Corso di laurea in Matematica
Insegnamento di Informatica generale
Lezioni in modalità mista o a distanza

Il problema dell'ordinamento: algoritmi lineari

Giancarlo Bongiovanni
Ivano Salvo

SAPIENZA
UNIVERSITÀ DI ROMA

Queste dispense sono state realizzate sulla base delle slide preparate da T. Calamoneri e G. Bongiovanni per il corso di Informatica Generale tenuto a distanza nell'A.A. 2019/20

Algoritmi lineari

- Abbiamo dimostrato il teorema che asserisce che ogni algoritmo di ordinamento ***che opera per confronti*** ha un costo computazionale di $\Omega(n \log n)$.

- Com'è possibile, allora, avere degli algoritmi di ordinamento di costo computazionale lineare?

- Occorre fare ipotesi **più forti sull'insieme delle chiavi da ordinare** in modo che si possa ordinare il vettore senza basarsi sui confronti.

Counting Sort - 1

- Ipotesi: ciascuno degli n elementi da ordinare è un intero di valore compreso in un intervallo $[1 .. k]$
- Il costo computazionale è di $\Theta(n+k)$. Se $k = O(n)$ allora l'algoritmo ordina n elementi in tempo lineare, cioè $\Theta(n)$.
- Idea: fare in modo che il valore di ogni elemento della sequenza **determini direttamente** la sua posizione nella sequenza ordinata.

Counting Sort - Esempio

A:	8	6	7	2	5	6	1	8	4	4	1	6

n=12, k=8

C:	0	0	0	0	0	0	0	0

1) Scorrendo A conteggiamo in C il numero di occorrenze di ciascun valore

Nota: $\sum_{i=1}^{k} C[i] = n$

	1	2	3	4	5	6	7	8
C:	2	1	0	2	1	3	1	2

2) Scorrendo C ricopiamo in A ciascun indice di C tante volte quanto è il valore in C di quell'indice

A:	1	1	2	4	4	5	6	6	6	7	8	8

Counting Sort - 2

```
Funzione Counting_sort (A: vettore; k, n: intero)
   for i = 1 to k                                  kΘ(1) + Θ(1)
      C[i] ← 0                                     Θ(1)
   for j = 1 to n                                  nΘ(1) + Θ(1)
      C[A[j]] ← C[A[j]] + 1                        Θ(1)
   //C[i] ora contiene il numero di elementi uguali a i
   j = 1                                           Θ(1)
   for i = 1 to k                                  kΘ(1) + Θ(1)
      while (C[i] > 0)                             (∑_{i=1}^{k} C[i]) Θ(1) + Θ(1)
         A[j] ← i                                   Θ(1)
         j ← j + 1                                  Θ(1)
         C[i] ← C[i] - 1                            Θ(1)
   return
```

Poiché $\sum_{i=1}^{k} C[i]$ = n, T(n) = Θ(k) + Θ(n) + Θ(k) + Θ(n) = Θ(k+n)

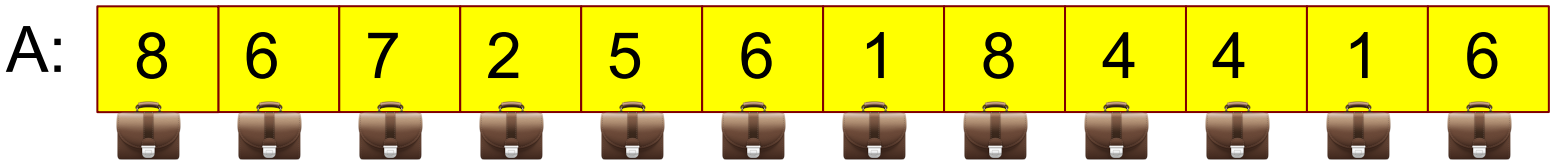
Counting Sort - 3

Lo pseudocodice appena illustrato è adeguato solamente se non vi sono dati satellite. Infatti, il ciclo che riscrive il vettore A di fatto ne sovrascrive i dati.

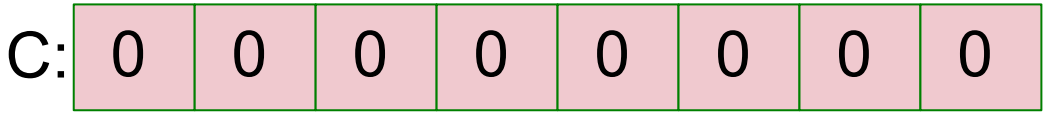
Se ci dovessero essere dati satellite, oltre ai vettori A e C si deve introdurre un nuovo vettore B di *n* elementi, che alla fine conterrà la sequenza ordinata, e si deve usare uno schema leggermente più complesso:

- dopo aver conteggiato gli elementi in C si fa una seconda passata su C, da sinistra a destra a partire da C[2], nella quale a ogni elemento C[i] si somma il precedente;
- alla fine di tale fase:
  - C[i] indica la posizione corretta, nel vettore ordinato, per l'elemento di valore pari a i più a destra fra quelli contenuti nel vettore A;
  - C[i] - C[i-1] indica quanti elementi ci sono con valore pari a C[i]
- si scorre quindi il vettore A da destra a sinistra e:
  - si copia in B ogni elemento A[j]=k nella posizione giusta che è C[k];
  - si decrementa di 1 il valore C[k].

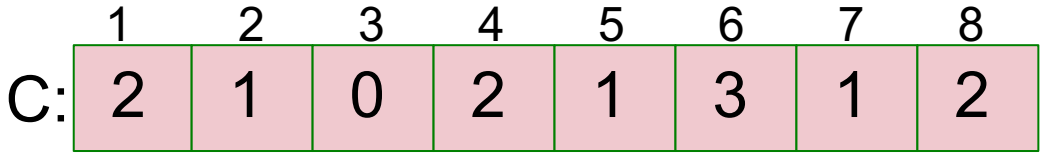
Counting Sort – Esempio 2



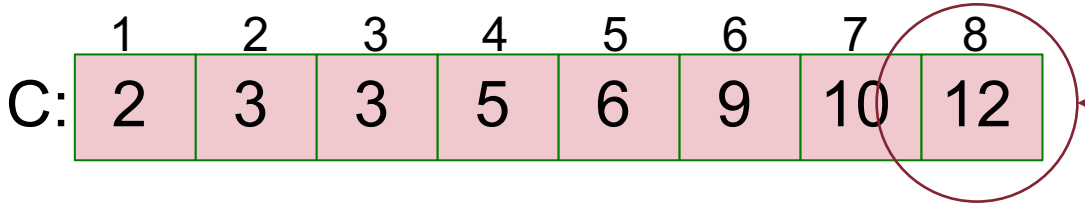
n=12, k=8



1) Prima passata: come nell'algoritmo precedente

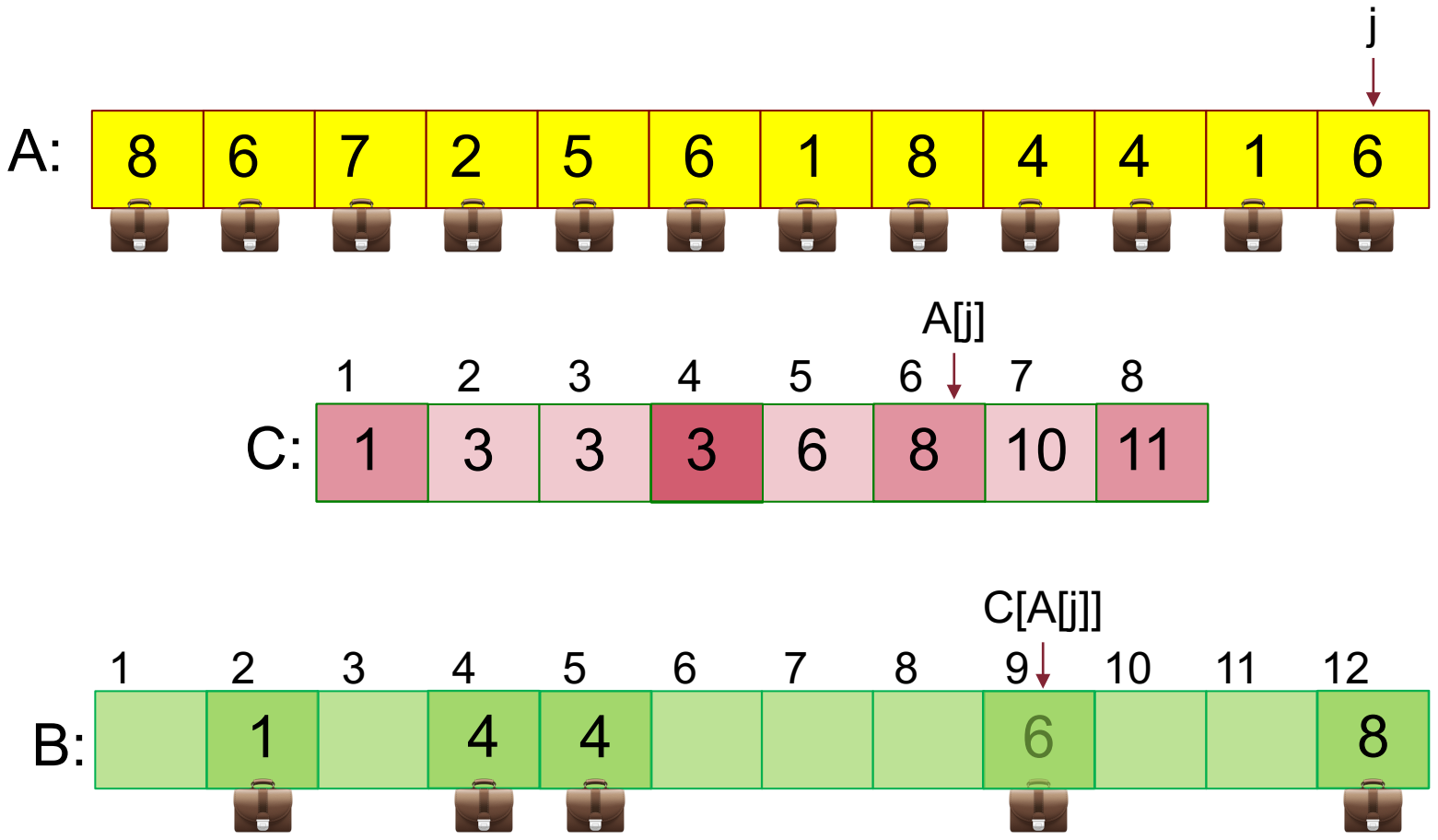


2) Seconda passata: partendo da C[2], ad ogni elemento si somma il precedente



Ci sono due elementi pari a 8 che vanno nelle posizioni 12 e 11

Counting Sort – Esempio 2



Counting Sort - 4

```
Funzione Counting_sort_con_Dati_Satellite (A, B; k, n)
   for i = 1 to k
      C[i] ← 0
   for j = 1 to n
      C[A[j]] ← C[A[j]] + 1
   //C[i] ora contiene il numero di elementi uguali a i
   for i = 2 to k
      C[i] ← C[i] + C[i-1]
   //C[i] ora contiene il numero di elementi minori o uguali a i
   for j = n downto 1
      B[C[A[j]]] ← A[j]]
      C[A[j]] ← C[A[j]] - 1
   for i=1 to n do A[i] ← B[i]
   return
```

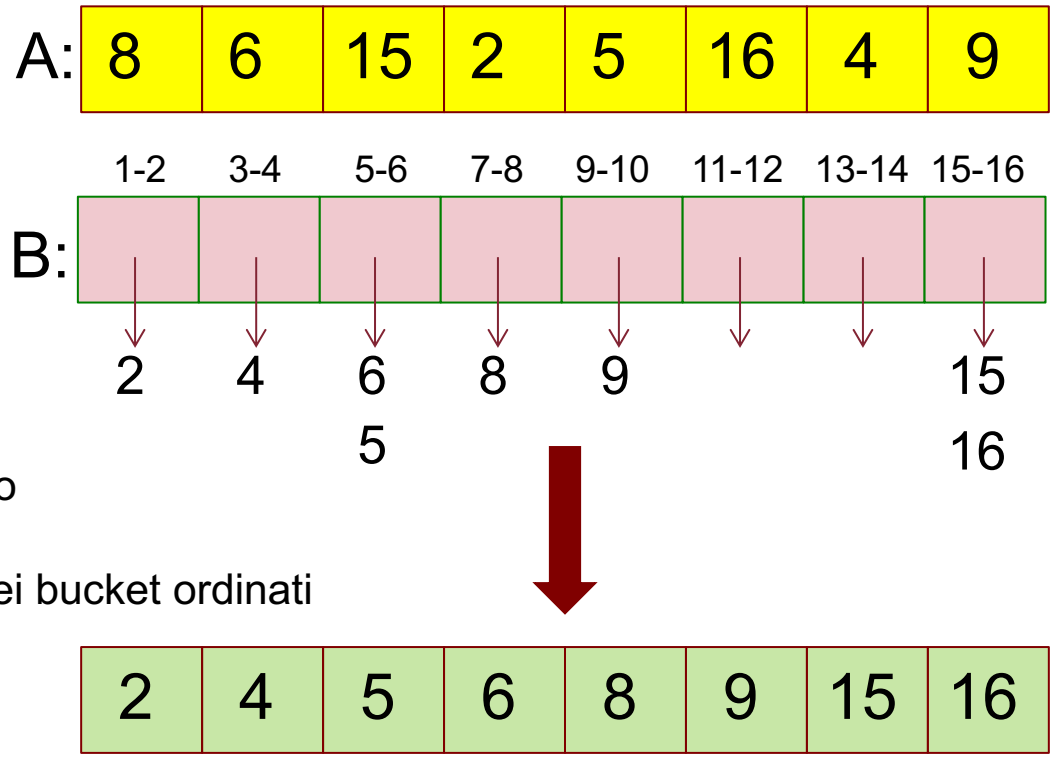
Il cui costo è chiaramente $T(n) = \Theta(k) + \Theta(n)$

Bucket sort - 1

- Ipotesi: gli *n* elementi da ordinare sono distribuiti in modo uniforme nell'intervallo *[1..k]*, senza alcuna ipotesi su *k.*
- Il costo computazionale **medio** è di $\Theta(n)$.
- Idea: dividere l'intervallo *[1..k]* in *n* sottointervalli di uguali dimensioni *k/n*, detti ***bucket***, e distribuire i valori nei bucket.
- Poiché gli elementi in input sono uniformemente distribuiti, non ci si aspetta che molti elementi cadano nello stesso bucket.

Bucket sort - Esempio

n=8, k=16



1) Inserimento degli elementi nei bucket
2) Ordinamento di ogni bucket
3) Copiatura dei bucket ordinati

Bucket sort - 2

```
Funzione Bucket_Sort (A: vettore, n: intero)
for i=1 to n                                    nΘ(1)
   inserisci A[i] nella lista B[  ]
for i=1 to n                                    ∑_{i=1}^{n}(costi di ordinamento)
        ordina la lista B[i]
        con insertion_sort
concatena le liste B[1] B[2] … B[n]             Θ(n)
        in quest'ordine
```

- I costi di ordinamento dipendono dalla lunghezza delle liste.
- Ci sono *n* bucket ed *n* valori; per l'ipotesi di equiprobabilità, in media ci sarà un numero costante di valori in ogni bucket, quindi la lista B[i] ha in media lunghezza costante.
- Quindi, in media T(n) = Θ(n).

Corso di laurea in Matematica
Insegnamento di Informatica generale
Lezioni a distanza

Esercizi per casa

SAPIENZA
UNIVERSITÀ DI ROMA

Esercizi

- Mostrare che il Counting sort è un algoritmo di ordinamento stabile.

- Qual è il tempo di esecuzione del Bucket sort nel caso peggiore? quale semplice modifica dell'algoritmo consente di conservare tempo medio lineare e costo $\Theta(n \log n)$ nel caso peggiore?

- Il bucket sort può essere modificato in modo che l'ordinamento all'interno delle liste sia eseguito tramite counting sort. Affinché il costo dell'algoritmo sia lineare anche nel caso peggiore, quale ipotesi bisogna fare su k?