

Soluzioni di Algoritmi e Strutture Dati

Ultimo aggiornamento: 26 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

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

Soluzione Sappiamo che l'algoritmo RadixSort applicato all'ordinamento di n valori rappresentati da d cifre in base b ha costo

$$O(d(n + b)) \tag{1}$$

Nel nostro caso, il numero di cifre in base b che sono necessarie per rappresentare gli interi da 0 a $k - 1$ è approssimativamente $\log_b k$. Poiché b è proporzionale a n , il numero di cifre è approssimativamente

$$d \approx \log_n k = \frac{\log k}{\log n} \tag{2}$$

(usando la regola di passaggio di base dei logaritmi). Sostituendo (2) in (1), e “assorbendo” il termine $b = \Theta(n)$ otteniamo

$$O\left(\frac{\log k}{\log n} n\right) \tag{3}$$

Si noti che non possiamo fermarci qui: infatti, se k fosse una costante molto minore di n , l'equazione (3) si ridurrebbe a $O(n/\log n)$ ma l'algoritmo RadixSort effettua comunque una scansione di tutti gli n elementi (operazione che ha costo

$O(n)$). È quindi necessario aggiungere un termine n a (3) per tenere conto di questa possibilità, ottenendo:

$$O\left(n\left(1 + \frac{\log k}{\log n}\right)\right) \quad (4)$$

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 MergeSort2(A, i, n); la chiamata QuickSort(A, m+1, j) ordina il sotto-vettore $A[m+1 \dots j]$ invocando l'algoritmo QuickSort. Infine, la chiamata merge(A[i, m], A[m+1, j]) effettua la fusione dei due sotto-vettori ordinati $A[i \dots m]$ e $A[m+1 \dots j]$ scrivendo il risultato nel sotto-vettore $A[i \dots j]$. La notazione $A[i \dots j]$ denota il sotto-vettore composto dagli elementi $A[i], A[i+1], \dots, A[j]$.

Determinare la complessità dell'algoritmo MergeSort2 nel caso pessimo.

Soluzione L'algoritmo MergeSort2 è ricorsivo, e possiamo scrivere la relazione di ricorrenza del costo computazionale $T(n)$ come segue:

$$T(n) = T(\lfloor n/2 \rfloor) + \Theta(\lceil n/2 \rceil^2) + \Theta(n)$$

infatti nel caso peggiore, QuickSort ha costo $\Theta(n^2)$, mentre l'operazione merge, come noto, richiede tempo $\Theta(n)$ in generale.

Osserviamo che, per n sufficientemente grande, i due termini Θ possono essere assorbiti in un unico termine $\Theta(n^2)$. Quindi possiamo riscrivere la ricorrenza come:

$$T(n) = T(\lfloor n/2 \rfloor) + \Theta(n^2)$$

Possiamo applicare il caso (3) del master theorem, dato che con $a = 1$, $b = 2$, $f(n) = n^2$ si ha $\Theta(n^2) = \Omega(n^{\log_b a + \epsilon})$ dove $\epsilon = 2$, e vale $af(n/b) \leq cf(n)$ per un $c < 1$ (ad esempio basta prendere $c = 1/4$). Concludiamo quindi che $T(n) = \Theta(n^2)$.

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?

Soluzione Consideriamo innanzitutto l'operazione `heap-insert`. Per inserire un nuovo elemento nel min-heap, è sufficiente inserirlo in ultima posizione, e scambiarlo ripetutamente col proprio padre fino a quando non raggiunge la posizione corretta. Lo pseudocodice è il seguente (assumiamo che n rappresenti il numero di elementi di H):

```
Algorithm heap-insert(H, k)
  n=n+1;
  H[n] = k;
  // NB: tutti i rapporti n/2 si intendono arrotondati
  // all'intero inferiore
  while( (n>1) && (H[n/2] > H[n]) ) do
    swap H[n] and H[n/2];
    n=n/2;
  end while
```

Lo pseudocodice di cui sopra richiede che le divisioni $n/2$ siano arrotondate all'intero inferiore, e che la condizione del ciclo `while` sia valutata in stile Java (cioè se il primo termine dell'`and` logico è falso, tutta l'espressione è falsa, senza bisogno di valutare anche il secondo termine).

Dimostriamo la correttezza della procedura di cui sopra, verificando che il ciclo `while` mantiene la seguente invariante:

1. $H[n]$ contiene la chiave k ;
2. L'albero radicato in $H[n]$ è un min-heap

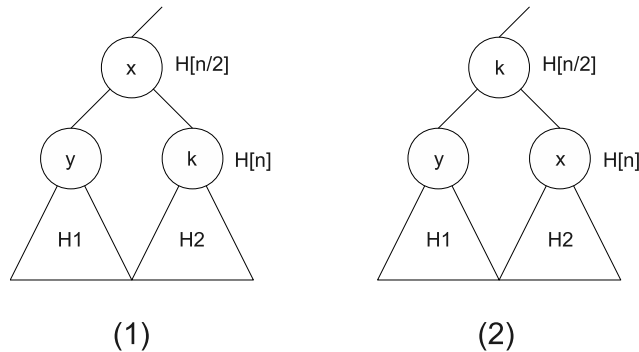


Figure 1: Dimostrazione dell’algoritmo di inserimento in uno heap

Inizialmente $H[n]$ contiene il valore k , ed è una foglia, quindi è un min-heap per costruzione. Dimostriamo ora che ogni volta che viene eseguita una iterazione del ciclo `while`, l’invariante rimane valida.

Prima dell’esecuzione del corpo del ciclo, la situazione è quella rappresentata in figura 1 (1). Poiché l’invariante vale, si ha che l’albero radicato in $H[n]$ (denotato come H2 in figura) è un min-heap e $H[n]$ contiene il valore k . L’esecuzione del corpo del ciclo `while` scambia i valori di $H[n]$ e $H[n/2]$, producendo il risultato mostrato in figura 1 (2). Dimostriamo ora che l’invariante di ciclo vale anche dopo lo scambio, dopo aver settato $n \leftarrow n/2$.

La radice $H[n/2]$ contiene ancora il valore k . In più, dato che la struttura dati H , prima dell’inserimento di k , era un min-heap, e dato che il sottoalbero H1 non è stato interessato da modifiche, si ha che $x \leq y$. L’iterazione del ciclo `while` è stata eseguita perché $k < x$ da cui $k < x \leq u$. Quindi avere messo il valore k nell’elemento $H[n/2]$ non ha violato la proprietà di min-heap nei confronti dei figli sinistro (che contiene il valore y) e destro (che contiene il valore x).

Dimostriamo ora che l’aver messo il valore x come radice dell’albero H2 non ha violato la proprietà di min-heap di H2. Osserviamo che, prima dell’inserimento della nuova chiave k , la struttura H era un min-heap; in particolare, il figlio sinistro del nodo $H[n/2]$ era costituito dall’albero H2 privo dell’elemento k (che è stato aggiunto in seguito). Quindi la proprietà di min-heap garantiva che $x \leq v$, per ogni $v \in H2 \setminus \{k\}$, ossia che il valore x era minore o uguale di ogni valore contenuto in H2, ad esclusione della nuova chiave k . Se si assegna x all’elemento $H[n]$, la considerazione appena fatta, unita all’invariante di ciclo che era valida per H2 prima dello scambio tra $H[n]$ e $H[n/2]$ ci permette di concludere che l’albero H2 in figura 1 (2) è ancora un min-heap.

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.

Soluzione L'algoritmo "banale" può essere descritto dallo pseudocodice seguente:

```
Algorithm MinDist(A) -> number
mindist = +inf; // Sarebbe meglio |A[1]-A[2]|
for i=1 to n do
  for j=1 to n do
    if (i != j) then
      dist = |A[i] - A[j]|;
      if (dist < mindist) then
        mindist = dist;
      endif
    endif
  endfor
endfor
return mindist
```

Questo algoritmo contiene due cicli "for" annidati, con il corpo del ciclo più interno che ha costo $O(1)$. Quindi il costo complessivo è $\Theta(n^2)$.

Supponiamo ora di ordinare gli elementi nell'array A . È possibile dimostrare che la distanza minima è il minimo valore della distanza di un elemento e il suo successivo nell'array ordinato.

Per assurdo, consideriamo l'array A ordinato e supponiamo che la distanza minima sia $d(i, j)$ per un qualche $j > i + 1$ nell'array ordinato (se $i > j$ possiamo

scambiare i ruoli di i e j). Allora esiste un indice k , con $i < k < j$ tale che $A[i] \leq A[k] \leq A[j]$. Se l'ultima disuguaglianza vale in senso stretto ($A[k] < A[j]$) si avrebbe che $d(i, k) < d(i, j)$ il che contraddice l'ipotesi.

Quindi un algoritmo piú efficiente per risolvere questo problema è rappresentato dallo pseudocodice seguente:

```

Algorithm MinDistEff(A) -> number
  MergeSort(A);
  mindist = A[2]-A[1];
  for i=2 to n-1 do
    if ( A[i+1]-A[i] < mindist ) then
      mindist = A[i+1]-A[i];
    endif
  endfor
  return mindist;

```

Il costo di questo algoritmo è dominato dal costo della procedura di ordinamento, che è $O(n \log n)$.

Per calcolare la distanza massima, è facile dimostrare che questa è la differenza tra il valore massimo e minimo in A . Poiché individuare il massimo e il minimo costa $\Theta(n)$, l'algoritmo per calcolare la distanza massima ha costo $\Theta(n)$.

3.2 Intervalli

Supponiamo di avere n intervalli chiusi nell'asse reale, memorizzati in un array A . In particolare, ciascun elemento $A[i]$ dell'array contiene l'estremo inferiore $A[i].inf$ e superiore $A[i].sup$ dell'intervallo i -esimo, che quindi corrisponderà all'intervallo chiuso $[A[i].inf, A[i].sup]$. Si può assumere che, per ogni $1 \leq i \leq n$ valga sempre $A[i].inf < A[i].sup$ (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.

Soluzione Come nell'esercizio precedente, l'algoritmo "banale" richiederebbe di confrontare ciascun intervallo con tutti gli altri (meno se stesso), con costo complessivo $\Theta(n^2)$.

Un algoritmo piú efficiente consiste nell'ordinare gli intervalli in base al valore dell'estremo inferiore ($A[i].inf$), e confrontare ciascun intervallo nella lista ordinata con il suo successore. Non appena si trovano due intervalli che si intersecano, si ritorna "true". Se si arriva al termine dell'array, si ritorna "false". Si noti che capire se due intervalli $A[i]$ e $A[j]$ si intersecano richiede tempo $O(1)$.



Figure 2: Se $[l_i, u_i]$ interseca $[l_j, u_j]$ per qualche $i < j$, allora $[l_i, u_i]$ interseca $[l_{i+1}, u_{i+1}]$

Vediamo innanzitutto lo pseudocodice

```

Algorithm Intersect(A) -> boolean
  sort A ascending, according to A.inf;
  for i=1 to n-1 do
    if ( A[i].sup >= A[i+1].inf )
      return true;
    endif
    // (*)
  endfor
  return false;

```

Osserviamo che, se gli intervalli sono ordinati in base all'estremo inferiore, per testare se $A[i]$ e $A[i + 1]$ si intersecano è sufficiente verificare se $A[i].sup \geq A[i + 1].inf$.

Dimostriamo ora la correttezza dell'algoritmo proposto. Consideriamo un insieme di intervalli $\{[l_1, u_1], [l_2, u_2], \dots, [l_n, u_n]\}$ con $l_i < u_i$ e per ogni $i = 1, 2, \dots, n - 1$ vale $l_i \leq l_{i+1}$. Allora due intervalli dell'insieme si intersecano se e solo se esiste $k \in [1, n - 1]$ tale che $u_k \geq l_{k+1}$.

\Rightarrow Supponiamo che $[l_i, u_i]$ e $[l_j, u_j]$ (con $i < j$) si intersechino, e dimostriamo che anche $[l_i, u_i]$ e $[l_{i+1}, u_{i+1}]$ si intersecano. Se $j = i + 1$, la tesi segue immediatamente dall'ipotesi. Supponiamo quindi che $j > i + 1$. Poiché tutti gli intervalli di A sono stati ordinati in base all'estremo inferiore, si ha per costruzione che (vedi figura 2)

$$l_i \leq l_{i+1} \leq l_j \tag{5}$$

Dato che per ipotesi $[l_i, u_i]$ e $[l_j, u_j]$ si intersecano, e poiché $l_i \leq l_j$, abbiamo necessariamente che

$$l_j \leq u_i \tag{6}$$

Quindi possiamo combinare (5) con (6) ottenendo

$$l_{i+1} \leq l_j \leq u_i$$

cioè, in particolare, abbiamo che

$$l_{i+1} \leq u_i \tag{7}$$

Combinando (5) con (7) otteniamo $l_i \leq l_{i+1} \leq u_i$, il che significa che $A[i]$ e $A[i+1]$ si intersecano, come volevamo dimostrare.

⇐ Vediamo ora il viceversa. Supponiamo che $u_k \geq l_{k+1}$, per un opportuno k . Poiché gli intervalli sono ordinati in base agli estremi inferiori, abbiamo che $l_k \leq l_{k+1}$. Combinata con l'ipotesi, abbiamo la catena di disuguaglianze:

$$l_k \leq l_{k+1} \leq u_k$$

da cui si conclude che $[l_k, u_k]$ e $[l_{k+1}, u_{k+1}]$ si intersecano.

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?

Soluzione Se gli elementi sono tutti distinti è sufficiente controllare i primi tre $A[1], A[2], A[3]$. È possibile scegliere in tempo $O(1)$ l'elemento che non è né minimo né massimo tra i primi tre. Tale elemento, per costruzione, non sarà minimo né massimo per l'intero vettore.

```
Algorithm notminmax(A) -> number
// min12 = minimo tra A[1] e A[2]
// max12 = massimo tra A[1] e A[2]
if ( A[1] < A[2] ) then
    min12 = A[1];
    max12 = A[2];
else
    min12 = A[2];
    max12 = A[1];
endif
if ( A[3] > max12 ) then
    return max12;
elseif ( A[3] < min12 ) then
    return min12;
else
    return A[3];
endif
```

Se gli elementi *non* sono tutti distinti, allora il problema diventa $\Theta(n)$ nel caso peggiore in quanto è necessario controllarli tutti. Il caso peggiore si verifica quando i primi $n - 1$ elementi contengono solo 2 valori distinti, e un terzo valore distinto si trova in posizione n . In questo caso è necessaria una scansione dell'intero vettore per individuare l'elemento che non è né minimo né massimo. Si noti che se il vettore contiene lo stesso valore, oppure contiene due soli valori distinti, l'elemento mediano non esiste.

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.

Soluzione L'algoritmo di ricerca in questo caso può essere rappresentato dal seguente pseudocodice, che accetta come parametro un albero binario di ricerca T e la chiave k da cercare, e restituisce come risultato la lista L dei nodi che contengono la chiave k .

```

Algorithm search(T,k) -> List
  L = empty list
  if ( T != null ) then
    v = root of T;
    if ( v.key > k ) then
      L = search(T.left, k);
    elseif ( v.key < k ) then
      L = search(T.right, k)
    else
      insert v in L;
      append (search(T.left,k)) to L;
      append (search(T.right, k)) to L;
    endif
  endif
  return L;

```

Si noti che, se la lista L è rappresentata mediante una struttura concatenata contenente un riferimento all'ultimo elemento, l'operazione **append** (che concatena due liste) può essere realizzata in tempo $O(1)$.

Il caso peggiore si verifica quando tutti gli n nodi dell'albero contengono la chiave k ; in questo caso l'algoritmo visita una volta ciascun nodo dell'albero, da cui il costo dell'operazione search è $O(n)$.