Corso di Programmazione

II Accertamento del 12 Giugno 2013

cognome e nome

Risolvi i seguenti esercizi giustificando sinteticamente le risposte.

1. Memoization

Il metodo statico `lcsx` risolve il problema della *sottosequenza comune più lunga* (LCS) restituendo la coppia di stringhe "allineate", cioè tali da rendere evidente la corrispondenza o meno dei simboli. A tal fine il carattere `'_'` (underscore) è trattato come simbolo speciale per rappresentare localmente la mancanza di allineamento e perciò non compare nelle stringhe passate come argomento. Le coppie di stringhe sono rappresentate da array di due elementi.

```
public static String[] lcsx( String u, String v ) {
  if ( u.equals("") && v.equals("") ) {
    return  new String[] { "", "" };
  } else if ( u.equals("") ) {
    String[] pair = lcsx( u, v.substring(1) );
    return  new String[] { '_'+pair[0], v.charAt(0)+pair[1] };
  } else if ( v.equals("") ) {
    String[] pair = lcsx( u.substring(1), v );
    return  new String[] { u.charAt(0)+pair[0], '_'+pair[1] };
  } else if ( u.charAt(0) == v.charAt(0) ) {
    String[] pair = lcsx( u.substring(1), v.substring(1) );
    return  new String[] { u.charAt(0)+pair[0], v.charAt(0)+pair[1] };
  } else {
    String[] pair1 = lcsx( u.substring(1), v );
    String[] pair2 = lcsx( u, v.substring(1) );
    return  better(
              new String[] { u.charAt(0)+pair1[0], '_'+pair1[1] },
              new String[] { '_'+pair2[0], v.charAt(0)+pair2[1] }
              );
  }
}

private static String[] better( String[] pair1, String[] pair2 ) {
  int  n1 = 0,  n2 = 0;
  for ( int i=0; i<pair1[0].length(); i=i+1 ) {
    if ( pair1[0].charAt(i) == pair1[1].charAt(i) ) {  n1 = n1 + 1;  }
  }
  for ( int i=0; i<pair2[0].length(); i=i+1 ) {
    if ( pair2[0].charAt(i) == pair2[1].charAt(i) ) {  n2 = n2 + 1;  }
  }
  if ( n1 < n2 ) {
    return pair2;
  } else if ( n1 > n2 ) {
    return pair1;
  } else if ( Math.random() < 0.5 ) {   // scelta causale
    return pair2;
  } else {
    return pair1;
  }
}
```

La valutazione di `lcsx( "arto", "atrio" )` può restituire, ad esempio, la coppia di stringhe riportata a lato, dove si vede che la sottosequenza comune è costituita dai caratteri `a`, `t`, `o`, nell'ordine.

```
art__o
a_trio
```

Trasforma il programma ricorsivo applicando opportunamente la tecnica di *memoization*.

```
public static String[] lcsxMem( String u, String v ) {

  int m = u.length();
  int n = v.length();

  String[][][] lcsDb = new String[ m+1 ][ n+1 ][];

  for ( int i=0; i<=m; i=i+1 ) {
    for ( int j=0; j<=n; j=j+1 ) {
      lcsDb[i][j] = null;
  }}
  return lcsxRec( u, v, lcsDb );
}
```

(esercizio 1)

```
private static String[] lcsxRec( String u, String v, String[][][] lcsDb ) {
  int i = u.length();
  int j = v.length();
  if ( lcsDb[i][j] == null ) {
    if ( u.equals("") && v.equals("") ) {
      lcsDb[i][j] = new String[] { "", "" };
    } else if ( u.equals("") ) {
      String[] pair = lcsxRec( u, v.substring(1), lcsDb );
      lcsDb[i][j] = new String[] { '_'+pair[0], v.charAt(0)+pair[1] };
    } else if ( v.equals("") ) {
      String[] pair = lcsxRec( u.substring(1), v, lcsDb );
      lcsDb[i][j] = new String[] { u.charAt(0)+pair[0], '_'+pair[1] };
    } else if ( u.charAt(0) == v.charAt(0) ) {
      String[] pair = lcsxRec( u.substring(1), v.substring(1), lcsDb );
      lcsDb[i][j] = new String[] { u.charAt(0)+pair[0], v.charAt(0)+pair[1] };
    } else {
      String[] pair1 = lcsxRec( u.substring(1), v, lcsDb );
      String[] pair2 = lcsxRec( u, v.substring(1), lcsDb );
      lcsDb[i][j] = better(
                      new String[] { u.charAt(0)+pair1[0], '_'+pair1[1] },
                      new String[] { '_'+pair2[0], v.charAt(0)+pair2[1] }
                      );
  }}
  return lcsDb[i][j];
}
```

2. Ricorsione e iterazione

Il metodo statico `flattenTree` codifica un *albero di Huffmann* (a meno dei pesi dei nodi) in una stringa. Il simbolo `'@'` rappresenta un nodo interno ed è seguito dalla codifica dei sottoalberi sinistro e destro; i nodi foglia sono invece identificati dai corrispondenti caratteri, eventualmente preceduti da `'\'` per risolvere ambiguità (nel caso di `'\'` e `'@'`).

```
private static String flattenTree( Node n ) {
  if ( n.isLeaf() ) {                    // foglia: codifica del carattere
    char c = n.character();
    if ( (c == '\\') || (c == '@') ) {
      return ( "\\" + c );               // caratteri speciali: \, @
    } else {
      return ( "" + c );                 // altri caratteri
    }
  } else {
    return ( "@"                         // @
           + flattenTree( n.left()  )    // codifica lineare del sottoalbero sinistro
           + flattenTree( n.right() )    // codifica lineare del sottoalbero destro
           );
  }
}
```

Trasforma il programma ricorsivo in un programma iterativo funzionalmente equivalente, utilizzando uno *stack*.

```
private static String flattenTree( Node root ) {
  String flat = "";
  Stack<Node> stack = new Stack<Node>();
  stack.push( root );
  while ( !stack.empty() ) {
    Node n = stack.pop();
    if ( n.isLeaf() ) {
      char c = n.character();
      if ( (c == '\\') || (c == '@') ) {
        flat = flat + "\\" + c;
      } else {
        flat = flat + c;
      }
    } else {
      flat = flat + "@";
      stack.push( n.right() );
      stack.push( n.left()  );
  }}
  return flat;
}
```

3. Verifica formale della correttezza

Dato un intero positivo *n*, il seguente metodo statico calcola la soluzione *gf(n)* del problema ispirato a un racconto di *Giuseppe Flavio*. Nel programma sono riportate precondizione, postcondizione, invariante e funzione di terminazione. Introduci opportune espressioni negli spazi denotati a tratto punteggiato e dimostra formalmente la correttezza *parziale* (cioè assumendo la terminazione) del programma iterativo.

```
public static int gFlavio( int n ) {      // Pre:    n > 0
  int  q = 1,  j = 0,  p = 1;
  while ( q + j < n ) {                   // Inv:    q + j ≤ n ,  ∃k . q = 2^k ,  0 ≤ j < q ,  p = 2j + 1
                                          // Term:   n − q − j
    if ( j + 1 < q ) {
      j = j + 1;

      p =  p + 2                                                              ;
    } else {

      q =  2 * q                                                              ;
      j = 0;  p = 1;
  }}
  return  p;                              // Post:   p = gf(n)
}
```

1) L'invariante vale all'inizio:

   $1 + 0 \le n$, $q = 2^0$, $0 \le 0 < 1$, $1 = 2 \cdot 0 + 1$ poiché $n > 0$ (precondizione)

2) L'invariante si conserva:

   Assumiamo $q + j \le n$ (a), $[\exists k .]\ q = 2^k$ (b), $0 \le j < q$ (c), $p = 2j + 1$ (d) e $q + j < n$ (e)

   b1) se inoltre $j + 1 < q$ (f)

   $q + j+1 \le n$ (per: e), $q = 2^k$ (per: b), $0 \le j+1 < q$ (per: c, f), $p+2 = 2(j+1) + 1$ (per: d)

   b2) se invece $j + 1 \ge q$ (g)

   $2q + 0 \le n$ (per: e, g), $2q = 2^{k+1}$ (per: b), $0 \le 0 < 2q$ (per: b), $1 = 2 \cdot 0 + 1$

3) Alla fine vale la postcondizione:

   Poiché $q + j \le n$ (a), $[\exists k .]\ q = 2^k$ (b), $0 \le j < q$ (c), $p = 2j + 1$ (d) e $q + j \ge n$ (e)

   ne consegue: $p = 2j + 1$ (per: d) $= gf(2^k+j) = gf(n)$

   in quanto $n = q+j = 2^k+j$ (per: a, b, e) e $0 \le j < q = 2^k$ (per: b, c)

4. Oggetti in Java

Considera il metodo `huffmanTree` per costruire l'*albero di Huffman* a partire dall'istogramma delle occorrenze dei caratteri in un documento, il cui codice è riportato qui sotto a destra. A differenza del programma discusso a lezione, la coda con priorità è un'istanza della classe `NodeQueue` per la quale è definito il protocollo sintetizzato sotto a sinistra.

```
// Coda con priorità di nodi: protocollo

NodeQueue()          // costruttore: struttura vuota

int size()           // numero di nodi nella struttura

void add( Node n )   // aggiunge il nodo n alla struttura

Node takeMin()       // rimuove dalla struttura e
                     // restituisce il nodo di peso minimo
```

```
public static Node huffmanTree( int[] freq ) {

  NodeQueue queue = new NodeQueue();
  for ( int c=0; c<CodingDevice.CHARS; c=c+1 ) {
    if ( freq[c] > 0 ) {
      Node n = new Node( (char) c, freq[c] );
      queue.add( n );
  }}
  while ( queue.size() > 1 ) {
    Node l = queue.takeMin();
    Node r = queue.takeMin();
    Node n = new Node( l, r );
    queue.add( n );
  }
  return queue.takeMin();
}
```

Definisci in Java una classe `NodeQueue` compatibile con quanto specificato sopra.

```
public class NodeQueue {


  private final Node[] nodes;
  private int size;


  public NodeQueue() {

    nodes = new Node[ CodingDevice.CHARS ];
    size = 0;
  }


  public int size() {

    return size;
  }


  public Node takeMin() {

    size = size - 1;
    return nodes[ size ];
  }


  public void add( Node n ) {

    size = size + 1;
    for ( int k=size-2; k>=0; k=k-1 ) {
      if ( n.weight() > nodes[k].weight() ) {
        nodes[k+1] = nodes[k];
      } else {
        nodes[k+1] = n;
        return;
    }}
    nodes[0] = n;
  }

}  // class NodeQueue
```