INTEGRAZIONE E TEST DI SISTEMI

GRUPPO: HackFAF

COMPONENTI:

Manfredi Francesco – 738906 - f.manfredi12@studenti.uniba.it

Alberta Motca Schnabel - 740769 - a.motcashnabel@studenti.uniba.it

Federica Picca - 749044 - f.picca4@studenti.uniba.it

ANNO ACCADEMICO 2022/2023

CORSO: ITPS

SOMMARIO

CAPITOLO 1 – PRIMO HOMEWORK

1. ASSEGNAZIONE

*Progettare e implementare una **suite di test black-box (specification-based testing)** per un codice a vostra scelta. La suite di test deve riferirsi al caso «**Unit test**», per cui dovete testare un **metodo o una classe** isolata (che non abbia quindi interazioni con altre classi o componenti esterni, quali DB, Web Service, ecc.) La suite di test deve comprendere non solo l'implementazione dei test con **JUnit**, ma anche tutta la fase di progettazione dei casi di test, vale a dire **l'approccio a sette step** spiegato a lezione (e presente sul libro e sulle slide).*

Il codice testato è dunque il seguente:

```java
package first.homework;

import java.util.Collection;


15 usages
public class CollsUtils {
    no usages
    private CollsUtils() {
    }
    14 usages
    public static boolean containsBoth(final Collection<?> coll1, final Collection<?> coll2) {
        if (coll1.size() < coll2.size()) {
            for (final Object aColl1 : coll1) {
                if (coll2.contains(aColl1)) {
                    return true;
                }
            }
        } else {
            for (final Object aColl2 : coll2) {
                if (coll1.contains(aColl2)) {
                    return true;
                }
            }
        }
        return false;
    }
}
```

Figura 1: Codice testato - Homework 1

Firma del metodo:

```java
/**
 * Returns <code>true</code> iff at least one element is in both collections.
 *
 * <p>
 *
 * @param coll1
 *            the first collection, must not be null
 * @param coll2
 *            the second collection, must not be null
 * @return <code>true</code> iff the intersection of the collections is
 *         non-empty
 * @since 2.1
 */
```

Figura 2: firma del metodo

CAPITOLO 1 – PRIMO HOMEWORK

2. COSA FA IL PROGRAMMA

- **COSA FA IL METODO**

  Si tratta dell'implementazione del metodo `containsBoth` che controlla se due diverse collezioni contengono almeno un elemento in comune. L'algoritmo itera su entrambe le collezioni e controlla se ogni elemento nella prima collezione è contenuto nella seconda collezione e viceversa, fino a trovare un elemento comune. Il metodo utilizza le collezioni delle API di Java, che non specificano valori di tipo primitivo come `int` o `double`. Invece, è possibile utilizzare l'incapsulamento di oggetti tramite classi wrapper come `Integer` o `Double`. È importante notare che il metodo utilizza l'istruzione `if-else` per selezionare la collezione più piccola da iterare prima. In questo modo si ottimizza l'algoritmo, riducendo il numero di iterazioni necessarie.

- **COSA NON FA IL METODO**

  Il metodo containsBoth non gestisce il caso in cui una collection (o entrambe) siano null, e quindi viene lanciata l'eccezione (NullPointerException).

- **DOMINIO**

  Le collection interessate accettano qualsiasi tipo di classi (eccetto le primitive) purché siano diverse da null.

- **PARAMETRI CHE RICEVE IL METODO**

  Il metodo prende in input due oggetti di tipo `Collection<?>`, ovvero due collezioni di oggetti generici.

- **OUTPUT**

  Il metodo restituisce un valore booleano che indica se le due collezioni contengono almeno un elemento in comune o meno.

CAPITOLO 1 – PRIMO HOMEWORK

3. ESEMPI

1. Avendo in input nella Coll1<Integer>(1,2,3) e nella Coll2<Integer>(3,4,5), le due collezioni hanno un elemento in comune quindi il metodo restituisce true.

```
@Test
void oneCommonElement(){
    Collection<Integer> coll1 = Arrays.asList(1,2,3);
    Collection<Integer> coll2 = Arrays.asList(3,4,5);
    assertTrue(CollsUtils.containsBoth(coll1, coll2));
}
```

Figura 3: esempio1

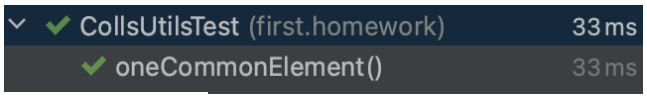

Figura 4: risultato1

2. Avendo in input nella Coll1<Integer>(1,2,3) e nella Coll2<Integer>(4,5,6), le due collezioni non hanno nessun elemento in comune, il metodo restituisce false.

```
@Test
void NoCommonElement(){
    Collection<Integer> coll1 = Arrays.asList(1,2,3);
    Collection<Integer> coll2 = Arrays.asList(4, 5, 6);
    assertFalse(CollsUtils.containsBoth(coll1,coll2));
}
```

Figura 5: esempio2

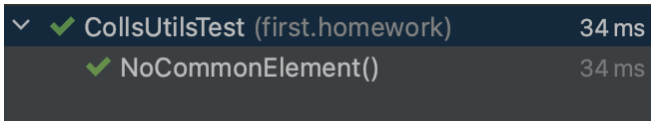

Figura 6: risultato2

3. Avendo in input nella Coll1<String>(“a”,” b”,”c”) e nella Coll2<Integer>(1,2,3), le due collezioni non hanno nessun elemento in comune (poiché di classi diverse), il metodo restituisce false.

```
@Test
void NoCommonElementWithDifferentType(){
    Collection<String> coll1 = Arrays.asList("a","b","c");
    Collection<Integer> coll2 = Arrays.asList(1,2,3);
    assertFalse(CollsUtils.containsBoth(coll1,coll2));
}
```

Figura 7: esempio3

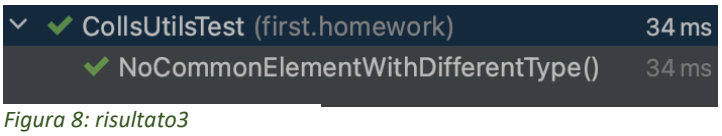

Figura 8: risultato3

4. Avendo in input nella Coll1<Double>(0.2, 0.6, 0.8) e nella Coll2<Double>(0.3, 0.6, 0.8), le due collezioni hanno due elementi in comune, il metodo restituisce true.

```
@Test
void TwoCommonElement(){
    Collection<Double> coll1 = Arrays.asList(0.2, 0.6, 0.8);
    Collection<Double> coll2 = Arrays.asList(0.3, 0.6, 0.8);
    assertTrue(CollsUtils.containsBoth(coll1,coll2));
}
```

Figura 9: esempio4

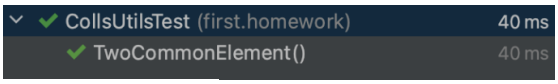

Figura 10: risultato4

5. Avendo in input nella Coll1<Integer>(2,5,7,9) e nella Coll2<Integer>(2,5,7,9), le due collezioni hanno tutti gli elementi in comune, il metodo restituisce true.

```
@Test
void AllCommonElement(){
    Collection<Integer> coll1 = Arrays.asList(2,5,7,9);
    Collection<Integer> coll2 = Arrays.asList(2,5,7,9);
    assertTrue(CollsUtils.containsBoth(coll1,coll2));
}
```

Figura 11: esempio5

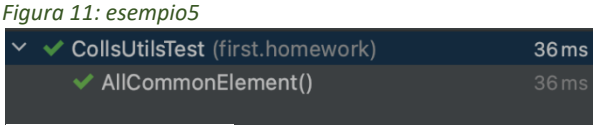

Figura 12: risultato5

6. Avendo in input nella Coll1<Integer>(2,5,null ,9) e nella Coll2<Integer>(2,5,7,9), due collezioni dove il primo, il secondo e il quarto elemento sono in comune, il metodo restituisce true.

```
@Test
void ThreeCommonElementWithNullElement(){
    Collection<Integer> coll1 = Arrays.asList(2,5,null,9);
    Collection<Integer> coll2 = Arrays.asList(2,5,7,9);
    assertTrue(CollsUtils.containsBoth(coll1,coll2));
}
```

Figura 13: esempio6

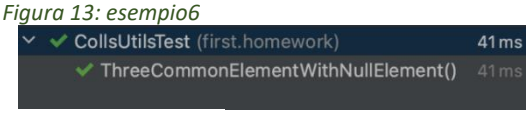

Figura 14: risultato6

7. Avendo in input nella Coll1<Integer>(null) e nella Coll2<String>(“a”,”b”,”c”), due collezioni dove una è nulla, il metodo solleva una NullPointerException.

```
@Test
void OneCollIsNull(){
    Collection<Integer> coll3 = null;
    Collection<String> coll4 = Arrays.asList("a","b","c");
    assertThrows(NullPointerException.class, ()-> CollsUtils.containsBoth(coll3,coll4));
}
```

Figura 12: esempio7

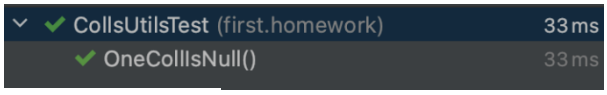

Figura 13: risultato7

CAPITOLO 1 – PRIMO HOMEWORK

4. INPUT E OUTPUT

i. INPUT

Valori di Coll1:

- Vuota
- Nulla
- Un solo elemento
- N elementi

Valori di Coll2

- Vuota
- Nulla
- Un solo elemento
- N elementi

ii. COMBINAZIONE DEGLI INPUT

a. Coll1 con duplicati e Coll2 con duplicati;
b. Coll1 con duplicati e Coll2 senza duplicati;
c. Coll1 con duplicati e Coll2 duplicati N volte;
d. Coll1 con duplicati N volte e Coll2 senza duplicati;
e. Coll1 con elementi null all'interno e Coll2 senza elementi null all'interno;
f. Coll1 con elementi null all'interno e Coll2 con elementi null all'interno;
g. Coll1 con N elementi e Coll2 con stessi elementi ma in ordine diverso;

iii. OUTPUT ATTESI

True: se viene trovato almeno un elemento in comune per le due collezioni;
False: altrimenti.

iv. BOUNDARY CASES

- **ON POINT** (punto per cui la condizione inizia a diventare vera): un elemento contenuto in entrambe le collezioni (stesso valore e stesso tipo).
- **OFF POINT** (punto che è più vicino al confine che fa sì che la condizione sia falsa): 0 elementi comuni ad entrambe le collezioni (perché di valore o tipo diversi).
- **IN POINT** (punto che fa sì che la condizione sia vera): 1 elemento in comune ad entrambe le collezioni.
- **OUT POINT** (punto che fa sì che la condizione sia falsa): 0 elementi in comune ad entrambe le collezioni.

CAPITOLO 1 – PRIMO HOMEWORK

5. TEST CASE

a. Casi eccezionali

T1: Coll1 nulla
T2: Coll1 vuota

T3: Coll2 nulla
T4: Coll2 vuota

b. Collezione con un solo elemento

T5: Coll1 ha un solo elemento che non è contenuto in Coll2

c. Combinazione di input

Coll1 = N elementi
Coll2 = 1 elemento

T6: Coll1 con elementi null all'interno e Coll2 senza elementi null all'interno
T7: Coll1 con elementi null all'interno e COll2 con elementi null all'interno

Quando Coll2 ha un solo elemento non hanno senso tutti gli altri casi individuati con la combinazione degli input.

Coll1 = N elementi
Coll2 = N elementi

T8: Coll1 con duplicati e Coll2 con duplicati
T9: Coll1 con duplicati e Coll2 senza duplicati
T10: Coll1 con duplicati e Coll2 duplicati N volte
T11: Coll1 con duplicati N volte e Coll2 senza duplicati
T12: Coll1 con N elementi e Coll2 con stessi elementi ma in ordine diverso

d. Boundary test

T13: Coll1 e Coll2 non hanno nessun elemento in comune
T14: Coll1 ha un solo elemento che è contenuto in Coll2

CAPITOLO 1 – PRIMO HOMEWORK

6. CODICE TEST

Si è scelto di implementare la suite di test suddividendo in 5 gruppi i test realizzati, per raggrupparli in modo semplice e chiaro:

1. **oneCollIsEmpty():**
   - Il primo caso, in cui coll1 è una collezione vuota e coll2 contiene elementi, restituirà false, poiché non vi è alcun elemento in comune.
   - Il secondo caso, in cui coll3 contiene elementi e coll4 è una collezione vuota, restituirà false, poiché non vi è alcun elemento in comune.

```
@Test
void oneCollIsEmpty(){
    Collection<String> coll1 = Collections.emptyList();
    Collection<String> coll2 = Arrays.asList("a", "b", "c");
    assertFalse(CollsUtilsFH.containsBoth(coll1, coll2)); //T2

    Collection<String> coll3 = Arrays.asList("a", "b", "c");
    Collection<String> coll4 = Collections.emptyList();
    assertFalse(CollsUtilsFH.containsBoth(coll3, coll4)); //T4
}
```

Figura 17: metodo di test oneCollIsEmpty

2. **oneCollIsNull():**
   - Nel primo caso, in cui coll1 è null e coll2 contiene elementi, il metodo solleverà un'eccezione di tipo NullPointerException, come atteso.
   - Nel secondo caso, in cui coll3 contiene elementi e coll4 è null, il metodo solleverà un'eccezione di tipo NullPointerException, come atteso.

```
@Test
void oneCollIsNull(){
    Collection<Integer> coll1 = null;
    Collection<Integer> coll2 = Arrays.asList(1,2,3);
    assertThrows(NullPointerException.class, ()-> CollsUtilsFH.containsBoth(coll1,coll2)); //T1

    Collection<Integer> coll3 = Arrays.asList(1,2,3);
    Collection<Integer> coll4 = null;
    assertThrows(NullPointerException.class, ()-> CollsUtilsFH.containsBoth(coll3,coll4)); //T3
}
```

Figura 18: metodo di test oneCollIsNull

3. **oneCollElement():**
   - Nel caso in cui una collezione presenti un solo elemento (coll3) e questo non sia presente nella seconda collezione (coll4), il metodo restituirà false.

```
@Test
void oneCollElement() {
    Collection<String> coll3 = List.of( e1: "a");
    Collection<String> coll4 = Arrays.asList("f", "b", "c");
    assertFalse(CollsUtilsFH.containsBoth(coll3, coll4)); //T5
}
```

Figura 19: metodo di test oneCollElement

4. **twoCollWithNullOrOneCollWithNull():**
   - Nel primo caso, in cui coll1 contiene un valore null e coll2 non contiene elementi in comune e nessun elemento null, il metodo restituirà false.
   - Nel secondo caso, in cui sia coll3 che coll4 contengono un valore null, il metodo restituirà true, poiché entrambe le collezioni condividono il valore null.

```
//Coll1 = N elements   Coll2 = 1 element
@Test
void twoCollWithNullOrOneCollWithNull() {
    Collection<Integer> coll1 = Arrays.asList(1, 2, null);
    Collection<Integer> coll2 = Arrays.asList(3, 4, 5);
    assertFalse(CollsUtilsFH.containsBoth(coll1, coll2)); //T6

    Collection<Integer> coll3 = Arrays.asList(1, 2, null);
    Collection<Integer> coll4 = Arrays.asList(3, null, 4);
    assertTrue(CollsUtilsFH.containsBoth(coll3, coll4)); //T7
}
```

Figura 20: metodo di test twoCallWithNullOrOneCallWithNull

5. **duplicatesAndRepetitions():**
   - Nel primo caso entrambe le collezioni contengono elementi che vengono ripetuti lo stesso numero di volte. Poiché hanno almeno un elemento in comune ("2"), il metodo restituirà true.
   - Nel secondo caso, coll3 e coll4 hanno un elemento in comune ("b"), quindi il metodo restituirà true anche se in una collezione compare una sola volta mentre nell'altra viene ripetuto più volte.
   - I casi successivi seguono lo stesso ragionamento, giocano sul numero di ripetizioni degli elementi che compaiono duplicati e restituiscono true poiché, in ognuno di questi casi, le collezioni hanno almeno un elemento in comune.
   - L'ultimo caso, rappresentato da coll9 e coll10, si concentra invece sull'ordine in cui compaiono gli elementi. Entrambe le collezioni presentano, infatti, gli stessi identici elementi ma disposti in ordine diverso. Il valore booleano restituito in questo caso è sempre true perché, anche se in ordine diverso, le due collezioni hanno almeno un elemento in comune.

```
//Coll1 = N elements  Coll2 = N elements
@Test
void duplicatesAndRepetitions(){
    Collection<Integer> coll1 = Arrays.asList(1, 2, 2, 4);
    Collection<Integer> coll2 = Arrays.asList(3, 6, 2, 6, 2);
    assertTrue(CollsUtilsFH.containsBoth(coll1,coll2));  //T8

    Collection<String> coll3 = Arrays.asList("a", "b", "b", "c");
    Collection<String> coll4 = Arrays.asList("b", "f", "z");
    assertTrue(CollsUtilsFH.containsBoth(coll3,coll4)); //T9


    Collection<Integer> coll5 = Arrays.asList(1, 2, 2, 4);
    Collection<Integer> coll6 = Arrays.asList(3, 6, 2, 2, 6, 2, 2, 5, 8, 2, 2, 78, 2, 2);
    assertTrue(CollsUtilsFH.containsBoth(coll5,coll6)); //T10

    Collection<String> coll7 = Arrays.asList("a", "b", "b", "c", "v", "b", "b", "b");
    Collection<String> coll8 = Arrays.asList("b", "f", "z");
    assertTrue(CollsUtilsFH.containsBoth(coll7,coll8)); //T11

    Collection<Double> coll9 = Arrays.asList(0.2, 0.6, 0.8, 0.10);
    Collection<Double> coll10 = Arrays.asList(0.6, 0.10, 0.2, 0.8);
    assertTrue(CollsUtilsFH.containsBoth(coll9,coll10)); //T12
}
```

Figura 21: metodo di test duplicateAndRepetitions

6. **boundary():**
   - Le collezioni coll1 e coll2 non hanno elementi in comune, quindi il metodo restituirà false.
   - Le collezioni coll3 e coll4 hanno almeno un elemento in comune, quindi, è il primo caso in cui il metodo trova un elemento contenuto in entrambe le collezioni.

```
@Test
void boundary(){
    Collection<Double> coll1 = Arrays.asList(0.42, 0.01, 0.10);
    Collection<Double> coll2 = Arrays.asList(0.66, 0.122, 1.0);
    assertFalse(CollsUtilsFH.containsBoth(coll1,coll2));  //T14

    Collection<String> coll3 = List.of( e1: "a");
    Collection<String> coll4 = Arrays.asList("a", "b", "c");
    assertTrue(CollsUtilsFH.containsBoth(coll3, coll4)); //T13
}
```

Figura 22: metodo di test boundary

CAPITOLO 1 – PRIMO HOMEWORK

TABELLE HOMEWORK 1

ID	DESCRIZIONE
T1	Coll1 null
SCENARIO DI TEST	
Una collezione è null mentre l'altra contiene elementi	
PRE-REQUISITI	
• Una collezione è null mentre l'altra è piena	
RISULTATO ATTESO	**RISULTATO OTTENUTO**
Ci si aspetta che con assertThrows il test passi perché una collezione è null e non potendo la collezione essere null deve sollevare un'eccezione.	Il test passa utilizzando assertThrows perché viene sollevata l'eccezione.

ID	DESCRIZIONE
T2	Coll1 empty
SCENARIO DI TEST	
Una collezione è vuota mentre l'altra contiene elementi	
PRE-REQUISITI	
• Una collezione è vuota mentre l'altra è piena	
RISULTATO ATTESO	**RISULTATO OTTENUTO**
Ci si aspetta che con assertFalse il test passi perché essendo una collezione vuota non ci potrebbero essere elementi in comune.	Il test passa utilizzando assertFalse perché non vi possono essere elementi in comune.

ID	DESCRIZIONE
T3	Coll2 null

SCENARIO DI TEST
Una collezione è null mentre l'altra contiene elementi

PRE-REQUISITI
• Una collezione è null mentre l'altra è piena

RISULTATO ATTESO	RISULTATO OTTENUTO
Ci si aspetta che con assertThrows il test passi perché una collezione è null e non potendo la collezione essere null deve sollevare un'eccezione.	Il test passa utilizzando assertThrows perché viene sollevata l'eccezione.

ID	DESCRIZIONE
T4	Coll2 empty

SCENARIO DI TEST
Una collezione è vuota mentre l'altra contiene elementi

PRE-REQUISITI
• Una collezione è vuota mentre l'altra è piena

RISULTATO ATTESO	RISULTATO OTTENUTO
Ci si aspetta che con assertFalse il test passi perché essendo una collezione vuota non ci potrebbero essere elementi in comune.	Il test passa utilizzando assertFalse perché non vi possono essere elementi in comune.

ID	DESCRIZIONE
T5	Coll1 ha un solo elemento che non è contenuto in Coll2

SCENARIO DI TEST
Una collezione contiene un solo elemento, l'altra contiene più elementi, ma non quell'unico che è contenuto nella prima (non hanno elementi in comune)

PRE-REQUISITI
• Una collezione che presenta un solo elemento • Una collezione che presenta più elementi • Le due collezioni non hanno elementi in comune

RISULTATO ATTESO	RISULTATO OTTENUTO
Ci si aspetta che con assertFalse il test passi perché il metodo non trova nessun elemento che sia presente in entrambe le collezioni.	Il test passa utilizzando assertFalse perché non c'è un elemento in comune ad entrambe le collezioni.

ID	DESCRIZIONE
T6	Coll1 con elementi null all'interno e Coll2 senza elementi null all'interno

SCENARIO DI TEST
Una collezione contiene vari elementi, tra cui uno o più pari a null. L'altra collezione contiene più elementi, ma nessuno di questi è null.

PRE-REQUISITI
• Una collezione con più elementi, tra cui null • Una collezione con più elementi, senza null • Nessun elemento in comune

RISULTATO ATTESO	RISULTATO OTTENUTO
Ci si aspetta che con assertFalse il test passi nel momento in cui non vi è nessun elemento, diverso da null, in comune. Questo perché elemento pari a null sicuramente non può rendere la condizione vera essendo presente in una sola collezione. Nel caso in cui nel test scritto ci fossero stati altri elementi, diversi da null, in comune sarebbe stato necessario ricorrere all'utilizzo di assertTrue.	Il test passa utilizzando assertFalse perché non è presente nessun elemento, diverso da null, presente in entrambe le collezioni.

ID	DESCRIZIONE
T7	Coll1 con elementi null all'interno e Coll2 con elementi null all'interno

SCENARIO DI TEST
Una collezione contiene vari elementi, tra i quali sono presenti uno o più elementi pari a null. La seconda collezione contiene a sua volta più elementi, tra i quali sono presenti uno o più elementi pari a null.

PRE-REQUISITI
• Una collezione con più elementi, tra cui null • Una collezione con più elementi, tra cui null

RISULTATO ATTESO	RISULTATO OTTENUTO
Ci si aspetta che con assertTrue il test passi perché, nonostante le due collezioni contengano elementi differenti, esse contengono entrambe almeno un elemento pari a null, che è proprio l'elemento in comune che fa sì che il metodo restituisca true.	Il test passa utilizzando assertTrue perché c'è almeno un elemento, null, in comune ad entrambe le collezioni .

ID	DESCRIZIONE
T8	Coll1 con duplicati e Coll2 con duplicati

SCENARIO DI TEST
Una collezione contiene vari elementi, tra i quali ci sono duplicati. L'altra collezione contiene vari elementi, tra i quali ci sono duplicati

PRE-REQUISITI
• Una collezione che presenta elementi duplicati • Una collezione che presenta elementi duplicati

RISULTATO ATTESO	RISULTATO OTTENUTO
Ci si aspetta che con assertTrue il test passi nel momento in cui le due collezioni presentino almeno un elemento in comune, che potrebbe corrispondere con un duplicato o meno.	Il test passa utilizzando assertTrue perché c'è almeno un elemento in comune ad entrambe le collezioni, il test fallisce con assertFalse quando non c'è alcun elemento comune ad entrambe le collezioni.

ID	DESCRIZIONE
T9	Coll1 con duplicati e Coll2 senza duplicati

SCENARIO DI TEST
Una collezione contiene vari elementi, tra i quali c'è almeno un elemento che viene duplicato una volta. L'altra collezione contiene elementi senza duplicati

PRE-REQUISITI
• Una collezione che presenta elementi duplicati • Una collezione che presenta elementi senza duplicati • L'elemento duplicato è presente anche nella collezione senza duplicati

RISULTATO ATTESO	RISULTATO OTTENUTO
Ci si aspetta che con assertTrue il test passi perché, anche se in una delle due collezioni è presente un elemento duplicato, quello che conta è quell'elemento compaia almeno una volta in entrambe le collezioni.	Il test passa utilizzando assertTrue perché c'è almeno un elemento in comune ad entrambe le collezioni.

ID	DESCRIZIONE
T10	Coll1 con duplicati e Coll2 duplicati N volte

SCENARIO DI TEST
Una collezione contiene vari elementi, tra i quali ci sono elementi che compaiono duplicati. L'altra collezione contiene gli stessi elementi che vengono duplicati nella prima collezione, ma in questa collezione questi vengono duplicati molte volte

PRE-REQUISITI
• Una collezione che presenta elementi duplicati • Una collezione che presenta gli stessi elementi duplicati, ma tante volte

RISULTATO ATTESO	RISULTATO OTTENUTO
Ci si aspetta che con assertTrue il test passi perché, anche se in una delle due collezioni un elemento duplicato è presente più volte quello che conta è quell'elemento compaia almeno una volta in entrambe le collezioni	Il test passa utilizzando assertTrue perché c'è almeno un elemento in comune ad entrambe le collezioni

ID	DESCRIZIONE
T11	Coll1 con duplicati N volte e Coll2 senza duplicati

SCENARIO DI TEST
Una collezione contiene vari elementi, tra i quali ci sono duplicati ripetuti più volte. L'altra collezione contiene elementi senza duplicati

PRE-REQUISITI
• Una collezione che presenta elementi duplicati varie volte • Una collezione che presenta elementi senza duplicati • L'elemento duplicato più volte in una collezione è presente anche nell'altra collezione ma non è duplicato

RISULTATO ATTESO	RISULTATO OTTENUTO
Ci si aspetta che con assertTrue il test passi perché, anche se in una delle due collezioni un elemento è presente più volte quello che conta è quell'elemento compaia almeno una volta in entrambe le collezioni.	Il test passa utilizzando assertTrue perché c'è almeno un elemento in comune ad entrambe le collezioni.

ID	DESCRIZIONE
T12	Coll1 con N elementi e Coll2 con stessi elementi ma in ordine diverso

SCENARIO DI TEST
Una collezione contiene N elementi. L'altra collezione contiene gli stessi elementi ma disposti in ordine diverso

PRE-REQUISITI
• Due collezioni riempite con gli stessi elementi • Elementi disposti in ordine diverso nelle due collezioni

RISULTATO ATTESO	RISULTATO OTTENUTO
Ci si aspetta che con assertTrue il test passi perché, anche se l'ordine è diverso, il metodo trova elementi presenti in entrambe le collezioni.	Il test passa utilizzando assertTrue perché vengono trovati elementi presenti in entrambe le collezioni indipendentemente dall'ordine con cui questi compaiono.

ID	DESCRIZIONE
T13	Coll1 e Coll2 non hanno nessun elemento in comune
SCENARIO DI TEST	
Due collezioni sono state riempite con un certo numero di elementi ma le due collezioni non presentano nessun elemento in comune.	
PRE-REQUISITI	
• Due collezioni piene • Elementi diversi tra loro • Nessun elemento comune ad entrambe le collezioni	
RISULTATO ATTESO	**RISULTATO OTTENUTO**
Ci si aspetta che con assertFalse il test passi perché il metodo non trova nessun elemento che sia presente in entrambe le collezioni.	Il test passa utilizzando assertFalse perché le due collezioni non presentano nessun elemento che sia contenuto in entrambe le collezioni.

ID	DESCRIZIONE
T14	Coll1 ha un solo elemento che è contenuto in Coll2
SCENARIO DI TEST	
Una collezione contiene un unico elemento. L'altra collezione contiene più elementi, tra cui vi è quello contenuto nella prima collezione (elemento in comune)	
PRE-REQUISITI	
• Una collezione che contiene un solo elemento • Una collezione con più elementi • L'elemento della prima collezione è contenuto anche nella seconda	
RISULTATO ATTESO	**RISULTATO OTTENUTO**
Ci si aspetta che con assertTrue il test passi perché l'unica cosa che conta è che l'unico elemento contenuto nella prima collezione sia presente anche nella seconda collezione.	Il test passa utilizzando assertTrue perché c'è un elemento in comune ad entrambe le collezioni.

CAPITOLO 1 – PRIMO HOMEWORK

CODE COVEREGE – TASK 1

CollsUtils.java

```
1.  package first.homework;
2.
3.  import java.util.Collection;
4.
5.
6.  public class CollsUtils {
7.      private CollsUtils() {
8.      }
9.      public static boolean containsBoth(final Collection<?> coll1, final Collection<?> coll2) {
10.         if (coll1.size() < coll2.size()) {
11.             for (final Object aColl1 : coll1) {
12.                 if (coll2.contains(aColl1)) {
13.                     return true;
14.                 }
15.             }
16.         } else {
17.             for (final Object aColl2 : coll2) {
18.                 if (coll1.contains(aColl2)) {
19.                     return true;
20.                 }
21.             }
22.         }
23.         return false;
24.     }
25. }
26.
27.
```

Created with JaCoCo 0.8.8.202204050719

Figura 23: code coverage primo homework

La code coverage indicata dal report di Jacoco, relativo alla suite di test sviluppata durante lo Specification based Test presenta già una copertura massima (100%).

Bisogna notare che è stata raggiunta innanzitutto la **line coverage**, in quanto ogni riga del codice del metodo risulta essere attraversata da almeno un test. Ciò indica che durante l'esecuzione dei test, tutte le righe di codice sono state prese in considerazione e testate.

Per quanto riguarda la **branch coverage**, è stata raggiunta una copertura completa dei rami. Sono stati eseguiti test specifici per garantire che siano stati coperti sia il ramo del vero (ad esempio con i test T8 e T10) che il ramo del falso (con i test T9 e T11) nell'istruzione if più esterna presente nel codice.

Attraverso questi test è stata raggiunta anche **la branch + condition coverage**, infatti, prendendo come esempio il test T8, questo rispetta sia l'if esterno perché la coll1 ha meno elementi della coll2 e rispetta anche la condizione dell'if interno perché almeno un elemento della coll1 è contenuto nella coll2.

Invece il test T11, per esempio, oltre ad entrare nel ramo dell'else rispetta la condizione dell'if più interno perché la collezione equivalente a coll1 ha al suo interno almeno un elemento della collezione equivalente a coll2

Questo fornisce una copertura quanto più possibile completa del codice e aumenta la fiducia nell'efficacia dei test eseguiti.

CAPITOLO 1 – PRIMO HOMEWORK

RAFFINAMENTO DEL CODICE

```java
package first.homework;
import java.util.Collection;
17 usages
public class CollsUtils {
    no usages
    private CollsUtils() {
    }
    16 usages
    public static boolean containsBoth(final Collection<?> coll1, final Collection<?> coll2) {
        if(coll1 == null || coll2 == null){
            throw new NullPointerException("Null collection exception");
        }
        if (coll1.size() < coll2.size()) {
            for (final Object aColl1 : coll1) {
                if (coll2.contains(aColl1)) {
                    return true;
                }
            }
        } else {
            for (final Object aColl2 : coll2) {
                if (coll1.contains(aColl2)) {
                    return true;
                }
            }
        }
        return false;
    }
}
```

Figura 24: bug fix primo homework

Dalla firma del metodo containsBoth() è possibile evincere che le collezioni coll1 e coll2 non devono essere nulle. Quindi, nel codice fornito, è stata fatta una piccola aggiunta che consiste nell'inserimento di un controllo esterno, con un'istruzione if, per verificare se coll1 o coll2 sono null.

In caso affermativo, viene lanciata un'eccezione di tipo NullPointerException. Questo controllo esterno è stato introdotto per gestire esplicitamente il caso in cui una delle due collezioni sia null e per assicurarsi che non si verifichino errori durante l'esecuzione del metodo.

CAPITOLO 2 – SECONDO HOMEWORK

ASSEGNAZIONE

Scelto un nuovo codice (che il docente deve approvare) eseguite il minor numero di test che garantisca il 100% code coverage. A valle della scrittura di questi test, elencare i casi di test non coperti dall'analisi di code coverage ma che è importante eseguire nella prospettiva di analisi black-box.

Il codice testato è il seguente:

```java
package second.homework;

public class calculate {
    no usages
    private calculate() {
    }
    no usages
    public static int calculateSumOfEvenNumbers(int[] numbers) {
        if (numbers == null || numbers.length == 0) {
            throw new IllegalArgumentException("Input array cannot be null or empty");
        }
        int sum = 0;
        for (int i = 0; i < numbers.length; i++) {
            if (numbers[i] > 0 && numbers[i] % 2 == 0) {
                if (numbers[i] > 10 && numbers[i] < 20) {
                    sum += numbers[i] * 2;
                } else if (numbers[i] > 20) {
                    sum += numbers[i] / 2;
                } else {
                    sum += numbers[i];
                }
            }
        }
        return sum;
    }
}
```

Figura 25: codice testato homework 2

COSA FA IL PROGRAMMA

Questo metodo, denominato "**calculateSumOfEvenNumbers**", calcola la somma dei numeri pari in un array di interi. Per prima cosa controlla se l'array di numeri è nullo o vuoto. Se lo è, viene lanciata un'eccezione IllegalArgumentException con un messaggio di errore.

Viene eseguito un ciclo for per scorrere l'array di numeri e all'interno di questo ciclo, viene controllato se il numero corrente è positivo e pari.
Se il numero corrente è compreso tra 10 e 20 (esclusi), viene moltiplicato per 2 e il risultato viene aggiunto alla somma data dalla variabile "sum".
Se invece il numero corrente è maggiore di 20, viene diviso per 2 e il risultato viene aggiunto alla somma "sum".
Infine, se il numero corrente non soddisfa nessuna delle condizioni precedenti, viene semplicemente aggiunto alla somma "sum" e alla fine del ciclo viene restituito il valore della variabile "sum", che rappresenta la somma dei numeri pari e positivi nell'array.

CAPITOLO 2 – SECONDO HOMEWORK

100% DI CODE COVERAGE CON IL MINOR NUMERO DI TEST – TASK 2

I test necessari per raggiungere il 100% di code coverage con il minor numero di test sono i seguenti:

T1: array nullo
T2: array con valori pari e positivi. Per esempio, prendendo 18,20,88 come valori otteniamo che:

- Sono tutti valori pari e maggiori di zero.
- Con il 18 si rispetta la condizione di avere un numero compreso tra 10 e 20, estremi esclusi.
- Con 88 si entra nella condizione dei numeri strettamente maggiori di 20.
- Con 20 come valore si entra nell'else finale.

Con i due test individuati si raggiunge il **100% di line coverage**. Si può notare che per coprire il primo if che lancia l'eccezione IllegalArgumentException nel caso di array nulli o vuoti è sufficiente testare uno dei due casi; quindi, si esegue il test sugli array nulli o su quelli vuoti, farli entrambi non ha effetti sulla percentuale di codice coperto.

Con i valori individuati e testati con T2 si ottiene anche la **branch + condition coverage**; infatti, avendo a che fare con array non nulli e non vuoti, contenenti valori pari e positivi, si entra almeno una volta nel for esterno. Successivamente grazie ai valori interi presi in considerazione di entra almeno una volta nell'if (con 18), nell'if-else (con 88) e infine nell'else (con 20) perché sono tutti valori che rispettano le rispettive condizioni.

```java
public class CalculateTest {
    @Test
    public void nullArray(){

        int[] numbers = null;
        assertThrows(IllegalArgumentException.class, () -> Calculate.calculateSumOfEvenNumbers(numbers));
    }

    @Test
    public void values(){

        int[] numbers = {18,20,88};
        assertEquals( expected: 100, Calculate.calculateSumOfEvenNumbers(numbers));
    }
}
```

Figura 26: test eseguiti per ottenere il 100% di code coverage

```java
package second.homework;

// Press Shift twice to open the Search Everywhere dialog and type `show whitespaces`,
// then press Enter. You can now see whitespace characters in your code.
public class Calculate {

    public static int calculateSumOfEvenNumbers(int[] numbers) {
        if (numbers == null || numbers.length == 0) {
            throw new IllegalArgumentException("Input array cannot be null or empty");
        }

        int sum = 0;

        for (int i = 0; i < numbers.length; i++) {
            if (numbers[i] > 0 && numbers[i] % 2 == 0) {
                if (numbers[i] > 10 && numbers[i] < 20) {
                    sum += numbers[i] * 2;
                } else if (numbers[i] > 20) {
                    sum += numbers[i] / 2;
                } else {
                    sum += numbers[i];
                }
            }
        }
        return sum;
    }
}
```

Figura 27: 100% di code coverage con il minor numero di test possibili

CAPITOLO 2 –SECONDO HOMEWORK

SPECIFICATION-BASED TESTING – CASI MANCANTI

COMPRENSIONE DEI REQUISITI

Il metodo calculateSumOfEvenNumbers ha lo scopo di sommare i numeri pari, applicando determinate trasformazioni a certe categorie di numeri. Vengono quindi sommari solo i numeri positivi e pari, mentre array nulli o vuoti sollevano una IllegalArgumentException.

PARAMETRI: Array di numeri interi.

RETURN: Il metodo ritorna un intero che corrisponde alla somma dei numeri pari e positivi presenti nell'array.

ECCEZIONI: Il metodo solleva una IllegalArgumentException se l'array passato risulta essere vuoto o nullo.

Input e output

Input

Valori dell'array:

- Vuoto
- Null
- Un solo elemento positivo pari
- Un solo elemento positivo dispari
- N elementi positivi pari
- N elementi positivi dispari
- Zero
- Elementi negativi

Combinazioni di input

- N elementi positivi pari e dispari
- N elementi positivi pari ed elementi negativi
- N elementi positivi dispari ed elementi negativi
- N elementi positivi pari e 0
- N elementi positivi dispari e 0
- N elementi negativi e 0

Output attesi

- Somma dei numeri purché siano positivi e pari.
- Eccezione in cui l'array sia vuoto o nullo.
- 0 nel caso in cui i valori presenti nell'array siano tutti dispari o negativi.

Boundary case

- ON POINT (punto per cui la condizione inizia a diventare vera): Array composto da un elemento pari a 2 (perché deve essere pari).
- OFF POINT (punto che è più vicino al confine che fa sì che la condizione sia falsa): Array composto da un elemento pari a 1.
- IN POINT (punto che fa sì che la condizione sia vera): Array composto da un elemento pari a 2 (perché deve essere pari).
- OUT POINT (punto che fa sì che la condizione sia falsa): Array composto da un elemento pari a 0.

TEST CASE

Casi eccezionali

T3: Array vuoto

Array con un solo elemento

T4: Array con un elemento pari negativo
T5: Array con un elemento dispari negativo
T6: Array con un elemento pari a 0

Combinazione di input

T7: N elementi pari e dispari
T8: N elementi positivi pari ed elementi negativi
T9: N elementi positivi dispari ed elementi negativi
T10: N elementi positivi pari e 0
T11: N elementi positivi dispari e 0
T12: Elementi negativi e 0

Boundary test

T13: Array con un elemento pari a 1 (unico elemento dispari positivo)
T14: Array con un elemento pari a 2 (unico elemento pari positivo)

CAPITOLO 2 –SECONDO HOMEWORK

Si è deciso di utilizzare **test parametrici**, per eseguire un insieme di casi di test simili con input diversi e valori attesi corrispondenti. Ciò riduce la duplicazione del codice in quanto non è necessario scrivere un metodo di test separato per ogni caso di test individuale.

Nei test parametrici realizzati vengono testati sia casi standard (come array con elementi pari e dispari) che casi limite (come array vuoti o array con un solo elemento). Inoltre, si è deciso di suddividere i test in due parti, una inerente l'**Happy Path** (per testare il comportamento corretto dell'applicazione) ed una inerente il **Sad Path** o **Unhappy Path** (per testare come l'applicazione gestisce situazioni di errore o scenari imprevisti), ciò al fine di una migliore organizzazione del report dei test.

L'obiettivo sia dell'Happy Path che dell'Unhappy Path è quello di identificare eventuali difetti o problemi nel software e verificare se l'applicazione è in grado di gestire correttamente una vasta gamma di scenari e condizioni diverse.

CODICE TEST

Il codice contiene sia casi di test Sad Path che di Happy Path. I casi di test Sad Path verificano il comportamento quando viene passato un array nullo o vuoto, mentre i casi di test Happy Path eseguono una serie di casi di test con input validi e si aspettano un risultato corretto dal metodo **calculateSumOfEvenNumbers()**.

SAD PATH

```
//T1 e T3
no usages
@DisplayName("Sad Path Tests")
@Test
public void nullOrEmptyArray() {
    int[] nullArray = null;
    int[] emptyArray = {};

    assertThrows(IllegalArgumentException.class, () -> Calculate.calculateSumOfEvenNumbers(nullArray));
    assertThrows(IllegalArgumentException.class, () -> Calculate.calculateSumOfEvenNumbers(emptyArray));
}
```

Figura 28: sad path dell'homework 2

Per quanto riguarda T1, il metodo **nullArray()** è un caso di test che verfica il comportamento quando viene passato un array nullo come input al metodo calculateSumEvenNumbers(). L'assertThrows verifica se viene sollevata correttamente un'eccezione **IllegalArgumentException**.

Invece, per T3 il metodo **emptyArray()** è un altro caso che verifica il comportamento quando viene passato un array vuoto come input al metodo calculateSumOfEvenNumbers(). Anche qui lo scopo è quello di verificare se viene sollevata correttamente un'eccezione **IllegalArgumentException**.

HAPPY PATH

```java
@ParameterizedTest
@DisplayName("Happy Path Tests")
@MethodSource("happyPathTestCases")
public void valuesHappyPath(int[] numbers, int excepted){

    assertEquals(excepted, Calculate.calculateSumOfEvenNumbers(numbers));
}


public static Stream<Arguments> happyPathTestCases() {
    return Stream.of(
            Arguments.of(new int[]{18,20,88}, 100), //T2

            //single element array
            Arguments.of(new int[]{-30},0), //T4
            Arguments.of(new int[]{-73},0), //T5
            Arguments.of(new int[]{0},0), //T6

            //combination of inputs
            Arguments.of(new int[]{2, 3, 4, 5},6), //T7
            Arguments.of(new int[]{2, 4, -4, -2},6), //T8
            Arguments.of(new int[]{1, 3, -3, -1},0), //T9
            Arguments.of(new int[]{2, 0, 4, 18},42), //T10
            Arguments.of(new int[]{1, 0, 3, 17},0), //T11
            Arguments.of(new int[]{-1, 0,-3, -10},0), //T12

            //boundary case
            Arguments.of(new int[]{1},0), //T13
            Arguments.of(new int[]{2},2) //T14
    );
}
```

Figura 29: happy path dell'homework 2

Il metodo **valuesHappyPath()**esegue diversi casi di test con input e valori attesi specificati nel metodo happyPathTestCases(). Esso contiene l'asserzione **assertEquals**, utilizzata per confrontare il valore restituito dal metodo calculateSumOfEvenNumbers con il valore atteso specificato nel caso di test.
Il metodo **happyPathTestCases()** restituisce uno stream di argomenti che rappresentano diversi casi di test per l'Happy Path. Questi casi di test coprono scenari in cui l'array di input contiene elementi validi.

Nel caso di **T4**, **T5** l'array contiene rispettivamente un unico elemento negativo pari e negativo dispari; essendo tutti valori che non rispettano la condizione dell'if esterno (numero positivo pari), il comportamento è lo stesso, ovvero quello di restituire 0, che è il valore con cui è stata inizializzata la variabile somma.

Nel caso di **T6**, l'unico valore contenuto è 0, di conseguenza il valore di ritorno nella variabile somma sarà pari a sé stesso, poiché non ci sarà nulla da aggiungere alla somma che è stata inizializzata a 0.

Relativamente al test **T7**, invece, l'array contiene elementi pari e dispari, di conseguenza verranno aggiunti alla somma solo quelli pari che passano la condizione dell'if esterno.

T8 è il test in cui l'array contiene N elementi positivi pari ed elementi negativi, per cui soddisferanno la condizione dell'if solo i primi, che verranno aggiunti alla variabile somma con le dovute operazioni preliminari necessarie.

In merito al caso di test **T9**, l'array contiene elementi positivi dispari ed elementi negativi che non soddisfano la condizione e quindi il risultato che torna nella variabile somma è pari a 0; stesso discorso vale per **T11**, che contiene elementi positivi dispari (anch'essi non rispettano la condizione) e 0, che non aggiungono nulla alla somma precedentemente inizializzata a 0.

Per quanto riguarda **T10**, l'array contiene valori positivi pari e lo 0; solo i primi, rispettando la condizione, verranno considerati nel calcolo della somma e nel valore di ritorno.

Infine, nel caso di **T12** si è scelto di prendere in considerazione un array contenente il valore 0 e numeri negativi in generale, senza una distinzione tra negativi pari e negativi dispari. Questa scelta è stata fatta perché, mentre nel caso di T10 e T11 la distinzione tra pari e dispari porta a risultati diversi, in questo caso il risultato che si ottiene è sempre 0 perché nessun valore rispetta la condizione dell'if esterno.

I **boundary case**, invece, sono rappresentati da due test **T13** e **T14**, in cui l'array contiene rispettivamente un unico elemento pari a 1 o a 2, essi quindi rientrano nei casi "unico elemento positivo dispari" e "unico elemento positivo pari"; per il primo caso il valore di ritorno è pari a 0, perché non è rispettata la condizione dell'if, mentre per il secondo si procede con l'aggiunta allo stesso alla variabile somma per il conteggio richiesto, senza dover effettuare alcun tipo di operazione (caso in cui è positivo pari e minore di 10).

CAPITOLO 2 –SECONDO HOMEWORK

TABELLE – HOMEWORK 2

ID	DESCRIZIONE
T1	Array nullo
SCENARIO DI TEST	
L'array è uguale a null	
PRE-REQUISITI	
• Array uguale a null	
RISULTATO ATTESO	**RISULTATO OTTENUTO**
Ci si aspetta che il test non passi l'if esterno, poiché l'array è nullo; viene quindi lanciata un'eccezione IllegalArgumentException.	Lanciata l'eccezione IllegalArgumentException.

ID	DESCRIZIONE
T2	N elementi positivi e pari
SCENARIO DI TEST	
L'array contiene elementi tutti positivi e pari	
PRE-REQUISITI	
• Array con più di un elemento • I valori sono tutti positivi e pari	
RISULTATO ATTESO	**RISULTATO OTTENUTO**
Ci si aspetta il valore restituito dalla somma sia proprio la somma dei valori presenti nell'array. Verranno prima eseguite le dovute operazioni e poi verranno aggiunti alla somma.	Il valore della somma, restituito dal metodo calculateSumOfEvenNumbers, corrisponde esattamente al valore atteso nel test.

ID	DESCRIZIONE
T3	Array vuoto
SCENARIO DI TEST	
L'array non contiene alcun valore	
PRE-REQUISITI	
• Array vuoto	
RISULTATO ATTESO	**RISULTATO OTTENUTO**
Ci si aspetta che non venga trovato alcun elemento; non viene aggiunto nulla alla variabile somma e viene lanciata un'eccezione IllegalArgumentException.	Lanciata l'eccezione IllegalArgumentException.

ID	DESCRIZIONE
T4	Un elemento pari e negativo
SCENARIO DI TEST	
L'array contiene un elemento pari e negativo	
PRE-REQUISITI	
• Array con un unico elemento pari e negativo	
RISULTATO ATTESO	**RISULTATO OTTENUTO**
Ci si aspetta che il valore restituito dalla somma sia pari a 0, poiché non è presente alcun elemento pari e positivo: l'if che controlla che sia pari e positivo non passa e ritorna il valore con cui è stata inizializzata la variabile sum.	Il valore della somma, restituito dal metodo calculateSumOfEvenNumbers, corrisponde esattamente al valore atteso nel test (pari a 0)..

ID	DESCRIZIONE
T5	Un elemento dispari e negativo

SCENARIO DI TEST
L’array contiene un elemento dispari e negativo

PRE-REQUISITI
• Array con un unico elemento pari e negativo

RISULTATO ATTESO	RISULTATO OTTENUTO
Ci si aspetta che il valore restituito dalla somma sia pari a 0, poiché non è presente alcun elemento pari e positivo: l’if che controlla che sia pari e positivo non passa e ritorna il valore con cui è stata inizializzata la variabile sum.	Il valore della somma, restituito dal metodo calculateSumOfEvenNumbers, corrisponde esattamente al valore atteso nel test (pari a 0).

ID	DESCRIZIONE
T6	Array con un elemento pari a 0

SCENARIO DI TEST
L’array contiene esattamente un elemento che è 0

PRE-REQUISITI
• Array con un unico elemento 0

RISULTATO ATTESO	RISULTATO OTTENUTO
Ci si aspetta che il valore restituito dalla somma sia pari a 0, poiché l’if che controlla che sia maggiore di 0 (strettamente) non passa e ritorna il valore con cui è stata inizializzata la variabile sum.	Il valore della somma, restituito dal metodo calculateSumOfEvenNumbers, corrisponde esattamente al valore atteso nel test (pari a 0).

ID	DESCRIZIONE
T7	N elementi pari e dispari
SCENARIO DI TEST	
L'array contiene N elementi pari e N elementi dispari	
PRE-REQUISITI	
• Un array con più di un elemento • Presenza di N elementi pari ed N elementi dispari	
RISULTATO ATTESO	**RISULTATO OTTENUTO**
Ci si aspetta che il valore restituito dalla somma sia pari alla somma dei valori positivi e pari che compongono l'array, poiché per gli elementi dispari l'if che controlla che sia pari non passa e ritorna il valore con cui è stata inizializzata la variabile sum.	Il valore della somma, restituito dal metodo calculateSumOfEvenNumbers, corrisponde esattamente al valore atteso nel test e rappresenta unicamente la somma, con le dovute trasformazioni, dei valori pari e dispari presenti nell'array.

ID	DESCRIZIONE
T8	N elementi positivi pari ed elementi negativi
SCENARIO DI TEST	
L'array contiene N elementi positivi pari ed N elementi negativi	
PRE-REQUISITI	
• Un array con più di un elemento • Presenza di N elementi positivi pari ed N elementi negativi	
RISULTATO ATTESO	**RISULTATO OTTENUTO**
Ci si aspetta che il valore restituito dalla somma sia pari alla somma dei valori positivi e pari che compongono l'array, poiché per gli elementi minori di 0 l'if che controlla che i numeri siano maggiori di 0 (strettamente) non passa e ritorna il valore con cui è stata inizializzata la variabile sum.	Il valore della somma, restituito dal metodo calculateSumOfEvenNumbers, corrisponde esattamente al valore atteso nel test e rappresenta unicamente la somma, con le dovute trasformazioni, dei valori pari e dispari presenti nell'array.

ID	DESCRIZIONE
T9	N elementi positivi dispari ed elementi negativi

SCENARIO DI TEST
L'array contiene N elementi positivi dispari ed N elementi negativi

PRE-REQUISITI
• Un array con più di un elemento • Presenza di N elementi positivi dispari ed N elementi negativi

RISULTATO ATTESO	RISULTATO OTTENUTO
Ci si aspetta che il valore restituito dalla somma sia pari a 0, poiché non è presente alcun elemento positivo pari: l'if che controlla che sia pari e positivo non passa e ritorna il valore con cui è stata inizializzata la variabile sum.	Il valore della somma, restituito dal metodo calculateSumOfEvenNumbers, corrisponde esattamente al valore atteso nel test (pari a 0).

ID	DESCRIZIONE
T10	N elementi positivi pari e 0

SCENARIO DI TEST
L'array contiene il valore 0 ed elementi tutti positivi e pari

PRE-REQUISITI
• Array con più di un elemento • Uno degli elementi deve essere uguale a 0 • I restanti valori sono tutti positivi e pari

RISULTATO ATTESO	RISULTATO OTTENUTO
Ci si aspetta il valore restituito dalla somma sia proprio la somma dei valori presenti nell'array, escluso lo zero che non rispetta la condizione dell'if esterno. Per i restanti elementi verranno prima eseguite le dovute operazioni e poi verranno aggiunti alla somma.	Il valore della somma, restituito dal metodo calculateSumOfEvenNumbers, corrisponde esattamente al valore atteso nel test.

ID	DESCRIZIONE
T11	N elementi positivi dispari e 0

SCENARIO DI TEST
L'array contiene il valore uguale a zero ed elementi che sono tutti positivi e dispari

PRE-REQUISITI
• Un array con più di un elemento • Uno degli elementi deve essere proprio uguale a 0 • I restanti elementi devono essere tutti positivi e dispari

RISULTATO ATTESO	RISULTATO OTTENUTO
Ci si aspetta che il valore restituito dalla somma sia uguale a 0. I valori dispari e il valore uguale a zero non rispettano la condizione dell'if esterno; quindi, nessuno di questi verrà aggiunto alla somma.	Il valore della somma, restituito dal metodo calculateSumOfEvenNumbers, corrisponde esattamente al valore atteso nel test.

ID	DESCRIZIONE
T12	Elementi negativi e 0

SCENARIO DI TEST
L'array contiene l'elemento pari a zero insieme ad elementi negativi, sia pari che dispari.

PRE-REQUISITI
• Un array con più di un elemento • Uno degli elementi presenti nell'array è uguale a 0 • I restati elementi presenti nell'array possono essere sia pari che dispari ma devono essere tutti negativi

RISULTATO ATTESO	RISULTATO OTTENUTO
Ci si aspetta che il valore restituito dalla somma sia 0. Trattandosi di valori negativi non è rilevante che questi siano pari o dispari, in ogni caso non verranno aggiunti alla somma. Ed essendo che la condizione richiede numeri strettamente positivi lo 0 verrà ignorato in ogni caso.	Il valore della somma, restituito dal metodo calculateSumOfEvenNumbers, corrisponde esattamente al valore atteso nel test.

ID	DESCRIZIONE
T13	Array con un elemento pari a 1
SCENARIO DI TEST	
L'array presenta un solo valore intero e questo è proprio il valore di confine che rende la condizione falsa	
PRE-REQUISITI	
• Array con almeno un elemento • L'elemento deve essere proprio il primo valore dispari, ossia il primo valore per cui la condizione diventa falsa	
RISULTATO ATTESO	**RISULTATO OTTENUTO**
Ci si aspetta che il valore restituito dalla somma sia 0. Trattandosi dell'operatore booleano && è sufficiente che una delle due condizioni non venga rispettata affinché tutto l'if esterno e le condizioni annidate non vegano eseguite.	Il valore della somma, restituito dal metodo calculateSumOfEvenNumbers, corrisponde esattamente al valore atteso nel test.

ID	DESCRIZIONE
T14	Array con un elemento pari a 2
SCENARIO DI TEST	
L'array presenta un solo valore intero è questo è proprio il valore di confine che rende la condizione vera	
PRE-REQUISITI	
• Array con almeno un elemento • L'elemento deve essere il primo valore positivo e pari, ossia il primo valore per cui la condizione diventa vera	
RISULTATO ATTESO	**RISULTATO OTTENUTO**
Ci si aspetta che il valore restituito dalla somma sia esattamente 2 perché il valore fornito è positivo, pari ed entra nell'else finale. Questo valore non subisce manipolazioni prima di essere aggiunto alla somma.	Il valore della somma, restituito dal metodo calculateSumOfEvenNumbers, corrisponde esattamente al valore atteso nel test.

CAPITOLO 3 –SECONDO HOMEWORK

PROPERTY-BASED TESTING – PROPRIETA' TESTATA

Il metodo ritenuto idoneo per lo svolgimento dei property based tests è il metodo containsBoth(), protagonista del primo homework.
La proprietà testata di questo codice è la **correttezza del comportamento** del metodo. Quindi viene sostanzialmente verificato che il metodo **containsBoth()** restituisca lo stesso risultato del metodo **containsBothTest()**, quando viene eseguito con le stesse collezioni come argomenti.
Inizialmente, come set sono state scelte collezioni contenenti rispettivamente 12 e 21 elementi interi in modo da ottenere una statistica equilibrata tra collezioni con elementi in comune e non.
Successivamente sono stati selezionati ulteriori set di dati caratterizzati da elementi che rappresentano casi di test significativi come, per esempio, il comportamento del metodo nel caso di elementi duplicati o nel caso di collezioni di tipi diversi.

PROPERTY-BASED TESTING – CODICE

Come primo test è stato scelto il caso più semplice e generale che è possibile testare con questo metodo.
Vengono generate randomicamente due collezioni, coll1 e coll2. Queste due collezioni presentano dimensioni diverse ma entrambe sono riempite con interi compresi nell'intervallo [-500, 500], estremi inclusi. Attraverso l'annotazione @UniqueElements ci si assicura che i valori generati siano unici e che quindi non ci siano elementi duplicati all'interno di una collezione.

Successivamente alla variabile "expected" viene assegnato il risultato del metodo di test containsBothTest(coll1, coll2), mentre actual contiene il risultato del metodo testato containsBoth(coll1, coll2) della classe CollsUtilsTH.
Attraverso assertEquals viene verificato che il valore delle due variabili sia uguale.

```
private boolean containsBothTest(Collection<?> coll1, Collection<?> coll2) {
    for (Object element : coll1) {
        if (coll2.contains(element)) {
            return true;
        }
    }
    for (Object element : coll2) {
        if (coll1.contains(element)) {
            return true;
        }
    }
    return false;
}
```

Figura 30: metodo containsBothTest()

```
//Test with collections of integers
@Report(Reporting.GENERATED)
@StatisticsReport(format = Histogram.class)
@Property
void testContainsBoth(
        @ForAll @Size(value = 12) Collection< @IntRange(min = -500, max = 500) @UniqueElements Integer> coll1,
        @ForAll @Size(value = 21) Collection< @IntRange(min = -500, max = 500) @UniqueElements Integer> coll2){

    final int matchCount = 0;
    boolean expected = containsBothTest(coll1, coll2);
    boolean actual = CollsUtilsTH.containsBoth(coll1, coll2);
    assertEquals(expected, actual);

    if (expected) {
        Statistics.collect( values: matchCount, 1);
    }else
        Statistics.collect( values: matchCount, 0);
}
```

Figura 31: metodo testContainsBoth()

Successivamente, facendo riferimento ai test sviluppati per il primo homework, si è scelto di implementare i casi ritenuti più significativi attraverso la metodologia dei property based testing in modo da poter condurre anche un confronto tra le due metodologie di test a posteriori.
Sono stati quindi selezionati casi di test per i quali si vuole verificare la presenza di eventuali differenze tra i test scritti inserendo manualmente i valori e i test su valori generati randomicamente, nell'ottica in cui possa venir fuori un valore, o una combinazione di valori, in grado di rompere la suite di test.

Innanzitutto l'attenzione si è concentrata su un caso particolare a cui sono stati dedicati diversi test nel primo homework in modo da testarne tutte le possibili combinazioni (T8: Coll1 con duplicati e Coll2 con duplicati, T9: Coll1 con duplicati e Coll2 senza duplicati, T10: Coll1 con duplicati e Coll2 duplicati N volte, T11: Coll1 con duplicati N volte e Coll2 senza duplicati). In tutti questi casi è stato utilizzato assertTrue perché il risultato atteso era sempre true purché un elemento fosse presente in entrambe le collezioni almeno una volta. Non ha quindi rilevanza la presenza reiterata di un elemento e nemmeno il numero di volte in cui compare in una collezione rispetto ad un'altra.

Questa consapevolezza ha portato alla scrittura di un unico caso di test, generico, dedicato alla presenza di duplicati in entrambe le collezioni. Tuttavia, il comportamento del metodo è stato sempre il medesimo, non evidenziando quindi nessun caso particolare a cui prestare attenzione.

```
//Test with duplicated elements
@Report(Reporting.GENERATED)
@StatisticsReport(format = Histogram.class)
@Property
void testContainsBothWithDuplicatedElements(
        @ForAll @Size(value = 21) Collection<@IntRange(min = -500, max = 500) Integer> coll1,
        @ForAll @Size(value = 21) Collection< @IntRange(min = -500, max = 500) Integer> coll2) {


    final int matchCount = 0;
    boolean expected = containsBothTest(coll1, coll2);
    boolean actual = CollsUtilsTH.containsBoth(coll1, coll2);
    assertEquals(expected, actual);

    if (expected) {
        Statistics.collect( ...values: matchCount, 1);
    }else
        Statistics.collect( ...values: matchCount, 0);
}
```

Figura 32: metodo testContainsBothWithDuplicatesElements()

Visti i risultati ottenuti fino a questo momento si è deciso di optare per test differenti rispetto al primo homework. Si è ritenuto quindi opportuno testare il comportamento del metodo su collezioni molto grandi, nella speranza che, su 1000 elementi generati randomicamente per la coll1 e altri 1000 elementi generati randomicamente per la coll2, possa verificarsi la situazione in cui una particolare combinazione di input possa evidenziare un eventuale caso anomalo dovuto a un bug nel codice o a output discordanti tra il metodo containsBoth() e il metodo containsBothTest().

Per la generazione di collezioni così ampie il range dei valori è stato ampliato e portato a [-5000, 5000], estremi inclusi. L'esecuzione del test ha richiesto maggiori tempi d'attesa ma nessuna situazione anomala si è verificata.

```
//Test with large collections
@Report(Reporting.GENERATED)
@StatisticsReport(format = Histogram.class)
@Property
void testContainsBothWithLargeCollections(
        @ForAll @Size(value = 1000) Collection<@IntRange(min = -5000, max = 5000) Integer> coll1,
        @ForAll @Size(value = 1000) Collection<@IntRange(min = -5000, max = 5000) Integer> coll2) {

    final int matchCount = 0;
    boolean expected = containsBothTest(coll1, coll2);
    boolean actual = CollsUtils.containsBoth(coll1, coll2);
    assertEquals(expected, actual);

    if (expected) {
        Statistics.collect( ...values: matchCount, 1);
    }else
        Statistics.collect( ...values: matchCount, 0);
}
```

Figura 33: metodo testContainsBothWithLargeCollections()

Per una maggiore completezza è stato considerato anche il caso in cui il metodo ha come parametri collezioni di tipo diverso. Secondo il risultato atteso non può esistere un elemento comune ad entrambe le collezioni se il tipo di elementi presenti non è uguale per entrambe. Attraverso il metodo della figura 34 è stata confermata la previsione fatta e nessun caso anomalo è stato riscontrato.

```
//Test with different types of elements (Integer and String)
@Report(Reporting.GENERATED)
@StatisticsReport(format = Histogram.class)
@Property
void testContainsBothWithDifferentTypesOfElements(
        @ForAll @Size(value = 21) Collection<@IntRange(min = -500, max = 500) Integer> coll1,
        @ForAll @Size(value = 21) Collection<String> coll2) {

    final int matchCount = 0;
    boolean expected = containsBothTest(coll1, coll2);
    boolean actual = CollsUtilsTH.containsBoth(coll1, coll2);
    assertEquals(expected, actual);

    if (expected) {
        Statistics.collect( ...values: matchCount, 1);
    }else
        Statistics.collect( ...values: matchCount, 0);
}
```

Figura 34: metodo testContainsBothWithDifferentTypesOfElements()

Infine, nella prospettiva di scrivere test quanto più completi e accurati, è stata presa in considerazione una nuova casistica. È stato infatti testato il comportamento del metodo quando riceve come parametri collezioni che non sfruttano il medesimo metodo contains(). Per testare il comportamento nel caso di diverse implementazioni quindi si è ricorso all'utilizzo di coll1 che è una collezione riempita con valori randomici, che quindi sfrutta il metodo boolean contains(Object o) dell'interfaccia Collecion. Coll2 invece è stata creata esattamente come coll1, ma poi passata come input ad HashSet, andando così a riempire coll2Set. In questo modo per coll2Set verrà utilizzato il metodo contains() della classe HashSet che implementa l'interfaccia Set che a sua volta è un'interfaccia di Collection.

```
//Test with different collection implementations
@Report(Reporting.GENERATED)
@StatisticsReport(format = Histogram.class)
@Property
void testContainsBothWithDifferentCollectionImplementations(
        @ForAll @Size(value = 12) Collection<@IntRange(min = -500, max = 500) Integer> coll1,
        @ForAll @Size(value = 21) Collection <@IntRange(min = -500, max = 500) Integer> coll2){

    Collection<Integer> coll2Set = new HashSet<>(coll2);

    final int matchCount = 0;
    boolean expected = containsBothTest(coll1, coll2Set);
    boolean actual = CollsUtils.containsBoth(coll1, coll2Set);
    assertEquals(expected, actual);

    if (expected) {
        Statistics.collect( ...values: matchCount, 1);
    }else
        Statistics.collect( ...values: matchCount, 0);
}
```

Figura 35: metodo testContainsBothWithDifferentCollectionImplementations()

Come atteso però non si è verificata alcuna situazione anomala.
Non sono, quindi, mai stati riscontrati problemi nel codice e non si è mai verificato un caso in cui il valore di expected si rivelasse essere diverso da quello di actual. L'esecuzione del test property based non ha quindi portato alla luce nessuna situazione particolare che non fosse già stato affronta con i specification based tests; tuttavia, ha fornito risultati interessanti dal punto di vista dell'analisi delle statistiche.

PROPERTY-BASED TESTING – STATISTICHE PSD

Le annotazioni **@Report(Reporting.GENERATED)** e **@StatisticsReport(format = Histogram.class)** specificano che vogliamo generare un report delle statistiche per il metodo di test utilizzando il formato **Histogram** e che vogliamo raccogliere le statistiche generate da jqwik durante l'esecuzione del test.

```
if (expected) {
    Statistics.collect( ...values: matchCount, 1);
}else
    Statistics.collect( ...values: matchCount, 0);
```

Figure 36: collezione delle statistiche

Quando viene eseguito il property-based testing per ogni set di dati generato, si determina se è stata trovata una corrispondenza tra le collezioni coll1 e coll2. Se la corrispondenza viene trovata (expected è true), viene incrementato matchCount di 1 e viene raccolta questa informazione tramite Statistics.collect(matchCount, 1). Questo significa che stiamo registrando che è stata trovata una corrispondenza per questo particolare set di dati.

Se la corrispondenza non è stata trovata (expected è false), raccogliamo il valore 0 tramite Statistics.collect(matchCount, 0). Questo significa che stiamo registrando che non è stata trovata alcuna corrispondenza per il set di dati.

Alla fine dell'esecuzione del test, jqwik raccoglierà le statistiche delle corrispondenze trovate durante tutti i test eseguiti. Le statistiche verranno visualizzate nel report dei test. L'utilizzo di tali statistiche consente di raccogliere informazioni sul numero di corrispondenze trovate durante l'esecuzione dei test, fornendo un'indicazione delle performance e dell'efficacia della logica implementativa.

```
timestamp = 2023-07-29T15:45:07.762651400, [CollsUtilsTest:testContainsBoth] (1000) statistics =
       # | label | count |
    -----|-------|-------|---------------------------------------------------------------------------------
       0 |   0 1 |   503 | ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
       1 |   0 0 |   497 | ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■

timestamp = 2023-07-29T15:45:07.762651400, CollsUtilsTest:testContainsBoth =
                              |-------------------jqwik--------------------
tries = 1000                  | # of calls to property
checks = 1000                 | # of not rejected calls
generation = RANDOMIZED       | parameters are randomly generated
after-failure = PREVIOUS_SEED | use the previous seed
when-fixed-seed = ALLOW       | fixing the random seed is allowed
edge-cases#mode = MIXIN       | edge cases are mixed in
edge-cases#total = 81         | # of all combined edge cases
edge-cases#tried = 81         | # of edge cases tried in current run
seed = -8104605396799171447   | random seed to reproduce generated values
```

Figura 37: statistiche ottenute con collezioni di dimensioni diverse

Questo risultato è stato ottenuto grazie all'utilizzo di due collezioni con dimensioni diverse tra loro: coll1 formata da 12 elementi, coll2 formata da 21 elementi.

Con collezioni di uguale dimensione è stato constatato che le statistiche ottenute sono diversamente distribuite, con una concentrazione importante sullo 0 nel caso in cui entrambe le collezioni siano di 21 elementi

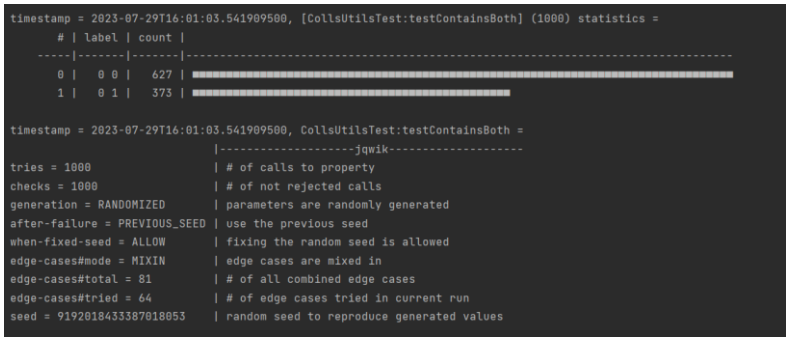

Figura 38: statistiche ottenute con collezioni abbastanza ampie di uguali dimensioni

e una maggiore concentrazione su 1 nel caso in cui le collezioni siano di dimensioni ridotte, per esempio entrambe di 12 elementi:

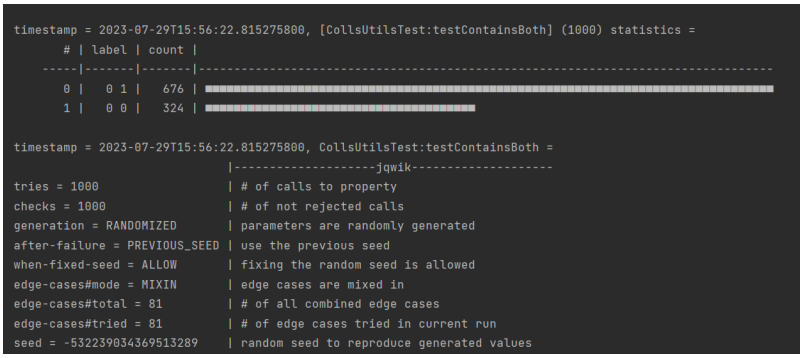

Figura 39: statistiche ottenute con collezioni abbastanza piccole di uguali dimensioni

Nel caso di due collezioni di stessa dimensione pari a 21, ma di tipi differenti (nel nostro caso interi e stringhe), le statistiche presentano la totalità dei valori pari a 0; il che risulta essere scontato ed evidente, data la diversità dei dati presi in considerazione.

Figura 40: statistiche ottenute con collezioni di tipi differenti

Quando, invece, le due collezioni, entrambe di dimensione 21, presentano elementi dello stesso tipo e sono ammessi elementi duplicati, le statistiche rilevano una maggiore concentrazione sull' 1. Questo potrebbe essere spiegato dal fatto che i duplicati potrebbero influenzare la probabilità di trovare elementi in comune; quindi, un maggior numero di corrispondenze rispetto ad una situazione in cui gli elementi sono tutti unici.

```
timestamp = 2023-08-14T15:04:02.082470, [CollsUtilsTHTest:testContainsBothWithDuplicatedElements] (1000) statistics =
       # | label | count |
    -----|-------|-------|--------------------------------------------------------------------------------
       0 |   0 1 |   666 | ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
       1 |   0 0 |   334 | ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■

timestamp = 2023-08-14T15:04:02.082470, CollsUtilsTHTest:testContainsBothWithDuplicatedElements =
                            |--------------------jqwik--------------------
tries = 1000                | # of calls to property
checks = 1000               | # of not rejected calls
generation = RANDOMIZED     | parameters are randomly generated
after-failure = PREVIOUS_SEED | use the previous seed
when-fixed-seed = ALLOW     | fixing the random seed is allowed
edge-cases#mode = MIXIN     | edge cases are mixed in
edge-cases#total = 81       | # of all combined edge cases
edge-cases#tried = 79       | # of edge cases tried in current run
seed = 6834876060345065206  | random seed to reproduce generated values
```

Figura 41: statistiche ottenute con collezioni con elementi duplicati

Per quanto riguarda il caso in cui le due collezioni contengono interi casuali molto grandi, entrambi di dimensione 1000, le statistiche registrano una maggiore concentrazione sul valore 1 rispetto allo 0. Quando le collezioni sono molto grandi, l'intersezione effettiva degli elementi può essere relativamente grande, anche se le collezioni contengono molti elementi unici.

```
timestamp = 2023-07-29T15:45:44.798724900, [CollsUtilsTest:testContainsBothWithLargeCollections] (1000) statistics =
       # | label | count |
    -----|-------|-------|--------------------------------------------------------------------------------
       0 |   0 1 |   941 | ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
       1 |   0 0 |    59 | ■■■■■

timestamp = 2023-07-29T15:45:44.798724900, CollsUtilsTest:testContainsBothWithLargeCollections =
                            |--------------------jqwik--------------------
tries = 1000                | # of calls to property
checks = 1000               | # of not rejected calls
generation = RANDOMIZED     | parameters are randomly generated
after-failure = PREVIOUS_SEED | use the previous seed
when-fixed-seed = ALLOW     | fixing the random seed is allowed
edge-cases#mode = MIXIN     | edge cases are mixed in
edge-cases#total = 81       | # of all combined edge cases
edge-cases#tried = 62       | # of edge cases tried in current run
seed = 2304045745723718827  | random seed to reproduce generated values
```

Figura 42: statistiche ottenute con collezioni molto grandi di interi

Infine, nel caso in cui le collezioni hanno diverse implementazioni (nello specifico, una Collection e un Hashset), la prima con dimensione pari a 12 e la seconda pari a 21, le statistiche risultano essere più o meno bilanciate sui valori 1 e 0. La dimensione delle collezioni può influenzare il numero totale di elementi e la probabilità di trovare elementi in comune tra di loro, come visto nel primo test ("testContainsBoth"). Con una collezione più piccola (12 elementi), il range di elementi è più limitato; quindi, la probabilità di avere valori in comune tra le due collezioni potrebbe essere più alta. Un altro aspetto che potrebbe essere rilevante è quello che l'Hashset non consente duplicati all'interno della stessa collezione, la rimozione degli elementi duplicati dalla collezione generata potrebbe influenzare il bilanciamento delle statistiche. Infatti, tale rimozione potrebbe ridurre la probabilità di avere elementi in comune tra le due collezioni, portando a risultati meno frequenti dell'1 nelle statistiche.

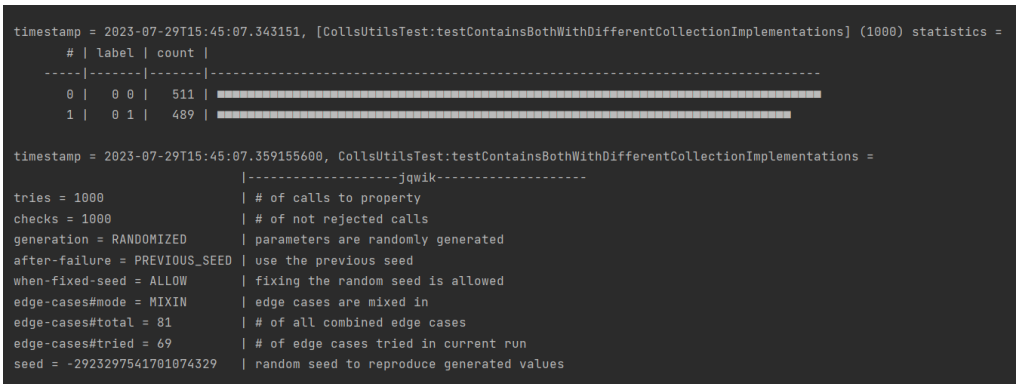

Figura 43: statistiche ottenute con collezioni con implementazione differente

Ovviamente, in tutti i casi affrontati, un aspetto importante da considerare è che le collezioni vengono generate casualmente, e la distribuzione degli elementi potrebbe essere tale da rendere meno probabile che ci siano elementi in comune tra le due. Potrebbe essere una casualità che nel set specifico considerato ci siano meno corrispondenze.