Corso di Programmazione

Esame del 2 Settembre 2020

cognome e nome

Risolvi i seguenti esercizi giustificando sinteticamente le risposte.

1. Ricorsione di coda e iterazione

La procedura (metodo statico in Java) `josephus` risolve il problema ispirato a una leggenda di Giuseppe Flavio:

```
public static int josephus( int n ) {   // n = 2^k + j  t.c.  2^(k+1) > n  e  j ≥ 0
  int x = 1,  y = 1,  z = 1;
  while ( x+y <= n ) {
    if ( x == y ) {
      x = 2*x;  y = 1;  z = 1;
    } else {
      y = y+1;  z = z+2;
  }}
  return z;                                 // z = 2j + 1
}
```

Oggetto di questo esercizio è la trasformazione del programma che realizza `josephus` in un programma in Scheme basato sulla ricorsione di coda.

1.1. La procedura Scheme `josephus` è intesa ad avviare una computazione essenzialmente equivalente a quella dell'omonimo metodo statico Java. Completane la definizione introducendo gli opportuni argomenti della procedura ricorsiva di coda `josephus-rec`:

```
(define josephus
  (lambda (n)
    (josephus-rec ..... )
    ))
```

1.2. Trasforma il costrutto iterativo (`while`) definendo la corrispondente procedura ricorsiva di coda `josephus-rec`:

2. Argomenti procedurali in Scheme

La procedura `tessellations` risolve una variante del problema delle tassellazioni lineari affrontato a lezione, dove le piastrelle hanno tre colori diversi: rosso (`R`), giallo (`G`) e blu (`B`). Più precisamente, si vuole conoscere in quanti modi diversi sia possibile coprire un cordolo di lunghezza *n* allineando piastrelle quadrate di lato unitario dei tre colori disponibili, rispettando il vincolo che le piastrelle rosse non possono essere collocate in posizioni adiacenti; in altri termini, due piastrelle rosse devono sempre essere separate da una o più piastrelle di colore diverso. Nel caso in esame, inoltre, `tessellations` restituisce la lista di tutte le sequenze diverse di *n* piastrelle colorate che rispettano il vincolo stabilito. Ciascuna sequenza è rappresentata da una stringa composta dalle lettere `"R"`, `"G"` e `"B"`, che identificano i corrispondenti colori.

Per esempio, valutando l'espressione `(tessellations 4)` si ricava una lista di 60 elementi, in cui sono contenute, fra le altre, le stringhe `"RGRB"`, `"RBBR"`, `"GBRG"` e `"GGGG"`; *non* vi appartengono, invece, stringhe come `"RRRB"`, `"RGRR"`, `"GRRB"` e `"BGRR"`, che presentano occorrenze adiacenti di `"R"`.

2.1. La lista restituita da `(tessellations 3)` include le seguenti stringhe:

`"RGR"` `"RGG"` `"RBG"` `"GRG"` `"GRB"` `"GGR"` `"GGB"` `"GBR"`
`"GBG"` `"GBB"` `"BRG"` `"BRB"` `"BGG"` `"BBR"` `"BBG"` `"BBB"`

Riporta tutte le ulteriori stringhe che completano la lista:

.....

.....

2.2. Completa la definizione della procedura `tessellations`, basata sull'applicazione di `map` alle liste che risolvono istanze un po'più semplici dello stesso problema (attraverso ricorsioni ad albero).

```
(define tessellations  ; val : intero
  (lambda (n)          ; n : intero non negativo
    (cond ((= n 0) (list ""))

          ((= n 1) (list ..... ))
          (else
           (let ((u (tessellations (- n 2)))

                 (v ..... )
                 )

             (append (map ..... u)

                     (map ..... u)

                     (map ..... v)

                     .....

                     ))
           ))
    ))
```

3. Memoization

Considera la seguente procedura funzionale (metodo statico), basata su una ricorsione ad albero:

```
public static long rec( int n, int k ) {   // n, k : interi qualsiasi
  int a = Math.abs( k );
  if ( n < a ) {
    return 0;
  } else if ( n == a ) {
    return 1;
  } else {
    return rec( n-1, k-1 ) + 2*rec( n-1, k ) + rec( n-1, k+1 );
  }
}
```

3.1. Completa il seguente programma che applica una tecnica *top-down* di memoization per realizzare in modo più efficiente la procedura `rec`.

```
public static long mem( int n, int k ) {   // funzionalmente equivalente a rec(n,k)
  if ( n < 0 ) {

    return ..... ;
  }

  long[][] h = new long[ ..... ][ ..... ];

  for ( int i=0; i<= ..... ; i=i+1 ) {

    for ( int j=0; j<= ..... ; j=j+1 ) {
      h[i][j] = UNKNOWN;
  }}
  return mrec( n, k, h );
}

private static long mrec( int n, int k, long[][] h ) {
  int a = Math.abs( k );
  if ( n < a ) {

    return ..... ;
  } else {

    if ( ..... == UNKNOWN ) {
      if ( n == a ) {

        ..... = ..... ;
      } else {

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

private static final int UNKNOWN = ..... ;
```

3.2. Fissato $n = 5$, per quali valori del parametro k `rec(n,k)` $\neq 0$? .....

E fissato $k = 5$, per quali valori del parametro n `rec(n,k)` $\neq 0$? .....

4. Ricorsione e iterazione

Dati un intero positivo `k` e la radice `root` di un *albero di Huffman*, il metodo statico `charsWithCodeLength` simula una visita ricorsiva dell'albero, utilizzando uno stack, per selezionare i caratteri il cui codice di Huffman ha lunghezza `k` (cioè i caratteri associati a nodi dell'albero di profondità `k`). Nell'array di booleani restituito, i caratteri selezionati sono quelli per cui l'elemento associato al corrispondente codice ASCII (indice dell'array) è *true*. Le coppie di parametri delle ricorsioni implicite, da registrarsi nello stack, sono rappresentate attraverso istanze della classe `Frame`.

4.1. Completa la definizione di `charsWithCodeLength`:

```
public static boolean[] charsWithCodeLength( int k, Node root ) {
  boolean[] selectedChars = new boolean[ InputTextFile.CHARS ];
  for ( int i=0; i<selectedChars.length; i=i+1 ) {

    selectedChars[i] = ..... ;
  }
  Stack<Frame> stack = new Stack<Frame>();
  stack.push( new Frame(0,root) );
  do {
    Frame fr = stack.pop();
    int  d = fr.depth;
    Node n = fr.node;

    if ( ..... ) {
      if ( d == k ) {
        selectedChars[ n.character() ] = true;
      }
    } else if ( d < k ) {

      ..... ;

      ..... ;
    }

  }  while ( ..... );
  return selectedChars;
}
```

4.2. Definisci in Java una classe `Frame` in accordo con l'uso che ne viene fatto in `charsWithCodeLength`:

5. Classi in Java

Fai riferimento alla definizione della classe `Board` le cui istanze sono oggetti dotati di uno stato che può evolvere nel corso dell'elaborazione, aggiungendo e rimuovendo regine. Si desidera modificare il protocollo e le funzionalità del metodo `addQueen` in accordo con le seguenti specifiche:

(i) Il protocollo aggiornato di `addQueen` prevede un valore restituito di tipo `String` che rappresenta testualmente le regine in potenziale conflitto (attacco reciproco) con un'eventuale regina collocata nella casella di coordinate (*i, j*):

```
public String addQueen( int i, int j )
```

(ii) Per quanto riguarda le funzionalità, se `b` è un oggetto di tipo `Board`, l'esecuzione di `b.addQueen(i,j)` colloca una nuova regina in posizione (*i, j*) quando questa non è minacciata da altre regine disposte sulla scacchiera `b`, così come avveniva per la versione di `addQueen` discussa a lezione, e inoltre il metodo restituisce `null`. Se invece la posizione (*i, j*) è minacciata in `b`, l'esecuzione di `b.addQueen(i,j)` non modifica la configurazione della scacchiera, ma restituisce una stringa in cui le regine di b che minacciano la casella (*i, j*) sono rappresentate nel modo consueto da coppie lettera-cifra (per identificare rispettivamente una colonna e una riga della scacchiera, come nelle stringhe restituite dal metodo `arrangement`).

Per esempio, considera una scacchiera `b` di dimensione 6 x 6 e supponi che la configurazione delle regine, descritta da `b.arrangement()`, sia: `" c1 f2 b3 e4 a5 "`.

Allora `b.addQueen(4,1)` non modifica lo stato di `b` e restituisce una stringa che codifica le coppie `e4`, `a5` e `b3` (l'ordine non è importante), in quanto le regine in queste posizioni minacciano la casella (*4,1*), ovvero `a4`; infatti `a4` e `e4` si trovano sulla stessa riga, `a4` e `a5` sulla stessa colonna, `a4` e `b3` su una stessa diagonale.

Invece `b.addQueen(6,4)` restituisce `null` e aggiunge una nuova regina in posizione (*6,4*), corrispondente a `d6` che non è in conflitto con le altre regine, portando `b` alla nuova configurazione `" c1 f2 b3 e4 a5 d6 "`.

Nei punti seguenti ti è richiesto di formalizzare le modifiche da apportare alla classe `Board` per realizzare quanto specificato sopra. Si intende che il resto del protocollo deve rimanere inalterato rispetto alla versione di riferimento.

5.1. Formalizza le integrazioni e/o modifiche relative alle variabili di istanza della classe `Board`, senza riprodurre le parti che non subiscono variazioni:

5.2. Formalizza la versione aggiornata del metodo `addQueen` e di eventuali altri metodi su cui si renda necessario intervenire in conseguenza alle modifiche delle variabili di istanza: