Laboratorio di Programmazione a.a. 2023-24

Esercitazione su temi d'esame

5 e 12 Giugno 2024

1. Memoization (15 Luglio 2014)

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 *Java* che applica opportunamente la tecnica *top-down* di *memoization*.

2. Programmazione in Java (15 Luglio 2014)

Come è noto dall'algebra lineare, una matrice quadrata Q si dice *simmetrica* se coincide con la propria trasposta, ovvero se $Q_{ij} = Q_{ji}$ per tutte le coppie di indici della matrice. Scrivi un metodo statico in Java per verificare se l'argomento è una matrice simmetrica — assumendo che tale argomento rappresenti una matrice *quadrata*.

Per quanto possibile, struttura il codice in modo tale da ridurre al minimo il numero di confronti effettuati dal programma nei casi in cui la matrice sia effettivamente simmetrica.

3. Oggetti in Java (9 Settembre 2015)

Il modello della scacchiera realizzato dalla classe `Board` per affrontare il rompicapo delle n regine deve essere integrato introducendo due nuovi metodi: `isFreeRow(int)`, che dato un indice di riga i compreso fra 1 e la dimensione n della scacchiera, restituirà *true* se e solo se nella riga i non c'è alcuna regina; `addQueen(String)`, che svolge la stessa funzione di `addQueen(int,int)` ma ricevendo come argomento la codifica della posizione tramite una stringa di due caratteri, una lettera per la colonna e una cifra per la riga secondo le convenzioni consuete. Inoltre, `addQueen` e `removeQueen` non devono modificare lo stato della scacchiera se l'operazione è inconsistente perché due regine verrebbero a trovarsi sulla stessa casella oppure perché nella posizione data non c'è una regina da rimuovere. Per esempio, il metodo statico `listOfCompletions`, definito sotto a destra, stamperà tutte le soluzioni del rompicapo, se ve ne sono, compatibili con una disposizione iniziale di regine che non si minacciano reciprocamente.
In base a quanto specificato sopra, modifica opportunamente la classe `Board`.

```
Board( int n )  // costruttore

void addQueen( int i, int j )
void removeQueen( int i, int j )

int size()
int queensOn()
boolean underAttack( int i, int j )
String arrangement()

boolean isFreeRow( int i )
void addQueen( String pos )

// esempio:  b.addQueen( "b6" )
```

```
public static void listOfCompletions( Board b ) {
  int n = b.size();  int q = b.queensOn();
  if ( q == n ) {
    System.out.println( b.arrangement() );
  } else {
    int i = 1;
    while ( !b.isFreeRow(i) ) {
      i = i + 1;      // ricerca di una riga libera
    }
    for ( int j=1; j<=n; j=j+1 ) {
      if ( ! b.underAttack(i,j) ) {
        b.addQueen( i, j );
        listOfCompletions( b );
        b.removeQueen( i, j );
  }}}
}
```

4. Ricorsione e iterazione (4 Luglio 2016)

Dato un *albero di Huffman*, il metodo statico `shortestCodeLength` determina la lunghezza del più corto fra i codici di Huffman associati ai caratteri che compaiono in un documento di testo. Più specificamente, la visita dell'albero, finalizzata alla determinazione di tale lunghezza, è realizzata attraverso uno schema iterativo.
Completa la definizione del metodo `shortestCodeLength` riportata nel riquadro.

```
public static int shortestCodeLength( Node root ) {
  int sc = ..... ;
  Stack<Node> stack = new Stack<Node>();
  Stack<Integer> depth = new Stack<Integer>();
  stack.push( root );
  depth.push( 0 );
  do {
    Node n = ..... ;
    int  d = ..... ;
    if ( n.isLeaf() ) {
      sc = Math.min( sc, d );
    } else if ( d+1 < ..... ) {
      .....
      .....
      .....
      .....
    }
  } while ( ..... );
  return sc;
}
```

5. Programmazione in Java (5 Settembre 2016)

Dato un *array* di numeri (`double`) con almeno due elementi, il metodo statico `closestPair` ne restituisce una coppia la cui differenza in valore assoluto è minima. La coppia è rappresentata da un array ordinato di due elementi. Esempio:

`closestPair( new double[] {0.3, 0.1, 0.6, 0.8, 0.5, 1.1} )` → `{0.5, 0.6}`

Definisci in Java il metodo statico `closestPair`.

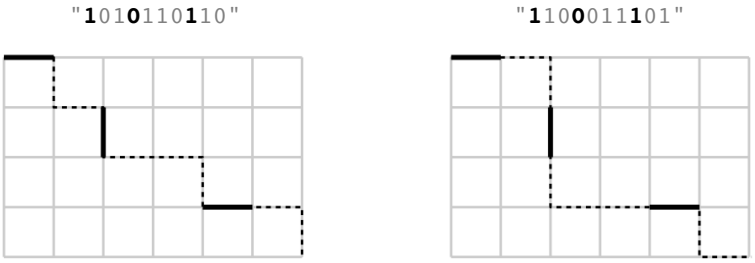
6. Programmazione in Java (24 Settembre 2019)

Gli argomenti del metodo statico `commonStretches` sono due stringhe `u`, `v` composte solo dai caratteri `'0'` e `'1'`, entrambe con esattamente *m* occorrenze di `'0'` ed *n* occorrenze di `'1'`. Tali stringhe rappresentano due *percorsi di Manhattan* in un reticolo di *m* x *n* tratti di strada, dove `'0'` denota uno spostamento verticale e `'1'` orizzontale. Date le stringhe `u`, `v`, il metodo `commonStretches` calcola il numero di tratti di strada comuni ai due percorsi, numero che può variare da *0* a *m*+*n*, nel secondo caso quando i due percorsi (e quindi le due stringhe) coincidono. Per esempio:

`commonStretches( "1010110110", "1100011101" )` → `3` *// m = 4, n = 6*

Affinché un certo tratto sia comune a due percorsi, questi devono passare per lo stesso nodo (incrocio) e lo spostamento successivo deve chiaramente procedere nella stessa direzione (orizzontale o verticale). I due percorsi passano per lo stesso nodo se una parte iniziale (prefisso) delle stringhe che li rappresentano è composta dallo stesso numero di `'0'` e dallo stesso numero di `'1'`; inoltre, lo spostamento che segue va nella stessa direzione se il carattere successivo delle due stringhe coincide.

Nell'esempio questa situazione si verifica in corrispondenza ai caratteri in posizione *0*, *3* e *7* come illustrato dalla figura qui a lato.



Definisci in Java il metodo statico `commonStretches` in accordo con le specifiche indicate.

7. Ricorsione e iterazione (20 Giugno 2022)

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, 0 );      // visita dell'albero di Huffman

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

private static long recSize( Node n, int depth ) {

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

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

```
public class Frame {

  public final Node node;
  public final int depth;

  public Frame( ..... ) {

    ..... ;

    ..... ;
  }
}   // class Frame


public static int codeSizeIter( Node root ) {

  long bits = 0;

  Stack<Frame> stack = new Stack<Frame>();

  ..... ;

  do {

    Frame current = ..... ;

    Node n = ..... ;

    int depth = ..... ;

    .....

    .....

    .....

    .....

    .....

    .....

  } while ( ..... );

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

8. Programmazione in Java (20 Giugno 2022)

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

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

```
heapCheck( new double[] { 5.0, 3.1, 5.7, 3.1, 8.5, 6.0, 3.8, 4.2, 9.3 } )  →  true

heapCheck( new double[] { 5.0, 3.1, 5.7, 3.1, 8.5, 6.0, 3.0, 4.2, 9.3 } )  →  false
```

9. Programmazione dinamica (16 Settembre 2022)

Una stringa è una palindrome, o *palindromica*, quando la sequenza dei caratteri risulta la stessa leggendola da sinistra a destra oppure da destra a sinistra. Data una stringa *s*, la procedura ricorsiva `lps`, riportata nella pagina seguente, ne determina la *sottosequenza palindromica più lunga* (*LPS = Longest Palindromic Subsequence*) — più precisamente, una delle sue sottosequenze palindromiche più lunghe. Il metodo statico `longer` è lo stesso introdotto a lezione per realizzare `lcs` e restituisce semplicemente la più lunga delle due stringhe passate come argomenti. Esempio:

```
lps( "irradiare" )  ⟶  "radar"
```

```
public static String lps( String s ) {            // longest palindromic subsequence
  int n = s.length();
  if ( n < 2 ) {                                  // stringa vuota o di un solo carattere: palindrome
    return s;
  } else if ( s.charAt(0) == s.charAt(n-1) ) {    // caratteri estremi uguali: fanno parte del risultato
    return s.charAt(0) + lps( s.substring(1,n-1) ) + s.charAt(n-1);
  } else {                                        // caratteri estremi diversi: almeno uno va scartato
    return longer( lps(s.substring(0,n-1)), lps(s.substring(1,n)) );
  }
}
```

Completa il programma impostato nel riquadro, basato sulla procedura `lpsDP`, che applica una tecnica *bottom-up* di *programmazione dinamica* per rendere più efficiente la computazione ricorsiva svolta da `lps`.

```
public static String lpsDP( String s ) {
  int n = s.length();
  ..........  mem = ..........  ;
  for ( int k=0; k<=n; k=k+1 ) {
    for ( int i=0; i<=n-k; i=i+1 ) {
      // k :  lunghezza della sottostringa s* di s considerata;
      // i :  posizione di s* in s:
      //      s* corrisponde al potenziale argomento di una invocazione ricorsiva di lps.
      if ( k < 2 ) {
        ..........
        ..........
      } else if ( s.charAt(i) == s.charAt(i+k-1) ) {
        ..........
        ..........
      } else {
        ..........
        ..........
      }
  }}
  return ..........  ;
}
```