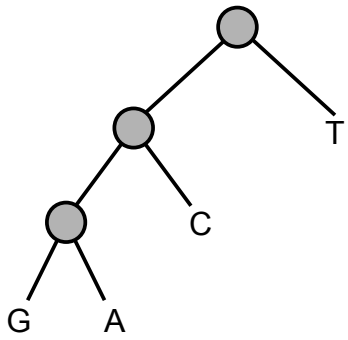
Corso di Programmazione

II Accertamento del 1 Luglio 2014

cognome e nome

Risolvi i seguenti esercizi giustificando sinteticamente le risposte.

1. Procedure con valori procedurali



Dato un albero di Huffman *tree*, rappresentato in termini di liste annidate e caratteri, il valore della procedura `decoding-fun` è la funzione di decodifica *dec* che applicata a una stringa di *0/1* restituisce il carattere corrispondente. Si assume che le stringhe passate come argomento a *dec* siano valide, cioè corrispondandano effettivamente al percorso dalla radice a una foglia di *tree*.

La struttura della rappresentazione degli alberi di Huffman è la seguente: un albero di un solo nodo è rappresentato direttamente dal carattere associato al nodo; un albero con più di un nodo è rappresentato da una lista di due elementi corrispondenti ai sottoalberi sinistro e destro. Per esempio, l'albero illustrato in figura è rappresentato dalla struttura:

```
'(((#\G #\A) #\C) #\T)
```

per cui le stringhe valide sono `"000"`, `"001"`, `"01"` e `"1"`. Quindi, sulla base della definizione

```
(define decoder (decoding-fun '(((#\G #\A) #\C) #\T)))
```

si avrà:

```
(decoder "000")   →   #\G          (decoder "01")   →   #\C
(decoder "001")   →   #\A          (decoder "1")    →   #\T
```

Completa la definizione della procedura `decoding-fun` riportata qui sotto, introducendo il codice appropriato negli spazi indicati a tratto punteggiato.

```
(define decoding-fun  ; valore procedurale
  (lambda (tree)      ; tree: albero di Huffman
    (lambda (code)    ; code: codifica valida di un carattere in base a tree

      (cond ((string=?  code                                  "")

                tree                                                    )
            ((char=? (string-ref code 0) #\0)

             ( (decoding-fun (car tree))                 (substring code 1)))
            (else

              ((decoding-fun (cadr tree)) (substring code 1))           )
            ))))
```

2. Programmazione dinamica

La procedura `llcs3` determina la *lunghezza della sottosequenza comune più lunga* (LLCS) di tre stringhe:

```
(define llcs3
  (lambda (t u v)
    (cond ((or (string=? t "") (string=? u "") (string=? v ""))
           0)
          ((char=? (string-ref t 0) (string-ref u 0) (string-ref v 0))
           (+ 1 (llcs3 (substring t 1) (substring u 1) (substring v 1))))
          (else
           (max (llcs3 (substring t 1) u v)
                (llcs3 t (substring u 1) v)
                (llcs3 t u (substring v 1))))
          )))
```

Trasforma la procedura `llcs3` in un programma non ricorsivo in *Java* che applica opportunamente la tecnica *bottom-up* di *programmazione dinamica*.

```
public static int llcs3( String t, String u, String v ) {

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

  int[][][] answer = new int[ k+1 ][ m+1 ][ n+1 ];

  for ( int i=0; i<=m; i=i+1 ) {
    for ( int j=0; j<=n; j=j+1 ) {
      answer[0][i][j] = 0;
  }}
  for ( int h=1; h<=k; h=h+1 ) {
    for ( int j=0; j<=n; j=j+1 ) {
      answer[h][0][j] = 0;
  }}
  for ( int h=1; h<=k; h=h+1 ) {
    for ( int i=1; i<=m; i=i+1 ) {
      answer[h][i][0] = 0;
  }}

  for ( int h=1; h<=k; h=h+1 ) {
    for ( int i=1; i<=m; i=i+1 ) {
      for ( int j=1; j<=n; j=j+1 ) {
        if ( (t.charAt(k-h) == u.charAt(m-i)) && (u.charAt(m-i) == v.charAt(n-j)) ) {
          answer[h][i][j] = 1 + answer[h-1][i-1][j-1];
        } else {
          answer[h][i][j]
            = Math.max( Math.max(answer[h-1][i][j], answer[h][i-1][j]),
                        answer[h][i][j-1]
                      );
        }
  }}}
  return  answer[k][m][n];
}
```

3. Verifica formale della correttezza

Dati due interi non negativi *i, j*, il seguente metodo statico calcola la soluzione del problema dei percorsi di *Manhattan*. Nel programma sono riportate precondizione, postcondizione, invariante e funzione di terminazione. Introduci opportune espressioni negli spazi denotati a tratto punteggiato e dimostra formalmente che l'invariante del comando iterativo si conserva (non è richiesta la dimostrazione degli altri passi della verifica formale di correttezza).

```
public static long manhattan( int i, int j ) {    // Pre:    i, j ≥ 0

  long p =  1                                                          ;
  int k = 0;

  while ( k < j                                  ) {    // Inv:    0 ≤ k ≤ j ,   p = (i+k)! / (i!·k!)
                                                        // Term:   j – k
    k = k + 1;

    p = p +  p * i / k                                                 ;
  }
  return  p                                      ;      // Post:   valore restituito: (i+j)! / (i!·j!)
}
```

All'inizio del generico passo iterativo si assume che siano verificati l'invariante (Inv) e la condizione del while, ovvero:

$$0 \le k < j, \quad p = \frac{(i+k)!}{i!\cdot k!}$$

Dopo il passo iterativo deve essere verificato l'invariante:

$$0 \le k' \le j, \quad p' = \frac{(i+k')!}{i!\cdot k'!}$$

cioè

$$0 \le k+1 \le j, \quad p + X = \frac{(i+k+1)!}{i!\cdot(k+1)!} = p\cdot\frac{i+k+1}{k+1}$$

che è soddisfatto per:

$$X = p\cdot\frac{i}{k+1} = p\cdot\frac{i}{k'}$$

4. Oggetti in Java

Si vuole modellare alcune funzionalità di un distributore automatico che gestisca la riserva di monete, funzionalità simili a quelle trattate in un problema di laboratorio. Supponi che si possano utilizzare solo monete da 5, 10, 20, 50, 100 (1 euro) e 200 (2 euro) centesimi. Le operazioni consentite sono l'inserimento e la resa di monete; inoltre deve essere possibile verificare la disponibilità di monete per restituire un dato resto (non è sufficiente che la riserva superi l'ammontare del resto: ci devono essere monete che consentano di restituire esattamente il resto richiesto). In sintesi, il modello è rappresentato dalla classe `DistributoreAutomatico`, per cui è definito il seguente protocollo:

`public DistributoreAutomatico()`	costruttore: la riserva iniziale di monete è vuota
`public void introduciMonete( n, v )`	carica nel distributore *n* monete da *v* centesimi
`public void rendiMonete( n, v )`	restituisce all'utente *n* monete da *v* centesimi
`public boolean restoDisponibile( v )`	verifica se ci sono monete per un resto di *v* centesimi

Realizza in Java la classe `DistributoreAutomatico` rispettando le specifiche informali illustrate sopra.

```
public class DistributoreAutomatico {


  private int[] riservaMonete;
  private int[] valoreCent;


  public DistributoreAutomatico() {

    riservaMonete = new int[ 6 ];
    valoreCent = new int[ 6 ];

    for ( int i=0; i<6; i=i+1 ) {
      riservaMonete[i] = 0;
    }
    valoreCent[0] =   5;
    valoreCent[1] =  10;
    valoreCent[2] =  20;
    valoreCent[3] =  50;
    valoreCent[4] = 100;
    valoreCent[5] = 200;
  }


  public void introduciMonete( int n, int val ) {

    int k = indice( val );
    riservaMonete[k] = riservaMonete[k] + n;
  }


  public void rendiMonete( int n, int val ) {

    int k = indice( val );
    riservaMonete[k] = riservaMonete[k] - n;
  }


  public boolean restoDisponibile( int val ) {

    int k = 5;

    while ( (val > 0) && (k >= 0) ) {

      int q = ( val / valoreCent[k] );
      val = val - Math.min( q, riservaMonete[k] ) * valoreCent[k] ;
      k = k - 1;
    }
    return  ( val == 0 );
  }


  private int indice( int val ) {

    int k = 0;
    while ( valoreCent[k] < val ) {
      k = k + 1;
    }
    return k;
  }

}  // class DistributoreAutomatico
```