Algoritmi e Strutture Dati

Sessione invernale A.A. 2003/2004

Appello 20.02.04

Prima parte

1. *Data la ricorrenza*

$$T(n) = 2 \cdot T(\frac{2n}{3}) + n^2$$

(a) *Utilizzando il metodo di sostituzione dimostrare che $T(n) = O(n^2)$.*

Dobbiamo dimostrare che esistono due costanti c e n_0 tale che $T(n) \leq cn^2$, per ogni $n \geq n_0$. Procediamo per induzione e supponiamo $T(\frac{2n}{3}) \leq c \cdot (\frac{2n}{3})^2$. Otteniamo:
$T(n) = 2 \cdot T(\frac{2n}{3}) + n^2 \leq 2c \cdot (\frac{2n}{3})^2 + n^2 = (\frac{8}{9} \cdot c + 1)n^2$
È ora facile vedere che $(\frac{8}{9} \cdot c + 1) \leq c$ per un qualsiasi costante $c \geq 9$. Quindi $T(n) \leq cn^2$, per ogni $c \geq 9$ e per ogni $n_0 > 0$.

(b) *Utilizzando poi il Teorema Principale, dare i limiti inferiore e superiore stretti per $T(n)$.* (Nota: $\log_{\frac{3}{2}} 2 \approx 1,710$ e $\log_2 \frac{3}{2} \approx 0,585$).

Abbiamo: $a = 2$, $b = \frac{3}{2}$ e $1 < \log_{\frac{3}{2}} 2 < 2$. Poiché $f(n) = n^2 = \Omega(n^{\log_{\frac{3}{2}} 2 + \epsilon})$ se vale la condizione di regolarità ($\exists c < 1$ tale che $af(\frac{n}{b}) < cf(n)$) siamo nel caso 3 del Master Method. La condizione vale in quanto scelto $\frac{8}{9} < c < 1$ è facile vedere che $2 \cdot (\frac{2n}{3})^2 = \frac{8}{9} \cdot n^2 < c \cdot n^2$. Allora $T(n) = \Theta(f(n)) = \Theta(n^2)$.

2. *Nell'ipotesi che* $\text{Proc}(m) = \Theta(\sqrt{m})$, *determinare la complessità asintotica (caso peggiore) della seguente procedura* $\text{Fun}(A, n)$ *al crescere di* $n \in N$.

```
Fun(A, n)
if n < 1 return 1
t ← Fun(A, n/2)                      T_Fun(n/2)
if t > n^2
    then t ← t − 1/2 · Fun(A, n/2)   T_Fun(n/2)
for j ← 1 to n
      do t ← t + A[j] + Proc(n)      n√n
return t
```

Nel caso peggiore vengono effettuate entrambe le chiamate ricorsive e quindi $T_{\text{Fun}}(n)$ soddisfa la ricorrenza $T_{\text{Fun}}(n) = 2 \cdot T_{\text{Fun}}(n/2) + \Theta(n\sqrt{n})$ e la complessità asintotica della procedura Fun si trova risolvendo tale ricorrenza. Questo si ottiene facilmente utilizzando il Master Method. Si ha $a = 2, b = 2$, $\log_2 2 = 1$, $n\sqrt{n} = \Omega(n^{1+\epsilon})$, e $n\sqrt{n}$ soddisfa la proprietà di regolarità. Quindi siamo nel caso 3 ed abbiamo $T_{\text{Fun}}(n) = \Theta(n\sqrt{n}) = \Theta(n^{\frac{3}{2}})$.

3. *Si scriva lo pseudocodice di un algoritmo che dato un albero binario T calcola il numero dei nodi di T che hanno due figli con lo stesso numero di discendenti.*

Si risolve facilmente con una visita in post-ordine di T che calcola il numero dei discendenti di ogni nodo (includendo il nodo stesso) ed incrementa una variabile globale

Equals inizializzata a 0. Il test ($nl \neq 0$) serve ad escludere i nodi che non hanno due figli (perchè entrambi uguali a NIL).

```
DISCENDENTI(x)
if x = NIL
    then return 0
nl ← DISCENDENTI(LEFT[x])
nr ← DISCENDENTI(RIGHT[x])
if (nl = nr) & (nl ≠ 0)
    then Equals ← Equals + 1
return nl + nr + 1
```

Seconda parte

4. *Si supponga di voler modificare gli elementi di un array* A *incrementando di una costante* k1 *tutti gli elementi minori di un'altra costante* k2. *Supposto che* A *soddisfi le proprietà di max-heap, scrivere lo pseudocodice di un algoritmo efficiente che trasforma* A *come richiesto producendo ancora un max-heap.*

Una semplice soluzione si ottiene modificando la procedura BUILD-MAX-HEAP.

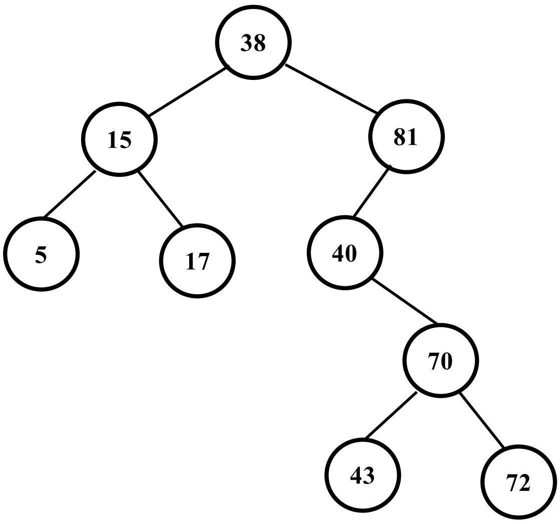
```
INCREASE-HEAP(A, i, n, k1, k2)
for i ← n downto 1
    if A[i] < k2
    then A[i] ← A[i] + k1
    else HEAPIFY(A, i, n)
```

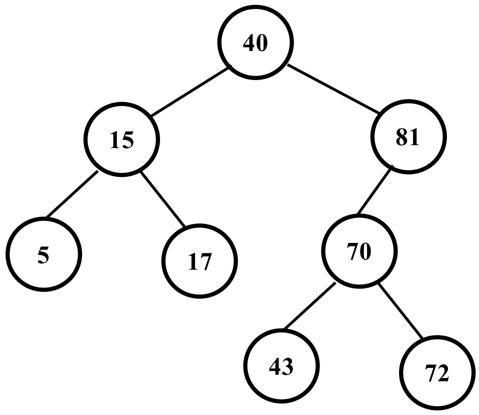
Si noti che quando si esegue l'incremento dell'elemento $A[i]$ non è necessario applicare la procedura HEAPIFY perchè l'array di partenza rappresentava un max-heap e quindi se $A[i] < k2$ anche tutti i discendenti di $A[i]$ lo erano, e sono quindi già stati incrementati senza alterare la proprietà di max-heap. Al contrario quando $A[i]$ non viene incrementato potrebbe essere diventato minore di uno dei figli (precedentemente incrementati) e in questo caso si rende necessaria una chiamata alla procedura HEAPIFY.

5. (a) *Dare la definizione di* albero binario di ricerca
(b) *Data la struttura ad albero indicata in figura, inserire gli elementi 38, 72, 70, 15, 81, 17, 40, 43, 5 in modo da ottenere un albero binario di ricerca*
(c) *Successivamente si mostri l'albero di ricerca ottenuto dopo la rimozione del nodo corrispondente alla chiave 38.*

Proprietà BST. Per ogni coppia di nodi x e y in T
- se y è nel sottoalbero sinistro di x allora $key[y] \leq key[x]$
- se y è nel sottoalbero destro di x allora $key[x] \leq key[y]$

ATTENZIONE! molti hanno dato la seguente risposta **errata**: *per ogni nodo* x *deve essere* $key[left[x]] \leq key[x] \leq key[right[x]]$.





6. *Descrivere le principali caratteristiche di una tabella hash realizzata con la tecnica ad indirizzamento aperto.*

Vedi testo.

7. *Sia* T *un* albero binario di ricerca *ai cui nodi sono associati gli attributi* key, left, right, color, bh. *I primi sono gli usuali attributi di un albero* Rosso/Nero *mentre* bh[x] *è progettato per indicare l'altezza nera dell'albero radicato in* x.
Supponendo che T soddisfi le prime 4 proprietà caratteristiche degli alberi Rosso/Neri e che i campi key, left, right, color *siano già stati correttamente assegnati, scrivere lo pseudocodice di un algoritmo efficiente che verifica se T soddisfa anche la proprietà sui cammini.*

Dobbiamo controllare solo la proprietà relativa alle altezze nere. Si noti che il testo dell'esercizio non garantisce il contenuto del campo emphbh[x]. Questo può venir utilizzato durante la visita di T come nel seguente algoritmo.

```
RN-CHECK(x)
if x = FOGLIA-NIL
   then BH[x] ← 1
        return TRUE
if color[x] = BLACK
   then s ← 1
   else s ← 0
ck ← RN-CHECK(LEFT[x]) & RN-CHECK(RIGHT[x])
if ck & (BH[LEFT[x]] = BH[RIGHT[x]])
   then BH[x] ← BH[LEFT[x]] + s
        return TRUE
   else return FALSE
```