8.2 Counting sort

L'algoritmo ***counting sort*** suppone che ciascuno degli n elementi di input sia un numero intero compreso nell'intervallo da 0 a k, per qualche intero k. Quando $k = O(n)$, l'ordinamento viene effettuato nel tempo $\Theta(n)$.

Il concetto che sta alla base di counting sort è determinare, per ogni elemento di input x, il numero di elementi minori di x. Questa informazione può essere

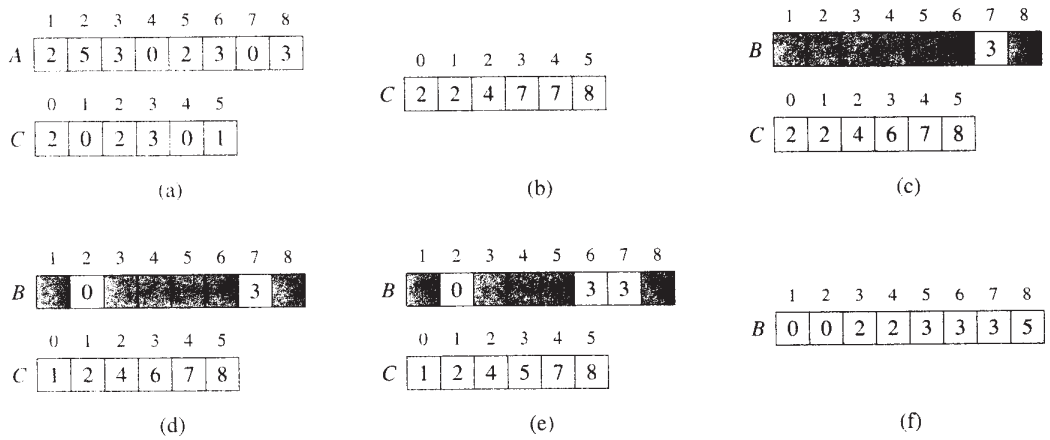

Figura 8.2 L'operazione di COUNTING-SORT su un array di input $A[1..8]$, dove ogni elemento di A è un intero non negativo non maggiore di $k = 5$. **(a)** L'array A e l'array ausiliario C dopo la riga 4. **(b)** L'array C dopo la riga 7. **(c)–(e)** L'array di output B e l'array ausiliario C, rispettivamente, dopo una, due e tre iterazioni del ciclo (righe 9–11). Le caselle di colore grigio chiaro nell'array B rappresentano gli elementi che sono stati inseriti. **(f)** L'array di output B è ordinato.

utilizzata per inserire l'elemento x direttamente nella sua posizione nell'array di output. Per esempio, se ci sono 17 elementi minori di x, allora x deve andare nella posizione di output 18. Questo schema deve essere modificato leggermente per gestire il caso in cui più elementi hanno lo stesso valore, per evitare che siano inseriti nella stessa posizione.

Nel codice di counting sort, supponiamo che l'input sia un array $A[1..n]$, quindi $length[A] = n$. Occorrono altri due array: l'array $B[1..n]$ contiene l'output ordinato; l'array $C[0..k]$ fornisce la memoria temporanea di lavoro.

```
COUNTING-SORT(A, B, k)
 1  for i ← 0 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] adesso contiene il numero di elementi uguali a i.
 6  for i ← 1 to k
 7      do C[i] ← C[i] + C[i − 1]
 8  ▷ C[i] adesso contiene il numero di elementi minori o uguali a i.
 9  for j ← length[A] downto 1
10      do B[C[A[j]]] ← A[j]
11         C[A[j]] ← C[A[j]] − 1
```

La Figura 8.2 illustra counting sort. Dopo l'inizializzazione nelle righe 1–2 del ciclo **for**, ogni elemento di input viene esaminato nelle righe 3–4 del ciclo **for**. Se il valore di un elemento di input è i, incrementiamo $C[i]$. Quindi, dopo la riga 4, $C[i]$ contiene il numero degli elementi di input uguali a i per ogni intero $i = 0.1, \ldots, k$. Le righe 6–7 determinano, per ogni $i = 0, 1, \ldots, k$, quanti elementi di input sono minori o uguali a i, mantenendo la somma corrente dell'array C.

Infine, le righe 9–11 del ciclo **for** inseriscono l'elemento $A[j]$ nella corretta posizione ordinata dell'array di output B. Se tutti gli n elementi sono distinti, quando viene eseguita per la prima volta la riga 9, per ogni $A[j]$, il valore $C[A[j]]$ rappresenta la posizione finale corretta di $A[j]$ nell'array di output, in quanto ci sono $C[A[j]]$ elementi minori o uguali ad $A[j]$. Poiché gli elementi potrebbero non essere distinti, $C[A[j]]$ viene ridotto ogni volta che viene inserito un valore $A[j]$ nell'array B. La riduzione di $C[A[j]]$ fa sì che il successivo elemento di input con un valore uguale ad $A[j]$, se esiste, venga inserito nella posizione immediatamente prima di $A[j]$ nell'array di output.

Quanto tempo richiede counting sort? Il ciclo **for** alle righe 1–2 impiega un tempo $\Theta(k)$, il ciclo **for** alle righe 3–4 impiega un tempo $\Theta(n)$, il ciclo **for** alle righe 6–7 impiega un tempo $\Theta(k)$ e il ciclo **for** alle righe 9–11 impiega un tempo $\Theta(n)$. Quindi, il tempo totale è $\Theta(k + n)$. Di solito counting sort viene utilizzato quando $k = O(n)$, nel qual caso il tempo di esecuzione è $\Theta(n)$.

Counting sort batte il limite inferiore di $\Omega(n \lg n)$ dimostrato nel Paragrafo 8.1 perché non è un ordinamento per confronti. Infatti, il codice non effettua alcun confronto fra gli elementi di input. Piuttosto, counting sort usa i valori effettivi degli elementi come indici di un array. Il limite inferiore $\Omega(n \lg n)$ non vale se ci allontaniamo dal modello di ordinamento per confronti.

Un'importante proprietà di counting sort è la ***stabilità***: i numeri con lo stesso valore si presentano nell'array di output nello stesso ordine in cui si trovano nell'array di input. Ovvero, l'uguaglianza di due numeri viene risolta applicando la seguente regola: il numero che si presenta per primo nell'array di input sarà inserito per primo nell'array di output. Normalmente, la proprietà della stabilità è importante soltanto quando i dati satellite vengono spostati insieme con gli elementi da ordinare. La stabilità di counting sort è importante per un'altra ragione: counting sort viene spesso utilizzato come subroutine di radix sort. Come vedremo nel prossimo paragrafo, la stabilità di counting sort è cruciale per la correttezza di radix sort.

Esercizi

8.2-1

Utilizzando la Figura 8.2 come modello, illustrate l'operazione di COUNTING-SORT sull'array $A = \langle 6, 0, 2, 0, 1, 3, 4, 6, 1, 3, 2 \rangle$.

8.2-2

Dimostrate che COUNTING-SORT è stabile.

8.2-3

Supponete che la riga 9 del ciclo **for** nella procedura COUNTING-SORT sia modificata così

9 **for** $j \leftarrow 1$ **to** $length[A]$

Dimostrate che l'algoritmo opera ancora correttamente. L'algoritmo modificato è stabile?

8.2-4

Descrivete un algoritmo che, dati n numeri interi compresi nell'intervallo da 0 a k, svolga un'analisi preliminare del suo input e poi risponda nel tempo $O(1)$ a qualsiasi domanda su quanti degli n interi ricadono nell'intervallo $[a\,..\,b]$. Il vostro algoritmo dovrebbe impiegare un tempo $\Theta(n + k)$ per l'analisi preliminare.