Algoritmi e Strutture Dati
&
Laboratorio di Algoritmi e Programmazione

— Appello del 16 Gennaio 2006 —

Esercizio 1 (ASD)

Si consideri la ricorrenza:

$$T(n) = T(\frac{n}{3}) + T(\frac{n}{6}) + 2n$$

e si dimostri la verità o falsità delle seguenti affermazioni, utilizzando il metodo di sostituzione o le proprietà delle classi di complessità.

1. $T(n) = \Theta(n)$
2. $T(n) = O(n \lg n)$
3. $T(n) = \Theta(n \lg n)$

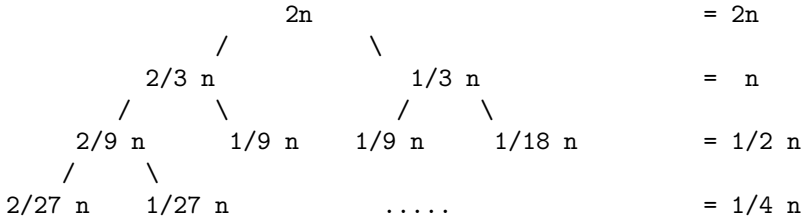
Si disegni inoltre l'albero di ricorsione relativo alla ricorrenza data.

Soluzione

E' facile dimostrare che $T(n) = \Theta(n)$. Dobbiamo dimostrare che esistono c_1, c_2, n_0 positive tali che $c_1 n \leq T(n) \leq c_2 n$, per ogni $n \geq n_0$. Procedendo con il metodo di sotituzione, verifichiamo che esiste $n_0 \geq 0$ e

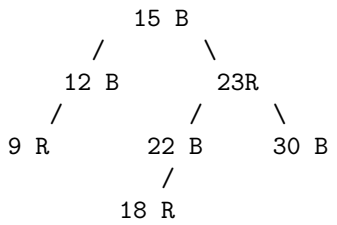
- (i) esiste $c_1 > 0$ tale che $c_1 n \leq c_1 \frac{n}{3} + c_1 \frac{n}{6} + 2n$, per ogni $n \geq n_0$.
  E' sufficiente scegliere $c_1 = 1$ per ottenere: $n \leq \frac{n}{2} + 2n$, per ogni $n \geq 0$.
- (ii) esiste $c_2 > 0$ tale che $c_2 \frac{n}{3} + c_2 \frac{n}{6} + 2n \leq c_2 n$, per ogni $n \geq n_0$. E' sufficiente scegliere $c_2 = 6$ per ottenere: $6\frac{n}{3} + 6\frac{n}{6} + 2n = 5n \leq 6n$, per ogni $n \geq 0$.

La verità della seconda affermazione segue dalle seguenti considerazioni: (a) $T(n) = \Theta(n)$ implica $T(n) = O(n)$ e (b) poiché $n \leq n \lg n$, per ogni $n > 1$ si ha $n = O(n \lg n)$. Quindi $T(n) = O(n \lg n)$ segue da (a) e (b) per la proprietà transitiva relativa alla classe O.
La terza affermazione è invece falsa perchè $n = o(n \lg n)$, come si dimostra facilmente calcolando il limite.



Esercizio 2 (ASD)

Dire se il seguente albero binario gode della proprietaà R/B. Giustificare la risposta e, in caso di risposta negativa, dire se è possibile ottenere un albero R/B tramite ricolorazione dei nodi.



Soluzione

Si, è un albero R/B. (Chiaramente si suppone che tutte le foglie-NIL lasciate implicite siano nere).

Esercizio 3 (ASD)

Si consideri l'algoritmo di ordinamento Quicksort realizzato con la seguente procedura di partizione di Lomuto (come nel testo):

```
Partition(A,p,r)
  x <- A[r]
  i <- p-1
  for j <- p to r-1
     do if A[j] <= x
           then i <- i+1
                scambia (A[i], A[j])
  scambia (A[i+1], A[r])
  return i+1
```

Si dica quali tra le seguenti affermazioni sono corrette:

(a) L'algoritmo ha un comportamento pessimo quando l'array di partenza è già ordinato;

(b) L'algoritmo ha un comportamento pessimo quando l'array di partenza è ordinato in ordine inverso;

(c) L'algoritmo ha un comportamento pessimo quando l'array di partenza contiene solo valori inclusi in un intervallo prefissato;

(d) L'algoritmo ha un comportamento pessimo quando l'array di partenza contiene valori positivi e negativi alternati;

(e) L'algoritmo ha un comportamento pessimo quando l'array di partenza contiene tutti valori uguali;

Giustificare la risposta.

Soluzione

Sono corrette le affermazioni a,b,e perchè in questi casi l'algoritmo di partizione costruisce (sempre) una partizione che contiene un solo elemento in una delle due parti. Con una tale partizione la ricorrenza che caratterizza il comportamento di Quicksort è $T(n) = T(n-1) + \Theta(n)$ la cui soluzione è $T(n) = \Theta(n^2)$, che sappiamo caratterizzare il comportamento pessimo di Quicksort.

Esercizio 4 (ASD e Laboratorio)

Scrivere un algoritmo ITERATIVO *isMinHeap* che, dato un array A contenente numeri interi, verifica in tempo lineare se A è un min-heap. Si richiede di:

1. scrivere l'algoritmo in pseudo-codice
2. dimostrare la correttezza dell'algoritmo
3. completare l'implementazione della seguente classe *Esercizio4.java*, il cui unico metodo *isMinHeapRic* deve risolvere in modo RICORSIVO lo stesso problema.

```
public class Esercizio4.java {

    // pre: A non nullo
    // post: ritorna true se l'array A e' un min-heap; ritorna false altrimenti
    public static boolean isMinHeapRic(int[] A) {...}
}
```

Se necessario, si possono aggiungere alla classe eventuali metodi privati di supporto.

Soluzione

1. ```
   isMinHeap(A)
         i <- 2
         while (i <= lenght[A]) and (A[i div 2] <= A[i])
             do i <- i+1
         return (i > length(A))
   ```

   Algoritmo alternativo:

   ```
   isMinHeap(A)
      flag <- true
      i <- 1
      while (flag and i <= (length(A) div 2))
         do  if (2*i+1 > length(A))
                then flag <- A[i] <= A[2*i]
                else flag <- A[i] <= A[2*i] and A[i] <= A[2*i +1]
             i <- i+1
      return flag
   ```

2. *Invariante*: minheap($A[1..i-1]$)] ovvero A[1..i-1] è un minheap.
   Inizializzazione: i=2 e A[1] contiene un solo elemento e quindi è un minheap.
   Mantenimento: L'incremento di i avviene solo se la condizione (A[padre[i]] $\leq A[i]$) è soddisfatta e

   $$[(A[\text{padre}[i]] \leq A[i]) \text{ and } \text{minheap}(A[1..i-1])] \implies \text{minheap}(A[1..i]).$$

   Terminazione: Il ciclo puo' terminare perché

   - $i = \text{lenght}[A] + 1$ e in questo caso l'invariante implica minheap(A[1..lenght[A]])
   - oppure perchè isheap è false ma questo può avvenire solo se A[padre[i]] $> A[i]$) con $i \leq$ lenght[A], e in questo caso A non è un minheap.

3. ```
   public class Esercizio4 {

       // pre: A non nullo
       // post: ritorna true se l'array A e' un min-heap; ritorna false altrimenti
       public static boolean isMinHeapRic(int[] A) {
           return verifyMinHeap(A,0);
       }

       private static boolean verifyMinHeap(int[] A, int i) {

           // i punta ad una foglia
           if (i >= (A.length / 2))
               return true;

           // manca il figlio destro
           if (2*(i+1) >= A.length)
               return (A[i] <= A[2*i+1]);
           else // ci sono entrambi i figli
               return ((A[i] <= A[2*i+1]) && (A[i] <= A[2*(i+1)]) &&
                      verifyMinHeap(A, 2*i+1) && verifyMinHeap(A, 2*(i+1)));

       }
   }
   ```

Esercizio 5 (Laboratorio)

Estendere la classe *BinaryTree.java* del package *BinTrees* relativo agli alberi binari, aggiungendo il metodo

```
// post: ritorna una stringa contenente la chiave di tutti i nodi dell'albero che hanno un solo cugino.
// NOTA: x e' cugino di y sse i padri di x e y sono fratelli
public String NodiConUnCugino() {...}
```

Si richiede di implementare il metodo mediante un algoritmo ricorsivo. Se necessario, si possono aggiungere alla classe eventuali metodi privati di supporto.

Soluzione

Proponiamo due soluzioni: la prima è più lineare e permette di provare semplicemente la correttezza.

PRIMA SOLUZIONE

```
// post: ritorna una stringa contenente la chiave di tutti i nodi
//       dell'albero che hanno un solo cugino.
// NOTA: x e' cugino di y sse i padri di x e y sono fratelli
public String NodiConUnCugino() {

    StringBuffer sb = new StringBuffer("Nodi con un solo cugino: ");
    visita(root, sb);
    return sb.toString();
}

// post: esegue una visita in preordine
private void visita(BTNode n, StringBuffer sb) {
    if (n != null) {
        cugini(n,sb);
        visita(n.left, sb);
        visita(n.right, sb);
    }
}

// pre: n diverso da null
// post: aggiunge n ad sb sse n ha un solo cugino
private void cugini(BTNode n, StringBuffer sb) {
    BTNode fp; // fratello del padre di n

    if (n.parent != null && n.parent.parent != null) {
       if (n.parent.parent.left == n.parent)
           fp = n.parent.parent.right;
       else
           fp = n.parent.parent.left;

       if ((fp != null) && ((fp.left == null) != (fp.right == null)))
           sb.append(n.key.toString() + " ");
    }
}
```

SECONDA SOLUZIONE

```
// post: ritorna una stringa contenente la chiave di tutti i nodi
//       dell'albero che hanno un solo cugino.
// NOTA: x e' cugino di y sse i padri di x e y sono fratelli
public String NodiConUnCugino() {
    StringBuffer sb = new StringBuffer("Nodi con un solo cugino: ");
    seleziona(root, sb);
    return sb.toString();
}
```

```
private void seleziona(BTNode n, StringBuffer sb) {

    if (n == null)
        return;
    BTNode l = n.left;
    BTNode r = n.right;

    if (l == null && r == null)
        return;

    if (l != null && r == null) {
        seleziona(l, sb);
        return;
    }

    if (l == null && r != null) {
        seleziona(r, sb);
        return;
    }

    if ((l.left != null) && ((r.left != null) != (r.right != null)))
        sb.append(l.left.key.toString() + " ");

    if ((l.right != null) && ((r.left != null) != (r.right != null)))
        sb.append(l.right.key.toString() + " ");

    if ((r.left != null) && ((l.left != null) != (l.right != null)))
        sb.append(r.left.key.toString() + " ");

    if ((r.right != null) && ((l.left != null) != (l.right != null)))
        sb.append(r.right.key.toString() + " ");

    seleziona(l, sb);
    seleziona(r, sb);
}
```

Esercizio 6 (ASD)

Scrivere un algoritmo che dato un albero binario di ricerca T con chiavi tutte distinte ed un nodo x di T stampa tutte le chiavi di T strettamente minori di key[x].

Soluzione

Utilizzando le funzioni min(x) e succ(x), possiamo scrivere:

```
y <- min(root[T])
while key[y] < key[x]
      do write key[y]
         y <- succ(y)
```

In alternativa possiamo richiamare la seguente procedura ricorsiva con y inizializzato a root[T].

```
minore(y,x)
     if (y == nil) then return
     if (y == x) then stampa-tree(left[x])
                      return
     if (key[x] < key[y])
        then minore(left[y],x)
        else write(key[y])
             stampa-tree(left[y])
             minore(right[y],x)
```

dove stampa-tree(z) stampa tutte le chiavi dell'albero radicato nel nodo z e può essere sviluppata con una semplice visita in profondità tipo:

```
stampa-tree(z)
    if z =/= NIl
       then write(key[z])
            stampa-tree(left[z])
            stampa-tree(right[z])
```

Nota: La soluzione di percorrere l'albero senza tener conto della proprietà BST, non è valutabile positivamente.

```
***************************** classe BTNode.java ***************************
package BinTrees;
class BTNode {

    Object key;       // valore associato al nodo
    BTNode parent;    // padre del nodo
    BTNode left;      // figlio sinistro del nodo
    BTNode right;     // figlio destro del nodo


    // post: ritorna un albero di un solo nodo, con valore value e sottoalberi
    //       sinistro e destro vuoti
    BTNode(Object ob) {
        key = ob;
        parent = left = right = null;
    }


    // post: ritorna un albero contenente value e i sottoalberi specificati
    BTNode(Object ob,
           BTNode left,
           BTNode right,
           BTNode parent) {
        key = ob;
        this.parent = parent;
        setLeft(left);
        setRight(right);
    }
    ....
    ....
}
***************************** classe BinaryTree.java ***************************
package BinTrees;
import java.util.Iterator;
public class BinaryTree implements BT {
    private BTNode root;        // la radice dell'albero
    private BTNode cursor;      // puntatore al nodo corrente
    private int count;          // numero nodi dell'albero


    // post: crea un albero binario vuoto
    public BinaryTree() {
        root = null;
        cursor = null;
        count = 0;
    }
    ....
    ....
}
```