Algoritmi di ordinamento (I parte)

Fulvio CORNO - Matteo SONZA REORDA
Dip. Automatica e Informatica
Politecnico di Torino

Definizione formale del problema

Input:

- Una sequenza di *n* numeri $<a_1, a_2, ..., a_n>$

Output:

- Una permutazione $<a'_1, a'_2, ..., a'_n>$ degli elementi, tale che $a'_1 \leq a'_2 \leq ... \leq a'_n$

Tipologie di ordinamenti

- Ordinamento interno
  - I dati da ordinare sono tutti contenuti nella memoria centrale
  - Accesso diretto ai vari elementi
- Ordinamento esterno
  - I dati da ordinare non possono essere tutti caricati in memoria centrale contemporaneamente
  - Occorre agire direttamente sui dati memorizzati in file
  - Acccesso tipicamente sequenziale

Considerazioni pratiche

- Gli elementi da ordinare sono solitamente delle strutture (`struct`)
- Uno dei campi (o un valore calcolato a partire da uno o più campi) costituisce la *chiave* della struttura
- I restanti campi sono dati aggiuntivi
- L'ordinamento viene fatto con riferimento ai valori crescenti della chiave

Esempio

```
struct studente {
     int matricola ;
     char cognome[30] ;
     char nome[30] ;
     int voto ;
} ;

struct studente classe[100] ;
```


Esempio

```
struct studente {
     int matricola ;
     char cognome[30] ;
     char nome[30] ;
     int voto ;
} ;

struct studente classe[100] ;
```

Ordinamento per matricola

Ordinamento per cognome e nome (chiave = concatenazione di cognome e nome)

Ordinamento per voto (con valori ripetuti)

Stabilità

Un algoritmo di ordinamento si dice stabile quando, in presenza di elementi con ugual valore della chiave, nella sequenza risultante tali elementi compaiono nello stesso ordine in cui comparivano nella sequenza originaria.

Semplificazione

Nello studiare gli algoritmi di ordinamento, solitamente si lavora con vettori di interi di n elementi.

- int A[n] ;

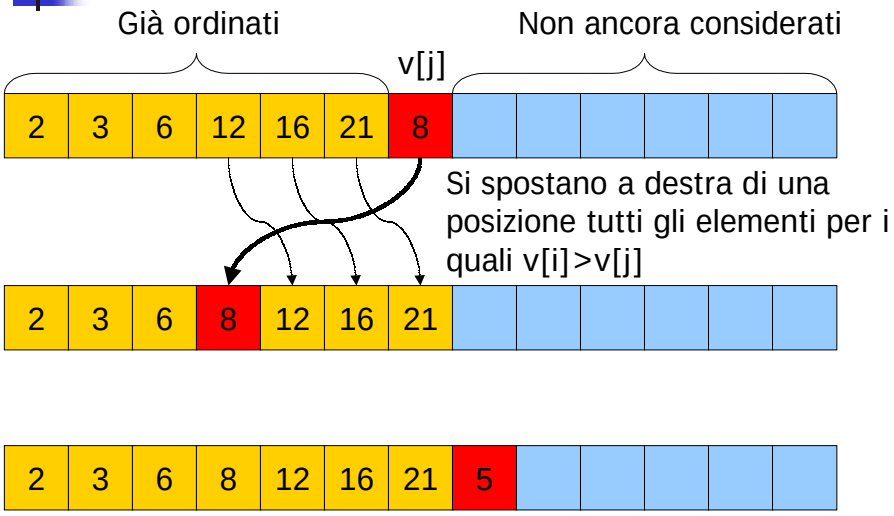
Algoritmi

Esistono numerosi algoritmi di ordinamento, di diversa complessità:

- $O(n^2)$: semplici, iterativi
  - Insertion sort, Selection sort, Bubble sort, ...
- O(n): applicabili solo in casi particolari
  - Counting sort, Radix sort, Bin (o Bucket) sort, ...
- O(n log n): più complessi, ricorsivi
  - Merge sort, Quicksort, Heapsort

Insertion sort

1.1



Pseudo-codice

```
INSERTION-SORT(A)
1   for j ← 2 to length[A]
2       do   key ← A[j]
3            ▷ Si inserisce A[j] nella sequenza ordinata A[1 . . j – 1]
4            i ← j – 1
5            while i > 0 e A[i] > key
6                 do  A[i + 1] ← A[i]
7                     i ← i – 1
8            A[i + 1] ← key
```


Implementazione C

```
void InsertionSort(int A[], int n)
{
     int i, j, key ;
     for(j=1; j<n; j++)
     {
          key = A[j] ;
          i = j - 1 ;
          while ( i >= 0 && A[i]>key )
          {
               A[i+1] = A[i] ;
               i-- ;
          }
          A[i+1] = key ;
     }
}
```

Dallo pseudo-codice al C

Avvertenze:

- In C, gli indici dei vettori vanno da `0` ad `n-1`, mentre lo pseudo-codice del Cormen usa l'intervallo da 1 ad n.
- L'indentazione del codice è utile, ma occorre anche inserire i delimitatori di blocco { … }

Complessità

Numero di confronti:

- $C_{min} = n-1$
- $C_{med} = \frac{1}{4}(n^2+n-2)$
- $C_{max} = \frac{1}{2}(n^2+n)-1$

Numero copie dati:

- $M_{min} = 2(n-1)$
- $M_{med} = \frac{1}{4}(n^2+9n-10)$
- $M_{max} = \frac{1}{2}(n^2+3n-4)$

Caso migliore: vettore già ordinato

Caso peggiore: vettore ordinato inversamente

$C = O(n^2)$, $M = O(n^2)$

$T(n) = O(n^2)$

$T(n)$ non è $\Theta(n^2)$

$T_{caso\ peggiore}(n) = \Theta(n^2)$

Altri algoritmi quadratici

		Min	Med	Max
Inserimento Diretto	$C =$	$n - 1$	$(n^2 + n - 2)/4$	$(n^2 - n)/2 - 1$
	$M =$	$2(n - 1)$	$(n^2 - 9n - 10)/4$	$(n^2 + 3n - 4)/2$
Selezione Diretta	$C =$	$(n^2 - n)/2$	$(n^2 - n)/2$	$(n^2 - n)/2$
	$M =$	$3(n - 1)$	$n(\ln n + 0.57)$	$n^2/4 + 3(n - 1)$
Scambio Diretto (Bubblesort)	$C =$	$(n^2 - n)/2$	$(n^2 - n)/2$	$(n^2 - n)/2$
	$M =$	0	$(n^2 - n)*0.75$	$(n^2 - n)*1.5$

Tabella 2.8 Confronto tra metodi di ordinamento diretti.

Tempi di esecuzione (in ms)

	Ordinati		Disordinati		Inversamente ordinati	
Inserimento diretto	12	23	366	1444	704	2836
Inserimento binario	56	125	373	1327	662	2490
Selezione diretta	489	1907	509	1956	695	2675
Bubblesort	540	2165	1026	4054	1492	5931
Bubblesort con segnalatore di scambio	5	8	1104	4270	1645	6542
Shakersort	5	9	961	3642	1619	6520
Shellsort	58	116	127	349	157	492
Heapsort	116	253	110	241	104	226
Quicksort	31	69	60	146	37	79
Fusione *	99	234	102	242	99	232
* *Vedi Sezione 2.3.1.*	n = 256	512	256	512	256	512

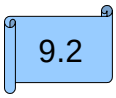
Tabella 2.9 Tempi d'esecuzione dei programmi di ordinamento.

Esercizio proposto

Si scriva lo pseudo codice, e si implementi in C l'algoritmo di Selection Sort. Si analizzi la complessità asintotica nel caso migliore, nel caso peggiore e nel caso medio.

Selection Sort: a ciascuna iterazione 'j' trovo il "min" nella parte di vettore non ancora ordinata, e lo metto alla posizione corretta 'j'.

Counting sort



Non è applicabile nel caso generale, in quanto si basa su un'ipotesi:

- Gli n elementi da ordinare sono numeri interi compresi in un intervallo tra 1 e k, con k intero.

Con questa ipotesi, se k = O(n), allora l'algoritmo ha una complessità solamente O(n).

Idea di base

Determinare, per ciascun elemento da ordinare x, quanti elementi vi sono minori di x.

Tale informazione permette di depositare x direttamente nella sua posizione finale nel vettore.

Strutture dati

- Si usano 3 vettori:
  - Vettore di partenza: A[1..n]
  - Vettore risultato: B[1..n]
  - Vettore di appoggio: C[1..k]
- Il vettore C tiene traccia del numero di elementi di A che hanno un certo valore: C[i] è il numero di elementi di A pari ad i.
- La somma dei primi i elementi di C determina il numero di elementi di A di valore <= i.

Pseudo-codice

```
COUNTING-SORT(A, B, k)
 1  for i ← 1 to k
 2       do C[i] ← 0
 3  for j ← 1 to length[A]
 4       do C[A[j]] ← C[A[j]] + 1
 5  ▷ C[i] now contains the number of elements equal to i.
 6  for i ← 2 to k
 7       do C[i] ← C[i] + C[i − 1]
 8  ▷ C[i] now contains the number of elements less than or equal to i.
 9  for j ← length[A] downto 1
10       do B[C[A[j]]] ← A[j]
11          C[A[j]] ← C[A[j]] − 1
```

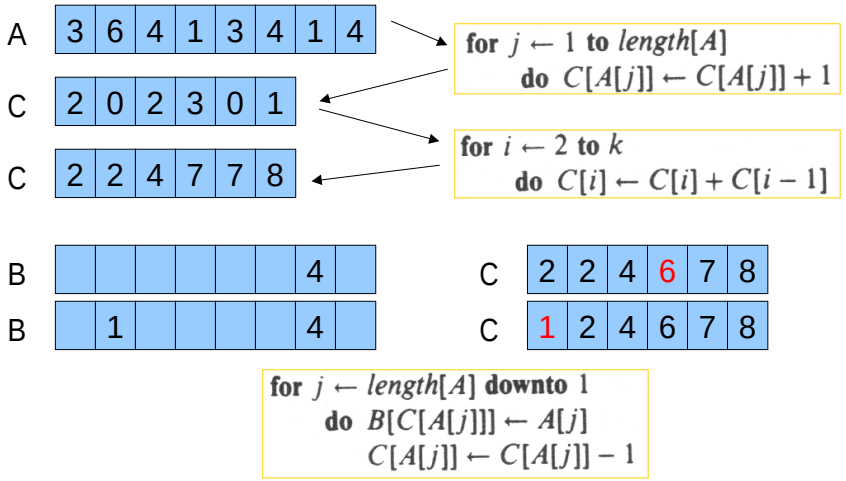

Analisi

Per ciascun j, C[A[j]] indica il numero di elementi minori od uguali ad A[j], e quindi indica anche la posizione finale di A[j] in B:

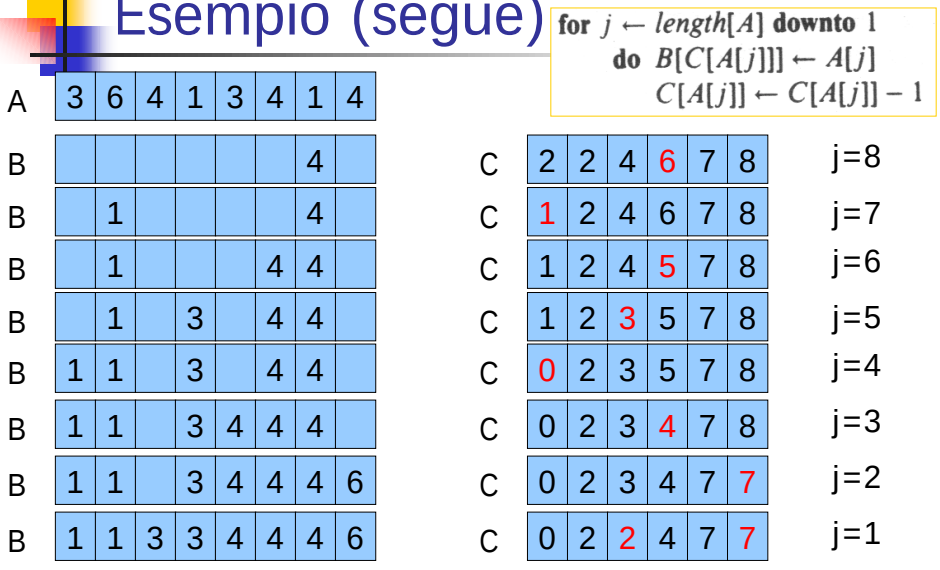
- B[C[A[j]]] = A[j]

La correzione C[A[j]] ← C[A[j]] – 1 serve a gestire la presenza di elementi duplicati.

Esempio (n=8, k=6)



Esempio (segue)



Complessità

- 1-2: Inizializzazione di C: O(k)
- 3-4: Calcolo di C: O(n)
- 6-7: Sommatoria in C: O(k)
- 9-11: Ricopiatura in B: O(n)

La complessità totale è quindi O(n+k).

L'algoritmo è utile solo quando k=O(n), per cui la complessità risultante è O(n).

Esercizio proposto

Si implementi l'algoritmo di Counting Sort in C, e se ne verifichi il corretto funzionamento.

Quali sono i valori accettabili per gli elementi del vettore di ingresso? Che cosa succede se il vettore in ingresso non rispetta l'ipotesi?