Corso di laurea in Informatica
Introduzione agli Algoritmi
A.A. 2024/25

Il problema dell'ordinamento: Ordinamenti lineari

Tiziana Calamoneri

SAPIENZA
UNIVERSITÀ DI ROMA

Slides realizzate sulla base di quelle 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?
- Si può limitare il numero di confronti facendo ipotesi aggiuntive.

Counting Sort (1)

- Ipotesi: ciascuno degli n elementi da ordinare è un intero di valore compreso in un intervallo $[0..k]$ (si può variare leggermente…)
- 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 SENZA fare confronti.

Counting Sort – Esempio

A:

0	6	7	2	5	6	1	0	4	4	1	6

n=12, k=7

C:

0	0	0	0	0	0	0	0

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

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

Nota: $\sum_{i=0}^{k} C[i] = n$

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

A:

0	0	1	1	2	4	4	5	6	6	6	7

Counting Sort (2)

```
def Counting_sort (A):
   k=max(A)                                  Θ(n)
   n=len(A)                                  Θ(1)
   C[0]*(k+1)                                Θ(k)
   for j in range(n):                        n volte
      C[A[j]] +=1                               Θ(1)
   //C[i] ora contiene il numero di elementi uguali a i
   j = 0                                     Θ(1)
   for i in range(k):                        ∑_{i=0}^{k}
      while (C[i] > 0)                       C[i] volte
         A[j]=i                                 Θ(1)
         j+=1                                   Θ(1)
         C[i]-=1                                Θ(1)
```

Poiché $\sum_{i=0}^{k} C[i]$ = n,

T(n) = Θ(n)+Θ(k)+Θ(1) +n Θ(1)+ $\sum_{i=0}^{k} C[i]$ Θ(1) = Θ(k+n)

Counting Sort (3)

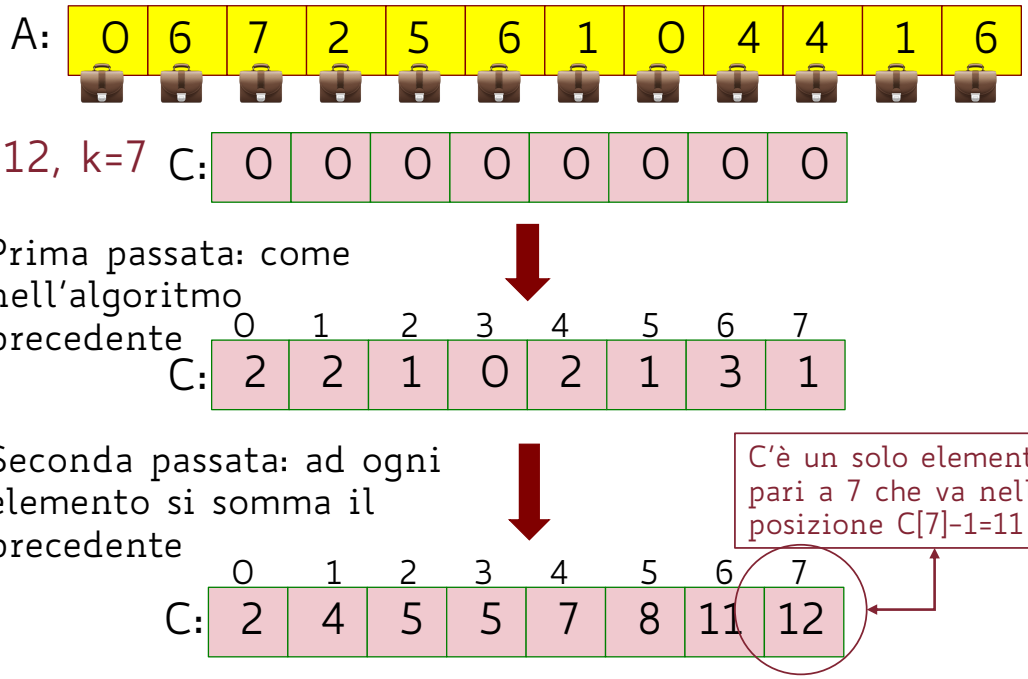
Lo pseudocodice appena illustrato è adeguato solamente se non vi sono dati satellite. Infatti, il ciclo che riscrive l'array *A* di fatto non è in grado di recuperare i dati.

Se ci sono dati satellite, oltre agli array *A* e *C* si deve introdurre un nuovo array *B* di *n* elementi, che alla fine conterrà la sequenza ordinata, e si deve usare uno schema leggermente più complesso…

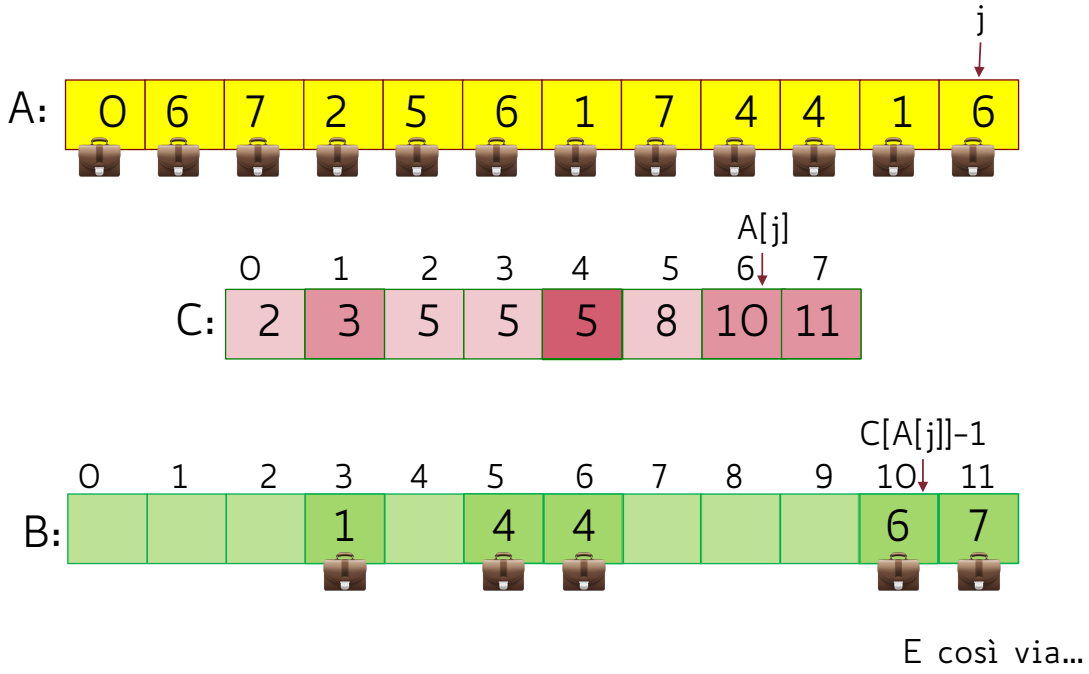
Counting Sort (4)

- Si conteggiano gli elementi di *A* in *C*, si fa una seconda passata su *C*, nella quale a ogni elemento C[i] si somma il precedente;
- A questo punto:
  - *C[i]* indica la posizione corretta, nell'array ordinato, per l'elemento di valore *i* più a destra;
  - *C[i] - C[i-1]* indica quanti elementi ci sono con valore *i*
- si scorre quindi l'array *A* da destra a sinistra e, per ogni elemento *A[j]*:
  - si copia in *B A[j]=k* nella posizione giusta, che è *C[k]*;
  - si decrementa di 1 il valore *C[k]*.

Counting Sort – Esempio 2



Counting Sort – Esempio 2



E così via…

Counting Sort - 4

```
Funzione Counting_sort_con_Dati_Satellite (A)
   k=max(A); n=len(A)
   C[0]*(k+1); B=[0]*n
   for j in range(n)
      C[A[j]]+=1       //in C[i] ora c'è il # di elem. = i
   for i in range(1,k)
      C[i]+=C[i-1]     //in C[i] ora c'è il # di elem. ≤ i
   for j in range(n,-1)
      B[C[A[j]]-1]=A[j]
      C[A[j]]-=1
   return B
```

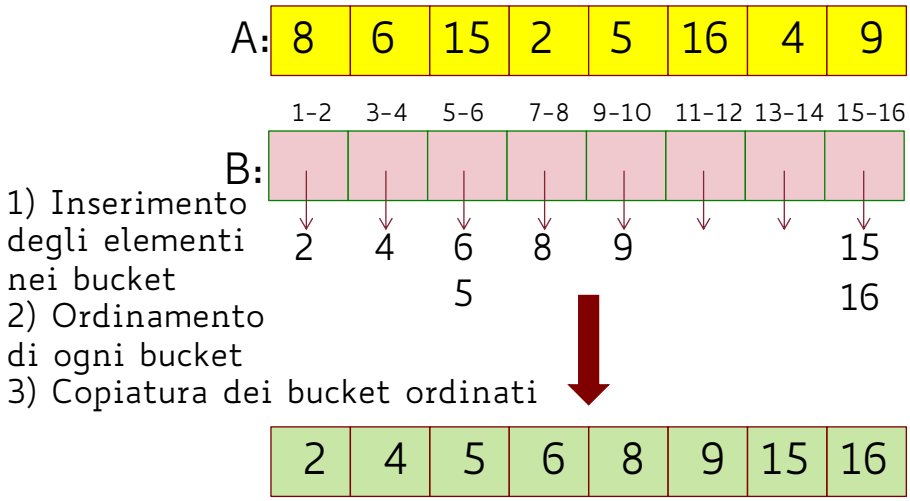
il cui costo è chiaramente $T(n) = \Theta(k) + \Theta(n)$

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 SENZA confrontare gli elementi tra loro.
- 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



Bucket sort - 2

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


Bucket sort - 3

- I costi di ordinamento dipendono dalla lunghezza delle liste.
- Ci sono n valori per n bucket; per l'ipotesi di equidistribuzione, 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 $T(n) = \Theta(n)$.

Esercizio svolto (1)

(dall'esame del 20/3/2023)

Dato un array A di n interi non negativi distinti, si vuole determinare se esistono almeno tre numeri consecutivi di valore inferiore a 100.

es.: se A = [101, 5, 9, 31, 33, 10, 100, 4, 8, 32, 500, 11, 99], gli elementi 8, 9 e 10 così come gli elementi 31, 32 e 33 rispettano la proprietà mentre 99, 100 e 101 no.

Progettare un algoritmo che, dato A, in tempo $\Theta(n)$ restituisce il valore dell'elemento centrale della terna se questa è presente, -1 altrimenti. Se esistono più terne allora bisogna restituire l'elemento centrale di valore massimo (nell'esempio sopra, l'algoritmo dovrebbe restituire 32).

Esercizio svolto (2)

Idea 1.

Scorrere l'array per ciascun elemento alla ricerca dei due elementi che possano far parte della terna: $\Theta(n^2)$ - NO

Idea 2.

Ordinare l'array e poi scorrerlo una sola volta alla ricerca di una eventuale terna: $\Omega(n \log n)$ (il costo esatto dipende dall'algoritmo di ordinamento usato) - NO

Esercizio svolto (3)

Idea 3.

Utilizzare una variante del Counting sort, con un array di lavoro C di dimensione 100: in posizione C[i] viene inserito il numero di occorrenze in A dell'intero i, solo se i < 100: $\Theta(n)$ – OK

Nota: una semplice chiamata del Counting Sort sull'intero array A NON garantisce costo lineare.

Elemento cercato: i tra 1 e 98 t.c. C[i-1], C[i], C[i+1] $\neq 0$; è il primo scorrendo C da destra; se si scorre C da sinistra bisogna tenere traccia dell'ultimo trovato.

Esercizio svolto (4)

Idea 4.

Applicare un algoritmo che antepone tutti gli elementi ≤ 100 agli altri (possiamo quindi scegliere 100 come soglia del classico algoritmo di partizionamento); poi procedere all'ordinamento della parte sinistra dell'array in tempo costante (poiché gli elementi sono tutti distinti non saranno più di 100) ed alla ricerca della terna: $\Theta(n)$ - OK

Esercizio svolto (5)

Idea 5.

Scorrere l'array A e, SOLO per ogni elemento A[i] < 100, cercare nell'array il precedente ed il successivo di A[i]. Anche qui viene garantito il costo lineare perché si scorre l'array A al più 100 volte: $\Theta(n)$ - OK

Corso di laurea in Informatica
Introduzione agli Algoritmi
A.A. 2024/25

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?