ESERCITAZIONI DI *INTRODUZIONE AGLI ALGORITMI*
(A.A. 09/10, CANALE E-O)

APPUNTI ESERCITAZIONE N. 7

1. Counting Sort

Abbiamo visto che nel modello basato sui confronti esiste un lower bound assoluto di $\Omega(n \cdot \log(n))$ alla complessità di un algoritmo di ordinamento.

Introduciamo ora un algoritmo di ordinamento che non è basato su confronti, il Counting Sort (o Integer Sort). L'algoritmo ordina n elementi. La complessità dell'algoritmo non dipende soltanto da n ma anche dall'intervallo in cui gli n elementi sono scelti! Se questi elementi sono presi nell'intervallo $[0, k]$, allora la complessità dell'algoritmo è $\Theta(n + k)$. Quindi, se k non è troppo grande rispetto ad n (ossia se i numeri non sono troppo distanti l'uno dall'altro), la complessità dell'algoritmo è ben al di sotto di $\Omega(n \log(n))$. Per esempio, se k è $O(n)$, allora la complessità è addirittura lineare in n.

L'idea dell'algoritmo è la seguente. Supponiamo per il momento di dover ordinare n elementi tutti distinti tra loro. Per conoscere la destinazione finale di un elemento i nel vettore ordinato, è sufficiente sapere quanti elementi $\leq i$ sono presenti nel vettore. Analogamente, se nel vettore ci sono ripetizioni di elementi, per ogni elemento i, per ogni occorrenza di i nel vettore, per conoscerne la posizione finale è sufficiente sapere quante occorrenze di elementi $\leq i$ ci sono nel vettore.

Possiamo organizzare questa informazione in una tabella, che associa ad ogni valore possibile i nell'intervallo $[0, k]$, il numero delle occorrenze di elementi $\leq i$ che appaiono nel vettore. Visualizziamo la tabella in ordine crescente rispetto al valore di i.

Per esempio, se il vettore iniziale è $A = 26112340$, allora la tabella è

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

Nella prima riga abbiamo i tipi di elementi possibili, gli interi da 0 a k. In corrispondenza di ciascuno di questi valori, abbiamo il numero delle occorrenze in A di valori più piccoli o uguali al valore in questione.

Data l'informazione contenuta nella tabella, possiamo costruire il vettore ordinato come segue. Scorriamo il vettore A da destra a sinistra e costruiamo il vettore ordinato B passo passo. Ad ogni passo posizioneremo una occorrenza di un elemento di A nella sua posizione finale nel vettore ordinato B. Partiamo dall'ultimo elemento di A e leggiamo il valore associato nella tabella. Questo valore ci dà la posizione finale (l'indice) dell'elemento nel vettore ordinato. Nell'esempio posizioniamo il valore 0 nella posizione 1 del vettore ordinato. Dobbiamo poi ricordarci di aggiornare la tabella, decrementando il valore associato all'elemento di cui abbiamo posizionato una occorrenza. In tal modo, quando incontreremo la prossima

Note preparate da Lorenzo Carlucci, carlucci@di.uniroma1.it.

occorrenza dello stesso elemento, avremo nella tabella il valore corretto rispetto alla nuova occorrenza da ordinare. La prima colonna della tabella diventerà allora $0|0$. All'inizio di ogni nuovo passo della procedura il valore che la tabella associa ad un elemento i è il numero delle occorrenze di elementi $\leq i$ in A *che non abbiamo ancora considerato*! Guardiamo ora il penultimo elemento di A, che è 4, e consultiamo la tabella per conoscerne la posizione finale, che è 6. Aggiorniamo la tabella ponendo $4|5$ nella quinta colonna. Procedendo così, quando avremo finito di scorrere il vettore A avremo anche posizionato tutti i suoi elementi nella loro posizione corretta, i.e. il vettore $B = 01122346$.

Vediamo ora come si può ricavare efficientemente l'informazione contenuta nella tabella dal vettore input *senza usare confronti* ossia senza mai confrontare due elementi di A tra loro.

Usiamo un vettore, C, per rappresentare la tabella. La tabella ha dimensione $k + 1$, perché contiene l'informazione per ogni possibile elemento nell'intervallo $[0, k]$, non solo per quelli presenti effettivamente in A? Perché? Perché l'idea che ci permette di evitare confronti è di usare i *valori* degli elementi di A come *indici* del vettore C. E' quanto abbiamo fatto implicitamente scrivendo la tabella come sopra: alla colonna che ha il valore i nella prima riga abbiamo associato l'informazione relativa all'elemento i del vettore A. L'idea di usare valori come indici (e viceversa) è molto diffusa in informatica (anche in informatica teorica).

Diamo ora lo pseudocodice del CountingSort. L'algoritmo usa un vettore di appoggio C di dimensione $k+1$, che supponiamo inizializzato tutto a 0 (altrimenti possiamo inizializzarlo al costo di un ulteriore ciclo di ordine k). L'algoritmo prende come parametri il vettore iniziale disordinato A e il vettore destinazione B, nonché la dimensione dell'intervallo in cui sono scelti i valori in A.

algoritmo *CountingSort*(A, B, k)
for $j = 1$ **to** n **do**
  $C[A[j]] = C[A[j]] + 1$
for $i = 1$ **to** k **do**
  $C[i] = C[i] + C[i - 1]$
for $j = n$ **to** 1 **do**
  $B[C[A[j]]] = A[j]$; $C[A[j]] = C[A[j]] - 1$

Si osservi che

- Dopo il primo ciclo la cella $C[i]$ contiene il numero delle occorrenze del valore i in A.
- Dopo il secondo ciclo la cella $C[i]$ contiene il numero delle occorrenze di valori fino a i (compreso) in A.

L'algoritmo evita di ricorrere a confronti sfruttando la "confusione" tra valori e indici, secondo lo schema seguente. Il *valore* $A[j]$ della j-esima cella di A viene usato come *indice* per accedere al vettore C. Il *valore* della cella di C di *indice* $A[j]$ contiene a sua volta l'*indice* di posizione del *valore* $A[j]$ nel vettore ordinato...

$$j \to A[j] \to C[A[j]] \to B[C[A[j]]].$$

La complessità dell'algoritmo si calcola facilmente: i tre cicli non sono annidati, il primo costa $\Theta(n)$, il secondo $\Theta(k)$, il terzo $\Theta(n)$. La complessità è allora

$$\Theta(n) + \Theta(k) + \Theta(n) = \Theta(n + k).$$

Esercizio: Qual è l'occupazione di memoria del Counting Sort? Confrontare con la memoria necessaria per gli algoritmi di ordinamento per confronti.

2. Esercizi

Esercizio 1 Diamo una soluzione del problema 4.7 del libro. Il problema e la soluzione hanno somiglianze con la procedura Partition del QuickSort, con due differenze da tenere presenti: (i) il perno rispoetto a cui si ordinano gli altri valori è fisso (è il colore Bianco), e (ii) il perno può avere molte occorrenze che vanno tutte accumulate al centro del vettore.

Usiamo tre variabili. v indica la prima posizione dove possiamo inserire un elemento verde, r indica la prima posizione dove possiamo inserire un elemento rosso, i è l'indice di scorrimento e indica la prima posizione dove inserire un elemento bianco. Inizializziamo

$$v = 1; \;\; i = 1; \;\; r = n.$$

algoritmo *Bandiera*(B)
while $i \leq r$ **do**
  if $A[i] = B$ **then**
    i = i+1;
  if $A[i] = V$ **then**
    scambia $A[i]$ e $A[v]$;
    $i = i + 1$;
    $v = v + 1$;
  else
    scambia $A[i]$ e $A[r]$;
    $r = r - 1$;

Esercizio: formulare un invariante e dimostrare la correttezza dell'algoritmo.

L'algoritmo ha costo lineare: la variabile i scorre sul vettore A da sinistra a destra, la variabile r da destra a sinistra. Ad ogni iterazione del ciclo While, o incrementiamo i o decrementiamo r. L'iterazione è dunque limitata dalla lunghezza del vettore.

Diamo un esempio di esecuzione dell'algoritmo. Sopra il vettore indichiamo le posizioni delle variabili v, i, r ad ogni passo. I valori sottolineati sono quelli che vengono scambiati. Un elemento sottolineato due volte viene scambiato con se stesso.

v,i								r
$\underline{\text{R}}$	V	B	R	V	B	R	V	$\underline{\text{B}}$
v,i							r	
B	V	B	R	V	B	R	V	R
v	i						r	
$\underline{\text{B}}$	$\underline{\text{V}}$	B	R	V	B	R	V	R
	v	i					r	
V	B	B	R	V	B	R	V	R
	v		i				r	
V	B	B	$\underline{\text{R}}$	V	B	R	$\underline{\text{V}}$	R

	v		i			r		
V	$\underline{\text{B}}$	B	$\underline{\text{V}}$	V	B	R	R	R
		v		i		r		
V	V	$\underline{\text{B}}$	B	$\underline{\text{V}}$	B	R	R	R
			v		i	r		
V	V	V	B	B	B	R	R	R
			v			i,r		
V	V	V	B	B	B	$\underline{\underline{\text{R}}}$	R	R
			v			r	i	
V	V	V	B	B	B	R	R	R

Si osserva che l'algoritmo può fare molti scambi inutili su alcuni tipi di input. Un esempio (dal secondo passo):

	v,i							r
V	$\underline{\text{R}}$	V	B	V	B	R	R	$\underline{\text{R}}$
	v,i						r	
V	$\underline{\text{R}}$	V	B	V	B	R	$\underline{\text{R}}$	R
	v,i					r		
V	$\underline{\text{R}}$	V	B	V	B	$\underline{\text{R}}$	R	R
	v,i				r			
V	$\underline{\text{R}}$	V	B	V	$\underline{\text{R}}$	R	R	R

Esercizio: Come si possono evitare questi sprechi? Si migliora così la complessità dell'algoritmo?

Esercizio 2 Si richiede di ricostruire un albero binario a partire dalle sue proiezioni in ordine e in preordine. I.e., dati in input due vettori *inOrd* e *preOrd* contenenti i risultati delle visite - rispettivamente - in ordine e in pre ordine di uno stesso albero T, progettare un algoritmo ricorsivo per ricostruire T. Si chiede qui di estrarre un algoritmo generale dalla procedura usata per risolvere il Problema 3.7 del libro. Se V è un vettore e $i \leq j$ si indica con $V[i, j]$ il vettore contenente le celle $V[i], \ldots, V[j]$ di V (nello stesso ordine che in V). Se $i > j$ conveniamo che $V[i, j]$ denota il vettore vuoto. Per semplicità di scrittura assumiamo che i sottovettori di V su cui avvengono le chiamate ricorsive siano copiati in nuovi vettori (così da essere indicizzati a partire da 1). Nella soluzione qui sotto il sottovettore cui vogliamo accedere inizia sempre con indice 1 e accediamo all'ultima posizione con una funzione $dim(V)$ che restituisce la dimensione del vettore. L'algoritmo si può modificare in modo da evitare di copiare i sottovettori per le chiamate ricorsive, a costo di usare parametri espliciti per l'accesso al primo e all'ultimo elemento dei sottovettori (Esercizio). L'algoritmo usa come subroutine una procedura trova(chiave k, vettore V) che restituisce l'indice di una cella con chiave k nel vettore V (stiamo assumendo che le chiavi dell'albero siano tutte distinte). L'idea dell'algoritmo è di usare il vettore in-ordine per identificare il prossimo nodo da aggiungere all'albero, e di usare il vettore in pre-ordine per identificare i sottoalberi sinistri e destri radicati in quel nodo.

algoritmo $R(inOrd, preOrd) \rightarrow$ tree
if $dim(inOrd) = 0$ **then**
    return NULL;
else
    crea nuovo nodo n;
    $n.val = preOrd[1]$;
    $i = trova(n.val, inOrd)$;
    $n.sin = R(inOrd[1, i-1], preOrd[2, i])$
    $n.des = R(inOrd[i+1, dim(inOrd)], preOrd[i+1, dim(preOrd)]$
    return n