Corso di Programmazione

Esame del 13 Luglio 2020

cognome e nome

Risolvi i seguenti esercizi giustificando sinteticamente le risposte.

1. Ricorsione di coda e correttezza dei programmi ricorsivi

La procedura `pattern-repeats` calcola il numero di occorrenze del pattern p nel testo s, dove p e s sono stringhe.

```
(define pattern-repeats                          ; val : intero
  (lambda (p s)                                  ; p ≠ "", s : stringhe
    (repeats-rec p s (string-length p) 0)
    ))

(define repeats-rec                              ; val : intero
  (lambda (p s k r)                              ; p ≠ "", s : stringhe, k = len(p), r : interi
    (cond ((< (string-length s) k)
           r)
          ((string=? p (substring s 0 k))
           (repeats-rec p (substring s 1) k (+ r 1)))
          (else
           (repeats-rec p (substring s 1) k r))
          )))
```

Formalmente, per ogni coppia di stringhe $p \neq ""$ (p diversa dalla stringa vuota) e s :

$$\texttt{(pattern-repeats } p\ s\texttt{)} \quad \rightarrow \quad |\{\, i :\ 0 \le i \le len(s)\text{–}len(p)\ \wedge\ p = sub(\, s, i, i+len(p)\,)\,\}|$$

dove $len(x)$ è la lunghezza della stringa x e $sub(x,i,j)$ è la sottostringa di x dal carattere di posizione i a quello di posizione j–1. (Il risultato è espresso in termini di cardinalità dell'insieme di indici per i quali valgono le prorpietà specificate a destra dei due punti.)

1.1. Considera la procedura ricorsiva `repeats-rec`: come può essere formalizzato il valore intero restituito da `repeats-rec` per poter eventualmente dimostrare la correttezza sia di `repeats-rec` che di `pattern-repeats`?
Più precisamente, date due stringhe $p \neq ""$, s, l'intero $k = len(p)$ e un intero r :

`(repeats-rec` $p\ s\ k\ r$`)` $\rightarrow$ .....

1.2. La procedura `repeats-rec` applica la *ricorsione di coda* (tail recursion) e pertanto la relativa computazione può essere rielaborata in forma iterativa introducendo variabili di stato che corrispondano a ciascuno dei parametri della procedura. Scrivi un programma in Java basato su un ciclo *while* per realizzare una computazione sostanzialmente equivalente a quella di `repeats-rec` (a prescindere dall'eventuale utilizzo di `pattern-repeats`).

2. Programmazione in Scheme

Data una lista *w* non vuota di stringhe binarie (composte solo dalle cifre 0 e 1), tutte della stessa lunghezza *n*, la procedura `parity-check` restituisce la stringa binaria *p* di lunghezza *n* che rappresenta il controllo di parità delle stringhe contenute in *w*. Ciò significa che il *k*-imo bit di *p* è 1 se e solo se 1 occorre un numero dispari di volte fra i bit in posizione *k*-ima nelle stringhe di *w*. In altri termini, immaginando di incolonnare le stringhe di *w*, il *k*-imo bit di *p* è 1 se c'è un numero dispari di 1 nella *k*-ima colonna, è invece 0 se la *k*-ima colonna contiene un numero pari di 1. Per esempio:

```
(parity-check '("00101011" "11000011" "10110101"))  →  "01011101"    (*)
```

2.1. Completa il programma per realizzare `parity-check` definendo la procedura `match`.

```
(define parity-check  ; val :  stringa binaria
  (lambda (w)         ; w :  lista non vuota di stringhe binarie della stessa lunghezza
    (parity-rec (cdr w) (car w))
    ))

(define parity-rec    ; val :  stringa binaria
  (lambda (w p)       ; w :  lista di stringhe binarie,  p :  stringa binaria
    (if (null? w)
        p
        (parity-rec (cdr w) (match (car w) p))
        )))
```

2.2. Nel corso della valutazione dell'esempio (*) riportato sopra, per due volte la procedura `match` è invocata dalla procedura `parity-rec` (ciò avviene in corrispondenza alla valutazione del ramo "else" dell'`if` di `parity-rec`). Quali sono gli argomenti e il risultato di ciascuna delle due invocazioni?

`(match` ..... `)` → .....

`(match` ..... `)` → .....

3. Memoization

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

```
public static long rec( int x, int y, int z ) {   // 1 ≤ x, y ≤ z

  if ( (x == 1) || (y == z) ) {
    return 1;
  } else {
    return rec( x-1, y, z ) + x * rec( x, y+1, z );
  }
}
```

3.1. Supponi che nel corso dell'esecuzione di un programma che utilizza `rec` venga valutata l'espressione:

```
rec( 7, 6, 13 )
```

Questa valutazione si svilupperà attraverso successive invocazioni ricorsive di `rec(x,y,z)` per diversi valori degli argomenti x, y e z. Quali sono il valore più piccolo e il valore più grande che assumerà ciascuno degli argomenti nelle ricorsioni che discendono da `rec(7,6,13)` ?

- Valore più piccolo di x : 1 e valore più grande di x : X ;
- Valore più piccolo di y : 4 e valore più grande di y : 7 ;
- Valore più piccolo di z : 7 e valore più grande di z : 7 .

3.2. Applica una tecnica *top-down* (ricorsiva) di memoization per realizzare la procedura `rec` in modo più efficiente.

4. Ricorsione e iterazione

La procedura `tessellations`, definita dal seguente programma, determina il numero di modi diversi in cui si può "piastrellare" un cordolo di lunghezza n x 1 utilizzando piastrelle quadrate rosse di dimensione 1 x 1 e piastrelle rettangolari blu di dimensione k x 1, dove $k \geq 1$. (Non ci sono comunque ulteriori vincoli: sia le piastrelle rosse che quelle blu possono essere disposte anche in posizioni adiacenti.)

4.1. Completa la definizione della procedura ricorsiva `tessRec`.

```
public static long tessellations( int n, int k ) {

  long[] c = new long[ 1 ];
  c[0] = 0;

  tessRec( n, k, c );

  return c[0];
}


public static void tessRec( int n, int k, long[] c ) {

  if ( n < k ) {
    c[0] = c[0] + 1;
  } else {
    tessRec( n-1, k, c );

    tessRec( n   k-1   c ....................................... );
  }
}
```

4.2. Completa la definizione del metodo `tessIter` che trasforma la ricorsione in iterazione utilizzando uno stack.

```
public static long tessIter( .......................................... ) {

  long[] c = new long[ 1 ];
  c[0] = 0;

  Stack< ............... > s = ........................................ ;

  s.push( ............................................................ );

  do {

    ............................................................ = s.pop();

    if ( ............................................................ ) {
      c[0] = c[0] + 1;
    } else {

      ................................................................ ;

      ................................................................ ;
    }
  } while ( ! s.empty() );

  return c[0];
}
```

5. Classi in Java

Per costruire l'albero di Huffman è stata applicata una coda con priorità. Sulla base dello stesso protocollo, una coda semplice garantisce invece che gli elementi vengano estratti rispettando esattamente l'ordine di inserimento: il primo ad essere inserito sarà il primo ad essere estratto. In particolare, considerando una coda di interi `IntQueue`, il protocollo può essere specificato come segue:

```
IntQueue q = new IntQueue(c);        // costruttore di una coda vuota di capacità massima c

q.size()     // numero di interi contenuti nella coda

q.add(n)     // aggiunge l'intero n alla coda se la capacità massima non viene superata

q.peek()     // restituisce l'intero che è stato inserito prima, fra quelli presenti nella coda

q.poll()     // restituisce e rimuove dalla coda l'intero inserito da più tempo
```

La capacità della coda, cioè il numero massimo di elementi che può contenere, viene assegnata una volta per tutte al momento della costruzione. L'eventuale inserimento di un elemento in una coda "piena" avrà come conseguenza che il contenuto della coda non venga modificato.

5.1. Definisci in Java una classe `IntQueue` che realizzi il protocollo specificato sopra.

5.2. Si potrebbe generalizzare la struttura rispetto al tipo degli elementi definendo `Queue<T>` ?
Indica brevemente quali modifiche andrebbero apportate relativamente alle variabili di istanza e al metodo `peek()`.