Algoritmi e Strutture Dati
6 luglio 2021

Note

1. La leggibilità è un prerequisito: parti difficili da leggere potranno essere ignorate.
2. Quando si presenta un algoritmo è fondamentale spiegare l'idea e motivarne la correttezza.
3. L'efficienza e l'aderenza alla traccia sono criteri di valutazione delle soluzioni proposte.
4. Si consegna la scansione dei fogli di bella copia, e come ultima pagina un documento di identità.

Domande

Domanda A (7 punti) Definire formalmente la classe $\Theta(g(n))$. Dimostrare le seguenti affermazioni o fornire un controesempio:

i. se $f(n) \in \Theta(g(n))$ e $f'(n) \in O(g(n))$ allora $f(n) + f'(n) \in \Theta(g(n))$;

ii. se $f(n) \in \Theta(g(n))$ e $f'(n) \in O(g(n))$ allora $f(n) - f'(n) \in \Theta(g(n))$;

Soluzione: Per la definizione di $\Theta(f(n))$, consultare il libro.

Per (i), sia $f(n) \in \Theta(g(n))$. Per definizione, esistono $c_1, c_2 > 0$ e n_0 tali che per ogni $n \geq n_0$ vale:

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$$

Inoltre, dato che $f'(n) \in O(g(n))$ esistono $c_2' > 0$ e n_0' tali che per ogni $n \geq n_0'$ vale:

$$0 \leq f'(n) \leq c_2' g(n)$$

Quindi, per ogni $n \geq \max\{n_0, n_0'\}$ abbiamo:

$$0 \leq c_1 g(n) \leq f(n) \leq f(n) + f'(n) \leq c_2 g(n) + c_2' g(n) = (c_2 + c_2')g(n)$$

dato che $c_1, c_2 + c_2' > 0$ questo conclude la prova che $f(n) + f'(n) \in \Theta(g(n))$.

Quindi, essendo $k > 0$, per ogni $n \geq \max n_0$ abbiamo:

$$(c_1 + c_1')g(n) = c_1 g(n) + c_1' g(n \leq f(n) + f'(n) \leq c_2 g(n) + c_2' g(n) = (c_2 + c_2')g(n)$$

dato che $c_1 + c_1', c_2 + c_1' > 0$ questo conclude la prova che $f(n) + f'(n) \in \Theta(g(n))$.

Per la parte (ii), l'affermazione è falsa. Basta considerare $f(n) = f'(n) = n \in \Theta(n)$ e quindi in $O(n)$, ma ovviamente $f(n) - f'(n) \notin \Omega(n)$ e quindi $\notin \Theta(n)$.

Domanda B (7 punti) Scrivere una funzione `toTree(A)` che dato un array `A` organizzato a max-heap (dimensione A.heapSize), lo trasforma in un albero binario realizzato con strutture linked, ancora organizzato a max-heap e ritorna la radice di tale albero. Il nuovo albero è costituito da nodi `x` con i campi `x.p` (parent), `x.k` (chiave), `x.l` e `x.r` (figlio sinistro e figlio destro). Per allocare un nuovo nodo si assuma di avere a disposizione un costruttore `node()`. Valutare la complessità.

Soluzione: L'idea è semplicemente quella di visitare l'albero, memorizzato come array, in profondità, e riprodurlo nella struttura linked.

```
toTree(A)
   T.root = toTreeRec(A,1,nil)
   return T

// toTreeRec(A,i,x):
// dato un max-heap A ritorna la radice di un albero, copia "linked" del
// sottoalbero di A radicato in i, l'argomento x e' il nodo padre
toTreeRec(A,i,x)
   if i <= A.heapSize
      y = node()                    // crea un nuovo nodo
      y.key = A[i]                  // la chiave e' A[i]
      y.p = x                       // il parent e' x
      left  = 2*i                   // costruisce i sottoalberi sx e dx,
      right = 2*i+1                 // che saranno figli sx e dx di y
      y.l = toTreeRec(A,left, y)
      y.r = toTreeRec(A,right,y)
   else
      return nil
```

Si tratta di una visita dell'albero e quindi la complessità è $\Theta(n)$ dove n è il numero di elementi del max-heap.

Esercizi

Esercizio 1 (8 punti) Realizzare una funzione `missing(A,n)` che dato un array `A[1..n]` che contiene interi nell'intervallo `[1,n]` verifica se esiste qualche valore $\mathtt{v} \in [1, \mathtt{n}]$ che non compare in `A`, ovvero tale che $\mathtt{A}[\mathtt{i}] \neq \mathtt{v}$ per ogni $\mathtt{i} \in [1, \mathtt{n}]$. Valutare la complessità.

Soluzione: Si osserva che in `A[1..n]` possono mancare dei valori se e solo se ci sono elementi duplicati.

Per verificare la presenza di duplicati, si scorre l'array `A` e si usa un array di booleani `B[1..n]` per ricordare i valori già incontrati. Più precisamente `B[v]` indica se è stato o meno incontrato il valore `v` nella parte di array `A` esplorata. Ci sarà un duplicato (e quindi un valore assente) solo se durante l'esplorazione si incontra un valore `v` che era già stato incontrato prima e quindi con `B[v]=true`.

```
missing(A,n)
   alloca B[1..n] array di booleani

   for i=1 to n
       B[i]=false

   i=1
   while (i<=n) and not B[A[i]]
       B[A[i]] = true
       i++

   return i<=n
```

L'invariante del ciclo è

- $A[1, i-1]$ non contiene duplicati e

- $\forall v \in [1,n]$. $B[v] = true$ se e solo se $\forall j \in [1, i-1]$. $A[j] \neq v$.

Quindi, se si esce perché $i = n+1$, significa che $A[1,n]$ non contiene duplicati e quindi non ci sono valori assenti.

Se invece si esce con $i \leq n$, significa che $B[A[i]] = true$. Pertanto deve esistere $j < i$ tale che $A[j] = A[i]$, ovvero c'è un duplicato e quindi necessariamente un valore assente.

La complessità è chiaramente $\Theta(n)$, dato che il ciclo di inizializzazione di B costa n, e il while fa al massimo n iterazioni, ciascuna di costo costante.

Esercizio 2 (9 punti) Data una stringa $X = x_1, x_2, \ldots, x_n$, si consideri la seguente quantità $\ell(i,j)$, definita per $1 \leq i \leq j \leq n$:

$$\ell(i,j) = \begin{cases} 1 & \text{se } i = j \\ 2 & \text{se } i = j-1 \\ 2 + \ell(i+1, j-1) & \text{se } (i < j-1) \text{ e } (x_i = x_j) \\ \sum_{k=i}^{j-1} (\ell(i,k) + \ell(k+1,j)) & \text{se } (i < j-1) \text{ e } (x_i \neq x_j). \end{cases}$$

1. Scrivere una coppia di algoritmi INIT_L(X) e REC_L(X, i, j) per il calcolo memoizzato di $\ell(1,n)$.
2. Si determini la complessità *al caso migliore* $T_{\text{best}}(n)$, supponendo che le uniche operazioni di costo unitario e non nullo siano i confronti tra caratteri.

Soluzione:

1. Pseudocodice:

```
INIT_L(X)
 n <- length(X)
 if n = 1 then return 1
 if n = 2 then return 2
 for i=1 to n-1 do
   L[i,i] <- 1
   L[i,i+1] <- 2
 L[n,n] <- 1
 for i=1 to n-2 do
   for j=i+2 to n do
     L[i,j] <- 0
 return REC_L(X,1,n)



REC_L(X,i,j)
 if L[i,j] = 0 then
   if x_i = x_j then L[i,j] <- 2 + REC_L(X,i+1,j-1)
   else for k=i to j-1 do
           L[i,j] <- L[i,j] + REC_L(X,i,k) + REC_L(X,k+1,j)
 return L[i,j]
```

2. Il caso migliore è chiaramente quello in cui tutti i caratteri sono uguali, poiché l'albero della ricorsione in quel caso è unario e i suoi nodi interni corrispondono alle chiamate con indici

$(1, n), (2, n-1), \ldots, (k, n-k+1)$, ognuno associato a un costo unitario. Ci sono al più $\lfloor n/2 \rfloor$ di queste chiamate, e quindi $T_{\text{best}}(n) = O(n)$.