Corso di Programmazione

II Prova di accertamento del 20 Giugno 2022 / B

cognome e nome

Riporta in modo chiaro negli appositi spazi le soluzioni degli esercizi, oppure precise indicazioni se alcune soluzioni si trovano in un foglio separato. Scrivi inoltre il tuo nome nell'intestazione e su ciascun ulteriore foglio che intendi consegnare.

1. Programmazione in Java

Un array *v* di `double` rappresenta uno *heap* se e solo se vale la relazione $v[i] \geq v[j]$ per ogni coppia (i, j) di indici dell'array tali che $j = 2i+1$ oppure $j = 2i+2$ — in altri termini quando l'indice i è il quoziente della divisione per due di $j–1$.

Definisci in Java un metodo statico `heapCheck` per verificare se un array di `double` rappresenta uno heap. Esempi:

```
heapCheck( new double[] { 8.5, 4.7, 8.5, 2.8, 3.2, 5.0, 6.3, 1.5, 2.6 } )  →  true

heapCheck( new double[] { 8.5, 4.7, 8.5, 2.8, 4.8, 5.0, 6.3, 1.5, 2.6 } )  →  false
```

2. memoization

Una sequenza *s* di `double` si definisce *smorzantesi* (damping) se ogni suo elemento ha un valore che ricade strettamente all'interno dell'intervallo delimitato dai due elementi precedenti, quando ci sono entrambi. Formalmente:

$$min(s[i–2], s[i–1]) < s[i] < max(s[i–2], s[i–1]) \quad \text{per } i \geq 2$$

Data una sequenza *s*, rappresentata da un array di `double`, il programma ricorsivo riportato nella pagina seguente ne determina la *lunghezza della sottosequenza smorzantesi più lunga* (*llds = length of the longest damping subsequence*).

```
public static int llds( double[] s ) {          // s[i] > 0 per i in [0,n-1], dove n = s.length

  return lldsRec( s, 0, UNKNOWN, UNKNOWN );
}

private static int lldsRec( double[] s, int k, int i, int j ) {

  if ( k == s.length ) {
    return 0;                                   // coda di s vuota
  } else if ( (i == UNKNOWN) ||
              ((Math.min(s[i],s[j]) < s[k]) && (s[k] < Math.max(s[i],s[j]))) ) {

    return Math.max( 1+lldsRec(s,k+1,j,k),
                       lldsRec(s,k+1,i,j) );    // s[k] può essere scelto o meno
  } else {
    return lldsRec( s, k+1, i, j );             // s[k] non può essere scelto
  }
}

private static final int UNKNOWN = -1;         // indice i indefinito
```

Completa il programma riportato qui sotto, che applica una tecnica *top-down* di *memoization* per rendere più efficiente la computazione ricorsiva avviata da `llds`.

```
public static int llds( double[] s ) {

  int n = s.length;

  ..........  mem = ..........  ;

  ..........

  ..........

  ..........

  ..........

  ..........

  ..........

  return lldsRec( s, ..........  , mem );
}

private static int lldsRec( double[] s, ..........  , ..........  mem ) {


  if ( mem ..........  == ..........  ) {

    ..........

    ..........

    ..........

    ..........

    ..........

    ..........

  }

  return ..........  ;
}
```

3. Ricorsione e iterazione

Dato l'albero di Huffman costruito sulla base di uno specifico documento, e completo dell'informazione sul numero di occorrenze dei caratteri utilizzarti, il seguente programma ricorsivo calcola il numero di byte che saranno richiesti per la codifica di Huffman di quel documento attraverso gli strumenti discussi a lezione.

```
public static int huffmanCodeSize( Node root ) {

  long bits = recSize( root, "" );      // visita dell'albero di Huffman

  return (int) ( bits / 7 ) + ( (bits%7 > 0) ? 1 : 0 );
}

private static long recSize( Node n, String path ) {

  if ( n.isLeaf() ) {
    return path.length() * n.weight();
  } else {
    return recSize( n.left(), path+"0" ) + recSize( n.right(), path+"1" );
  }
}
```

Completa le definizioni della classe `Pair` e del metodo `codeSizeIter` per trasformare la ricorsione in iterazione applicando uno stack.

```
public class Pair {
  public final Node node;
  public final String path;
  public Pair( Node n , String p ) {
      node = n;
      path = p;
  }
}   // class Pair

public static int codeSizeIter( Node root ) {
  long bits = 0;
  Stack<Pair> stack = new Stack<Pair>();
  stack.push( new Pair( root, "" ) );
  do {
    Pair current = stack.pop();
    Node n = current.node;
    String path = current.path;
    if (n.isLeaf()) {   bits += path.length()*n.weigth()   }
    else {
          stack.push ( new Pair ( n.right(), path+"1" ) ) ;
          stack.push ( new Pair ( n.left(), path+"0" ) ) ;
    }

  } while ( ! stack.empty() );
  return (int) ( bits / 7 ) + ( (bits%7 > 0) ? 1 : 0 );
}
```

4. Classi in Java

Con riferimento all'esercizio precedente, immagina di voler definire direttamente una classe `PairStack` che possa sostituire `Stack<Pair>` realizzando tutte le funzionalità utilizzate. In particolare, l'unica differenza relativa al programma iterativo `codeSizeIter` consisterà semplicemente nel sostiutire il comando:

```
Stack<Pair> stack = new Stack<Pair>();
```

con:

```
PairStack stack = new PairStack();
```

senza che si renda necessario apportare altre modifiche al codice. Inoltre, per rappresentare lo stato interno delle istanze della classe `PairStack` si introdurrà un'unica variabile di istanza di tipo `SList<Pair>`.

Completa la definizione della classe `PairStack` riportata nel riquadro.

```
public class PairStack {

  private SList<Pair> pairs;

  public PairStack() {
        pairs = new Slist<Pair>();
      // LISTA VUOTA = STACK VUOTO

  }

  public boolean empty() {
       return pairs.isNull();
     // metodo appartenente alla classe SList<E>

  }
  public void push( Pair pair ) {
      pairs = pairs.cons(pair);
     // mettendolo come primo elemento verrà anche prelevato per primo

  }
  public Pair peek() {
       return pairs.car();
     // primo elemento della SList pairs

  }
  public Pair pop() {
     Pair p = pairs.car(); //salva il primo elemento
     pairs = pairs.cdr();
     return p;     /// PRIMO ELEMENTO DELLA VECCHIA LISTA

  }

}   // class PairStack
```