Corso di Introduzione agli algoritmi
Prof.ssa Tiziana Calamoneri

Il problema dell'Ordinamento: algoritmi lineari

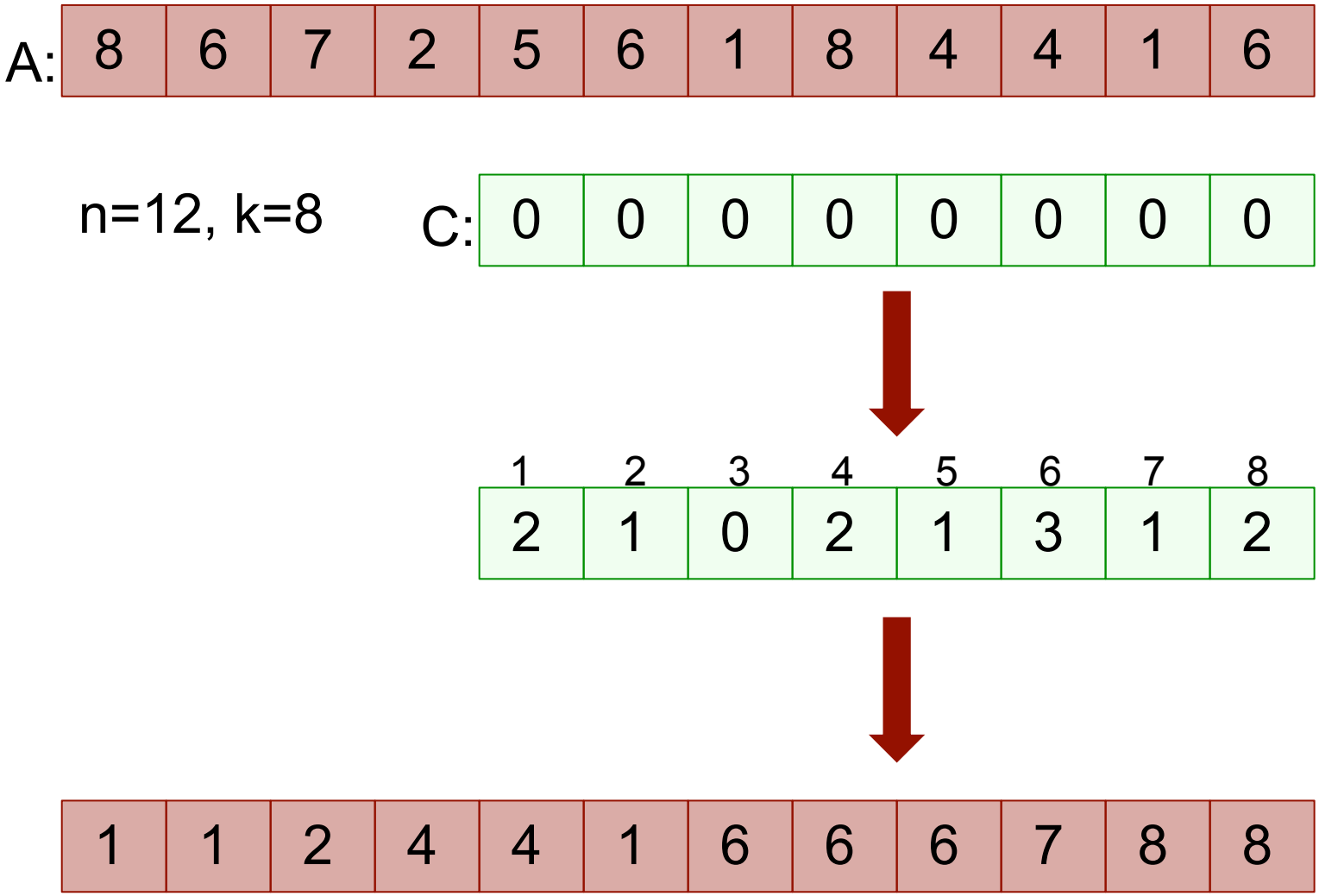
Algoritmi lineari

- 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?
- Rimuoviamo l'ipotesi

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 (2)



counting sort (3)

Funzione Counting_sort (A: vettore; k, n: intero)

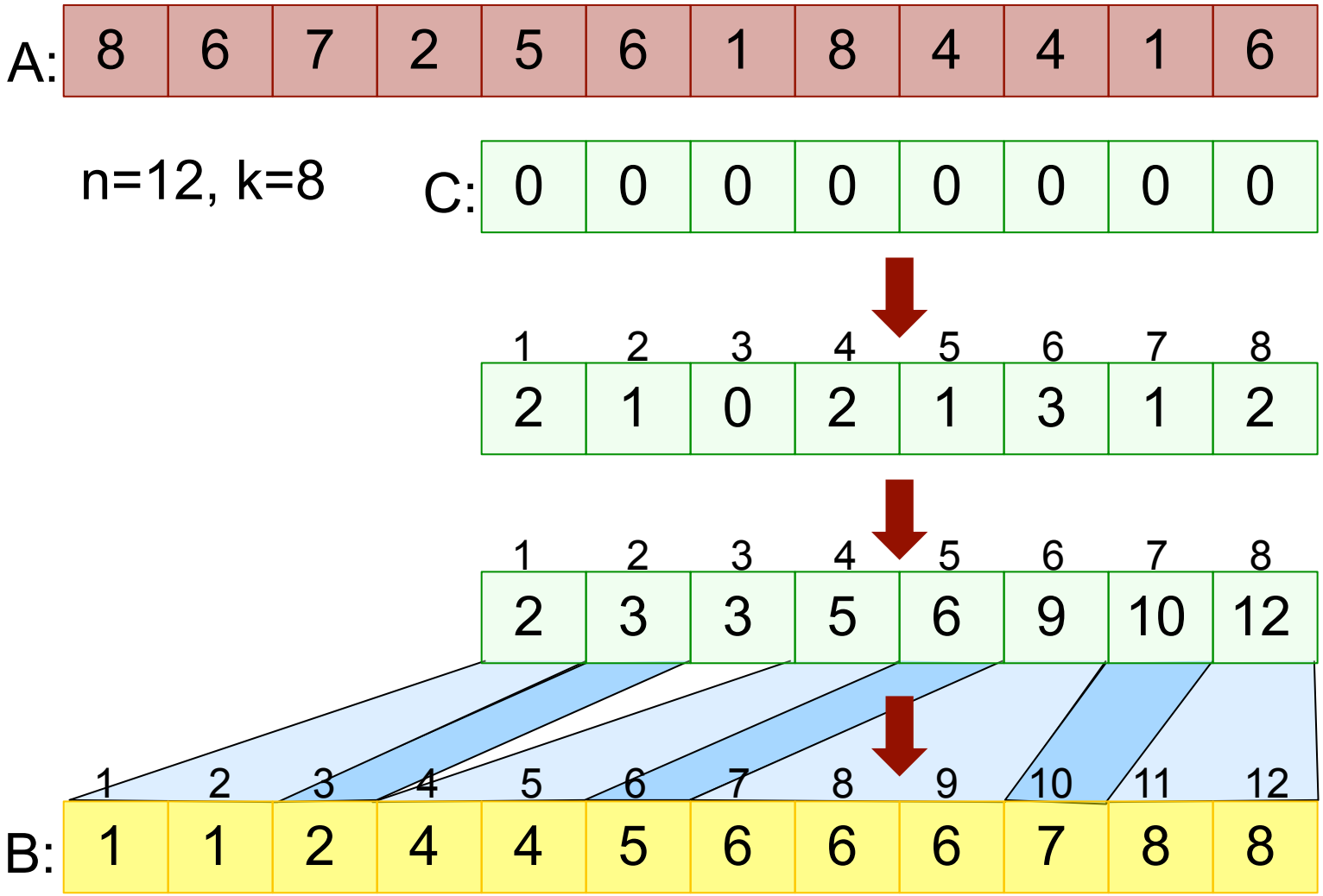
for i = 1 to k	Θ(k)
C[i] ← 0	
for j = 1 to n	Θ(n)
C[A[j]] ← C[A[j]] + 1	
//C[i] ora contiene il numero di elementi uguali a i	
j = 1	Θ(1)
for i = 1 to k	$\sum_{i=1}^{k}$
while (C[i] > 0)	C[i] volte
A[j] ← i	Θ(1)
j ← j + 1	Θ(1)
C[i] ← C[i] - 1	Θ(1)
return	

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

counting sort (4)

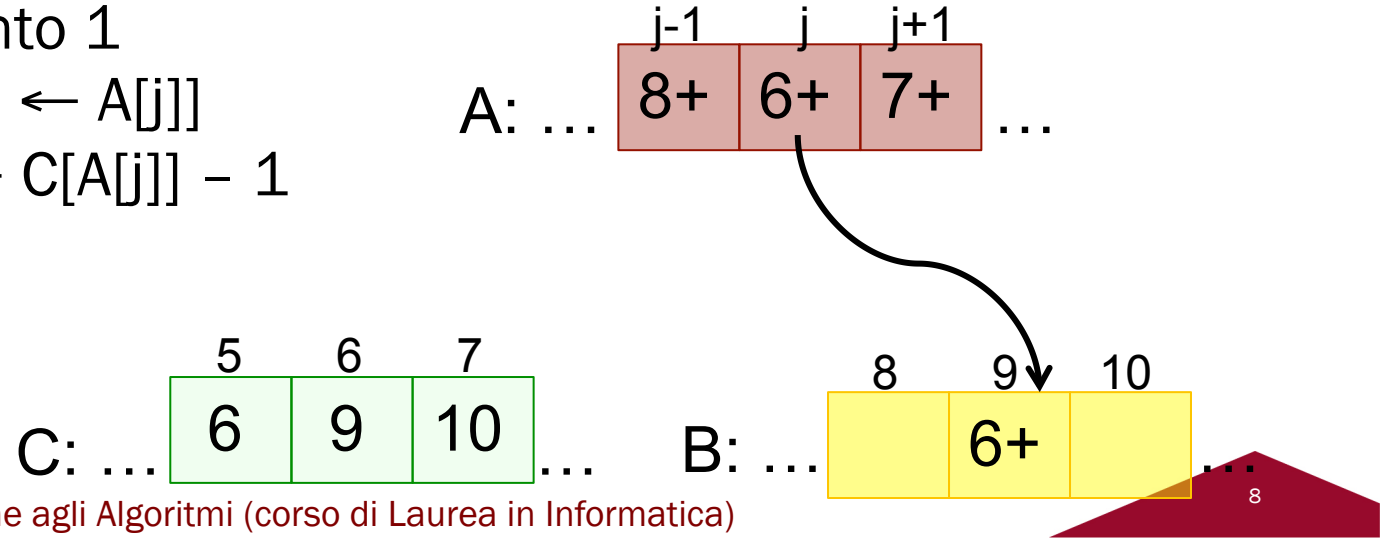
- Nota: lo pseudocodice sopra illustrato è adeguato solamente se non vi sono dati satellite. Infatti, il ciclo che riscrive il vettore A di fatto ne sovrascrive i dati.
- se ci dovessero essere dati satellite, oltra i vettori A e C, si deve introdurre un nuovo vettore B di *n* elementi, che alla fine conterrà la sequenza ordinata.

counting sort (5)



counting sort (5)

```
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
    return
```

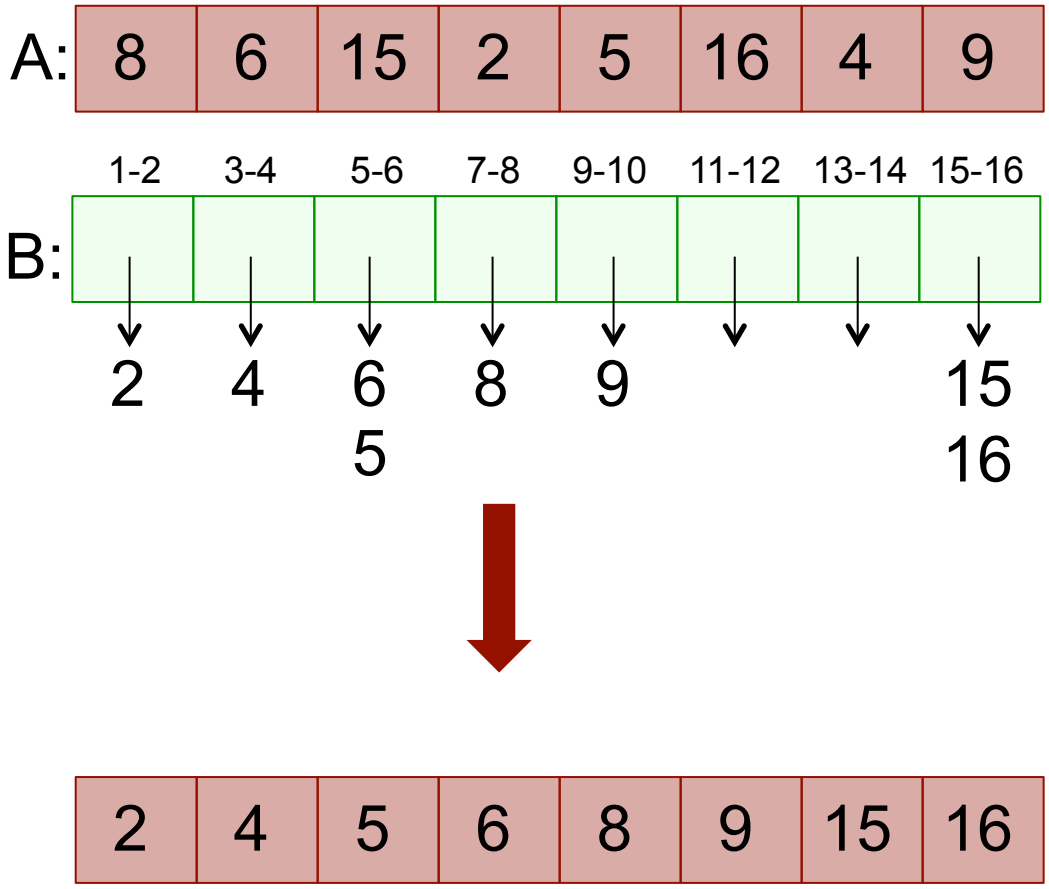


bucket sort (1)

- 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 (2)

n=8, k=16



bucket sort (3)

Funzione Bucket_Sort (A: vettore, n: intero)

for i=1 to n	n volte
inserisci A[i] nella lista B[$\lceil A[i] \cdot n / k \rceil$]	Θ(1)
for i=1 to n	n volte
ordina la lista B[i] con insertion_sort	dipende dalla lungh. di B[i]
concatena le liste B[1] B[2] … B[n] in quest'ordine	Θ(n)

Ci sono *n* bucket ed *n* valori. Per l'ipotesi di equiprobabilità, in media ci saranno 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).

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 Θ(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*?