Politecnico di Milano
Facoltà di Ingegneria dell'Informazione
Dipartimento di Elettronica e Informazione

Esercitazioni di Ingegneria del Software

08: JML – Astrazioni procedurali

1. getInterval

Sia data la classe Interval:

```
public class Interval {
 private float low;
 private float high;
 public float getLow() { … }
 public float getHigh() { … }
}

 public static Interval getInterval (float[] times,
                                     float timePoint) {
  …
 }
}
```

1. getInterval

- Definire in JML il contratto che rispetti le seguenti specifiche:
  - *times* non nullo
  - *times* con valori in ordine strettamente crescente
  - restituisce un oggetto di tipo *Interval* che corrisponde a un intervallo temporale avente come estremi due punti contigui di *times*.
  - *timePoint* deve essere maggiore o uguale all'estremo minore e strettamente minore dell'estremo maggiore

1. getInterval

PRE
- *times* non nullo
- *times* con valori in ordine strettamente crescente

POST
- restituisce un oggetto di tipo *Interval* che corrisponde a un intervallo temporale avente come estremi due punti contigui di *times*.
- *timePoint* deve essere maggiore o uguale all'estremo minore e strettamente minore dell'estremo maggiore

```
public static Interval getInterval (float[] times,
                                    float timePoint)
```

1. getInterval

PRE

```
//@ requires times != null
//@ && (\forall int i; 0 <=i < times.length -1;
//@         times[i] < times[i+1]);
```

POST

- restituisce un oggetto di tipo *Interval* che corrisponde a un intervallo temporale avente come estremi due punti contigui di *times*.
- *timePoint* deve essere maggiore o uguale all'estremo minore e strettamente minore dell'estremo maggiore

```
public static Interval getInterval (float[] times,
                        float timePoint)
```


1. getInterval

PRE

```
//@ requires times != null
//@ && (\forall int i; 0 <= i < times.length -1;
//@         times[i] < times[i+1]);
```

POST

```
//@ ensures (\exists int i; 0 <= i < times.length -1;
//@ times[i]==\result.getLow() &&
    times[i+1]==\result.getHigh())
```

- *timePoint* deve essere maggiore o uguale all'estremo minore e strettamente minore dell'estremo maggiore

```
public static Interval getInterval (float[] times,
                        float timePoint)
```

1. getInterval

PRE

```
//@ requires times != null
//@ && (\forall int i; 0 <= i < times.length -1;
//@         times[i] < times[i+1]);
```

POST

```
//@ ensures (\exists int i; 0 <= i < times.length -1;
//@ times[i]==\result.getLow() &&
   times[i+1]==\result.getHigh())
//@ && timePoint >= \result.low
//@ && timePoint < \result.high;
```

```
public static Interval getInterval (float[] times,
                        float timePoint)
```


2. subString

- Dato il seguente metodo:

```
public static boolean subString (char[] text,char[] chunk)
```

scrivere la specifica che renda l'operazione sensata, in modo che il metodo si comporti nel seguente modo:

text = [abbcdddeff], chunk = [bcddd] → true

text = [abbcdddeff], chunk = [baaddd] → false

2. subString

- Dall'esempio, si deduce che se chunk è contenuto in text il risultato è true; false altrimenti
- In altre parole: se esiste una posizione in text a partire dalla quale tutti i caratteri corrispondano nell'ordine a quelli di chunk, per tutta la lunghezza di chunk, il risultato è true (false altrimenti)

2. subString: soluzione

- PRECONDIZIONI

```
//@ requires text != null && chunk != null &&
//@  chunk.length <= text.lenght;
```

- POSTCONDIZIONI

```
//@ ensures \result == true <==>
//@   (\exists int i; 0<= i<= text.length - chunk.length;
//@     (\forall int j; 0 <= j < chunk.length;
//@      text[i+j] == chunk[j]));
```

3. computeScore

```
public static int computeScore (int P1, int P2, int L)
```

P1 = primo compitino; P2 = secondo compitino; L = laboratorio

Ogni prova intermedia assegna fino a 13 punti. Lo studente deve prendere almeno 6 punti in ognuna, altrimenti deve recuperare entrambe le prove.

Il laboratorio assegna fino a 4 punti. Lo studente deve prenderne almeno 2 punti, pena la ripetizione dell'intero corso l'anno successivo.

Per superare l'esame, la somma di P1 e P2 deve essere almeno 16, ed il totale con il laboratorio deve essere almeno 18, pena il recupero.

Il metodo restituisce il voto, oppure 0 in caso di recupero, oppure -1 in caso di ripetizione del corso.

3. computeScore : soluzione

```
/*@ requires
  @  0<=P1<=13 && 0<=P2<=13 && 0<=L<=4;
  @ ensures
  @  (\result == -1) <==> (L < 2)
  @   && (\result == 0) <==>
  @  (L >= 2)&&(P1<6 || P2<6 || P1+P2<16))
  @  && (\result == P1+P2+L <==>
  @  (L >= 2)&&(P1>=6 && P2>=6 || P1+P2>=16);
  @*/
public static int computeScore (int P1, int P2, int L)
```

4. isPermutation

```
public static boolean isPermutation(int x[], int y[])
```

True se y è una permutazione di x, false altrimenti

Sotto l'ipotesi di assenza di duplicati nei due array

4. isPermutation : soluzione

```
/*@ requires x != null && y != null &&
  @(\forall int i; 0 <= i < x.length -1;
      (\forall int j; i <= j < x.length; x[i] != x[j]))
  @ && (* same for y *);
  @ ensures (\result == true) <==>
  @    (x.length == y.length) &&
  @(\forall int i; 0 <= i < x.length;
  @   (\exists int j; 0 <= j < y.length; x[i] == y[j]));
  @*/
public static boolean isPermutation(int x[], int y[])
```

4. isPermutation (con duplicati)

```
public static boolean isPermutation(int[] x, int[] y)
```

True se y è una permutazione di x, false altrimenti

Rimuovere l'ipotesi di assenza di duplicati nei due array

4. isPermutation (con duplicati) : soluzione

```
/*@ requires x != null && y != null;
  @ ensures (\result == true) <==>
  @    (x.length  = y.length) &&
  @(\forall int i; 0 <= i < x.length;
  @   (\numof int j; 0 <= j < x.length; x[i] == x[j]))
  @== (\numof int k; 0 <= k < y.length; x[i] == y[k]));
  @*/
public static boolean isPermutation(int[] x, int[] y)
```

5. highLowNums

```
public static void highLowNums(int[] nums, int[] highs, int n)
```

Nums non contiene duplicati

Highs è lungo esattamente *n*

Il metodo trova gli *n* numeri più grandi in *nums* e li inserisce in *highs* in ordine decrescente. *Nums* non viene modificato.

5. highLowNums: soluzione (1)

```
/*@ assignable highs[*];
  @ requires nums != null && highs  != null && highs.length == n
  @ && nums.length >= n
  @ && (\forall int i; 0<=i<nums.length-1;
  @   !(\exists int j; i<j<nums.length; nums[i] == nums[j]));
  @ ensures (* highs contiene gli n numeri più grandi di nums *)
  @ && (\forall int i; 0<=i<n-1; highs[i]>=highs[i+1])
  @*/
public static void highLowNums(int[] nums, int[] highs, int n)
```

5. highLowNums: soluzione (2)

```
/*@ assignable highs[*];
  @ requires nums != null && highs  != null && highs.length == n
  @ && nums.length >= n
  @ && (\forall int i; 0<=i<nums.length-1;
  @   !(\exists int j; i<j<nums.length; nums[i] == nums[j]));
  @ ensures (* per ogni numero in highs esiste il corrispondente
   in nums ed esistono in nums esattamente "posizione_in_highs"
   numeri maggiori di esso *)
  @ && (\forall int i; 0<=i<n-1; highs[i]>=highs[i+1])
  @*/
public static void highLowNums(int[] nums, int[] highs, int n)
```


5. highLowNums: soluzione (3)

```
/*@ assignable highs[*];
  @ requires nums != null && highs  != null && highs.length == n
  @ && nums.length >= n
  @ && (\forall int i; 0<=i<nums.length-1;
  @   !(\exists int j; i<j<nums.length; nums[i] == nums[j]));
  @ ensures
  @  (\forall int i; 0<=i<n;
  @    (\exists int j; 0<=j<nums.length; highs[i] == nums[j])
  @ && (\numof int k; 0<=k<nums.length; nums[k] > highs[i])
                                                    == i);
  @ && (\forall int i; 0<=i<n-1; highs[i]>=highs[i+1])
  @*/
public static void highLowNums(int[] nums, int[] highs, int n)
```

Ridondante: la condizione sopra fa già sì che i numeri in highs siano ordinati

6. interval

- Dato un array di valori numerici interi senza duplicati, il metodo *interval* produce l'insieme di valori che ricadono all'interno dell'intervallo chiuso [a, b].
- Si supponga definito il tipo di dato astratto *Set* che fornisca il metodo *boolean isIn(int n)*

```
public static Set interval(int[] x, int a, int b)
```


6. Interval : soluzione (1)

```
/*@ requires x != null && a <= b
  @ && (\forall int i; 0 <= i < x.length-1;
  @    !(\exists int j; i < j <x.length; nums[i] == nums[j]));
  @ ensure (* il risultato contiene solo numeri di x compresi tra
                          a e b *)
  @*/
public static Set interval(int[] x, int a, int b)
```

6. Interval : soluzione (2)

```
/*@ requires x != null && a <= b
  @ && (\forall int i; 0 <= i < x.length-1;
  @   !(\exists int j; i < j <x.length; nums[i] == nums[j]));
  @ ensure (* ogni numero di x compreso tra a e b appartiene al
    risultato e il risultato non contiene altri numeri *)
  @*/
public static Set interval(int[] x, int a, int b)
```


6. Interval : soluzione (3)

```
/*@ requires x != null && a <= b
  @ && (\forall int i; 0 <= i < x.length-1;
  @   !(\exists int j; i < j <x.length; nums[i] == nums[j]));
  @ ensure (\forall int i; 0 <= i < x.length;
  @    \result.isIn(x[i]) <==> (a <= x[i] <= b));
  @*/
public static Set interval(int[] x, int a, int b)
```