Corso di Programmazione

Prova scritta d'esame del 22 Luglio 2022

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 $v[i] \le v[j]$ per ogni coppia di indici *positivi* dell'array $i, j \ge 1$ 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 `heapTest` che, dato un array di `double`, restituisce `null` se questo rappresenta uno heap, oppure restituisce una coppia di indici (un array di due elementi) per cui non è soddisfatta la condizione dello heap. Esempi:

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

heapTest( new double[] {5.0, 3.1, 5.7, 3.1, 8.5, 6.0, 3.0, 4.2, 9.3} )  →  {3, 6}
```

2. Programmazione dinamica

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:

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

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 indefinito
```

Completa il programma impostato nel riquadro, applicando una tecnica iterativa *bottom-up* di *programmazione dinamica* per rendere più efficiente la computazione ricorsiva avviata da `llds`.

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

  int n = s.length;

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

  ..........

  ..........

  ..........

  ..........

  ..........

  ..........

  ..........

  ..........

  ..........

  ..........

  ..........

  ..........

  ..........

  ..........

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

3. Argomenti procedurali in Scheme

Il seguente programma calcola il numero di sequenze binarie diverse di lunghezza *n* che si possono comporre utilizzando esclusivamente i simboli `0` e `1`, rispettando l'ulteriore vincolo che in ogni prefisso (porzione sinistra) della sequenza, inclusa l'intera sequenza, la differenza fra il numero di occorrenze di `0` e di `1` sia compresa fra $-k$ e k. Per esempio, `"1101100"` non rispetta il vincolo per $k = 2$ relativamente al prefisso `"11011"` (differenza $= -3$).

```
(define seq-num   ; val: intero
  (lambda (n k)   ; n ≥ 0, k ≥ 1 interi
    (num-rec n k 0)
    ))
```

```
(define num-rec
  (lambda (n k j)       ; j: eccedenza di 0 a sinistra
    (cond ((= n 0)
           1)
          ((= j (- k)) ; prossimo simbolo: deve essere 0
           (num-rec (- n 1) k (+ j 1)))
          ((= j k)     ; prossimo simbolo: deve essere 1
           (num-rec (- n 1) k (- j 1)))
          (else        ; prossimo simbolo: 0 oppure 1
           (+ (num-rec (- n 1) k (+ j 1))
              (num-rec (- n 1) k (- j 1))))
          )))
```

Il programma impostato nel riquadro, ha invece l'obiettivo di restituire *la lista di tali sequenze binarie*. Esempi:

```
(bin-sequences 1 2)  →  ("0" "1")
(bin-sequences 2 2)  →  ("00" "01" "10" "11")
(bin-sequences 3 2)  →  ("001" "010" "011" "100" "101" "110")
(bin-sequences 4 1)  →  ("0101" "0110" "1001" "1010")
```

Completa la procedura ricorsiva `bin-sequences` inserendo espressioni appropriate negli spazi indicati.

```
(define bin-sequences  ; val: intero
  (lambda (n k)        ; n ≥ 0, k ≥ 1 interi
    (seq-rec n k 0)
    ))

(define seq-rec
  (lambda (n k j)        ; j: eccedenza di 0 a sinistra (occorrenze di 0 meno occorrenze di 1)
    (cond ((= n 0)
           ..........................................................................................................)
          ((= j (- k))
           (map ..........................................................................................
                (seq-rec (- n 1) k (+ j 1)) ))
          ((= j k)
           ..........................................................................................
           ..........................................................................................
          (else
           (append
            (map ..........................................................................................
                 (seq-rec (- n 1) k (+ j 1)) )
            ..........................................................................................
            ..........................................................................................
            ))
          )))
```

4. Verifica formale della correttezza dei programmi ricorsivi

Considera la procedura `f` riportata qui a lato, il cui argomento *b* è una stringa non vuota costituita esclusivamente dalle cifre `0` e `1` e il cui primo carattere (quello più a sinistra) deve essere `1`.

Per ogni stringa *t* di lunghezza $2k$ composta da *k* coppie `10` si può dimostrare che:

`(f t)` → $(4^k - 1) / 3$ (*)

```
(define f       ; val: intero
  (lambda (b)   ; b: stringa di 0/1 che inizia con 1
    (let ( (k (- (string-length b) 1)) )
      (cond ((string=? b "1")
             1)
            ((char=? (string-ref b k) #\0)
             (- (* 2 (f (substring b 0 k))) 1))
            (else
             (+ (* 2 (f (substring b 0 k))) 1))
      )) ))
```

(In altri termini *t* potrà essere: `"10"`, `"1010"`, `"101010"`, `"10101010"`, `"1010 ... 10"`.)

Dimostra la proprietà (*) per induzione su *k*, attenendoti allo schema delineato qui sotto.

- Formalizza la proprietà che esprime il caso / i casi base:

- Ipotesi induttiva: scelto $k > 0$ intero, per t' = `"1010 ... 10"` composto da *k* ripetizioni di `"10"` si assume che:

  `(f t')` → $(4^k - 1) / 3$

- Formalizza la proprietà da dimostrare come passo induttivo:

- Dimostra caso/i base e passo induttivo: