Esercitazione 9
Algorithmi e Strutture Dati (Informatica)
A.A 2015/2016

Tong Liu

May 5, 2016

Elementi fondamentali

I passaggi di Heapsort, Kruskal, Prim etc
Pseudocodice della ricerca binaria (divide et impera) ...

Esercizio

[21/1/2014] Si scriva lo pseudo-codice dell'algoritmo di Kruskal per calcolare il minimo albero di copertura di un grafo non orientato pesato, dimostrandone correttezza e complessità. Successivamente, si esegua a mano l'algoritmo sul grafo $G = (N, A)$, con $N = \{1,2,3,4,5\}$, $A = \{[1,3],[1,4],[1,5],[2,3],[3,4],[4,5]\}$ e pesi $w[1,3] = 1$, $w[1,4] = 3$, $w[1,5] = 4$, $w[2,3] = 6$, $w[3,4] = 5$, $w[4,5] = 2$, mostrando ad ogni passo il contenuto delle strutture di dati usate.

Soluzione

Lo Pseudocodice dell'algoritmo Kruskal si trova nella Pagine 276 del libro il che ha il seguente passo di esecuzione, 1) ordina gli archi con i pesi, aggiungi gli archi nell'albero in ordine tenendo conto di non creare cicli. L'ordinamento iniziale degli archi costa $O(m \log n)$, usando i mfset, con le operazioni find() e merge(), la verifica dell'appartenenza dell'arco e l'eventuale inserimento in T costa $O(1)$. Il costo diventa $O(m \log n + m * 1)$ quindi $O(m \log n)$.

Esercizio

[18/2/2014] Si scriva la procedura HEAPSORT vista a lezione e la si esegua (a mano) per ordinare i 10 elementi: $9,10,3,5,8,2,6,4,7,1$. Si illustri con disegni, passo dopo passo, il contenuto dello heap durante l'esecuzione

Soluzione

L'array iniziale collocato nell'heap è sempre $9,10,3,5,8,2,6,4,7,1$. Build heap inizia dal nodo $n/2$, cioè 8, il 8 ha relazione corretta con il nodo padre, perchè più grande dei suoi figli quindi tocca a 5, il 5 si scambia con il nodo padre perchè è più grande, poi ... poi si elimina dalla testa il nodo, l'ultimo nodo in fondo andrà alla testa, dopodiché parte restoreHeapMax ... Il cambiamento è mostrato nella figura 1.1.

Esercizio

Dato un vettore ordinato contenente n elementi interi distinti compresi tra 1 e $n+1$ si calcoli l'elemento mancante in tempo $O(\log n)$

Soluzione

Poiché manca un unico valore k, tutti i valori $i, 1 \leq i < k$ sono memorizzati nella posizione i-esima del vettore; tutti i valori $i, k < i \leq n$ sono memorizzati nella posizione i - 1-esima. Troviamo quindi il più alto indice i tale per cui $A[i] = i$ e il valore mancante sarà $k = i + 1$.
L'idea è basata sulla ricerca binaria; analizzando le posizioni comprese fra i e j (estremi inclusi), calcoliamo $m = \lceil (i + j)/2 \rceil$. Se $A[m] = m$, il più alto indice tale per cui $A[i] = i$ è compreso fra m ed j; se $A[m] > m$, il più alto indice tale per cui $A[i] = i$ è compreso fra i ed $m - 1$. Quando ci si riduce ad un solo elemento ($i = j$), abbiamo trovato il nostro indice. La chiamata iniziale è mancante$(A, 1, n)$; la complessità è $O(\log n)$. l'Algoritmo 1

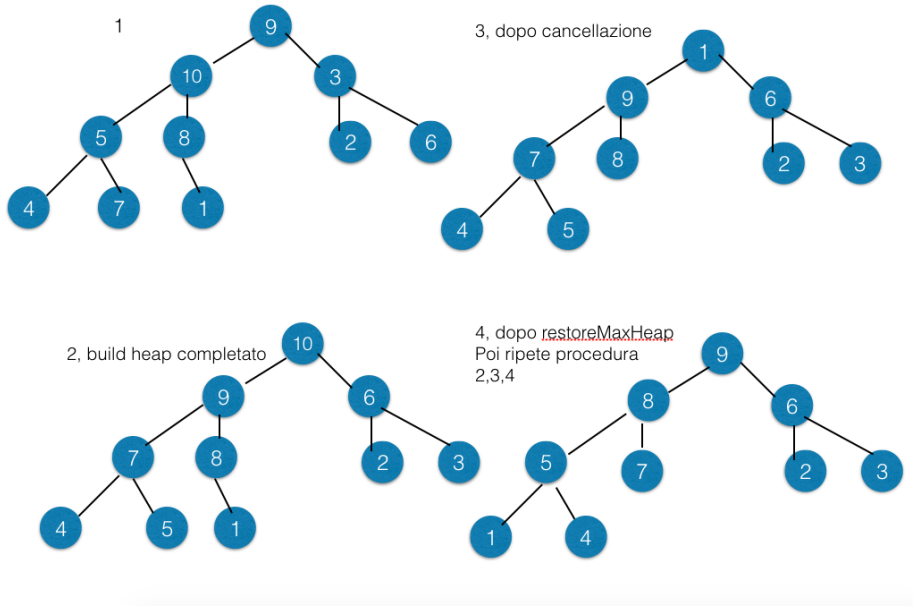

Figure 1.1: Figura Esercizio Heapsort

Algorithm 1 trova mancante

```
1: procedure MANCANTE(integer[] A, integer i, integer j)
2:     if i = j then
3:         return i + 1
4:     end if
5:     m ← ⌈(i + j)/2⌉
6:     if A[m] = m then
7:         return mancante(A, m, j)
8:     end if
9:     return mancante(A, i, m − 1)
10: end procedure
```

Esercizio

[27/1/2015] Dato un vettore di n interi, se ne vuole calcolare il massimo. Si scriva lo pseudocodice di un algoritmo divide et impera che partizioni i dati in tre parti bilanciate e richieda tempo ottimo.

Soluzione

La soluzione si vede nell'Algoritmo 2 L'algoritmo viene invocato con max3$(A, 1, n)$. Osserviamo che la ricorsione presenta due casi 'base': il caso di sottovettore vuoto ($i > j$), e il caso di sottovettore con un singolo elemento $A[i]$. Ad ogni chiamata ricorsiva, si calcolano le po-

sizioni iniziali d_1 e d_2 del secondo e terzo sottovettore, rispettivamente. Fatto quessto, il sottovettore viene implicitamente scomposto in $A[i..d_1-1], A[d_1..d_2-1] e A[d_2..j]$, e l'algoritmo invocato ricorsivamente su ogni parte. Il calcolo di d_1 e d_2 è fatto in modo tale da garantisce il comportamento corretto dell'algoritmo anche quando $A[i..j]$ è composto da 3 oppure 2 elementi. Il costo computazionale soddisfa la seguente equazione di ricorrenza:

$$T(n) = \begin{cases} c_1 & n \leq 1 \\ 3T(n/3) + c_2 & \text{altrimenti} \end{cases} \tag{1.1}$$

che in base al Teorema delle ricorrenze ha soluzione $O(n)$

Algorithm 2 calcola max con 3 partizioni

```
1: procedure MAX3(integer[] A, integer i, integer j)
2:     if i > j then
3:         return A[0]
4:     else if i = j then
5:         return A[i + 1]
6:     else
7:         d1 ← ⌈(i + (j − i + 1)/3)⌉
8:         d2 ← ⌈(i + (j − i + 1) ∗ 2/3)⌉
9:         return max(max3(A, i, d1 − 1),max3(A, d1, d2 − 1),max3(A, d2, j))
10:    end if
11: end procedure
```

Esercizio

[17/2/2015] Una sequenza $a_1, ..., a_n$ di n interi distinti è detta unimodale se esiste un indice m tale che $a_1 > ... > a_m < ... < a_n$ (cioè la sequenza è dapprima decrescente e poi crescente ed a_m è il minimo; se $m = 1$ allora la sequenza è crescente, mentre se $m = n$ allora è decrescente). Data una sequenza unimodale, si vuole trovare m. Si scriva lo pseudo-codice di un algoritmo divide et impera che richieda tempo ottimo.

Soluzione

Possiamo utilizziamo il meccanismo di ricerca binaria per risolvere il problema. Il minimo del vettore è sicuramente $A[h]$, perché minore di tutti gli elementi che lo seguono e lo precedono. L'idea 'e la seguente: prendiamo l'elemento mediano $A[m]$ del sottovettore considerato, e consideriamo l'elemento a sinistra $A[m-1]$ e a destra $A[m+1]$:

Se $A[m-1] > A[m] < A[m+1]$, allora l'elemento mediano è proprio h; abbiamo finito;
Altrimenti, Se $A[m-1] > A[m]$, l'elemento che cerco si trova a destra; riduco l'insieme;
Altrimenti, l'elemento che cerco si trova a sinistra. Ci sono anche un paio di casi base: il sottovettore considerato ha 1 o 2 elementi, nel qual caso scelgo il minimo. La procedura è illustrata nell'Algoritmo 3

Algorithm 3 unimodale

```
1:  procedure UIM(integer[] A,integer i,integer j)
2:      if i = i then
3:          return A[i]
4:      end if
5:      if j = i + 1 then
6:          return min(A[i], A[j])
7:      end if
8:      Integer m = (i + j)/2
9:      if A[m − 1] > A[m] and A[m + 1] > A[m] then
10:         return A[m]
11:     end if
12:     if A[m − 1] > A[m] then
13:         return uim(A, m + 1, j)
14:     else
15:         return uim(A, i, m − 1)
16:     end if
17: end procedure
```