

# Algoritmi e Strutture Dati & Laboratorio di Algoritmi e Programmazione

— Appello del 10 Gennaio 2005 —

## Esercizio 1 (ASD)

1. Sia  $T(n) = T(n/4) + T(3/4n) + O(n)$ . Supponendo  $T(1) = 1$ , dire, quale delle seguenti risposte è quella esatta. Giustificare la risposta.

(a)  $T(n) = O(\lg n)$

(b)  $T(n) = O(n)$

(c)  $T(n) = O(n \lg n)$

(d) Nessuna delle precedenti risposte è esatta.

2. Qual è la complessità dell'algoritmo di ricerca binaria, in funzione del numero di elementi  $n$ ? Dire quale delle seguenti risposte è quella esatta. Giustificare la risposta.

(a)  $O(n)$  nel caso peggiore

(b)  $O(\log n)$  nel caso peggiore

(c)  $O(\log \log n)$  nel caso medio

(d)  $O(\log n)$  nel caso medio ed  $O(n)$  nel caso peggiore

## Traccia di Soluzione

1. La risposta corretta è la (c). Si può dimostrare sia tramite l'albero di ricorsione sia con il metodo di sostituzione. E' forse più semplice sviluppare l'albero di ricorsione che ha somma  $c n$  su tutti i livelli pieni ed altezza  $O(\lg n)$ . Con il metodo di sostituzione si può procedere come segue. Assumiamo  $T(n) \leq dn \lg n$

$$T(n) = T\left(\frac{1}{4}n\right) + T\left(\frac{3}{4}n\right) + O(n)$$

$$\leq T\left(\frac{1}{4}n\right) + T\left(\frac{3}{4}n\right) + cn \quad \text{per definizione di } O(n), \text{ con } c > 0$$

$$\leq \frac{1}{4}dn \lg \frac{1}{4}n + \frac{3}{4}dn \lg \frac{3}{4}n + cn \quad \text{per ipotesi induttiva}$$

$$\leq \frac{1}{4}dn \lg \frac{1}{4}n + \frac{3}{4}dn \lg n + cn \quad \text{infatti } \lg \frac{3}{4}n \leq \lg n$$

$$= \frac{1}{4}dn(\lg n - 2) + \frac{3}{4}dn \lg n + cn$$

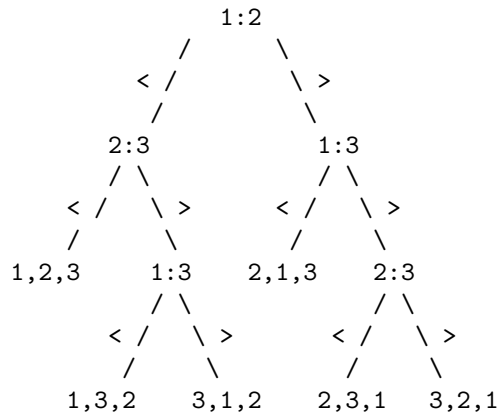
$$= dn \lg n - \left(\frac{1}{2}d - c\right)n$$

$$\leq dn \lg n \quad \text{se } d \geq 2c$$

2. La risposta corretta è la (b).

## Esercizio 2 (ASD)

Si consideri il seguente albero di decisione. Quale algoritmo di ordinamento rappresenta?



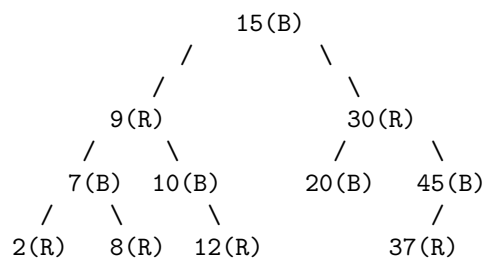
- (a) Selection Sort
- (b) Mergesort
- (c) Insertion Sort
- (d) Non rappresenta alcun algoritmo di ordinamento

### Traccia di Soluzione

La risposta corretta è la (c).

### Esercizio 3 (ASD)

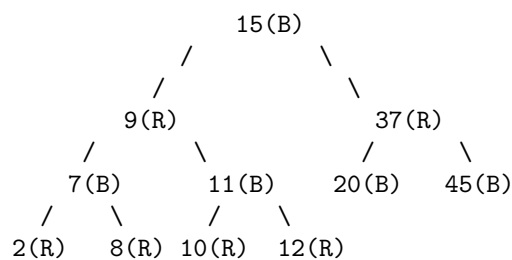
- In quanto tempo è possibile trovare il minimo in un albero binario di ricerca bilanciato di  $n$  elementi? Giustificare la risposta.
- Dato il seguente albero R/B



si consideri l'inserimento della chiave 11 seguito dalla cancellazione della chiave 30 e si disegni l'albero risultante.

### Traccia di Soluzione

- La risposta corretta è  $O(\lg n)$ .
- La parte piu' complessa è certamente l'inserimento della chiave 11 che richiede due rotazioni. L'albero finale è:



## Esercizio 4 (ASD)

Scrivete lo pseudocodice di un algoritmo che rimuove l'elemento massimo in un max-heap memorizzato in un array  $A$ . Valutare la complessità dell'algoritmo descritto.

## Traccia di Soluzione

Si tratta di realizzare l'algoritmo che elimina il massimo da un max-heap. Vedi testo.

## Esercizio 5 (ASD e Laboratorio)

Si consideri il seguente metodo della classe *SLList*, che ritorna l' $i$ -esimo elemento della lista:

```
// pre: indice i valido (cioe' 1 <= i <= size())
// post: ritorna il valore (campo key) dell'elemento i-esimo della lista;
public Object getAtIndex(int i) {...}
```

Si richiede di:

1. scrivere lo pseudocodice dell'algoritmo (ASD e Laboratorio)
2. provare la correttezza dell'algoritmo (ASD e Laboratorio)
3. determinare la complessità dell'algoritmo nel caso pessimo, giustificando la risposta (ASD e Laboratorio)
4. scrivere l'implementazione Java del metodo (Laboratorio)

## Traccia di Soluzione

1. **getAtIndex(i)**  
index  $\leftarrow$  head  
for  $j \leftarrow 1$  to  $i-1$   
do index  $\leftarrow$  next[index]  
return key[index]
2. Dobbiamo dimostrare che l'algoritmo ritorna il valore (cioè il contenuto del campo key) dell' $i$ -esimo elemento della lista. L'invariante del ciclo for è il seguente:

INV = index dista  $(i-j)$  elementi dall' $i$ -esimo della lista

**Inizializzazione:** all'inizio  $j=1$  e index è già posizionato sul primo elemento della lista. Quindi mancano  $i-1$  elementi per arrivare all' $i$ -esimo: l'invariante è vero.

**Mantenimento:** supponiamo che l'invariante sia vero per  $j$  fissato. Allora index dista  $(i-j)$  elementi dall' $i$ -esimo della lista. Il corpo del ciclo sposta index al successivo elemento. Quindi, index dista  $(i-(j+1))$  elementi dall' $i$ -esimo della lista, cioè l'invariante viene mantenuto per il ciclo successivo.

**Terminazione:** all'uscita dal ciclo  $j=i$  e quindi index dista 0 elementi dall' $i$ -esimo della lista.

L'algoritmo ritorna correttamente key[index], cioè il valore dell' $i$ -esimo elemento della lista.

3. Sia  $n$  il numero degli elementi contenuti nella lista. Nel caso pessimo  $i=n$  e index attraversa tutti gli elementi della lista. In tal caso il ciclo viene ripetuto  $n-1$  volte e quindi la complessità nel caso pessimo è  $\Theta(n)$ .
4. 

```
// pre: indice i valido (cioe' 1 <= i <= size())
// post: ritorna il valore (campo key) dell'elemento i-esimo della lista;
public Object getAtIndex(int i) {
    SLRecord index = head;
    for (int j = 1; j<i; j++)
        index = index.next;
    return index.key;
}
```

## Esercizio 6 (Laboratorio)

Si vogliono implementare due stack utilizzando un solo array. Implementare i metodi **isEmpty1**, **isFull2**, **push1**, **pop2** e il costruttore della classe *TwoStacksOneArray*. I metodi devono avere complessità costante.

```
public class TwoStacksOneArray {
    private static final int MAXSIZE = 100; // dimensione massima dell'array
    private Object[] A; // array che rappresenta i due stack
    /* ... dichiarare qui eventuali altri campi della classe ... */

    // post: inizializza i due stack vuoti
    public TwoStacksOneArray() {...}

    // post: ritorna true sse il primo stack e' vuoto
    public boolean isEmpty1() {...}

    // post: ritorna true sse il secondo stack e' vuoto
    public boolean isEmpty2() {...}

    // post: ritorna true sse il primo stack e' pieno
    public boolean isFull1() {...}

    // post: ritorna true sse il secondo stack e' pieno
    public boolean isFull2() {...}

    // post: se il primo stack non e' pieno, inserisce ob; altrimenti non fa nulla
    public void push1(Object ob) {...}

    // post: se il secondo stack non e' pieno, inserisce ob; altrimenti non fa nulla
    public void push2(Object ob) {...}

    // pre: primo stack non vuoto
    // post: rimuove e ritorna l'elemento in cima al primo stack
    public Object pop1() {...}

    // pre: secondo stack non vuoto
    // post: rimuove e ritorna l'elemento in cima al secondo stack
    public Object pop2() {...}
}
```

## Traccia di Soluzione

```
public class TwoStacksOneArray {
    private static final int MAXSIZE = 100; // dimensione massima dell'array
    private Object[] A; // array che rappresenta i due stack
    int top1; // top del primo stack
    int top2; // top del secondo stack

    // post: inizializza i due stack vuoti
    public TwoStacksOneArray() {
        A = new Object[MAXSIZE];
        top1 = -1;
        top2 = MAXSIZE;
    }

    // post: ritorna true sse il primo stack e' vuoto
    public boolean isEmpty1() {
        return (top1 == -1);
    }

    // post: ritorna true sse il secondo stack e' vuoto
    public boolean isEmpty2() {
        return (top2 == MAXSIZE);
    }
}
```

```

// post: ritorna true sse il primo stack e' pieno
public boolean isFull1() {
    return (top1 + 1 == top2);
}

// post: ritorna true sse il secondo stack e' pieno
public boolean isFull2() {
    return (top2 - 1 == top1);
}

// post: se il primo stack non e' pieno, inserisce ob; altrimenti non fa nulla
public void push1(Object ob) {
    if (!isFull1())
        A[++top1] = ob;
}

// post: se il secondo stack non e' pieno, inserisce ob; altrimenti non fa nulla
public void push2(Object ob) {
    if (!isFull2())
        A[--top2] = ob;
}

// pre: primo stack non vuoto
// post: rimuove e ritorna l'elemento in cima al primo stack
public Object pop1() {
    return A[top1--];
}

// pre: secondo stack non vuoto
// post: rimuove e ritorna l'elemento in cima al secondo stack
public Object pop2() {
    return A[top2++];
}
}

```

## Esercizio 7 (Laboratorio)

Si consideri il package *Trees* visto durante il corso e relativo agli alberi generali. Si vuole aggiungere alla classe *GenTree* il seguente metodo, che ritorna una stringa contenente tutti i nodi dell'albero di livello k:

```

// post: ritorna una stringa contenente tutti i nodi dell'albero di livello k
public String levelNodes(int k) {
    StringBuffer sb = new StringBuffer();
    sb.append("elenco nodi di livello " + k + ": ");
    if (root != null)
        getlevelNodes(root, sb, k);

    return sb.toString();
}

```

Si richiede di completare il metodo aggiungendo l'implementazione del metodo ricorsivo:

```

// pre: parametri diversi da null
// post: memorizza in sb i nodi del livello richiesto
private void getlevelNodes(TreeNode n, StringBuffer sb, int k) {...}

```

Si osservi che non ci sono precondizioni relative al livello k ricevuto in input. Quindi il metodo deve gestire anche i casi di k non valido (es. k minore di zero o maggiore dell'altezza dell'albero).

## Traccia di Soluzione

```

// pre: parametri diversi da null

```

```
// post: memorizza in sb i nodi del livello richiesto
private void getlevelKnodes(TreeNode n, StringBuffer sb, int k) {

    if (k == 0)
        sb.append(n.key.toString() + " ");

    if (k > 0 && n.child != null)
        getlevelKnodes(n.child, sb, k-1);

    if (k >= 0 && n.sibling != null)
        getlevelKnodes(n.sibling, sb, k);
}
```

```

***** CLASSE SLRecord *****
package BasicLists;
class SLRecord {
    Object key;           // valore memorizzato nell'elemento
    SLRecord next;        // riferimento al prossimo elemento

    // post: costruisce un nuovo elemento con valore v, e prossimo elemento nextel
    SLRecord(Object ob, SLRecord nextel) { key = ob; next= nextel; }

    // post: costruisce un nuovo elemento con valore v, e niente next
    SLRecord(Object ob) { this(ob,null); }
}

***** CLASSE SList *****
package BasicLists;
import Utility.Iterator;
public class SList {
    SLRecord head;        // primo elemento
    int count;            // num. elementi nella lista

    // post: crea una lista vuota
    public SList() { head = null; count = 0; }
    ...
}

***** CLASSE TreeNode *****
package Trees;
class TreeNode {

    Object key;           // valore associato al nodo
    TreeNode parent;      // padre del nodo
    TreeNode child;       // figlio sinistro del nodo
    TreeNode sibling;      // fratello destro del nodo

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

    // post: ritorna un albero contenente value e i sottoalberi specificati
    TreeNode(Object ob, TreeNode parent, TreeNode child, TreeNode sibling) {
        key = ob;
        this.parent = parent;
        this.child = child;
        this.sibling = sibling;
    }
}

***** CLASSE GenTree *****
package Trees;
import Utility.*;
public class GenTree implements Tree{
    private TreeNode root; // radice dell'albero
    private int count;     // numero di nodi dell'albero
    private TreeNode cursor; // riferimento al nodo corrente

    // post: costruisce un albero vuoto
    public GenTree() {
        root = cursor = null;
        count = 0;
    }
    ...
}

```