

Esercizi di Algoritmi e Strutture Dati

Ultimo aggiornamento: 23 novembre 2009

Moreno Marzolla
marzolla@cs.unibo.it

20 Novembre 2009

1 Algoritmi di ordinamento

1.1 Complessità di RadixSort

Si consideri l'algoritmo di ordinamento RadixSort visto a lezione. Supponiamo di voler ordinare n interi appartenenti all'intervallo $[0, k - 1]$, rappresentati in base $b = \Theta(n)$. Dimostrare che la complessità di RadixSort in questo caso è

$$O\left(n\left(1 + \frac{\log k}{\log n}\right)\right)$$

1.2 MergeSort modificato

(Esercizio 4.12 del libro di testo). Consideriamo la seguente variante dell'algoritmo MergeSort, in cui una delle chiamate ricorsive a MergeSort è rimpiazzata con una chiamata all'algoritmo QuickSort:

```
Algoritmo MergeSort2( Array A, indici i e j )
begin
  if ( i < j ) then
    m = (i+j)/2;
    MergeSort2(A, i, m);
    QuickSort(A, m+1, j);
    merge( A[i, m], A[m+1, j] );
  endif
end
```

Assumendo che l'array A abbia indici che vanno da 1 a n , l'array viene ordinato con la chiamata $\text{MergeSort2}(A, i, n)$; la chiamata $\text{QuickSort}(A, m+1, j)$ ordina il sotto-vettore $A[m+1, j]$ invocando l'algoritmo QuickSort. Infine, la chiamata $\text{merge}(A[i, m], A[m+1, j])$ effettua la fusione dei due sotto-vettori ordinati $A[i, m]$ e $A[m+1, j]$ scrivendo il risultato nel sotto-vettore $A[i, j]$. La notazione $A[i, j]$ denota il sotto-vettore composto dagli elementi $A[i], A[i+1], \dots, A[j]$.

Determinare la complessità dell'algoritmo MergeSort2 nel caso pessimo.

2 Struttura dati heap

2.1 Inserimenti di elementi negli heap

(Problema 4.3 del libro di testo) Si consideri una struttura dati di tipo min-heap, memorizzata in un vettore H come visto a lezione. Se lo heap contiene n elementi, questi sono memorizzati in $H[1], \dots, H[n]$.

- Definire una funzione `heap-insert(H,k)` per inserire il nuovo valore k in un min-heap H .
- Quanto costa l'operazione `heap-insert(H,k)` nel caso peggiore?
- Supponiamo di voler costruire un min-heap partendo da zero, inserendo ripetutamente gli elementi usando la funzione `heap-insert`. Quanto costa la costruzione di un min-heap in questo modo?
- Costruire uno heap usando `heap-insert` anziché `heapify` modificherebbe il costo dell'algoritmo `heapSort` nel caso peggiore?

3 Esercizi vari

3.1 Minima distanza

Consideriamo un array A di numeri reali $A[1], A[2], \dots, A[n]$, non necessariamente ordinati. Definiamo la *distanza* $d(i, j)$ tra gli elementi $A[i]$ e $A[j]$ (con $i \in [1, n]$, $j \in [1, n]$, $i \neq j$) come la differenza in valore assoluto tra $A[i]$ e $A[j]$:

$$d(i, j) = |A[i] - A[j]|$$

- Descrivere un algoritmo $\Theta(n^2)$ per determinare il valore minimo tra tutte le distanze $d(i, j)$. (Suggerimento: questo è l'algoritmo banale che calcola esplicitamente tutte le distanze $d(i, j)$ per ogni $i \neq j$).
- Descrivere un algoritmo *efficiente* per determinare il valore minimo della distanza $d(i, j)$, dimostrandone la correttezza e calcolandone il costo computazionale. (Suggerimento: l'algoritmo dovrebbe avere costo $O(n \log n)$).
- Descrivere un algoritmo *efficiente* per determinare il valore massimo della distanza $d(i, j)$, dimostrandone la correttezza e calcolandone il costo computazionale. (Suggerimento: l'algoritmo dovrebbe avere costo $O(n)$).

Nota: non è richiesto di individuare gli indici i e j che minimizzano la distanza $d(i, j)$. È richiesto solo il calcolo del valore minimo (e massimo) di tale distanza.

3.2 Intervalli

Supponiamo di avere n intervalli chiusi nell'asse reale, memorizzati in un array A . In particolare, ciascun elemento dell'array contiene l'estremo inferiore $A[i].\text{inf}$ e superiore $A[i].\text{sup}$ dell'intervallo i -esimo, che quindi corrisponderà all'intervallo chiuso $[A[i].\text{inf}, A[i].\text{sup}]$. Si può assumere che, per ogni $1 \leq i \leq n$ valga sempre $A[i].\text{inf} < A[i].\text{sup}$ (quindi non ci sono mai intervalli degeneri, e il limite inferiore è sempre strettamente minore di quello superiore).

Descrivere una procedura efficiente che, dato in input un array A di n intervalli, restituisce un valore booleano che vale **true** se esistono almeno due intervalli distinti che si intersecano, e **false** altrimenti.

Ad esempio, dati in input gli intervalli $\{[1, 3], [8, 9], [7.5, 8], [2, 7]\}$ la funzione restituisce **true** perché ad esempio $[1, 4]$ e $[2, 7]$ si intersecano (anche $[7.5, 8]$ e $[8, 9]$ si intersecano). Dati in input gli intervalli $\{[1, 3], [9, 10], [4, 5], [7, 8], [-1, 0]\}$ la funzione restituisce **false** perché tutti gli intervalli sono disgiunti.

3.3 Un elemento “qualsiasi”

(Esercizio 5.1 del libro di testo). Progettare un algoritmo che dato in input un array A di n numeri *distinti*, con $n > 2$, restituisce in tempo $O(1)$ un elemento di A che non sia né il minimo né il massimo.

Cosa cambia se si assume che i numeri non siano necessariamente distinti?

3.4 Ricerca su ABR con chiavi duplicate

A lezione abbiamo visto come effettuare la ricerca di un nodo in un ABR contenente una chiave k data. Se la chiave è presente più volte, l'algoritmo di ricerca visto a lezione si ferma una volta individuato un nodo (qualsiasi) contenente k .

Modificare l'algoritmo per restituire la lista di *tutti* i nodi che contengono la chiave k . Stimare il costo dell'operazione di ricerca nel caso peggiore.