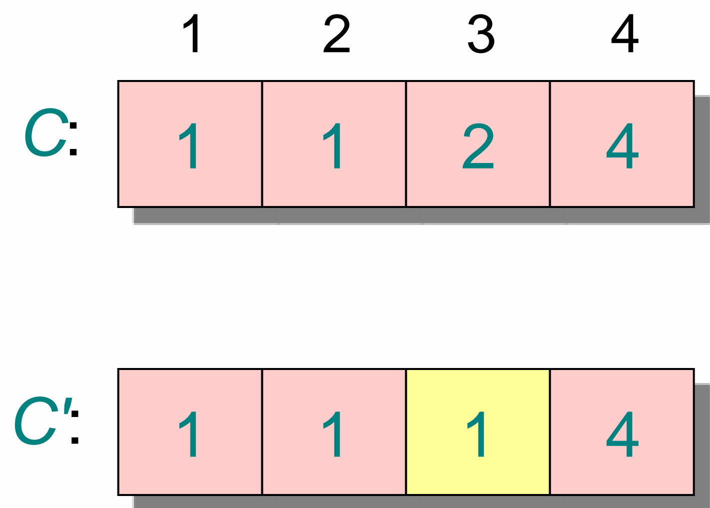





```
for j ← n downto 1
   do B[C[A[ j]]] ← A[ j]
      C[A[ j]] ← C[A[ j]] - 1
```

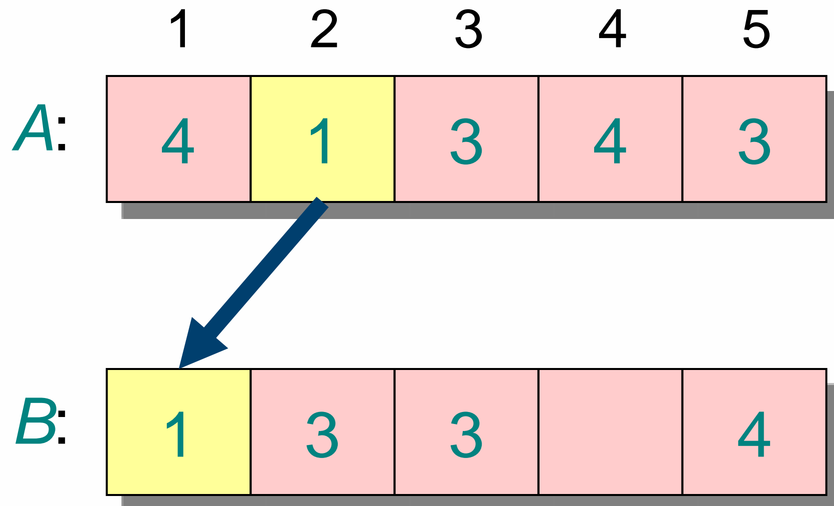
Quarto Loop

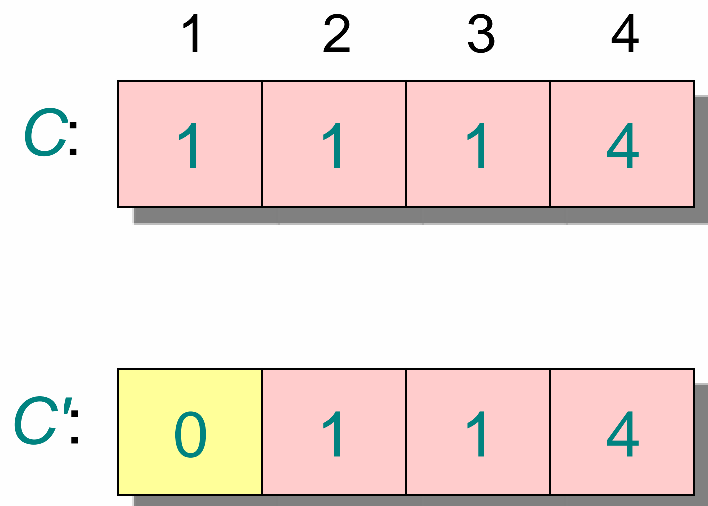





```
for j ← n downto 1
    do B[C[A[ j]]] ← A[ j]
       C[A[ j]] ← C[A[ j]] - 1
```

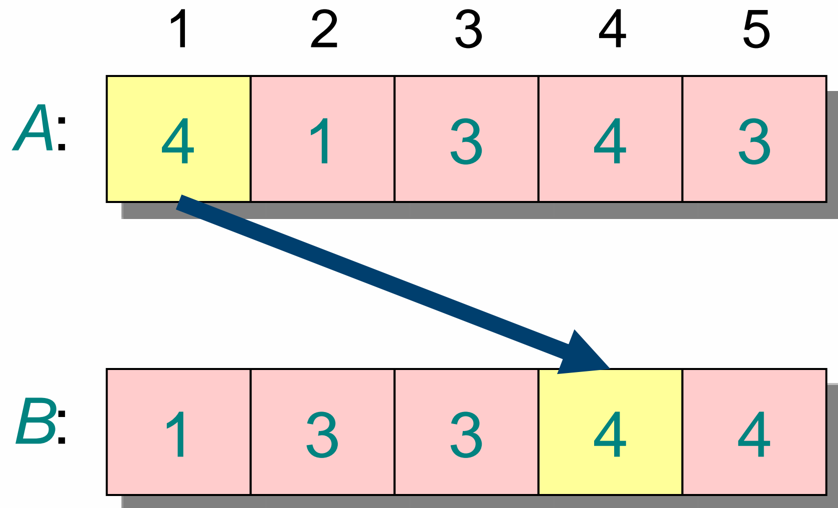
Quarto Loop

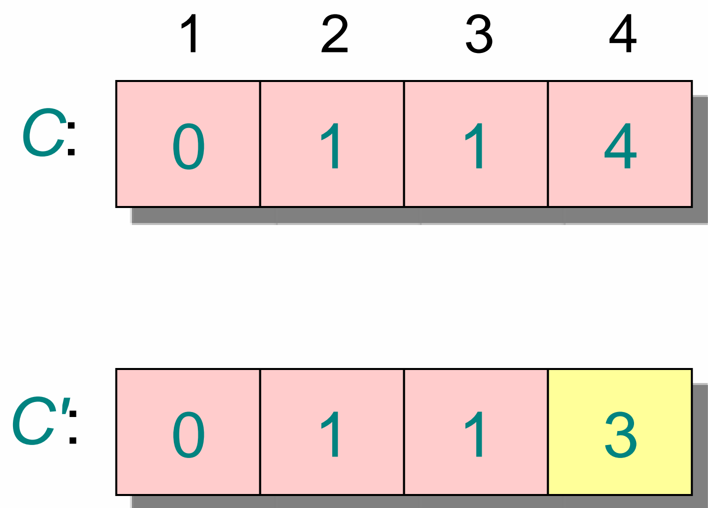





```
for j ← n downto 1
    do B[C[A[ j]]] ← A[ j]
       C[A[ j]] ← C[A[ j]] - 1
```

Quarto Loop






```
for j ← n downto 1
  do B[C[A[ j]]] ← A[ j]
     C[A[ j]] ← C[A[ j]] - 1
```

$\Theta(k)$

```
for i ← 1 to k
    do C[i] ← 0
```

$\Theta(n)$

```
for j ← 1 to n
    do C[A[ j]] ← C[A[ j]] + 1
```

$\Theta(k)$

```
for i ← 2 to k
    do C[i] ← C[i] + C[i−1]
```

$\Theta(n)$

```
for j ← n downto 1
    do B[C[A[ j]]] ← A[ j]
       C[A[ j]] ← C[A[ j]] − 1
```

$\Theta(n + k)$

Se k=n, allora il counting sort è $\Theta(n)$

Ma l'ordinamento è $\Omega(n \log n)$

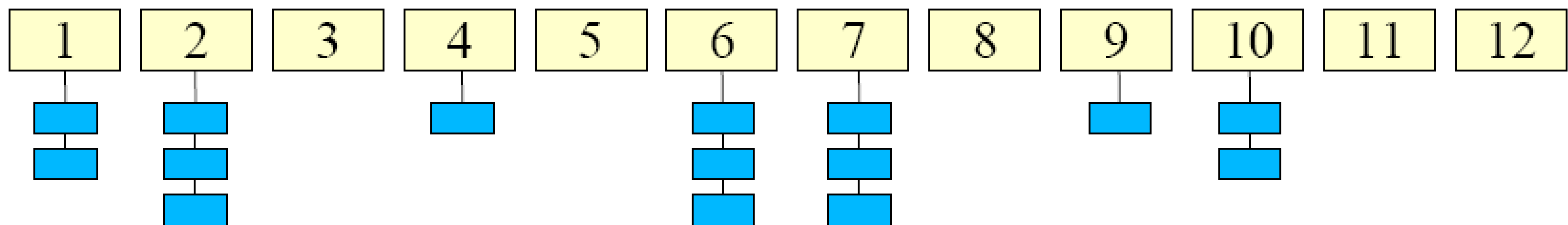
Dov'è l'errore?

L'ordinamento basato su confronto è $\Omega(n \log n)$

Il counting sort non si basa su confronti!

Pigeonhole Sort (Casellario)

- Se i valori non sono numeri interi, ma record associati ad una chiave da ordinare, non possiamo usare il counting
- Ma possiamo usare liste concatenate
- Esempio
  - record contenenti nome, cognome, e mese di nascita
  - per ordinare secondo il mese creiamo 12 caselle (uno per ogni valore) e incaselliamo i record nella posizione corrispondente



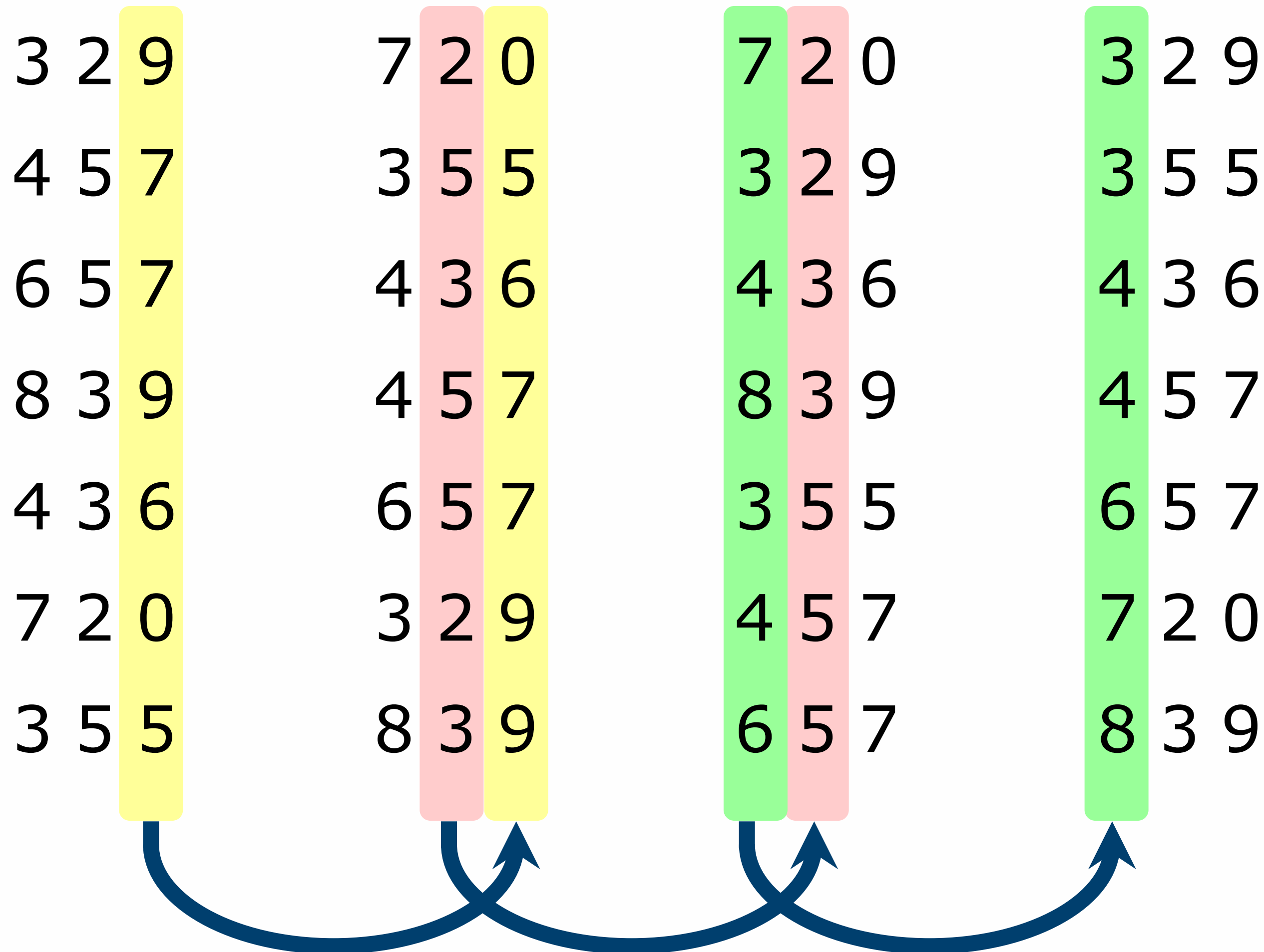
Algoritmi di Ordinamento Stabile

- Un algoritmo di ordinamento è stabile se preserva l'ordine iniziale tra due elementi con la stessa chiave
- Il counting sort è stabile
- Quali dei seguenti algoritmi sono stabili? Insertion Sort, Merge Sort, Heap Sort, Quick Sort, Pigeonhole Sort
- E' possibile rendere ogni algoritmo stabile, usando come chiave di ordinamento la coppia (chiave, posizione iniziale)

Radix Sort

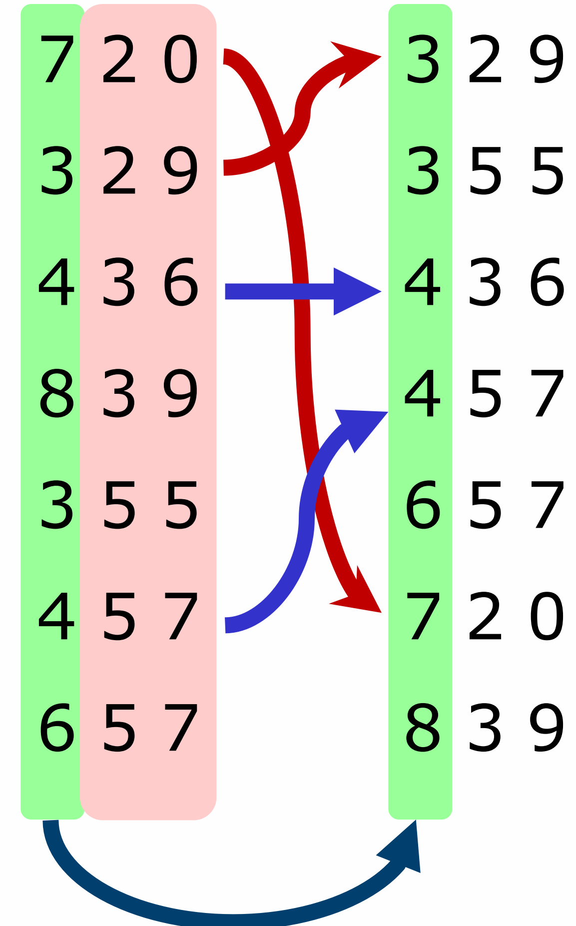
- Il Pigeonhole sort va bene quando k è piccolo
- Esempio
  - Ordinare n numeri con 4 cifre decimali
  - Richiede n+10000 operazioni
  - Se $n \log n < n+10000$, allora non è conveniente
- Radix Sort
  - Ha origine dalla macchina di Herman Hollerith per ordinare le schede del censimento US del 1890
  - Ordina digit per digit
  - Idea originariamente sbagliata:
    ordinare prima i digit più significativi
  - Idea giusta: ordinare prima i digit meno significativi utilizzando un sort stabile

Esempio



Correttezza del Radix Sort

- Dimostrazione per induzione sulla posizione dei digit
- Assumendo che i numeri siano ordinati rispetto i t-1 digit meno significativi
- Ordina il digit t
  - Due numeri che differiscono nel digit t vengono ordinati
  - Due numeri che sono uguali sul digit t sono posizionati nello stesso ordine dell'input (il sort è stabile) ottenendo l'ordine corretto



Analisi del Radix Sort

- Teorema: Dati n numeri di d cifre, dove ogni cifra può avere b valori distinti, il Radix Sort ordina correttamente i numeri in tempo $\Theta(d(n+b))$
- Dimostrazione (correttezza): Per induzione: dopo i chiamate a Pigeonhole Sort, i numeri sono ordinati in base alle prime i cifre meno significative.
- Dimostrazione (complessità): d chiamate a PigeonHole Sort, che ha complessità $\Theta(n+b)$

Sommario

Sommario

- Gli algoritmi di sort basati su confronto sono $\Omega(n \log n)$
- Counting Sort
  - Complessità $\Theta(n+k)$ dove k è il numero di valori
  - Se k alto allora può essere peggio di $\Theta(n \log n)$
- Radix Sort
  - Applica più volte il counting sort partendo dai digit meno significativi
  - Complessità $\Theta(d(n+b))$ con d digit ognuno di b valori
  - Molto veloce, semplice da programmare e mantenere
- Al contrario del quicksort, il radix sort ha una località rispetto al vettore limitata e quindi un quicksort ottimizzato ha una performance migliore sui moderni processori

Riassunto

Ordinamento: Riassunto

- Insertion Sort:
  - $O(n^2)$, stabile, sul posto, iterativo. Adatto per piccoli valori, sequenze quasi ordinate.
- Merge Sort:
  - $O(n \log n)$, stabile, richiede $O(n)$ spazio aggiuntivo, ricorsivo (richiede $O(\log n)$ spazio nello stack). Buona performance in cache, buona parallelizzazione.
- Heap Sort:
  - $O(n \log n)$, non stabile, sul posto, iterativo. Cattiva performance in cache, cattiva parallelizzazione. Preferito in sistemi embedded.
- Quick Sort:
  - $O(n \log n)$ in media, $O(n^2)$ nel caso peggiore, non stabile, ricorsivo (richiede $O(\log n)$ spazio nello stack). Buona performance in cache, buona parallelizzazione, buoni fattori moltiplicativi.

Ordinamento: Riassunto

- Counting Sort
  - O(n+k), richiede O(k) memoria aggiuntiva, iterativo.
  - Molto veloce quando k=O(n)
- Pigeonhole Sort
  - O(n+k), stabile, richiede O(n+k) memoria aggiuntiva, iterativo.
  - Molto veloce quando k=O(n)
- Radix Sort
  - O(d(n+b)), richiede O(n+b) memoria aggiuntiva. Molto veloce quando b=O(n).

- Divide-et-impera
  - Approcci differenti danno risultati diversi
  - Merge Sort: “divide” semplice, “combina” complesso
  - Quick Sort: “divide” complesso, “combina” nullo
- Utilizzo di strutture dati efficienti
  - Heap Sort basato su Heap
- Randomizzazione
  - La tecnica di randomizzazione ci permette di “evitare” il caso pessimo
- Dipendenza dal modello
  - Cambiando l'insieme di assunzioni, è possibile ottenere algoritmi più efficienti (nessun confronto, couting sort)