Corso di Programmazione

III Accertamento del 22 Giugno 2007 / A

cognome e nome

Risolvi i seguenti esercizi giustificando sinteticamente le risposte.

1. Programmi in Java

Traduci la seguente procedura Scheme in un corrispondente metodo statico formalizzato nel linguaggio *Java*:

```
(define s
  (lambda (i j)        ; i, j valori naturali
    (cond ((= i 0) 1)
          ((= j 0) 0)
          (else (+ (s i (- j 1)) (* j (s (- i 1) j))))
    )))
```

2. Memoization

Trasforma il programma in *Java* che risolve l'esercizio 1 applicando opportunamente la tecnica di *memoization*.

3. Asserzioni e invarianti

Questo esercizio fa riferimento alla classe *PriorityQueue* discussa a lezione. Il metodo *add* è corretto se ogni sua esecuzione conserva l'invariante di classe: $(0 \le n \le M) \wedge (\forall i \in [2,n] \,.\, q[\lfloor i/2 \rfloor] \ge q[i])$, dove n è il numero di elementi presenti nella coda, limitato da M, e q l'array in cui sono rappresentati (secondo lo schema dello *heap*). Qui sotto è riportata una versione di *add* leggermente semplificata, ma intercambiabile con quella vista a lezione. Immaginando di impostare la verifica della correttezza del codice di *add*, riporta precondizioni, postcondizioni, invarianti di ciclo, funzione di terminazione e opportune asserzioni in corrispondenza agli spazi introdotti dalle keyword *require*, *invariant*, *variant*, *check* e *ensure*. (La dimostrazione non è invece richiesta.) A tua scelta, puoi formalizzare le asserzioni nel linguaggio *Jass* oppure utilizzando la consueta notazione logico-matematica.

```
public void add( int x ) {
  /**  require  ..........
          ..........
  **/
  n = n + 1;
  int j = n, i = j/2;
  while ( (i > 0) && (x > q[i]) )
    /**  invariant  ..........
            ..........
            ..........
    **/
    /**  variant  ..........  **/ {
    q[j] = q[i];
    j = i; i = j/2;
  }
  /**  check  ..........
          ..........
  **/
  q[j] = x;
  /**  ensure  ..........
          ..........
  **/
}
```

4. Oggetti in Java

Una coda con priorità può essere applicata per ordinare un array in modo efficiente. Definisci un metodo statico *sort*, con l'intestazione riportata sotto, per ordinare in ordine *decrescente* (non strettamente) un array u di interi utilizzando un'istanza di *PriorityQueue*. Sono a disposizione il costruttore e i metodi del protocollo di *PriorityQueue*, specificamente: *size*, *max*, *add*, *delMax*, ma non è consentito intervenire sul codice della classe *PriorityQueue*.

```
public static void sort( int[] u ) {

  ...   ...
}
```

Corso di Programmazione

III Accertamento del 22 Giugno 2007 / B

cognome e nome

Risolvi i seguenti esercizi giustificando sinteticamente le risposte.

1. Programmi in Java

Traduci la seguente procedura Scheme in un corrispondente metodo statico formalizzato nel linguaggio *Java*:

```
(define s
  (lambda (i j)        ; i, j >= 0
    (cond ((= j 0) 1)
          ((= i 0) 0)
          (else (+ (* i (s i (- j 1))) (s (- i 1) j)))
    )))
```

2. Memoization

Trasforma il programma in *Java* che risolve l'esercizio 1 applicando opportunamente la tecnica di *memoization*.

3. Asserzioni e invarianti

Questo esercizio fa riferimento alla classe *PriorityQueue* discussa a lezione. Il metodo *add* è corretto se ogni sua esecuzione conserva l'invariante di classe: $(0 \le n \le M) \wedge (\forall i \in [2,n] \,.\, q[\lfloor i/2 \rfloor] \ge q[i])$, dove n è il numero di elementi presenti nella coda, limitato da M, e q l'array in cui sono rappresentati (secondo lo schema dello *heap*). Qui sotto è riportata una versione di *add* leggermente semplificata, ma intercambiabile con quella vista a lezione. Immaginando di impostare la verifica della correttezza del codice di *add*, riporta precondizioni, postcondizioni, invarianti di ciclo, funzione di terminazione e opportune asserzioni in corrispondenza agli spazi introdotti dalle keyword *require*, *invariant*, *variant*, *check* e *ensure*. (La dimostrazione non è invece richiesta.) A tua scelta, puoi formalizzare le asserzioni nel linguaggio *Jass* oppure utilizzando la consueta notazione logico-matematica.

```
public void add( int x ) {
  /**  require  ..........................................................
       ..........................................................
  **/
  n = n + 1;
  int j = n, i = j/2;
  while ( (i > 0) && (x > q[i]) )
    /**  invariant  ..........................................................
         ..........................................................
         ..........................................................
    **/
    /**  variant  ..........................................................  **/ {
    q[j] = q[i];
    j = i; i = j/2;
  }
  /**  check  ..........................................................
       ..........................................................
  **/
  q[j] = x;
  /**  ensure  ..........................................................
       ..........................................................
  **/
}
```

4. Oggetti in Java

Una coda con priorità può essere applicata per ordinare un array in modo efficiente. Definisci un metodo statico *sort*, con l'intestazione riportata sotto, per ordinare in ordine *crescente* (non strettamente) un array *v* di interi utilizzando un'istanza di *PriorityQueue*. Sono a disposizione il costruttore e i metodi del protocollo di *PriorityQueue*, specificamente: *size*, *max*, *add*, *delMax*, ma non è consentito intervenire sul codice della classe *PriorityQueue*.

```
public static void sort( int[] v ) {

  ...   ...
}
```

1. Programmi in Java / A

```
public static long s( int i, int j ) {

  if ( i == 0 ) {
    return 1;
  } else if ( j == 0 ) {
    return 0;
  } else {
    return s( i, j-1 ) + j * s( i-1, j );
  }
}
```

2. Memoization / A

```
public static long s( int i, int j ) {

  long[][] history = new long[i+1][j+1];
  for ( int x=0; x<=i; x=x+1 ) {
    for ( int y=0; y<=j; y=y+1 ) {
      history[x][y] = UNDEFINED;
  }}
  return sMem( i, j, history );
}

public static long sMem( int i, int j, long[][] history ) {

  if ( history[i][j] == UNDEFINED ) {
    if ( i == 0 ) {
      history[i][j] = 1;
    } else if ( j == 0 ) {
      history[i][j] = 0;
    } else {
      history[i][j] = sMem( i, j-1, history ) + j * sMem( i-1, j, history );
    }
  }
  return history[i][j];
}

public static final long UNDEFINED = -1;
```

3. Asserzioni e invarianti / A

```
public void add( int x ) {
  /**  require     ( n < M ) ;
                   ( forall k : { 2 .. n } # (q[k/2] >= q[k]) ) ;
  **/
  n = n + 1;
  int j = n, i = j/2;
  while ( (i > 0) && (x > q[i]) )
    /**  invariant   ( i == j / 2 ) ;
                     ( (j == n) || (x > q[j]) ) ;
                     ( forall k : { 2 .. n } # (q[k/2] >= q[k]) ) ;
    **/
    /**  variant     j                                              **/ {
    q[j] = q[i];
    j = i; i = j/2;
  }
  /**  check       ( i == j / 2 ) ;
                   ( (j == n) || (x > q[j]) ) ;   ( (i == 0) || (q[i] >= x) ) ;
  **/
  q[j] = x;
  /**  ensure      ( n <= M ) ;
                   ( forall k : { 2 .. n } # (q[k/2] >= q[k]) ) ;
  **/
}
```

4. Oggetti in Java / A

```
public static void sort( int[] u ) {
  PriorityQueue q = new PriorityQueue();
  for ( int i=0; i<u.length; i=i+1 ) {
    q.add( u[i] );
  }
  for ( int i=0; i<u.length; i=i+1 ) {
    u[i] = q.max();
    q.delMax();
  }
}
```

1. Programmi in Java / B

```
public static long s( int i, int j ) {

  if ( j == 0 ) {
    return 1;
  } else if ( i == 0 ) {
    return 0;
  } else {
    return i * s( i, j-1 ) + s( i-1, j );
  }
}
```

2. Memoization / B

```
public static long s( int i, int j ) {

  long[][] history = new long[i+1][j+1];
  for ( int x=0; x<=i; x=x+1 ) {
    for ( int y=0; y<=j; y=y+1 ) {
      history[x][y] = UNDEFINED;
  }}
  return sMem( i, j, history );
}

public static long sMem( int i, int j, long[][] history ) {

  if ( history[i][j] == UNDEFINED ) {
    if ( j == 0 ) {
      history[i][j] = 1;
    } else if ( i == 0 ) {
      history[i][j] = 0;
    } else {
      history[i][j] = i * sMem( i, j-1, history ) + sMem( i-1, j, history );
    }
  }
  return history[i][j];
}

public static final long UNDEFINED = -1;
```

3. Asserzioni e invarianti / B

```
public void add( int x ) {
  /**  require      ( n < M ) ;
                    ( forall k : { 2 .. n } # (q[k/2] >= q[k]) ) ;
  **/
  n = n + 1;
  int j = n, i = j/2;
  while ( (i > 0) && (x > q[i]) )
    /**  invariant  ( i == j / 2 ) ;
                    ( (j == n) || (x > q[j]) ) ;
                    ( forall k : { 2 .. n } # (q[k/2] >= q[k]) ) ;
    **/
    /**  variant    j                                              **/ {
    q[j] = q[i];
    j = i; i = j/2;
  }
  /**  check        ( i == j / 2 ) ;
                    ( (j == n) || (x > q[j]) ) ;   ( (i == 0) || (q[i] >= x) ) ;
  **/
  q[j] = x;
  /**  ensure       ( n <= M ) ;
                    ( forall k : { 2 .. n } # (q[k/2] >= q[k]) ) ;
  **/
}
```

4. Oggetti in Java / B

```
public static void sort( int[] v ) {
  PriorityQueue q = new PriorityQueue();
  for ( int i=0; i<v.length; i=i+1 ) {
    q.add( v[i] );
  }
  for ( int i=v.length-1; i>=0; i=i-1 ) {
    v[i] = q.max();
    q.delMax();
  }
}
```