Corso di Programmazione

II Accertamento del 18 Giugno 2015

cognome e nome

Risolvi i seguenti esercizi giustificando sinteticamente le risposte.

1. Verifica formale della correttezza

Dati due interi positivi *m*, *n*, il seguente metodo statico restituisce una coppia (array) di interi *i*, *j* tali che il valore dell'espressione *im* + *jn* sia il massimo comun divisore di *m*, *n*. Nel programma sono riportate precondizione, postcondizione, invariante e funzione di terminazione. Introduci opportune espressioni negli spazi denotati a tratto punteggiato in modo tale che i valori assunti dalle variabili soddisfino le relazioni specificate dalle asserzioni.

```
public static int[] extGcd( int m, int n ) {   // Pre:   m, n > 0

  int x = m, u = 1, v =   0                    ;

  int y = n, i = 0, j =   1                    ;

  int q = x / y, r =   x % y                   ;

  while (   r > 0                          ) {  // Inv:   x, y > 0,  x = qy + r,  0 ≤ r < y,
                                                //        MCD(x,y) = MCD(m,n),  x = um+vn,  y = im+jn
    x = y;  y = r;                              // Term:  y
    int s = u - q * i;
    u = i;  i = s;

    int t = v -   q * j                        ;

    v =   j                                  ;  j = t;

    q = x / y;  r =   x % y                    ;
  }
  return  new int[] { i, j };                   // Post:  MCD(m,n) = im + jn
}
```

2. Classi in Java

La classe `HiddenWord` realizza un modello del gioco della "parola nascosta" in cui il conduttore del gioco sceglie una parola (nel caso di un programma la scelta può essere effettuata casualmente a partire da un vocabolario), dicendo di quante lettere si compone, mentre gli altri partecipanti cercano di indovinarla proponendo a turno delle ipotesi. Ad ogni proposta il conduttore risponde mostrando una stringa delle corrispondenze, dove le lettere che occupano esattamente la stessa posizione nella parola nascosta e nell'ultima ipotesi formulata sono "scoperte", mentre tutte le altre sono sostituite da asterischi (incluse eventualmente quelle per cui non si può stabilire una corrispondenza perché la parola ipotizzata è più corta della parola nascosta). Naturalmente vince chi alla fine riesce a riconoscere la parola nascosta. In sintesi, per la classe `HiddenWord` è definito il seguente protocollo:

`HiddenWord( String hw )`	costruttore, a cui la parola nascosta è passata come argomento
`int length()`	metodo che restituisce lunghezza della parola nascosta
`String guess( String gw )`	metodo che, data un'ipotesi, restituisce la stringa delle corrispondenze

Per esempio, dopo aver creato l'istanza `hw = new HiddenWord("pluto")`, valutando nell'ordine le espressioni `hw.length()`, `hw.guess("pippo")`, `hw.guess("paperino")`, `hw.guess("terra")`, `hw.guess("blu")`, `hw.guess("prato")` si ottiene rispettivamente `5`, `"p***o"`, `"p****"`, `"*****"`, `"*lu**"` e `"p**to"`.

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

```
public class HiddenWord {


  private final String hword;
  private final int n;


  public HiddenWord( String hword ) {

    this.hword = hword;
    n = hword.length();
  }


  public int length() {

    return n;
  }


  public String guess( String gword ) {

    int k = Math.min( gword.length(), n );
    String match = "";

    for ( int i=0; i<k; i=i+1 ) {
      if ( gword.charAt(i) == hword.charAt(i) ) {
        match = match + hword.charAt(i);
      } else {
        match = match + "*";
    }}
    for ( int i=k; i<n; i=i+1 ) {
      match = match + "*";
    }
    return match;
  }


}  // class HiddenWord
```

3. Programmazione dinamica

```
public static long q( int i, int j, boolean b ) {  // i, j >= 0
  if ( b ) {
    if ( i*j == 0 ) {
      return  i + j + 1;
    } else {
      return  q( i-1, j, b ) + q( i, j-1, b ) + q( i, j, !b );
    }
  } else {
    if ( i*j == 0 ) {
      return  1;
    } else {
      return  q( i-1, j, b ) + q( i, j-1, b );
  }}
}
```

Considera il metodo statico `q` definito sopra. Trasforma il programma ricorsivo in un programma iterativo applicando opportunamente la tecnica *bottom-up* di *programmazione dinamica*.

```
public static long qDP( int i, int j, boolean b ) {  // i, j >= 0
  long[][][] h = new long[ i+1 ][ j+1 ][ 2 ];
  for ( int v=0; v<=j; v=v+1 ) {
    h[0][v][0] = 1;
    h[0][v][1] = v + 1;
  }
  for ( int u=1; u<=i; u=u+1 ) {
    h[u][0][0] = 1;
    h[u][0][1] = u + 1;
  }
  for ( int u=1; u<=i; u=u+1 ) {
    for ( int v=1; v<=j; v=v+1 ) {
      h[u][v][0] = h[u-1][v][0] + h[u][v-1][0];
      h[u][v][1] = h[u-1][v][1] + h[u][v-1][1] + h[u][v][0];
  }}
  if ( b ) {
    return  h[i][j][1];
  } else {
    return  h[i][j][0];
  }
}
```

4. Codifica di Huffman

Scrivi un programma che, analizzando un file di testo di nome (`String`) `src`, produce un file di testo di nome (`String`) `dst` contenente la tabella dei codici di Huffman associati a ciascuno dei caratteri contenuti nel primo file. La tabella deve essere costituita da una riga per ciascun carattere, riga che riporta il simbolo del carattere e il codice di Huffman corrispondente (stringa di 0/1) separati da uno spazio bianco; inoltre le righe devono essere ordinate per lunghezza del codice di Huffman crescente e a parità di lunghezza in base al codice ASCII del carattere.
Si intende che il tuo programma integri le unità `Node.java` e `Huffman.java` sviluppate a lezione, per cui puoi utilizzare direttamente gli strumenti che queste unità mettono a disposizione senza riportarne il codice qui. Inoltre, per ordinare opportunamente le coppie carattere / codice di Huffman, modellate dalla classe `Pair` definita sotto, utilizza una coda con priorità (`PriorityQueue`).

```
public class Pair implements Comparable<Pair> {
  private final char c;
  private final String code;
  public Pair( char c, String code ) {
    this.c = c;
    this.code = code;
  }
  public char c() {
    return c;
  }
  public String code() {
    return code;
  }
  public int codeLength() {
    return code.length();
  }
  public int compareTo( Pair p ) {
    if        (codeLength() < p.codeLength()) {
      return -1;
    } else if (codeLength() > p.codeLength()) {
      return +1;
    } else if ( c() < p.c() ) {
      return -1;
    } else {
      return +1;
    }
  }
}  // class Pair
```

```
int[] freq = Huffman.charHistogram( src );
Node root = Huffman.huffmanTree( freq );
String[] codes = Huffman.huffmanCodesTable( root );
PriorityQueue<Pair> queue = new PriorityQueue<Pair>();
for ( int c=0; c<InputTextFile.CHARS; c=c+1 ) {
  if ( freq[c] > 0 ) {
    Pair p = new Pair( (char) c, codes[c] );
    queue.add( p );
}}
OutputTextFile out = new OutputTextFile( dst );
while ( queue.size() > 0 ) {
  Pair p = queue.poll();
  out.writeTextLine( p.c() + ": "+ p.code() );
}
out.close();
```