1 Soluzioni primo compitino AA 2003-04

1.1 Primo esercizio

Si dimostri la verità o la falsità di ciascuna delle seguenti affermazioni

(a) $\frac{1}{2}n + lg\ n^2 + \sqrt{n}$ *è nella classe* $\Theta(n)$

VERO. Verifichiamo che esistono tre costanti c_1, c_2, n_0 positive tali che per ogni $n \geq n_0$:

$$c_1 n \leq \frac{1}{2}n + \lg n^2 + \sqrt{n} \leq c_2 n$$

ovvero, per $n > 0$,

$$c_1 \leq \frac{1}{2} + \frac{2\lg n}{n} + \frac{\sqrt{n}}{n} \leq c_2.$$

Poiché, per $n \geq 2$, $\frac{2\lg n}{n} + \frac{\sqrt{n}}{n} \leq 2$ le due diseguaglianze sono verificate scegliendo $c_1 = \frac{1}{2}$, $c_2 = 3$ ed $n_0 = 2$.

(b) $n^2 + nlg\ n$ *è nella classe* $\Theta(n)$

FALSO. Dimostriamo che non possono esistere tre costanti c_1, c_2, n_0 positive tali che per ogni $n \geq n_0$:

$$c_1 n \leq n^2 + n\lg n \leq c_2 n.$$

In particolare dimostriamo che non esistono c_2 ed n_0 con le proprietà richieste. Infatti, per ogni valore di c_2 si ha $n\lg n > c_2 n$ per ogni $n > 2^{c_2}$.

(c) *La ricorrenza* $T(n) = 3T(\frac{n}{4}) + nlg\ n$ *individua una funzione nella classe* $\Theta(nlg\ n)$

VERO. Possiamo utilizzare il Master Theorem. Poiché $n\lg n = \Omega(n^{\log_4 3+\epsilon})$, dove $\epsilon = 1 - \log_4 3$, ricadiamo nel caso 3. Per poter affermare questo si deve anche verificare che sia soddisfatta la condizione di regolarità:

$$\forall n \geq n_0 \quad 3\frac{n}{4} \lg (\frac{n}{4}) \leq cn\lg n$$

per un qualche $c < 1$ e $n_0 > 0$. Poiché per ogni $n \geq 1$

$$\frac{3}{4}n \lg (\frac{n}{4}) \leq \frac{3}{4}n \lg n$$

è sufficiente prendere proprio $c = \frac{3}{4}$ ed $n_0 = 1$. Allora per il terzo caso del Master Theorem $T(n) = \Theta(n\lg n)$.

(d) *La ricorrenza*$T(n) = 3T(\frac{n}{2}) + lg\, n$ *individua una funzione nella classe* $O(n^2)$

VERO. Possiamo utilizzare il Master Theorem. Poiché $\lg n = O(n^{\log_2 3-\epsilon})$, dove $\epsilon = \log_2 3 - 1$, possiamo applicare il primo caso del Master Theorem ed ottenere $T(n) = \Theta(n^{\log_2 3})$ e quindi $T(n) = O(n^2)$.

1.2 Secondo esercizio

Date le seguenti procedure A *e* B*, si determini la complessità asintotica della procedura* A(n) *su input* $n \in N$

A(n)

$s \leftarrow 0$	1
for $i \leftarrow 1$ **to** n	$n + 1$
do $s \leftarrow s + \mathrm{B}(i)$	$\sum_{i=1}^{n} T_B(i)$
return s	1

B(m)

$s \leftarrow 0$	1
for $i \leftarrow 1$ **to** m	$m + 1$
do $s \leftarrow s + i$	m
return s	1

Allora $T_B(i) = \Theta(i)$ e la complessità di A è data da:

$$\sum_{i=1}^{n} T_B(i) = \sum_{i=1}^{n} \Theta(i) = \Theta(n^2)$$

1.3 Terzo esercizio

(a) *Scelta una rappresentazione per le liste, si descriva un algoritmo che data una lista* L *ed un intero* k *modifichi* L *eliminando tutti gli elementi con chiave minore di* k*.*

Per la lista singola:

List-Delete-Smaller(L, k)

$x \leftarrow head[L]$
while $x \neq$ NIL
    do $y \leftarrow next[x]$
        if $key[x] < k$
            then List-Delete(L, x)
        $x \leftarrow y$

Per l'implementazione con lista doppia il codice non cambia; per l'implementazione con lista circolare invece è sufficiente cambiare NIL in $nil[L]$.

(b) *Si dimostri la correttezza dell'algoritmo proposto utilizzando un opportuno invariante.*

Invariante: nessun nuovo elemento è stato inserito nella lista L; tutti gli elementi con chiave maggiore o uguale a k originariamente in L solo ancora in L; tutti gli elementi prima di x hanno chiave maggiore o uguale a k;

Inizializzazione: quando $x = head[L]$ la lista non è stata ancora modificata e non ci sono elementi prima di x, per cui l'invariante è soddisfatta;

Mantenimento: all'inizio del ciclo **while** per l'invariante tutti gli elementi prima di x hanno chiave maggiore o uguale a k; ci sono due casi:

- l'elemento in x ha chiave minore di k: questo viene eliminato dalla lista e x passa al prossimo elemento; allora gli elementi prima di x nella lista L sono gli stessi elementi che c'erano all'inizio del ciclo; poiché questi per l'invariante erano originariamente nella lista l'invariante è mantenuta;
- l'elemento in x ha chiave maggiore o uguale a k: x passa al prossimo elemento; gli elementi che precedono x sono quelli che c'erano all'inizio del ciclo più quest'elemento con chiave maggiore o uguale a k; inoltre quest'elemento è un elemento della lista originaria; allora l'invariante è mantenuta;

Terminazione: $x =$ NIL, per cui è posizionato dopo l'ultimo elemento; allora tutti gli elementi prima di x sono la lista intera L che contiene ora solo gli elementi originari maggiori o uguali a k.

(c) *Si discuta la complessità dell'algoritmo proposto e si dica se e come questa varia al variare della rappresentazione scelta.*

La complessità varia nel seguente modo:

- lista semplice: $\Theta(n^2)$ (la LIST-DELETE è $\Theta(n)$)
- lista doppia: $\Theta(n)$ (la LIST-DELETE è $\Theta(1)$)
- lista circolare doppia: $\Theta(n)$ (la LIST-DELETE è $\Theta(1)$)

1.4 Quarto esercizio

Si sviluppi un algoritmo di tipo divide-et-impera per calcolare la somma degli elementi di un array $A[1 \dots n]$ e se ne valuti la complessità.

Sum-R(A, p, r)

if $p > r$
  then return 0
    $q \leftarrow \lfloor (p + r)/2 \rfloor$
    return Sum-R$(A, p, q - 1) + A[q] +$ Sum-R$(A, q + 1, r)$

La complessità è definita dalla seguente equazione ricorsiva:

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 0 \\ 2T(\frac{n}{2}) + \Theta(1) & \text{altrimenti} \end{cases}$$

La soluzione dell'equazione è $T(n) = \Theta(n)$ come si può dimostrare facilmente con il Master Theorem, o con alberi di ricorsione e metodo di sostituzione.

Usando il Master Theorem è pressoché immediato risolvere l'equazione: $1 = O(n^{1-\epsilon})$ per qualunque $\epsilon \leq 1$ per cui, per il primo caso del Master Theorem: $T(n) = \Theta(n)$.

1.5 Quinto esercizio

Si descriva un algorimo che dato un intero k ed un albero generale T, con attributi $key[x]$, $child[x]$, $sibling[x]$ e $parent[x]$, modifica il campo chiave di tutti i nodi di T ponendo $key[x]$ uguale al numero dei discendenti di x la cui chiave è minore di k.

Si può risolvere con una visita *depth-first-search*, ricorrendo sia sui figli che sui fratelli:

Tree-Count-Lesser-R(x, k)

if $x =$ nil
  then return 0
if $key[x] < k$
  then $key[x] \leftarrow 1$
  else $key[x] \leftarrow 0$
$key[x] \leftarrow key[x] +$ Tree-Count-Lesser-R$(child[x], k)$
return $key[x] +$ Tree-Count-Lesser-R$(sibling[x], k)$

Tree-Count-Lesser(T, k)

Tree-Count-Lesser-R$(root[T], k)$