Ingegneria del Software

Esercitazione 7

Esercizio 4

```
public static boolean sottoStringa (char[] testo, char[]
parola);
```

- Scrivere la specifica che renda l'operazione sensata, in modo che il metodo si comporti nel seguente modo:
  - testo = [abbcdddeff], parola = [bcddd] -> true
  - testo = [abbcdddeff], parola = [baaddd] -> false
- Dall'esempio, si deduce che se parola è contenuto in testo il risultato è true; false altrimenti
- In altre parole: se esiste una posizione in testo a partire dalla quale tutti i caratteri corrispondano nell'ordine a quelli di parola, per tutta la lunghezza di parola, il risultato è true (false altrimenti)

Esercizio 4

```
/*@
@ requires testo!=null && parola!=null &&
@                          length(testo)>=length(parola);
@ assignable \nothing
@ ensures \result <==>
@     (\exists int i; 0<=i && i<testo.length-
parola.length;
@         (\forall int j; 0 <= j && j<parola.length;
           testo[i+j] == parola[j])
@      );
@ */
boolean sottoStringa(char[] testo, char[] parola);
```

Esercizio 5

```
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.

Esercizio 5

```
/*@ 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)
```

Esercizio 6

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

- L’array nums contiene interi tutti diversi tra di loro
- L’array highs è lungo esattamente n
- Il metodo trova gli n numeri interi più grandi di nums e li inserisce in ordine decrescente nell’array highs
- L’array nums non viene modificato

Esercizio 6

```
/*
@ 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]);
@
*/
```

Esercizio 6

```
/*@
@ 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]);
@*/
```

Esercizio 6

```
/*
@ 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]);
@*/
```

Esercizio 7

```
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

Esercizio 7

```
/*
@ 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[])
```

Esercizio 7bis

```
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

Esercizio 7bis

```
/*
@ 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[])
```

Esercizio 8

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

- 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)**

Esercizio 8

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

Esercizio 8

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

Specifiche totali

Esercizio 9

Si consideri il metodo:

```
//@ requires in >= 0;
//@ ensures Math.abs(\result*\result - in) < 0.0001;
public static float sqrt(float in)
```

E i seguenti usi:

- `float zero = sqrt(0);`
- `float uno = sqrt(1);`
- `float due = sqrt(4);`
- `float tre = sqrt(9);`
- `float boh = sqrt(-1);`

Visto che non stiamo rispettando le precondizioni, il comportamento è impredicibile

Esercizio 9

Come possiamo creare sistemi Robusti?
Il comportamento è predicibile anche nei casi in cui i patti non siano stati rispettati. Si possono:

- togliere le precondizioni

```
//@ requires true
```

- evidenziare i casi problematici

```
//@ ensures in >= 0 && (* altre postcondizioni *)
```

- sollevando le opportune eccezioni

```
//@ signals (NegativeException e) in < 0;
```

Esercizio 9

Specifica con pre e post condizioni

```
//@ requires in >= 0;
//@ ensures Math.abs(\result*\result-in)<0.0001;
public static float sqrt(float in)
```

Specifica totale

```
//@ requires true;
//@ ensures in >= 0 && Math.abs(\result*\result-
in)<0.0001;
//@ signals(NegativeException ne) in < 0;
public static float sqrt(float in)
```

Esercizio 10

- Rendere specifiche totali 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

Con specifica parziale...

```
/*@@ assignable \nothing
@ requires times != null && times.length >= 2 &&
@ timePoint >= times[0] && timePoint < times[times.length -1]
@ && (\forall int i; 0<=i && i<times.length-1;times[i]<times[i+1]);
@ ensures (\exists int i; 0<=i<times.length-1;
@                         times[i] == \result.getLowerBound() &&
@                         times[i+1] == \result.getUpperBound()) &&
@  \result.getLowerBound() <= timePoint &&
@  \result.getUpperBound() > timePoint;
@*/
Interval getInterval(float[] times, float timePoint);
```

Resa totale

```
/*@@ assignable \nothing
@ requires true
@ ensures times != null && times.length >= 2 &&
@ timePoint >= times[0] && timePoint < times[times.length -1]
@ && (\forall int i; 0<=i && i<times.length-1;times[i]<times[i+1])
@ && (\exists int i; 0<=i<times.length-1;
@                           times[i] == \result.getLowerBound() &&
@                           times[i+1] == \result.getUpperBound()) &&
@ \result.getLowerBound()<=timePoint && \result.getUpperBound() > timePoint
@ signals (NullPointerException npe) times == null;
@ signals (UnorderedArrayException uae)
@     !(\forall int i; 0 <= i < times.length -1; @ times[i] < times[i+1]);
@ signals (TooShortDataException tsde)
@     times.length < 2;
@ signals (OutOfBoundException oobe)
@     timePoint < times[0] && timePoint >= times[times.length -1];
@*/
Interval getInterval(float[] times, float timePoint);
```

Esercizio 11

Rendere totali le specifiche di:

```
public static boolean sottoStringa (char[] testo, char[] parola);
```

- Scrivere la specifica che renda l'operazione sensata, in modo che il metodo si comporti nel seguente modo:
  - testo = [abbcdddeff], parola = [bcddd] -> true
  - testo = [abbcdddeff], parola = [baaddd] -> false
- Dall'esempio, si deduce che se parola è contenuto in testo il risultato è true; false altrimenti
- In altre parole: se esiste una posizione in testo a partire dalla quale tutti i caratteri corrispondano nell'ordine a quelli di parola, per tutta la lunghezza di parola, il risultato è true (false altrimenti)

Parziali:

```
/*@@ requires testo!=null && parola!=null &&
@                           length(testo)>=length(parola);
@ assignable \nothing
@ ensures \result <==>
@     (\exists int i; 0<=i && i<testo.length-parola.length;
@        (\forall int j; 0 <= j && j<parola.length;
testo[i+j] == parola[j])
@      );
@ */
boolean sottoStringa(char[] testo, char[] parola);
```

Totali:

```
/*@@ requires true
@ assignable \nothing
@ ensures testo != null && parola!= null  &&
@                          length(testo)>=length(parola) &&
@  \result <==>
@     (\exists int i; 0<=i && i<testo.length-parola.length;
@        (\forall int j; 0 <= j && j<parola.length; testo[i+j] ==
parola[j])
@      );
@ signals (NullPointerException npe) testo==null || parola==null;
@ signals (InvertedDimensionException ide)
length(testo)<length(parola);
@ */
boolean sottoStringa(char[] testo, char[] parola);
```

Esercizio 12

Rendere totali le specifiche di:

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

Parziali:

```
/*@ 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[])
```

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

Sostituibilità

Esercizio 13

```
public class Math {
  //@ requires value >= 0;
  //@ ensures Math.abs(\result * \result - value) < 0.01;
  public double sqrt(double value) { … }
}
```

- Mettiamoci nell'ottica degli utenti:
  - Non useremo mai un parametro attuale negativo, in quanto le precondizioni del metodo sarebbero violate.
  - Se le precondizioni non sono soddisfatte il metodo è infatti libero di fare quello che vuole (il contratto è nullo)
  - Se stiamo rispettando i patti (value >= 0), ci viene garantito che il risultato è la radice quadrata

```
public class SafeMath extends Math {
  //@  also
  //@  requires value < 0;
  //@  ensures  \result = -1;
  public double sqrt(double value) {…}
}
```

La classe SafeMath è un’estensione valida di Math?

La classe SafeMath è un'estensione valida di Math?

- Regola delle signature: sì, sqrt viene sovrascritta
- Regola dei metodi: le chiamate a sqrt di SafeMath si comportano come le chiamate a sqrt di Math
- Regola delle proprietà: tutti gli invarianti pubblici sono rispettati

Mettiamoci nei panni di un utente di Math: sarà sorpreso se gli viene passata un'istanza di tipo SafeMath?

- Per quanto ne sa lui, non è possibile calcolare la radice quadrata di un numero negativo, quindi non eserciterà mai le nuove funzionalità… di fatto per lui Math e SafeMath sono equivalenti

Ricaviamo il contratto complessivo di SafeMath.sqrt

Le precondizioni si ottengono mettendo in or quelle di SafeMath e quelle di Math:

```
//@ requires (value >= 0) || (value < 0)
```

Le postcondizioni si ottengono con la seguente formula:
(presup => postsup) && (presott => postsott)

```
//@ ensures (value >= 0 => Math.abs(...))
//@ && (value < 0  => \result = -1 )
```

Esercizio 14

Supponiamo che esista un tipo di dato astratto CD con un metodo getArtist(), che restituisce l'artista di un dato CD

```
public class Adviser {
  //@ requires artist != null;
  //@ ensures \result.getArtist().equals(artist);
  public CD consiglia(String artist) { … }
}
```

La seguente classe rispetta i principi di sostituibilità?

```
public class SmartAdviser extends Adviser {
  //@ also
 //@ requires artist != null;
 //@  ensures !(\exist CD c; isIn(c); c.giudizio >
\result.giudizio);
  public CD consiglia(String artist) { … }
}
```

La classe SmartAdviser è un'estensione valida di Adviser?

- Regola delle signature: sì, consiglia viene sovrascritta
- Regola dei metodi: le chiamate a "consiglia" di SmartAdviser si comportano come le chiamate a consiglia di Adviser
- Regola delle proprietà: tutti gli invarianti pubblici sono rispettati

Mettiamoci nei panni di un utente di Adviser : sarà sorpreso se gli viene passata un'istanza di tipo SmartAdviser?

- Il contratto indica che viene restituito un cd di un autore (a patto che questo non sia nullo)… SmartAdviser rispetta questo contratto (anzi, restituisce il miglior cd di quell'artista)

Ricaviamo il contratto complessivo
Le precondizioni si ottengono mettendo in or quelle di Adviser e quelle di SmartAdviser:

```
//@ requires (artist != null) || (artist != null)
```

Ovvero:

```
//@ requires (artist != null)
```

Le postcondizioni si ottengono con la seguente formula:
(presup => postsup) && (presott => postsott)

```
//@ ensures (artist != null => \result.getArtist().equals(artist) ) &&
//@               (artist != null => !(\exist CD c; isIn(c); c.giudizio > \result.giudizio) );
```

Ovvero:

```
//@ ensures artist != null => ( \result.getArtist().equals(artist) &&
//@                                !(\exist CD c; isIn(c); c.giudizio > \result.giudizio) );
```

Testing

Esercizio

- Il metodo statico suGiuOgiuSu riceve come parametro un array *a* contenente tre numeri interi e restituisce al chiamante un valore di tipo *boolean*; se a[0]>a[1] allora se a[1]<a[2] suGiuOgiuSu restituisce true, altrimenti restituisce false; se a[0]<a[1] allora se a[1]>a[2] restituisce true, altrimenti false.

Quesito 1

- Considerando il paragrafo precedente come una specifica del metodo suGiuOgiuSu *evidenziare un suo difetto* particolarmente rilevante, motivando adeguatamente la risposta.

Soluzione 1

- **La specifica è incompleta, perché non è definito il valore restituito nel caso in cui a[0] è uguale ad a[1].**

Quesito 2

- Sempre considerando il primo paragrafo come una specifica del metodo, *fornire 4 casi di test funzionali* per tale metodo, scegliendoli, e motivando tale scelta.

Soluzione 2

- **Si possono scegliere 4 casi che rendano vere o false le due proposizioni a[0]>a[1] e a[1]>a[2]. Per esempio, si possono quindi ottenere**

a[0]	a[1]	a[2]	Val. Restituito
1	2	1	true
1	2	3	false
2	1	3	true
2	1	0	false

Quesito 3

- Si consideri la seguente implementazione del metodo, non necessariamente corretta rispetto alla specifica.

```
static boolean suGiuOgiuSu (int [] a) {
        if ( a[0]>a[1] && a[1]<a[2] )
                return true;
        else
                if ( a[0]<a[1] )
                        if ( a[1]>a[2] )
                                return true;
                        else return false;
                else return false;
}
```

- Indicare, motivando la risposta, *quanti dati di test occorrono* per effettuare un test strutturale che copra tutte le diramazioni. Fornire un esempio di tali dati di test.

Soluzione 3

- **Bastano 4 dati di test, che corrispondono ai 4 cammini che portano alle 4 istruzioni return.**

1) a[0]>a[1] e a[1]<a[2] e.g. a[0]=2, a[1]=1, a[2]=3

2) a[0]<a[1] e a[1]>a[2] e.g. a[0]=2, a[1]=3, a[2]=1

3) a[0]<a[1] e a[1]<=a[2] e.g. a[0]=2, a[1]=3, a[2]=4

4) a[0]>a[1] e a[1]>=a[2] e.g. a[0]=3, a[1]=2, a[2]=1

Quesito 4

- Per ottenere la copertura rispetto al criterio delle condizioni *è necessario aggiungere altri dati di test oltre a quelli indicati* al punto (3)? Se sì, indicare quali, se no, spiegare perché.

Soluzione 4

- **No, perché quelli sopra riportati permettono di rendere vere e false ognuna delle tre condizioni delle istruzioni if e anche ognuna delle due componenti della prima di esse (a[0]>a[1] e a[1]<a[2]).**

Quesito 5

- Definire *un'eccezione TuttiUgualiException* di tipo checked, che possieda solo due costruttori, uno senza argomenti e uno con un argomento di tipo stringa. *Modificare il codice del metodo suGiuOgiuSu* in modo che esso lanci un'eccezione *TuttiUgualiException* nel caso in cui i tre elementi dell'array siano tutti uguali.

Soluzione 5

```
class TuttiUgualiException extends Exception {
    public TuttiUgualiException () { super(); }
    public TuttiUgualiException (String s) { super(s); }
}

static boolean suGiuOgiuSu (int [] a) throws TuttiUgualiException {
    if (a[0]==a[1] && a[1]==a[2])
        throw new TuttiUgualiException ("suGiuOgiuSu ");
    if ( a[0]>a[1] && a[1]<a[2] )
        return true;
    else
        if ( a[0]<a[1] )
            if ( a[1]>a[2] )
                return true;
            else return false;
        else return false;
}
```

Esempio

- Si consideri il seguente frammento di programma. Quanti casi di test sono necessari per coprire tutte le diramazioni (decisioni) e quanti sono necessari per coprire tutti i cammini? Motivare sinteticamente la risposta.

```
if (condizione_1) istruz_1;
else
   if (condizione_2) istruz_2;
   else istruz_3;
if (condizione_2) istruz_4;
else istruz_5;
```

Soluzione

- **Risposta: occorre aggiungere ulteriori ipotesi sulle condizioni e sulle istruzioni per poter determinare il numero necessario (ossia minimo) di cammini. Supponiamo allora che le due condizioni logiche siano indipendenti tra di loro (ossia, che possano essere verificate e falsificate indipendentemente l'una dall'altra) e atomiche, che le istruzioni non siano return, e che non modifichino condizione_2. Con queste assunzioni un test con tre casi è sufficiente: ad esempio un test nel quale il primo caso fa si che condizione_1 == true e condizione_2 == true (primo if, ramo then; terzo if, ramo then), il secondo fa si che condizione_1 == false e condizione_2 == true (primo if, ramo else; secondo e terzo if, ramo then), e il terzo fa si che condizione_1 == false e condizione_2 == false (primo if, ramo else; secondo e terzo if, ramo else). I cammini percorribili sono 4 (occorre aggiungere il caso: primo if, ramo then; terzo if, ramo else, gli altri cammini sul grafo di flusso non sono percorribili). Pertanto occorre aggiungere il caso di test con condizione_1 == true e condizione_2 == false.**

Esercizio

Si consideri la specifica del seguente metodo:

```
static int triangolo (int a, int b, int c) {
   // REQUIRES: a>0 and b>0 and c>0
   // ENSURES: restituisce 1 se i tre lati definiscono
   // un triangolo equilatero,
   // 2 se definiscono un triangolo isoscele,
   // 3 se definiscono un triangolo scaleno,
   // 0 se a, b, e c non possono essere lati di un
   triangolo
}
```

Quesito 1

- **Si definisca un insieme di dati di test in base a una strategia di tipo “black box” (cioè funzionale), motivando sinteticamente ciascuna scelta.**

Soluzione 1

- I dati <5, 5, 5>, <5, 7, 7>, <5, 6, 7> e <5, 6, 12> corrispondono alle quattro parti della clausola ENSURES e per essi il metodo deve restituire, rispettivamente, i valori 1, 2, 3 e 0. Si potrebbero anche considerare i tre casi in cui un triangolo può essere isoscele, e quindi anche (7, 5, 7) e (5, 7, 7), nonché il caso degenere in cui un lato è la somma degli altri due (non è infatti chiaro se questo debba essere considerato un triangolo oppure no). Si potrebbe anche considerare il valore 1 come un valore di confine per i lati, e quindi aggiungere almeno un test che considera un lato di lunghezza 1. Come si vede da queste considerazioni, il test black box è suscettibile di interpretazione individuale, in quanto le specifiche sono spesso espresse in maniera informale.

Quesito 2

- Si consideri la seguente implementazione (che non tiene conto dell'ultima parte della clausola ENSURES) del metodo precedente:

```
static int triangolo (int a, int b, int c) {
    if ((a==b) && (b==c)) return 1;
    if ((a==b) || (b==c) || (a==c)) return 2;
    else return 3;
}
```

- Si consideri la seguente formulazione di una strategia di tipo "white box"(cioè strutturale): deve esistere, per ciascun cammino percorribile del flusso di controllo, uno e un solo dato di test che ne genera la copertura.
- **Quanti casi di test sono necessari?**
- **Si descrivano i test scelti:**

Soluzione 2

- Servono 3 dati di test. I dati possono essere <5, 5, 5>, <5, 5, 7>, <5, 6, 7>.

Quesito 3

- Si consideri la seguente implementazione alternativa della funzione precedente:

```
static int triangolo (int a, int b, int c) {
    if ((a==b) && (b==c)) return 1;
    else if (a==b) return 2;
    else if (b==c) return 2;
    else if (a==c) return 2;
    else return 3;
}
```

- Considerando sempre la precedente formulazione del criterio di copertura dei cammini:
- **Quanti casi di test sono necessari per questo programma?**
- **Si descrivano i test scelti:**

Soluzione 3

- Servono 5 dati di test.
- I dati possono essere <5, 5, 5>, <5, 5, 7>, <5, 7, 7>, <5, 7, 5>, <5, 6, 7>.

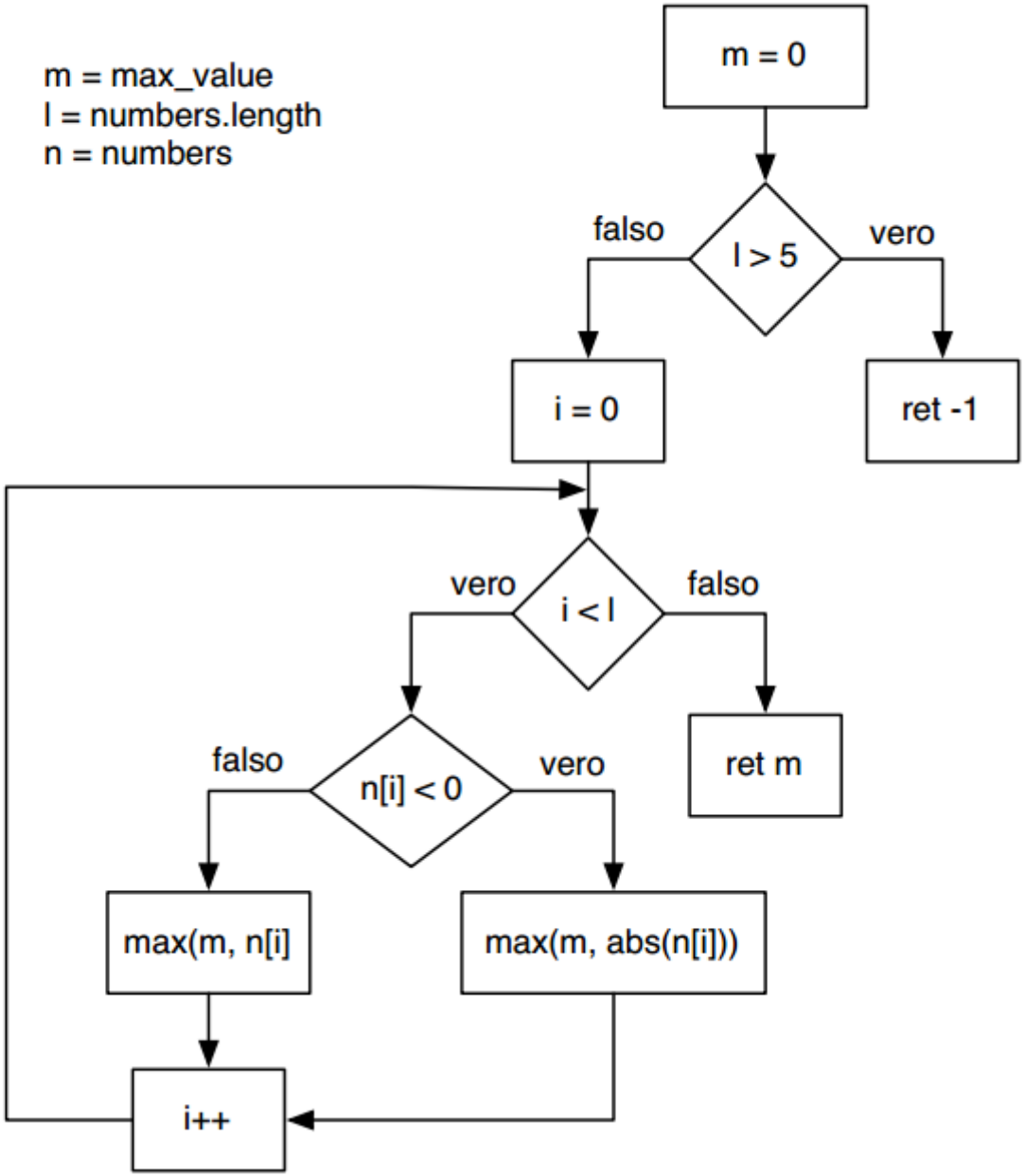
Esercizio (tde 17-7-2013)

Il seguente metodo Java prende in ingresso un array di al più 5 interi e restituisce l'elemento più grande in valore assoluto, oppure 0 se l'array fosse vuoto oppure -1 se troppo grande.

```
public int max(int[] numbers) {
   int i, max_value = 0;
   if (numbers.length > 5) return -1;
   for (i = 0; i<numbers.length; i++) {
      if (numbers[i] < 0)
         max_value = Math.max(max_value,Math.abs(numbers[i]));
      else max_value = Math.max(max_value, numbers[i]);
   }
   return max_value;
}
```

- Disegnare il diagramma del flusso di controllo.

Soluzione:


m = max_value
l = numbers.length
n = numbers
m = 0
l > 5
falso
vero
i = 0
ret -1
i < l
vero
falso
ret m
n[i] < 0
falso
vero
max(m, n[i]
max(m, abs(n[i]))
i++

- Il metodo è collaudato con i casi di test seguenti (input; valore restituito):
  - T1: {0,0,0,0,0}; 0;
  - T2: {1,2,3,4,5}; 5;
  - T3: {-1,-2,-3,-4,-5}; 5;
  - T4: {1,2,3,4,5,6}; -1;
  - T5: {-10,10,3,5,-6}; 10;
  - T6: {}; -1;

1. Definire le percentuali di copertura delle istruzioni (*statement*) e delle decisioni (*branch*) ottenute eseguendo ogni test separatamente.

2. Quale test darebbe errore?

Soluzione:

T	statement (10)	branch/archi (11)
T1	8	8
T2	8	8
T3	8	8
T4	3	2
T5	9	10
T6	5	4

T6: restiuisce 0 e non -1;

Esercizio (tde 5-2-2016)

Si consideri il seguente metodo statico Java:

```
public static int foo(int a, int b) {
        int n = 3;
        if (a == 0 ll b == 0)
                return 1;
        else
                while (a%n != 0)
                        if (a > b)
                                a -= b;
        return b;
}
```

e si definisca un insieme minimo di casi di test che coprano tutte le istruzioni e un insieme di casi di test che copra tutti i branch.
Quali (probabili) errori sono evidenziati dall'insieme di test definito al punto precedente?

Copertura istruzioni
(A=1, B=0), (A=4, B=1)

Copertura branch
(A=1, B=0), (A=4, B=1), (A=4, B=2)

Probabili errori
infinite loop per i valori A=4, B=2

Esercizio (tde 14-9-2015)

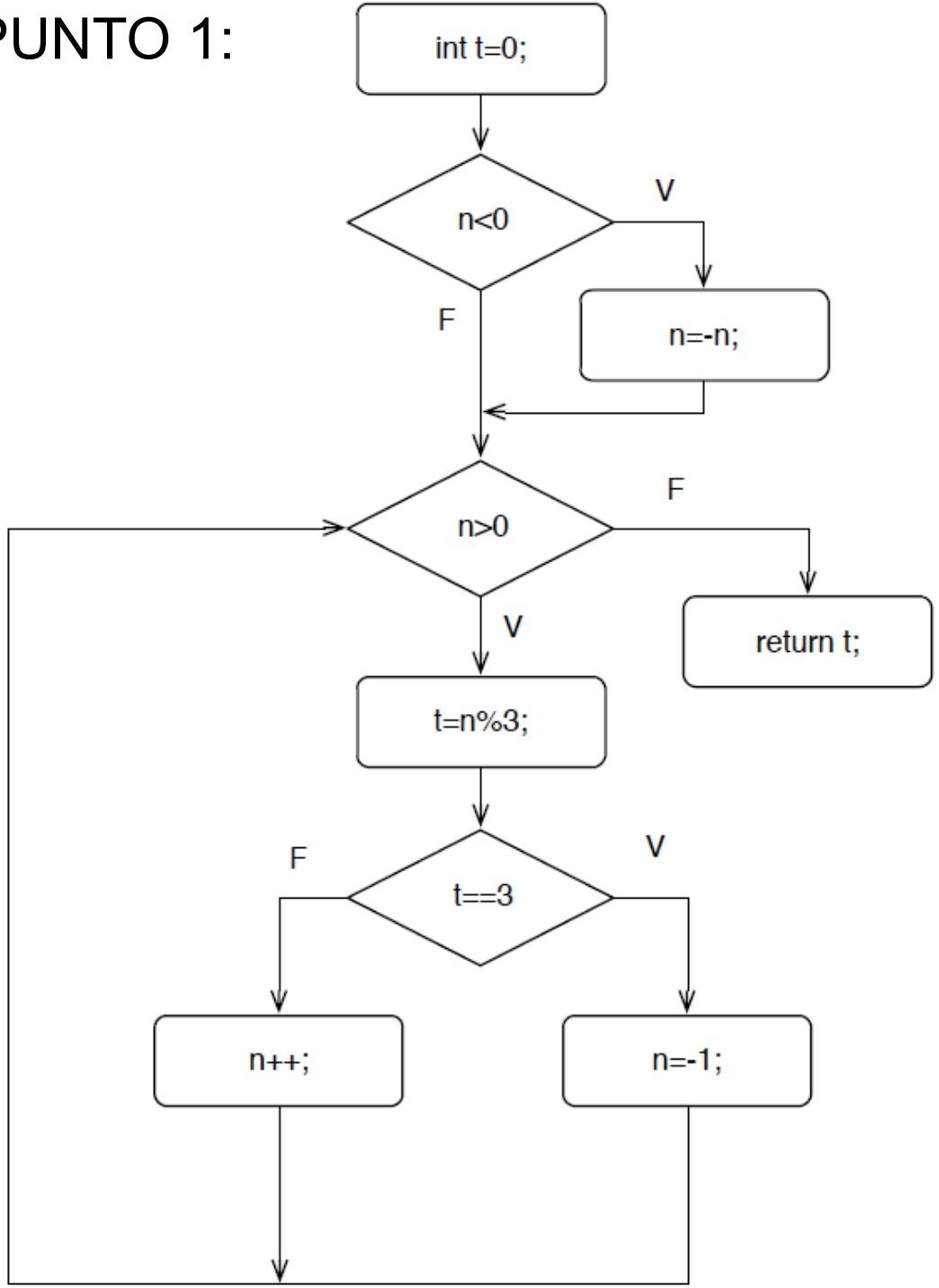
Si consideri il seguente metodo:

```
public static int foo(int n) {
        int t=0;
        if (n < 0)
                n = -n;
        while (n >0) {
                t = n%3;
                if (t == 2)
                        n = -1;
                else
                        n++;
        }
        return t;
}
```

Si definisca

1. Il diagramma del flusso di controllo.
2. Un insieme minimo di test che copra tutte le istruzioni. Nel caso non fosse possibile, definire la copertura (in percentuale) dell'insieme di test definiti.
3. La condizione che n deve rispettare per eseguire il programma senza entrare nel ciclo while. È possibile? Cosa restituirebbe il programma?
4. Si modifichi il test dell'istruzione if da t==3 in t==2 e si calcoli la precondizione che n dovrebbe rispettare affinché si entri nel ciclo due volte e la prima volta si esegua il ramo then mentre la seconda si esegua il ramo else. Calcolare anche un caso di test, se esiste, che soddisfa la precondizione.

SOLUZIONE PUNTO 1:



SOLUZIONE PUNTO 2:
n=-1
Copre tutte le istruzioni tranne il ramo then del secondo if. Infatti, all'ingresso del ciclo while, n=1. Ad ogni iterazione il valore di n viene sempre incrementato finché il valore di n non causa overflow e diviene negativo.
A quel punto si esce dal ciclo while.
Copertura 8/9, 88,88%

SOLUZIONE PUNTO 3:
n=0
restituisce 0

SOLUZIONE PUNTO 4:

$$\begin{cases} n \neq 0 \\ |n|\%3 = 2 \\ -1 > 0 \\ -1\%3 \neq 2 \\ 0 \leq 0 \end{cases}$$

Esercizio (tde 25-9-2015)

Si consideri il seguente metodo (in cui ogni istruzione è numerata):

```
1 static void esegui(int a, int b) {
2           if (b>= 0 && a>0) {
3                       while (b !=0) {
4                                   if (b >= 3)
5                                               a++;
6                                   else a--;
7                                   b--;
                        };
            };
8           return a;
}
```

1. Si scriva la path condition e si sintetizzi un dato di test per percorrere il cammino seguente: 1, 2, 3, 4, 5, 7, 3, 4, 6, 7, 3, 4, 6, 7, 8.
2. Qual’`e il numero minimo di dati di test necessari per coprire tutte le condizioni del programma?
3. Si sintetizzino i dati di test che soddisfano il requisito di cui al punto (2).

SOLUZIONE PUNTO 1:
b=3 && a>0 (es. b=3 && a = 1)

SOLUZIONE PUNTO 2:
Il caso trovato nel punto precedente copre già le condizioni del while e del secondo if (durante le varie iterazioni del ciclo). Basta quindi coprire il caso false del primo if (con due casi di test).

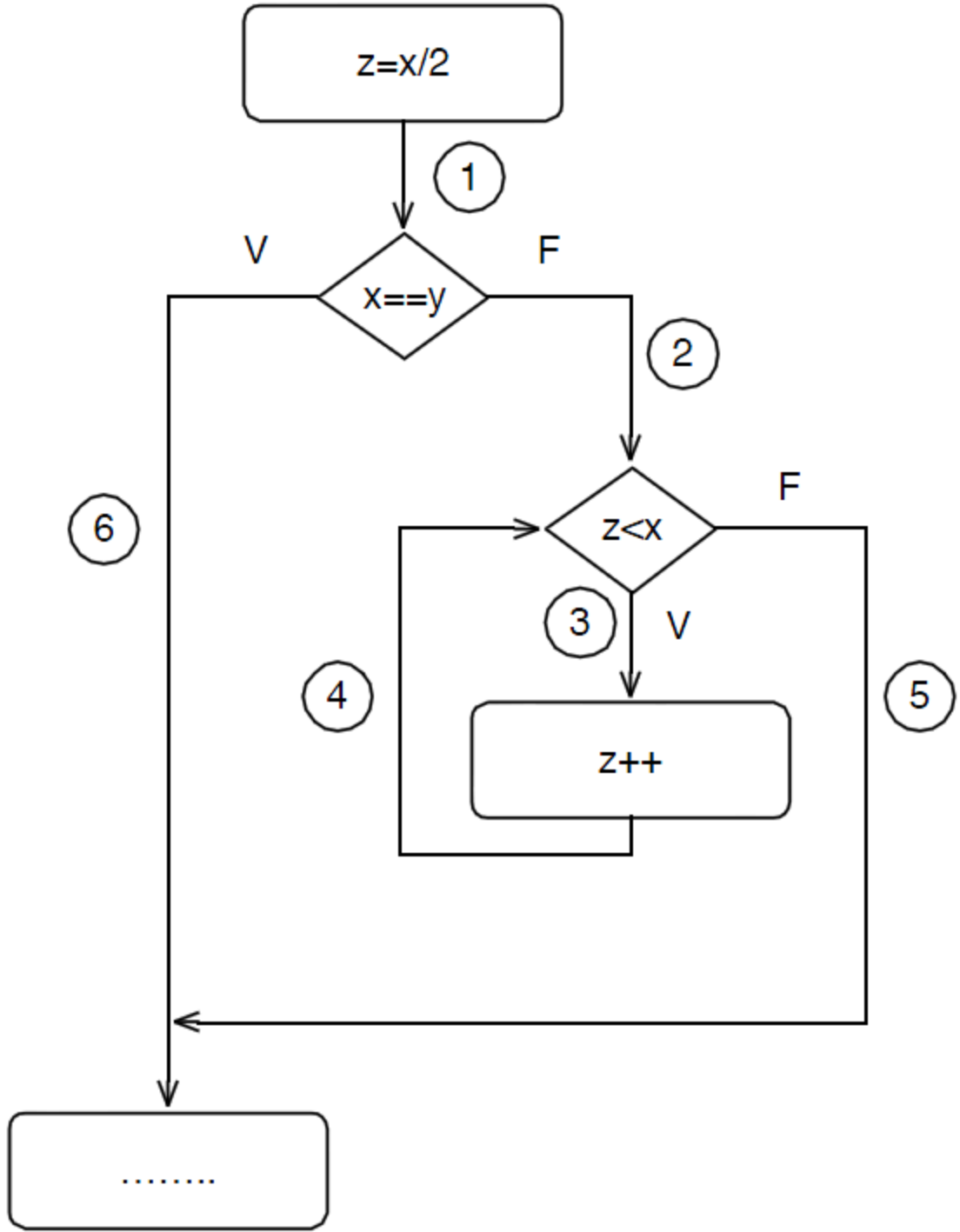
SOLUZIONE PUNTO 3:
I casi di test minimi sono quindi solo 3:
1) b<0 (es- b=-1)
2) b>=0 && a<=0 (es. b=1 && a=0)
3) b=3 && a=1

Esercizio (tde 23-7-2015)

Si consideri questo frammento di programma che opera su interi:

```
static void method(int x, int y) {
        int z = x/2;
        if (x == y)
                ...
        else while (z < x)
                z++;
}
```

1. Si definisca un insieme di casi di test che copre tutti i branch
2. Si calcoli il predicato e si sintetizzi il caso di test che deve essere soddisfatto da un caso di test che
    a) Entra nel ramo else e percorre il ciclo 0 volte;
    b) Entra nel ramo else e percorre il ciclo 1 volta;
    c) Entra nel ramo else e percorre il ciclo 2 volte;
    d) Entra nel ramo then



SOLUZIONE PUNTO 1:
Dopo aver rappresentato il control flow è necessario coprire i branch/edge marcati con 1, 2, 3, 4, 5, 6.

x = 2, y = 2 copre i branches 1 e 6.

x = 2, y = 1 copre i branches 1, 2, 3, 4 e 5 .

Punto 2b

$x \neq y \wedge 1 \leq x \leq 2$

Punto 2c

$x \neq y \wedge 3 \leq x \leq 4$

Punto 2c

$x = y$

Punto 2d

$x = 0, y = 1$

$x = 2, y = 3$

$x = 4, y = 3$

$x = y = 1.$

Esercizio

Un metodo statico *tipoParallelogramma* riceve come parametri sei numeri float che rappresentano le lunghezze dei quattro lati *l1*, *l2*, *l3*, *l4,* presi in senso antiorario, e delle due diagonali, *d1* e *d2,* di un quadrilatero e restituisce una stringa che indica di che tipo di figura si tratta.

```
static String tipoParallelogramma (
      float l1, float l2, float l3, float l4, float d1, float d2)
```

Il metodo è specificato come segue:

```
Se i quattro lati l1 … l4 sono tutti uguali
      allora      se d1=d2
                  allora restituisci "quadrato"
                  altrimenti restituisci "rombo"
      altrimenti  se l1=l3 ed l2=l4
                  allora se d1=d2
                              allora restituisci "rettangolo"
                              altrimenti restituisci "altroParallelogramma"
                  altrimenti restituisci "nonParallelogramma"
```

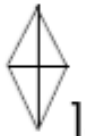
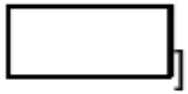
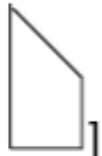
Quesito 1

Il metodo è specificato come segue:

```
Se i quattro lati l1 … l4 sono tutti uguali
      allora      se d1=d2
                  allora restituisci "quadrato"
                  altrimenti restituisci "rombo"
      altrimenti  se l1=l3 ed l2=l4
                  allora se d1=d2
                              allora restituisci "rettangolo"
                              altrimenti restituisci "altroParallelogramma"
                  altrimenti restituisci "nonParallelogramma"
```

Definire un insieme di *casi* di test (ossia dati di test più risultato atteso per quei dati) funzionali basati sulla precedente specifica.
NB: qui e nella risposta alle successive domande, indicando un caso di test non è indispensabile fornire le misure esatte di lati e diagonali; si richiede tuttavia di corredare ogni caso di test con un disegnino illustrativo.

Soluzione 1

{ [1 1 1 1 $\sqrt{2}$ $\sqrt{2}$ quadrato ], [$\sqrt{5}$ $\sqrt{5}$ $\sqrt{5}$ $\sqrt{5}$ 2 4 rombo ], [1 2 1 2 $\sqrt{5}$ $\sqrt{5}$ rettangolo ], [2 $\sqrt{2}$ 2 $\sqrt{2}$ $\sqrt{10}$ $\sqrt{2}$ altroParallelogramma ], [1 1 $\sqrt{2}$ 2 $\sqrt{2}$ $\sqrt{5}$ nonParallelogramma ] }

Quesito 2

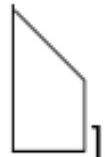
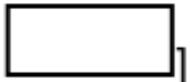
Si supponga che il metodo venga implementato come segue.

```
static String tipoParallelogramma (
      float l1, float l2, float l3, float l4, float d1, float d2){
            if (l1==l3) && (l2==l4)
                  if (l1==l2) && (d1==d2)
                        return "quadrato";
                  else  if (d1 != d2)
                              return "rombo";
                        else return "rettangolo";
            else return "nonParallelogramma";
}
```

È facile vedere che l'implementazione è scorretta, perché non riconosce il caso di altroParallelogramma".

Fornire un insieme di *casi* di test (indicando perciò anche il risultato atteso) che soddisfi il criterio della copertura delle diramazioni (decisioni) ma che NON permetta di evidenziare il difetto sopra citato.

Soluzione 2

{ [1 1 1 1 $\sqrt{2}$ $\sqrt{2}$ quadrato ], [1 1 $\sqrt{2}$ 2 $\sqrt{2}$ $\sqrt{5}$ nonParallelogramma ], [$\sqrt{5}$ $\sqrt{5}$ $\sqrt{5}$ 2 4 rombo ], [1 2 1 2 $\sqrt{5}$ $\sqrt{5}$ rettangolo ] }

Quesito 3

Spiegare, fornendo anche opportune esemplificazioni, perché utilizzando un insieme di dati di test che soddisfi il criterio di copertura delle condizioni il difetto viene rivelato.

Soluzione 3

Un siffatto insieme di casi di test deve contenere un caso che renda entrambe false le due componenti della condizione del secondo if (cioè tale che l1!=l2 e d1!=d2), oltre che vera la condizione del primo if (cioè tale che l1=l3 ed l2=l4), e queste condizioni individuano un parallelogramma che non è un rettangolo (quindi non un quadrato) né un rombo: in accordo alle specifiche, il metodo dovrebbe restituire "altroParallelogramma" mentre invece, scorrettamente, restituisce "rombo". Un esempio di caso di test di questo tipo è il seguente [2

$\sqrt{2}$ 2 $\sqrt{2}$ $\sqrt{10}$ $\sqrt{2}$ altroParallelogramma ].

Esercizio

- Si consideri la seguente specifica informale sotto riportata che definisce le regole riguardanti il superamento dell'esame di Ingegneria del Software.
- "Verranno svolte due prove intermedie. Ogni prova intermedia assegna un massimo di 13 punti ed è considerata valida se lo studente ottiene almeno un punteggio minimo di 6 punti per la I prova e di 6 punti per la II prova; chi ottiene un punteggio inferiore a quello minimo in una prova è obbligato a ripeterla nelle prove di recupero. La ripetizione riguarda comunque entrambe le prove; pertanto il mancato raggiungimento della soglia minima in una delle due prove richiede la partecipazione a una prova di recupero che riguarda l'intero programma del corso. L'attività svolta in laboratorio permette di ottenere un massimo di 4 punti, ed è considerata sufficiente se lo studente ottiene almeno 2 punti. Non è previsto il recupero del laboratorio. Pertanto in caso di valutazione insufficiente lo studente dovrà ripetere il corso nell'anno accademico successivo: saranno annullati gli eventuali risultati ottenuti durante le prove in itinere e non sarà possibile partecipare agli appelli. Per superare l'esame è inoltre necessario che la somma dei punteggi delle due prove in itinere sia almeno di 16 punti sui 26 disponibili e che il risultato complessivo (che comprende anche il voto relativo all'attività di laboratorio) sia almeno 18; lo studente che non soddisfa le precedenti condizioni dovrà recuperare entrambe le prove in un appello a propria scelta."

- In base alla specifica fornita, si definiscano dati a supporto del test funzionale (black-box testing) e i risultati attesi per ciascun dato.

Soluzione

< 7, 7, 0 > valore atteso: -1

< 3, 4, 3 > valore atteso: 0

< 7, 7, 3 > valore atteso: 0

< 10, 9, 3 > valore atteso: 22

Esercizio (tde 28-6-2013)

Si consideri il seguente frammento di programma C:

```
for (n=0; n<max_size && (c=getc(yyin)) != EOF && c!= '\n'; ++n)
buf[n] = (char) c;
```

1. Supponendo di voler coprire tutti i **branch**, qual'è il numero minimo di test che devono essere eseguiti? Giustificare la risposta.

Soluzione:

Il codice presenta un solo branch in corrispondenza del ciclo for per determinare se si deve uscire dal ciclo o se deve essere eseguita un'altra iterazione.

Basta quindi un solo caso di test per cui le condizioni di ingresso nel ciclo sono verificate almeno una volta. Questo caso di test copre sia l'ingresso sia l'uscita dal ciclo che avverrà al più dopo `max_size` iterazioni.

2. Supponendo di voler anche coprire tutte le **condizioni**, quanti e quali test dovrebbero essere eseguiti? Specificare esattamente i casi di test, con i valori delle variabili

Soluzione:

Chiamando `C1` la condizione `n < max_size`, `C2` la condizione `c != EOF` e `C3` la condizione `c != '\n'`, per poter coprire tutte le condizioni servono i seguenti casi di test.

- `C1 = False`
- `C1 = True && C2 = False`
- `C1 = True && C2 = True && C3 = False`
- `C1 = True && C2 = True && C3 = True`

L'operatore && in C, come in Java, è short circuited, quindi le altre possibili combinazioni delle condizioni non sono necessarie.

`getc` una funzione C che permette di leggere un carattere da un file, quindi i casi di test devono indicare il valore della variabile `max_size` e il contenuto del file cui fa riferimento `yyin`.

Dei possibili valori delle variabili per i diversi casi sono rispettivamente:

- `max_size = 0`
- `max_size = 1, yyin = riferimento a un file vuoto`
- `max_size = 1, yyin = riferimento a un file non vuoto, il cui prossimo carattere disponibile per la lettura è '\n'`
- `max_size = 1, yyin = riferimento a un file non vuoto, il cui prossimo carattere disponibile per la lettura è 'a'`

Si noti che i casi 2 e 4 potrebbero essere facilmente fusi in un solo test.