Integrazione e Test di Sistemi Software

Prof.ssa Azzurra Ragone

2023/2024

SOFTWARE CHASERS ME

Emanuele Piemontese, 758242, e.piemontese@stdenti.uniba.it
Michele Minervini, 758230, m.minervini50@stdenti.uniba.it

INDICE

HOMEWORK 1

Introduzione

Two Sum Problem

La classe *TwoSumProblem* contiene il metodo `twoSum()` che riceve in input un array di interi e un intero, restituendo una coppia di indici tali che la somma degli elementi dell'array che si trovano in quelle posizioni sia uguale all'intero specificato come parametro.

Parametri Input

`Values`: un array di interi;
`Target`: è il risultato che si vuole ottenere dalla somma di due elementi dell'array Values.

Parametri Output

`Return`: Un array di coppie di indici tali che la somma dei valori corrispondenti in Values sia uguale a Target.

Esempi

Esempio 1
`Values`: [2,5,9,3]
`Target`: 14
`Return`: [1,2]

Esempio 2
`Values`: [8,8,7,9]
`Target`: 16
`Return`: [0,1], [2,3]

Requisiti

- Se `Values` è null oppure empty, allora la funzione `twosum()` deve restituire null;
- Se `twosum()` non trova nessuna coppia di indici tali che la somma dei valori corrispondenti sia uguale a `Target`, allora ritorna un `Array` vuoto;
- Se `Values` contiene un solo elemento, allora ritorna un `Array` vuoto;
- Se `Values` contiene più coppie di elementi tali che la loro somma sia uguale a `Target`, la funzione `twoSum()` ritorna tutte le coppie.

I requisiti iniziali erano solo parzialmente definiti, per questo motivo abbiamo deciso di dettagliarli ulteriormente. In particolare, la classe non è responsabile di eseguire operazioni critiche e rimanendo in linea con l'implementazione, abbiamo ritenuto opportuno che debba restituire dei *soft values* piuttosto che lanciare delle eccezioni e bloccare l'esecuzione del programma.
La classe utilizza un oggetto di tipo Optional, si tratta di un container che può contenere un valore `non-null` oppure nessun valore, in tal caso l'oggetto sarà `empty`.

La classe è stata introdotta in Java8 con l'obbiettivo di fornire agli sviluppatori uno strumento che permettesse di ridurre il numero di `null` restituiti e quindi di `NullPointerException`.

Originariamente il codice restituiva un solo elemento di tipo `Optional`, questo rendeva impossibile rispettare i requisiti, in quanto è richiesto che vengano restituite tutte le coppie individuate. In questo modo i test sarebbero sempre falliti, abbiamo quindi sostituito il valore di ritorno con un `Array` di `Optional`.

Specification-based testing

1.Comprendere i requisiti

Goal: l'obiettivo del metodo `twoSum()` è quello di restituire coppie di indici tali che la somma degli elementi corrispondenti in un array si uguale ad un valore target.

Input:
a) `Values`: un array di interi;
b) `Target`: è il risultato che si vuole ottenere dalla somma di due elementi dell'array Values

Output:
il programma restituisce un array di coppie.

2.Esplora cosa fa il programma per vari input

Abbiamo effettuato alcuni semplici test per comprendere meglio il funzionamento del programma.

```
@Test
void casoSemplice() {
    int[] array = {2,3};
    assertEquals(List.of(Optional.of(Pair.of( left: 0,  right: 1))), TwoSumProblem.twoSum(array, target: 5));
}

@Test
void casoComplesso() {
    int[] array = {1,4,2,9,8,3};
    assertEquals(List.of(Optional.of(Pair.of( left: 1,  right: 5))), TwoSumProblem.twoSum(array, target: 7));
}

@Test
void casoNumeriNegativi() {
    int[] array = {2,-3,1,8,4,9,-9};
    assertEquals(List.of(Optional.of(Pair.of( left: 1,  right: 6))), TwoSumProblem.twoSum(array, target: -12));
}
```

3.Esplora input, output e identifica le partizioni

Input individuali

Values
Null
Empty
Array con un elemento
Array con più di un elemento

Target
Intero maggiore di 0
Intero minore di 0
Intero uguale a 0

Combinazioni di input

Values,Target
Values con più di un elemento, Target maggiore di 0 (una coppia di positivi)
Values con più di un elemento, Target maggiore di 0 (una coppia con un positivo e un negativo)
Values con più di un elemento, Target minore di 0 (una coppia di negativi)
Values con più di un elemento, Target minore di 0 (una coppia con un positivo e un negativo)
Values con più di un elemento, Target uguale a 0 (una coppia)
Values con più di un elemento, Target maggiore di 0 (più coppie)
Values con più di un elemento, Target minore di 0 (più coppie)
Values con più di un elemento, Target uguale a 0 (più coppie)
Values con più di un elemento, non contiene coppie

Classi di output

Array di coppie
Null
Empty
Array con una coppia
Array con più coppie

4.Identificare i boundary cases

Per identificare i boundary case bisogna assicurarsi che il programma funzioni correttamente per i casi limite. I casi limite individuati sono 2:

- Values contiene più di un elemento ed una coppia come soluzione;
- Values contiene più di un elemento e nessuna soluzione.

Tutti i casi di test individuati erano già stati coperti, non è quindi necessario includere altri test nella test suite.

5.Elaborare i casi di test

Il passo successivo è stato la creazione della test suite in linguaggio naturale. Abbiamo deciso di realizzare i casi di test in modo indipendente, ognuno di noi ha realizzato una test suite che poi abbiamo confrontato. Questo confronto ha permesso di non considerare alcuni casi ritenuti superflui e unire i casi particolari trovati da ognuno.

a) Casi eccezionali

T1: Values è null (la funzione ritorna null);
T2: Values è empty (la funzione ritorna null);
T3: Values con un elemento (la funzione ritorna un array empty);

b) Values contiene più di un elemento e Target maggiore di zero

T4: Una coppia di positivi
T5: Una coppia di un positivo e un negativo
T6: Più coppie

c) Values contiene più di un elemento e Target minore di zero

T7: Una coppia di negativi
T8: Una coppia di un positivo e un negativo
T9: Più coppie

d) Values contiene più di un elemento e Target uguale a zero

T10: Una coppia
T11: Più coppie

T12: Values con più di un elemento ma non contiene coppie (la funzione ritorna un array empty);

6.Automatizzare i casi di test

Il passo successivo prevedeva l'automatizzazione dei casi di test attraverso l'ausilio di `JUnit`. Viene di seguito mostrata una schematizzazione dei test effettuati mediante una tabella riassuntiva.

Class Name:	**TwoSumProblem**
Group Name	**Software Chasers ME**
Test Case Description	**Realizzazione di una test suite per effettuare un Specification-based testing**
Class Test Name	**TwoSumProblemTest**
Date of creation:	**1/12/2023**
Date of review:	

TEST CASE ID	TEST CASE	TEST INPUT	EXPECTED RESULT	ACTUAL RESULT	STATUS (PASS/FAIL)	NOTE
T1	Values è null, la funzione ritorna null	Values = Null	Null	Empty Array	FAIL	
T2	Values è empty, la funzione ritorna null	Values = empty	Null	Empty Array	FAIL	
T3	Values con un elemento, la funzione ritorna un array empty	Values.size() == 1	Empty Array	Empty Array	PASS	
T4	Values contiene piu' di un elemento e target maggiore di 0, una coppia	Values = [2, -5, 16, 4, -11]; Target=18	[0, 2]	[0, 2]	PASS	
T5	Values contiene piu' di un elemento e target maggiore di 0, una coppia	Values = [10, -23, 15, 2, -8]; Target=7	[2, 4]	[2, 4]	PASS	
T6	Values contiene piu' di un elemento e target maggiore di 0, più coppie	Values = [6, 5, -2, 5, -11]; Target=3	[1,2], [2,3]	[1,2]	FAIL	
T7	Values contiene più di un elemento e Target minore di zero, una coppia di	Values = [-1,12,-1,80] ; Target = -2	[0,2]	[0,2]	PASS	
T8	Values contiene più di un elemento e Target minore di zero, una coppia di un	Values = [5,40,-6,5,2] ; Target = -4	[2,4]	[2,4]	PASS	
T9	Values contiene più di un elemento e Target minore di zero, più coppie	Values = [2,-4,-30,-8,-6,9,24] ; Target = -6	[0,3], [2,6]	[0,3]	FAIL	
T10	Values contiene più di un elemento e Target uguale a zero, una coppia	Values = [-1,4,1,7] ; Target = 0	[0,2]	[0,2]	PASS	
T11	Values contiene più di un elemento e Target uguale a zero, più coppie	Values = [-2,20,-2,-20,-6,6,1] ; Target = 0	[1,3], [4,5]	[1,3]	FAIL	
T12	Values con più di un elemento ma non contiene coppie	Values = [9, 40, -1, 28] ; Target = 10	Empty Array	Empty Array	PASS	

Per facilitare la scrittura dei test abbiamo creato la funzione `createArray` la cui responsabilità è quella di creare un array di oggetti `Optional` in modo tale da non doverlo fare per ognuno dei test.

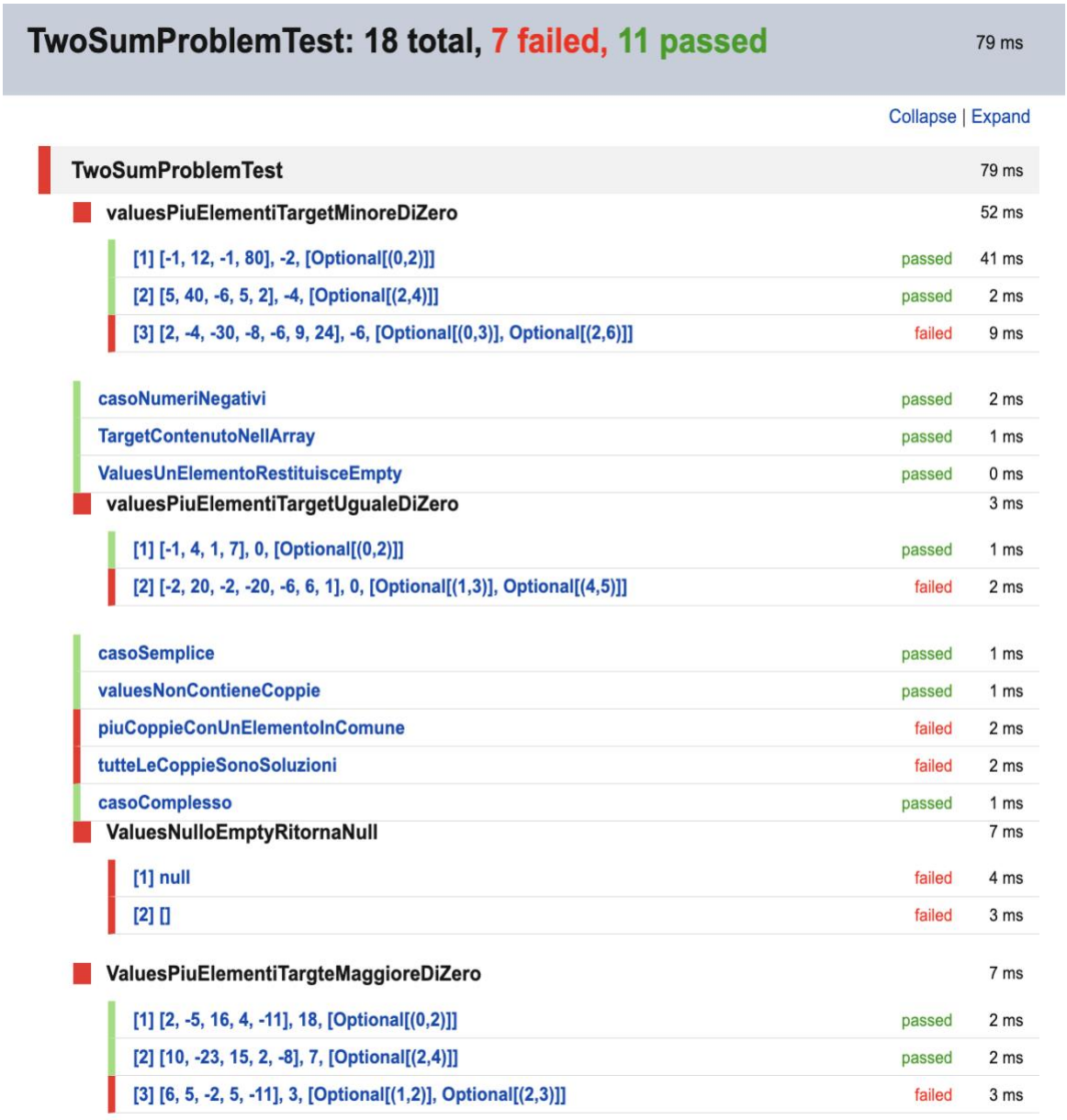
```
42      //metodo per la creazione di un array di oggetti Optional
        11 usages
43 @    public static ArrayList<Optional> createArray(int[] coppie){
44          ArrayList<Optional> arrayOfOptional = new ArrayList<>();
45          for(int i=0; i<coppie.length; i += 2){
46              arrayOfOptional.add(Optional.of(Pair.of(coppie[i], coppie[i+1])));
47          }
48          return  arrayOfOptional;
49      }
```

7.Aumenta la suite di test con creatività ed esperienza

T13: Target uguale a un elemento dell'array (Target=3: [1, 12, 3, 4, 0] --> [2,4])
T14: Tutte le possibili coppie di Values sono soluzioni al problema (Target = 4: [2,2,2,2] --> restituisce tutte le possibili combinazioni di coppie)
T15: Ci sono più coppie con un elemento in comune (Target=4: [5,1,3,1,6] --> [1,2], [2,3])

TEST CASE ID	TEST CASE	TEST INPUT	EXPECTED RESULT	ACTUAL RESULT	STATUS (PASS/FAIL)	NOTE
T13	Target uguale a un elemento dell'array	Values = [1, 12, 3, 4, 0] ; Target = 3	[2,4]	[2,4]	PASS	
T14	Tutte le possibili coppie di Values sono soluzioni al problema	Values = [2,2,2,2] ; Target = 4	[0,1], [0,2], [1,2], [0,3], [1,3], [2,3]	[0,1]	FAIL	
T15	Ci sono più coppie con un elemento in comune	Values = [5,1,3,1,6] ; Target = 4	[1,2], [2,3]	[1,2]	FAIL	

Risultati dei Test, Correzioni e Considerazioni



Sono falliti 7 sui 15 (+3 di prova) test previsti, in particolare i test:

- T1, T2: Tutti e tre i test avevano come valore atteso `Null` ma come valore effettivo di ritorno un `Empty Array`; Il problema dipendeva dalla mancanza di un controllo sui dati in ingresso; è stato risolto aggiungendo un'if che verifichi i dati in input restituendo correttamente `Null` quando previsto;
- T6, T9, T11, T15: I test avevano come valore atteso più coppie ma il metodo restituisce sempre una sola coppia, in particolare la prima identificata; Il problema dipendeva dal fatto che il metodo restituiva il risultato non appena una coppia era individuata; Il problema è stato risolto eliminando il `return` definito all'interno del ciclo in cui veniva effettuata la ricerca;
- T14: Il test fallisce poiché il metodo non è capace di individuare tutte le possibili coppie, la funzione non teneva conto di tutte le occorrenze di uno stesso valore ma si limitava alla prima individuata. Il problema è stato risolto includendo un ulteriore iterazione all'interno di quella già presente. Inoltre, `l'HashMap` utilizzato per tenere traccia dei valori e la loro posizione nell'array, ora contine un `Integer` e un `ArrayList` di `Integer` (prima `Integer`, `Integer`) in modo da tener traccia di tutte le occorrenze del valore come array di indici.

Classe prima della correzione:

```java
public final class TwoSumProblem {
    no usages
    private TwoSumProblem() {
    }

    13 usages
    public static ArrayList<Optional> twoSum(final int[] values, final int target) {
        HashMap<Integer, Integer> valueToIndex = new HashMap<>();
        ArrayList<Optional> result = new ArrayList<>();
        for (int i = 0; i < values.length; i++) {
            final var rem = target - values[i];
            if (valueToIndex.containsKey(rem)) {
                result.add(Optional.of(Pair.of(valueToIndex.get(rem), i)));
                return result;
            }
            if (!valueToIndex.containsKey(values[i])) {
                valueToIndex.put(values[i], i);
            }
        }
        return result;
    }
}
```

Classe dopo la correzione:

```java
public final class TwoSumProblem {
    no usages
    private TwoSumProblem() {
    }
    12 usages
    public static ArrayList<Optional> twoSum(final int[] values, final int target) {
        // Mappa che tiene traccia degli indici associati ai valori.
        HashMap<Integer, ArrayList<Integer>> valueToIndices = new HashMap<>();
        // Lista risultante contenente le coppie di indici.
        ArrayList<Optional> result = new ArrayList<>();
        // Se l'array è nullo o vuoto, restituisci null.
        if (values == null || values.length == 0) {
            return null;
        }
        // Itera su ogni elemento dell'array.
        for (int i = 0; i < values.length; i++) {
            // Calcola il restante (rem) necessario per raggiungere il target.
            final var rem = target - values[i];
            // Se il rem è presente nella mappa degli indici, aggiungi tutte le coppie possibili.
            if (valueToIndices.containsKey(rem)) {
                // Iterazione necessaria per identificare tutte le coppie.
                for (int index : valueToIndices.get(rem)) {
                    result.add(Optional.of(Pair.of(index, i)));
                }
            }
            // Se il valore corrente non è presente nella mappa, aggiungi l'indice corrente.
            if (!valueToIndices.containsKey(values[i])) {
                valueToIndices.put(values[i], new ArrayList<>());
            }
            valueToIndices.get(values[i]).add(i);
        }
        return result;
    }
}
```

Di seguito mostriamo la tabella riassuntiva per i test case e i risultati dei test in seguito alle modifiche sul codice:

Class Name:	**TwoSumProblem**
Group Name	**Software Chasers ME**
Test Case Description	**Realizzazione di una test suite per effettuare un Specification-based testing**
Class Test Name	**TwoSumProblemTest**
Date of creation:	**1/12/2023**
Date of review:	**15/12/2023**

TEST CASE ID	TEST CASE	TEST INPUT	EXPECTED RESULT	ACTUAL RESULT	STATUS (PASS/FAIL)	NOTE
T1	Values è null, la funzione ritorna null	Values = Null	Null	Null	PASS	
T2	Values è empty, la funzione ritorna null	Values = empty	Null	Null	PASS	
T3	Values con un elemento, la funzione ritorna un array empty	Values.size() == 1	Empty Array	Empty Array	PASS	
T4	Values contiene piu' di un elemento e target maggiore di 0, una coppia di positivi	Values = [2, -5, 16, 4, -11]; Target=18	[0, 2]	[0, 2]	PASS	
T5	Values contiene piu' di un elemento e target maggiore di 0, una coppia di un positivo e un negativo	Values = [10, -23, 15, 2, -8]; Target=7	[2, 4]	[2, 4]	PASS	
T6	Values contiene piu' di un elemento e target maggiore di 0, più coppie	Values = [6, 5, -2, 5, -11]; Target=3	[1,2], [2,3]	[1,2], [2,3]	PASS	
T7	Values contiene più di un elemento e Target minore di zero, una coppia di negativi	Values = [-1,12,-1,80] ; Target = -2	[0,2]	[0,2]	PASS	
T8	Values contiene più di un elemento e Target minore di zero, una coppia di un positivo e un negativo	Values = [5,40,-6,5,2] ; Target = -4	[2,4]	[2,4]	PASS	
T9	Values contiene più di un elemento e Target minore di zero, più coppie	Values = [2,-4,-30,-8,-6,9,24] ; Target = -6	[0,3], [2,6]	[0,3], [2,6]	PASS	
T10	Values contiene più di un elemento e Target uguale a zero, una coppia	Values = [-1,4,1,7] ; Target = 0	[0,2]	[0,2]	PASS	
T11	Values contiene più di un elemento e Target uguale a zero, più coppie	Values = [-2,20,-2,-20,-6,6,1] ; Target = 0	[1,3], [4,5]	[1,3], [4,5]	PASS	
T12	Values con più di un elemento ma non contiene coppie	Values = [9, 40, -1, 28] ; Target = 10	Empty Array	Empty Array	PASS	
T13	Target uguale a un elemento dell'array	Values = [1, 12, 3, 4, 0] ; Target = 3	[2,4]	[2,4]	PASS	
T14	Tutte le possibili coppie di Values sono soluzioni al problema	Values = [2,2,2,2] ; Target = 4	[0,1], [0,2], [1,2], [0,3], [1,3], [2,3]	[0,1], [0,2], [1,2], [0,3], [1,3], [2,3]	PASS	
T15	Ci sono più coppie con un elemento in comune	Values = [5,1,3,1,6] ; Target = 4	[1,2], [2,3]	[1,2], [2,3]	PASS	

Risultati dei test dopo le correzioni:

TwoSumProblemTest: 18 total, 18 passed 60 ms

Collapse | Expand

TwoSumProblemTest		60 ms
valuesPiuElementiTargetMinoreDiZero		41 ms
[1] [-1, 12, -1, 80], -2, [Optional[(0,2)]]	passed	39 ms
[2] [5, 40, -6, 5, 2], -4, [Optional[(2,4)]]	passed	1 ms
[3] [2, -4, -30, -8, -6, 9, 24], -6, [Optional[(0,3)], Optional[(2,6)]]	passed	1 ms
casoNumeriNegativi	passed	1 ms
TargetContenutoNellArray	passed	1 ms
ValuesUnElementoRestituisceEmpty	passed	1 ms
valuesPiuElementiTargetUgualeDiZero		3 ms
[1] [-1, 4, 1, 7], 0, [Optional[(0,2)]]	passed	2 ms
[2] [-2, 20, -2, -20, -6, 6, 1], 0, [Optional[(1,3)], Optional[(4,5)]]	passed	1 ms
casoSemplice	passed	1 ms
valuesNonContieneCoppie	passed	1 ms
piuCoppieConUnElementoInComune	passed	1 ms
tutteLeCoppieSonoSoluzioni	passed	1 ms
casoComplesso	passed	2 ms
ValuesNulloEmptyRitornaNull		4 ms
[1] null	passed	2 ms
[2] []	passed	2 ms
ValuesPiuElementiTargteMaggioreDiZero		3 ms
[1] [2, -5, 16, 4, -11], 18, [Optional[(0,2)]]	passed	1 ms
[2] [10, -23, 15, 2, -8], 7, [Optional[(2,4)]]	passed	1 ms
[3] [6, 5, -2, 5, -11], 3, [Optional[(1,2)], Optional[(2,3)]]	passed	1 ms

Structural testing

Dopo aver completato la fase di Specification Based Testing siamo passati allo Structural Testing. Il primo passo è stato eseguire la test suite con uno strumento di code coverege per verificare la copertura raggiunta.

Di seguito il risultato dell'analisi ottenuta tramite Intellij.

Class	Class, %	Method, %	Branch, %	Line, %
TwoSumProblem	100% (1/1)	100% (1/1)	100% (12/12)	100% (14/14)

```java
import java.util.HashMap;
import java.util.Optional;
import org.apache.commons.lang3.tuple.Pair;
import java.util.ArrayList;

public final class TwoSumProblem {
    private TwoSumProblem() {
    }

    public static ArrayList<Optional> twoSum(final int[] values, final int target) {
        HashMap<Integer, ArrayList<Integer>> valueToIndices = new HashMap<>();
        ArrayList<Optional> result = new ArrayList<>();
        //Controllo su values
        if(values == null || values.length == 0){
            return null;
        }
        for (int i = 0; i < values.length; i++) {
            final var rem = target - values[i];
            if (valueToIndices.containsKey(rem)) {
                //Iterazione necessaria per identificare tutte le coppie
                for (int index : valueToIndices.get(rem)) {
                    result.add(Optional.of(Pair.of(index, i)));
                }
            }
            if (!valueToIndices.containsKey(values[i])) {
                valueToIndices.put(values[i], new ArrayList<>());
            }
            valueToIndices.get(values[i]).add(i);
        }
        return result;
    }
}
```

Criterio di code coverage

Line Covreage e Branch Coverage

Lo strumento di analisi mostra come sia stata raggiunta la Line Coverage del 100%, questo è il criterio di valutazione più debole possibile. Anche la Branch Coverage è raggiuta al 100%. Abbiamo quindi voluto verificare se si potessero considerare criteri più forti.

Path Covreage

La Path Coverage è la più forte delle condizioni, ma molto spesso impossibile o troppo costosa da raggiungere. Normalmente si parla di 2^n possibili combinazioni con n: numero delle condizioni nel programma. Nel nostro caso 2^4 Combinazioni senza considerare le iterazioni che porterebbero ad un aumento consistente del numero di test da effettuare.

MC/DC Coverage

Avremmo potuto considerare l'MC/DC Coverage che permette di raggiungere il giusto compromesso tra il numero di bug individuabili e il costo per costruire la test suite ma nel nostro caso le condizioni sono tra loro indipendenti, inoltre gli `if` presenti all'interno del ciclo `for` contengono una sola condizione binaria: Per raggiungere una copertura al 100% con l'MC/DC sarebbe sufficiente eseguire 2 test (N + 1) per ognuna delle condizioni (7 test in totale). Dovremmo quindi verificare i casi in cui queste siano vere o false ricadendo così nella branch coverage.

C+B Covreage

Il criterio di code coverage più forte che possiamo raggiungere è il Condition + Branch Coverage.
Il criterio di branch coverege è stato raggiunto, come precedentemente mostrato, al 100%, lo stesso anche per la condition coverage:

- `if(values == null || values.lenght() == 0)`: per la condition coverage sono necessari soli 2 test, quando una condizione è vera l'altra sarà necessariamente falsa. È quindi sufficiente testare values null e values con lunghezza pari a zero. Tali test sono già stati inclusi nella test suite;
- `if(valueToIndex.containsKey(rem)){...}`: in questo caso la condizione è vera quando una coppia è individuata come soluzione, falsa quando non ne viene trovata alcuna;
- `if(!valueToIndices.containsKey(values[i])){...}`: la condizione serve per tener traccia dei valori presenti all'interno di `Values`, è vera la prima volta che si incontra un particolare numero in `Values`, falsa se si incontra un numero ripetuto in `Values`.

Per il secondo e terzo `if` (avendo una sola condizione binaria) la branch e condition coverage coincidono. Tutte le condizioni sono già state coperte nella test suite sviluppata con gli specification based testing.
È stata raggiunta una copertura del 100% per la condition + branch coverage infatti:

$$c + b = \frac{branches\ covered\ +\ conditions\ covered}{number\ of\ branches\ +\ number\ of\ conditions} \times 100\% = \frac{12+8}{12+8} \times 100\% = 100\%$$

Loops boundary adequacy criterion

Inoltre, nella funzione sono presenti 2 iterazioni, per ognuna di queste bisogna verificare i seguenti casi:

- Prima iterazione

```
for (int i = 0; i < values.length; i++) {...}
```

- il loop è eseguito 0 volte: questo caso si verifica quando "`values==null || values.length==0`";
- il loop è eseguito una volta: questo caso si verifica quando la stringa ha lunghezza 1;
- il loop è eseguito più volte: questo caso si verifica quando la lunghezza della stringa è maggiore di 1.

- Seconda iterazione

```
for (int index : valueToIndices.get(rem)) {...}
```

- il loop è eseguito 0 volte: questo caso si verifica quando nell'iterazione corrente (questo for è annidato nel primo) non è stata trovata una coppia;
- il loop è eseguito una volta: questo caso si verifica quando l'attuale valore di `rem` è stato incontrato una sola volta (verrà formata una sola coppia contenente `rem`);
- il loop è eseguito più volte: questo caso si verifica quando l'attuale valore di `rem` è stato incontrato più volte (verranno formate più coppie contenenti `rem`).

Tutti i casi sono già stati coperti nella test suite sviluppata con gli specification based testing.

Nonostante la code coverege raggiunta garantisca una copertura totale del codice, questo non garantisce che il codice sia privo di bug, abbiamo quindi riesaminato nuovamente la classe per identificare eventuali bug non emersi finora con particolare focus sui boundary case.
I boundary case individuati in seguito all'analisi del codice sono i seguenti:

- `if(valueToIndex.containsKey(rem)){...}` è sempre vero: Questo caso si verifica quando tutte le coppie in `Values` sono soluzioni;
- `if(valueToIndex.containsKey(rem)){...}` è sempre falso: Questo caso si verifica quando non ci sono coppie che rispettano la condizione;
- `if(!valueToIndices.containsKey(values[i])){...}` è sempre vero: Questo caso si verifica quando tutti i numeri in `Values` sono diversi;
- `if(!valueToIndices.containsKey(values[i])){...}` è sempre falso (ad eccezione della prima iterazione in cui la condizione sarà sempre vera): Questo caso si verifica quando tutti i numeri in `Values` sono uguali.

Tutti i casi di test emersi sono già stati inclusi nella test suite definita attraverso gli specification based testing, non è stato quindi necessario includere altri casi di test.

Notiamo che alcuni di questi test, in particolare il caso in cui tutte le coppie sono soluzioni e tutti i numeri in Values sono uguali, sono stati testati con il test case "T14" definito nel passo 7 degli specification based testing. Questo caso di test è stato quindi aggiunto grazie alla nostra creatività ed esperienza, ma anche se non avessimo pensato a questo test nella fase precedente, sarebbe comunque stato individuato con gli structural testing, grazie al procedimento rigoroso e schematico che abbiamo mantenuto per l'individuazione dei test.

HOMEWORK 2

Introduzione

LetterOrderChecker

La classe `LetterOrderChecker` contiene il metodo `checkOrder()` il quale verifica se tutte le occorrenze di una lettera in una stringa sono posizionate prima di tutte le occorrenze di una seconda lettera all'interno della stessa stringa.

Parametri Input

`str`: stringa sulla quale si vuole effettuare il controllo;
`firstLetter`: la prima lettera che deve essere trovata nella stringa;
`secondLetter`: la seconda lettera che deve essere trovata nella stringa.

Parametri Output

`Boolean`: la funzione ritorna

- true: se tutte le occorrenze di `firstLetter` precedono le occorrenze di `secondLetter;`
- false: se almeno un'occorrenza di `secondLetter` precede un'occorrenza di `firstLetter;`

Esempi

Esempio 1
`str`: abzcdefxg
`firstLetter`: z
`secondLetter`: x
`return`: true

Esempio 2
`str`: axbxcdzefgxh
`firstLetter`: x
`secondLetter`: z
`return`: false

Requisiti

- Se `str` è empty o null, allora la funzione `checkOrder()` ritorna `false;`
- Se `str` ha lunghezza pari a 1, allora la funzione `checkOrder()` ritorna `false;`
- Se in `str` non è presenta almeno un'occorrenza di `firstLetter` o `secondLetter`, allora la funzione `checkOrder()` ritorna `false;`
- Se in `str` è presente almeno un'occorrenza di `firstLetter` e `secondLetter`, allora la funzione `checkOrder()` ritorna `false` se `secondLetter` compare almeno una volta prima di `firstLetter;`
- Se in `str` è presente almeno un'occorrenza di `firstLetter` e `secondLetter`, allora la funzione `checkOrder()` ritorna `true` se tutte le occorrenze di `secondLetter` compaiono dopo tutte le occorrenze di `firstLetter;`
- Se `str` contiene numeri o caratteri speciali la funzione opera normalmente, considerando questi come fossero lettere;

- Se `firstLetter` e `secondLetter` sono uguali, allora la funzione `checkOrder()` ritorna `false;`

Property based testing

Abbiamo individuato tre partizioni con proprietà differenti, ovvero:

- pass: tutte le occorrenze di firstLetter sono posizionate prima di secondLetter, il metodo restituirà true;
- fail: almeno un'occorrenza di secondLetter è posizionata prima di firstletter, il metodo restituirà false;
- invalid: la stringa è vuota o null, ha lunghezza pari a 1 o non contiene firstLetter o secondLetter, il metodo restituirà false.

Per ognuna delle partizioni sono state definite proprietà di test differenti.

Partizione 1: pass

I test sono stati realizzati costruendo 1000 stringhe con una lunghezza compresa tra 2 e 15 ed escludendo le lettere 'a' e 'z' che sono state scelte in modo arbitrario come parametri per il metodo. È stato poi generato un indice casuale tra 1 e la lunghezza della stringa. Con la funzione `generateIndices` sono state generate le posizioni in cui inserire le lettere 'a' e 'z' all'interno della stringa. Tutte le 'a' vengono inserite prima dell'indice e le 'z' dopo. Costruendo la stringa in questo modo, è garantito che il metodo debba restituire `true`.

```
26      // PASS: tutte le occorenze di firstLetter sono posizionate prima di secondLetter
27      @Property
28      @Report(Reporting.GENERATED)
29      @StatisticsReport(format = Histogram.class)
30      void passCase(
31              @ForAll @StringLength(min = 2, max = 15) @CharRange(from = 'b', to = 'y') String string) {
32
33          Random random = new Random();
34          int indice = random.nextInt( origin: 1, string.length());
35
36          // Genera indici per il posizionamento di firstLetter e secondLetter
37          List<Integer> indicesFirstLetter = generateIndices( startIndex: 0,  endIndex: indice - 1, indice);
38          List<Integer> indicesSecondLetter = generateIndices(
39                  indice,  endIndex: string.length() - 1,  size: string.length() - indice);
40
41          // Modifica la stringa in base agli indici generati posizionando firstLetter e secondLetter
42          StringBuilder modifiedString = new StringBuilder(string);
43          indicesFirstLetter.forEach(i -> modifiedString.setCharAt(i, firstLetter));
44          indicesSecondLetter.forEach(i -> modifiedString.setCharAt(i, secondLetter));
45
46          // Verifica che l'ordine sia rispettato
47          assertTrue(LetterOrderChecker.checkOrder(modifiedString.toString(), firstLetter, secondLetter));
48
49          Statistics.label("Lunghezza").collect(string.length());
50          Statistics.label("Indice").collect(indice);
51          Statistics.label("Numero di A").collect(indicesFirstLetter.size());
52          Statistics.label("Numero di z").collect(indicesSecondLetter.size());
53      }
```

Partizione 2: fail

Come nel caso precedente sono state generate 1000 stringhe con una lunghezza compresa tra 2 e 15 ed escludendo sempre le lettere ‘a’ e ‘z’. In questo caso è stata creata una sola lista di indici dove posizionare `firstLetter` e `secondletter`. Iterando sugli indici generati, abbiamo inserito le lettere in modo casuale: ‘a’ sé `nextBoolean` restituisce true, altrimenti abbiamo inserito ‘z’. Per garantire che il metodo debba restituire `false`, la prima lettera della stringa è stata trasformata in una ‘z’ mentre l’ultima in una ‘a’. In questo modo è garantito che almeno un’occorrenza di z preceda a e quindi che il metodo deve restituire false. L’aggiunta di altre ‘a’ e ‘z’ casuali all'interno della stringa è stata fatta con lo scopo di rendere i dati di input dei test quanto più simili possibili a quelli reali.

Nel caso in cui la lunghezza della stringa fosse 2, non era necessario aggiungere altre ‘a’ e ‘z’ in modo casuale, era sufficiente modificare la stringa in ‘za’.

```
55      // FAIL: almeno un occorenza di secondLetter è posizionata prima di secondLetter
56      @Property
57      @Report(Reporting.GENERATED)
58      @StatisticsReport(format = Histogram.class)
59      void failCase(
60              @ForAll @StringLength(min = 2, max = 15) @CharRange(from = 'b', to = 'y') String string) {
61
62          Random random = new Random();
63          int numeroA = 1;
64          int numeroZ = 1;
65
66          // Modifica la stringa in base agli indici generati posizionando firstLetter e secondLetter
67          StringBuilder modifiedString = new StringBuilder(string);
68          if(string.length() > 2){
69              // Genera indici per il posizionamento di firstLetter e secondLetter:
70              List<Integer> indices = generateIndices(
71                      startIndex: 1, endIndex: string.length() - 2, size: string.length() - 2);
72              for(int i = 0; i < indices.size(); i++){
73                  if(random.nextBoolean()){
74                      modifiedString.setCharAt(i, firstLetter);
75                      numeroA++;
76                  }
77                  else{
78                      modifiedString.setCharAt(i, secondLetter);
79                      numeroZ++;
80                  }
81              }
82              Statistics.label("Indici").collect(indices.size());
83          }
85          //Modifica della stringa inserendo firstletter in ultima posizione e secondletter in prima posizione
86          modifiedString.setCharAt( index: 0, secondLetter);
87          modifiedString.setCharAt( index: string.length() - 1, firstLetter);
88          // Verifica che l'ordine non è rispettato
89          assertFalse(LetterOrderChecker.checkOrder(modifiedString.toString(), firstLetter, secondLetter));
90          Statistics.label("Lunghezza").collect(string.length());
91          Statistics.label("Numero di A").collect(numeroA);
92          Statistics.label("Numero di Z").collect(numeroZ);
93          Statistics.label("Numero di Z prima di A").
94                  collect(countLetterBeforeAnother(modifiedString, firstLetter, secondLetter));
95      }
```

Partizione 3: invalid

Per questa partizione abbiamo generato 1000 stringhe casuali con una lunghezza minima 0 (di default) e massima di 15 caratteri. Abbiamo verificato che il programma intercettasse correttamente le stringhe vuote, di lunghezza uno e quelle in cui almeno uno dei caratteri '`a`' o '`z`' non comparisse restituendo `false` come risultato.

```
96      // INVALID: la stringa è vuota o null, ha lunghezza pari a 1 o non contiene firstLetter o secondLetter
97      @Property
98      @Report(Reporting.GENERATED)
99      void invalidCase(
100             @ForAll @StringLength(max = 15) @CharRange(from = 'a', to = 'z') String string) {
101
102         // Verifica se la stringa è vuota, ha lunghezza pari a 1 o non contiene firstLetter o secondLetter
103         if (string.isEmpty() || string.length() == 1 ||
104                 !string.contains(String.valueOf(firstLetter)) || !string.contains(String.valueOf(secondLetter))) {
105             assertFalse(LetterOrderChecker.checkOrder(string, firstLetter, secondLetter));
106         }
107
108         Statistics.label("Vuota").collect(string.isEmpty());
109         Statistics.label("Lunghezza 1").collect( ...values: string.length() == 1);
110         Statistics.label("Contiene first e second letter").collect(
111                 ...values: string.contains(String.valueOf(firstLetter)) && string.contains(String.valueOf(secondLetter)));
112         Statistics.label("Risultato").collect(LetterOrderChecker.checkOrder(string, firstLetter,secondLetter));
113     }
114
```

Per la generazione degli indici ci siamo avvalsi della classe `Arbitraries` e abbiamo deciso di creare la funzione `generateIndices`, la quale genera indici in modo random con valori compresi tra `startIndex` e `endIndex` e dimensione tra 1 e `size`.

```
138     // metodo per generare indici random
        3 usages
139     private List<Integer> generateIndices(int startIndex, int endIndex, int size) {
140         return Arbitraries.integers().between(startIndex, endIndex).list().ofMaxSize(size).ofMinSize(1).sample();
141     }
142
```

Code Coverage

Dopo aver utilizzato gli strumenti di verifica della code coverage, abbiamo notato che una condizione non era coperta. In particolare, non erano generate e quindi testate stringhe `null`.

```java
package org.example.Homework2;

import java.util.ArrayList;
import java.util.List;

public class LetterOrderChecker {
    public static boolean checkOrder(String str, char firstLetter, char secondLetter) {

        // controlla se la stringa è nulla, vuota o ha lunghezza 1
        if (str == null || str.isEmpty() || str.length() == 1) {
            return false;
        }

        // controlla se firstLetter è uguale a secondLetter
        if (firstLetter == secondLetter){
            return false;
        }

        // lista per contenere gli indici della prima lettera nella stringa
        List<Integer> indices = new ArrayList<>();
        // trova il primo indice della prima lettera nella stringa
        int firstIndex = str.indexOf(firstLetter);
        // Trova l'indice della seconda lettera nella stringa
        int secondIndex = str.indexOf(secondLetter);

        // Itera finché ci sono ulteriori occorrenze della prima lettera e le aggiunge a indices
        while (firstIndex != -1) {
            indices.add(firstIndex);
            firstIndex = str.indexOf(firstLetter, firstIndex + 1);
        }

        // Controlla se entrambe le lettere sono presenti nella stringa
        if (!indices.isEmpty() && secondIndex != -1) {
            // Controlla se tutte le occorrenze della prima lettera sono prima della seconda lettera
            return indices.stream().allMatch(index -> index < secondIndex);
        }

        // Se una delle due lettere non è presente, restituisce false
        return false;
    }
}
```

Abbiamo quindi modificato il codice di test, includendo un Parameterized Test con l'annotazione `@NullSource`, in quanto si trattava di un caso particolare che non era necessario testare tramite i property based test.

```java
@ParameterizedTest
@DisplayName("stringa null")
@NullSource
void stringaNull(String str) {
    assertFalse(LetterOrderChecker.checkOrder(str,firstLetter,secondLetter));
}
```

Grazie alle modifiche effettuate è stata raggiunta una Branch Coverage del 100%. La Line Coverage è del 92,9% poiché la classe non viene istanziata nel codice di test. Questa è un'operazione superflua da testare che obbligherebbe ad accedere al metodo della classe attraverso una sua istanza, trattandosi di un metodo statico è preferibile farlo direttamente attraverso la classe.

Class	Class, %	Method, %	Branch, %	Line, %
LetterOrderChecker	100% (1/1)	50% (1/2)	100% (16/16)	92,9% (13/14)

```java
package org.example.Homework2;

import java.util.ArrayList;
import java.util.List;

public class LetterOrderChecker {
    public static boolean checkOrder(String str, char firstLetter, char secondLetter) {

        // controlla se la stringa è nulla, vuota o ha lunghezza 1
        if (str == null || str.isEmpty() || str.length() == 1) {
            return false;
        }

        // controlla se firstLetter è uguale a secondLetter
        if (firstLetter == secondLetter){
            return false;
        }

        // lista per contenere gli indici della prima lettera nella stringa
        List<Integer> indices = new ArrayList<>();
        // trova il primo indice della prima lettera nella stringa
        int firstIndex = str.indexOf(firstLetter);
        // Trova l'indice della seconda lettera nella stringa
        int secondIndex = str.indexOf(secondLetter);

        // Itera finché ci sono ulteriori occorrenze della prima lettera e le aggiunge a indices
        while (firstIndex != -1) {
            indices.add(firstIndex);
            firstIndex = str.indexOf(firstLetter, firstIndex + 1);
        }

        // Controlla se entrambe le lettere sono presenti nella stringa
        if (!indices.isEmpty() && secondIndex != -1) {
            // Controlla se tutte le occorrenze della prima lettera sono prima della seconda lettera
            return indices.stream().allMatch(index -> index < secondIndex);
        }

        // Se una delle due lettere non è presente, restituisce false
        return false;
    }

}
```

Casi particolari

I casi particolari che non vengono testati con i property based testing e che possono aumentare la test suite sono i seguenti:

- La stringa è `null`;
- firstLetter e secondLetter sono uguali;
- La stringa contiene numeri o caratteri speciali.

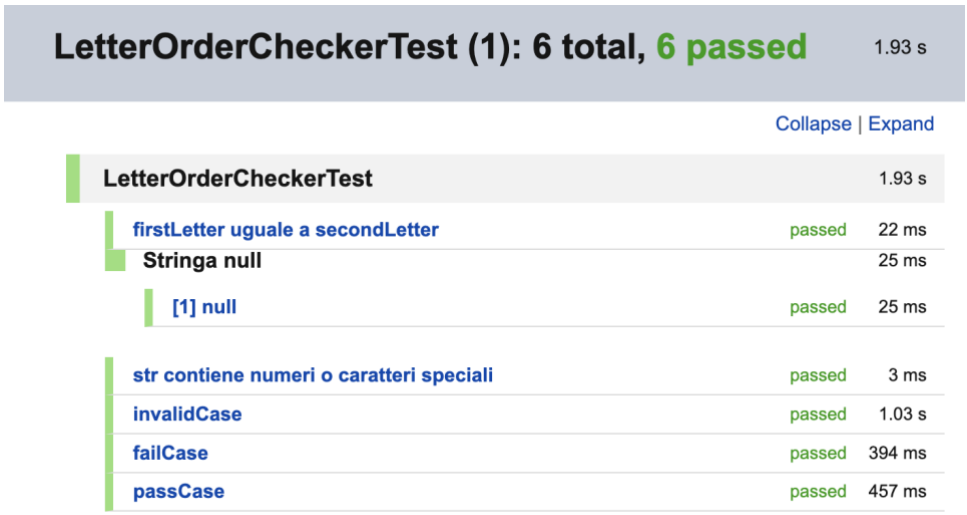
```java
// caso particolare in cui la stringa è null
@ParameterizedTest
@DisplayName("Stringa null")
@NullSource
void stringaNull(String str) {
    assertFalse(LetterOrderChecker.checkOrder(str,firstLetter,secondLetter));
}

// caso particolare in cui firstLetter è uguale a secondLetter
@Test
@DisplayName("firstLetter uguale a secondLetter")
void firstLetterUgualeASecondLetter() {
    assertFalse(LetterOrderChecker.checkOrder( str: "abcdef",  firstLetter: 'a', secondLetter: 'a'));
}

// caso particolare in cui la stringa contiene numeri o caratteri speciali
@Test
@DisplayName("str contiene numeri o caratteri speciali")
void strContieneNumeriOCaratteriSpeciali() {
    assertTrue(LetterOrderChecker.checkOrder( str: "a!bc de1f0#@",  firstLetter: 'a', secondLetter: 'f'));
}
```

Risultati

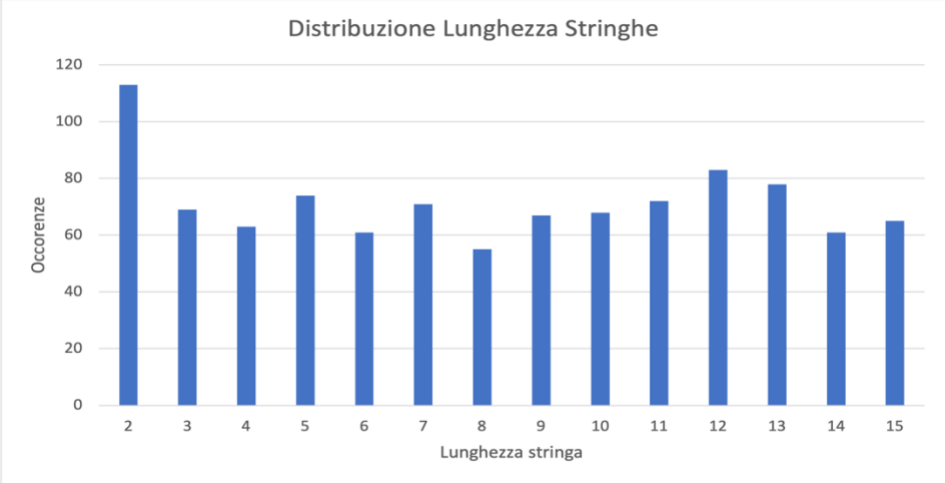
Tutti i 6 test definiti sono stati superati, sia i 3 `property based testing` sia i 3 `example based`.



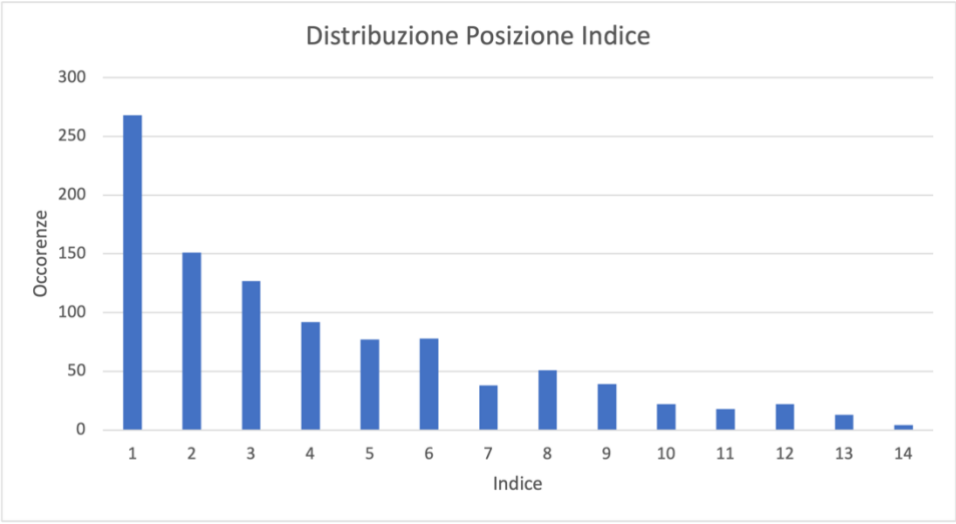
Statistiche

Abbiamo infine sfruttato Jqwik per raccogliere statistiche utili sui dati generati ed i test effettuati. Per migliorare la leggibilità i dati raccolti sono stati riportati attraverso grafici più efficaci visivamente rispetto a quelli di Jqwik.

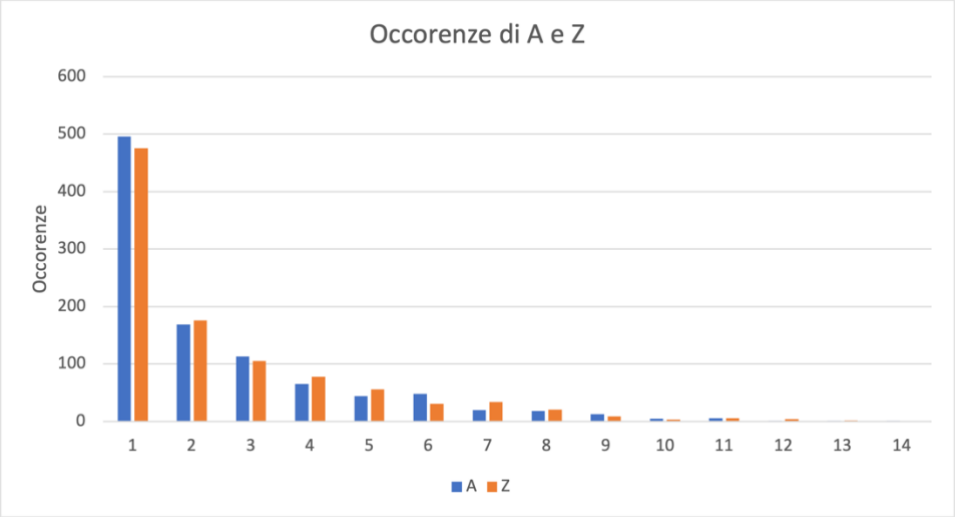
Caso: pass



Questa statistica mostra come *Jqwik* ha generato in modo casuale le stringhe con una lunghezza variabile da 2 a 15. È possibile osservare come il numero di occorrenze delle stringhe di lunghezza 2 è maggiore rispetto al numero delle altre stringhe, questo perché tali stringhe sono boundary cases che *Jqwik* ha testato. Invece, il numero di occorrenze di tutte le altre lunghezze si mantiene costante tra i 60 e gli 80.



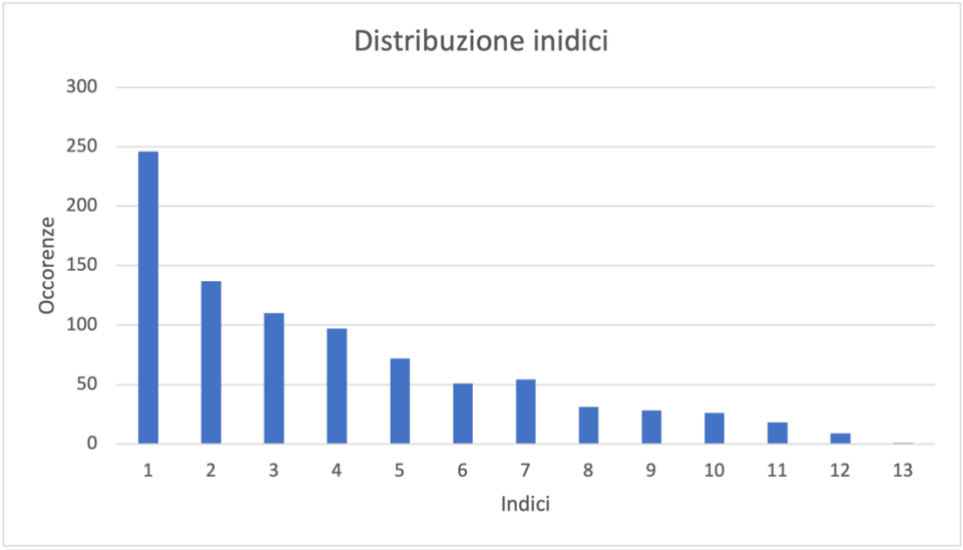
Questa statistica mostra come l'indice, per determinare il posizionamento di `firstLetter` e `secondLetter`, è stato generato in modo casuale. Le occorrenze dell'indice in posizione 1 sono maggiori perchè le stringhe di lunghezza 2 sono di più delle altre, come abbiamo visto nella statistica precedente; quindi, l'unico valore dell'indice possibile in questo caso è la posizione 1. Inoltre, notiamo che al crescere del valore dell'indice, il numero di occorrenze diminuisce, questo dipende dal fatto che le probabilità di avere un indice più alto sono minori (per le stringhe di lunghezza maggiore è anche possibile che venga generato un indice piccolo).

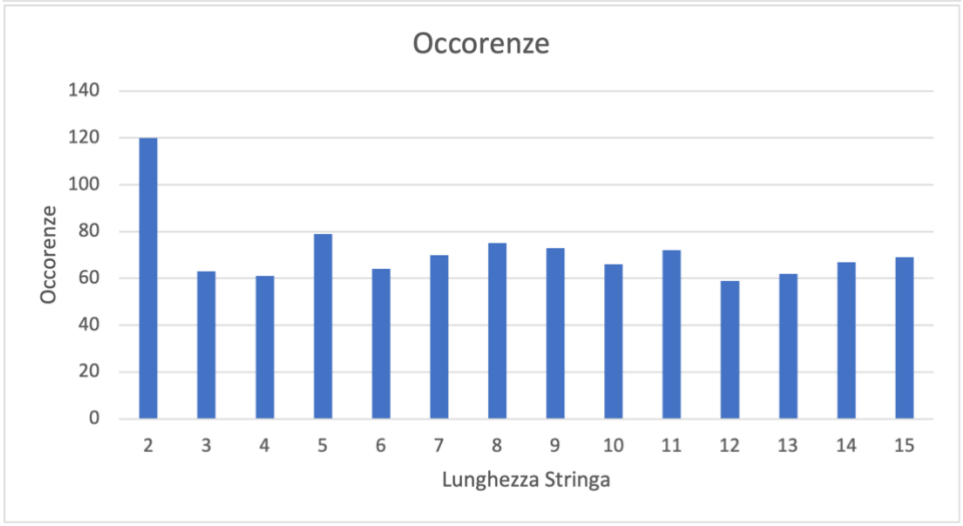


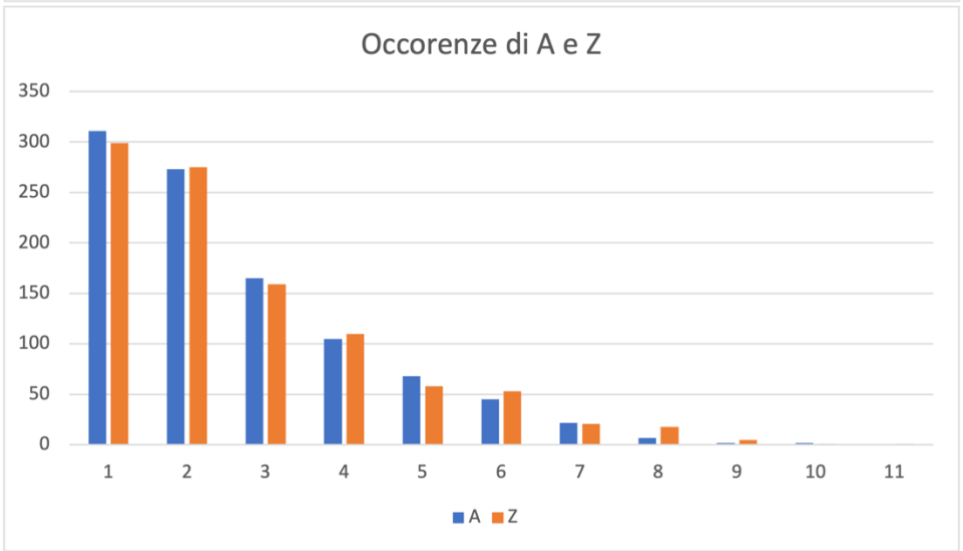
Abbiamo infine deciso di mostrare il numero di occorrenze di A (`firstLetter`) e Z (`secondLetter`) e la loro distribuzione per mostrare come le stringhe testate siano varie, in modo da assicurarci che i test rispecchino i dati con cui la funzione verrà realmente utilizzata.

Caso: fail

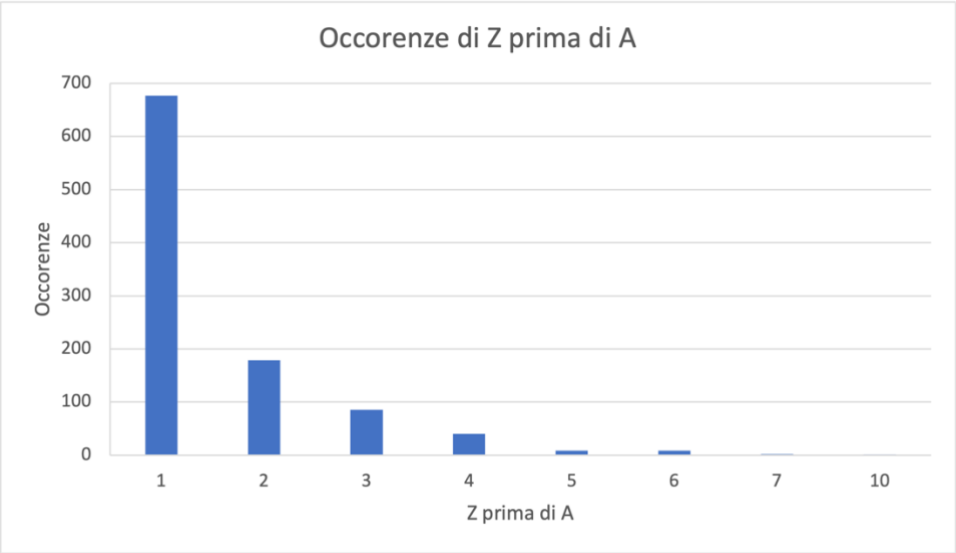
Per il caso fail abbiamo deciso di raccogliere le stesse statistiche del caso pass. Abbiamo ritenuto opportuno verificare anche il numero di Z che precedevano la prima A.







I risultati ottenuti sia per il caso pass che fail sono analoghi tra loro, questo poiché le caratteristiche delle stringhe generate sono simili per entrambe le partizioni.

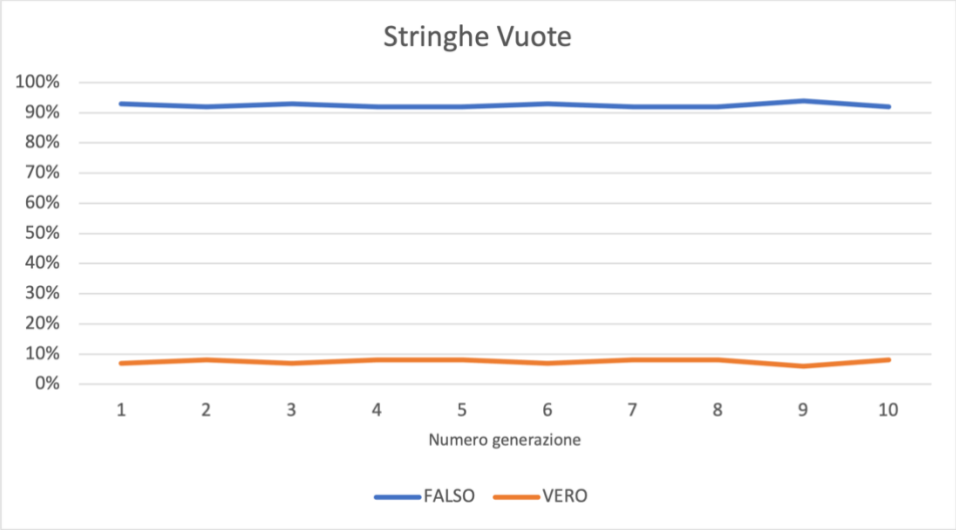


Per assicurarci che i test rispecchiassero casi d'uso reali della funzione abbiamo voluto verificare con questa statistica che dati di input fossero vari ed in particolare che le stringhe generate casualmente contenessero effettivamente altri casi oltre a quello base in cui solo la prima lettera è una '`z`' e solo l'ultima una '`a`'.
Per verificare il numero di 'z' prima di 'a' abbiamo definito la funzione `countLetterBeforeAnother` che conta il numero di occorrenze di letter2 che precedono la prima occorrenza di letter1 all'interno di `string`.

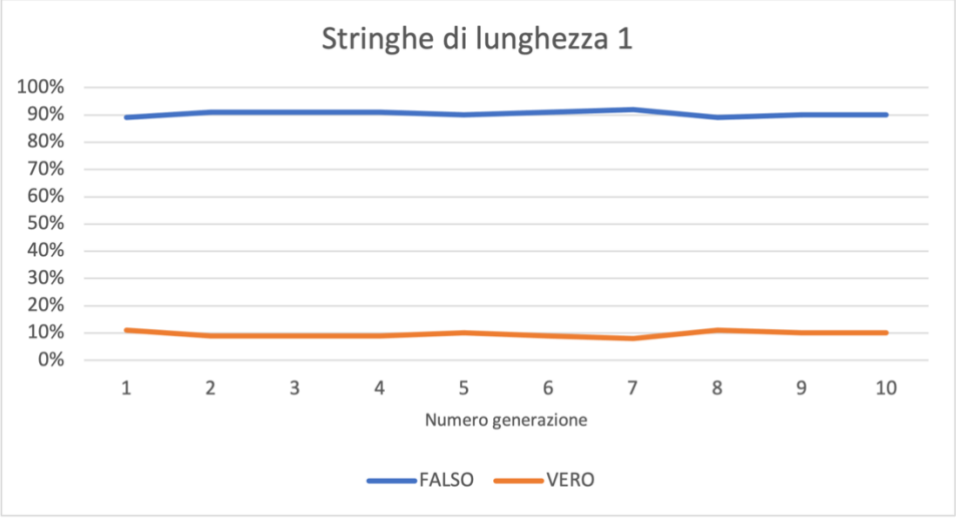
```
143 @       private int countLetterBeforeAnother(StringBuilder string, char letter1, char letter2){
144             int count = 0;
145             int i = 0;
146
147             while (string.charAt(i) != letter1){
148                 if(string.charAt(i) == letter2){
149                     count++;
150                 }
151                 i++;
152             }
153             return  count;
154         }
155     }
```

Caso: invalid

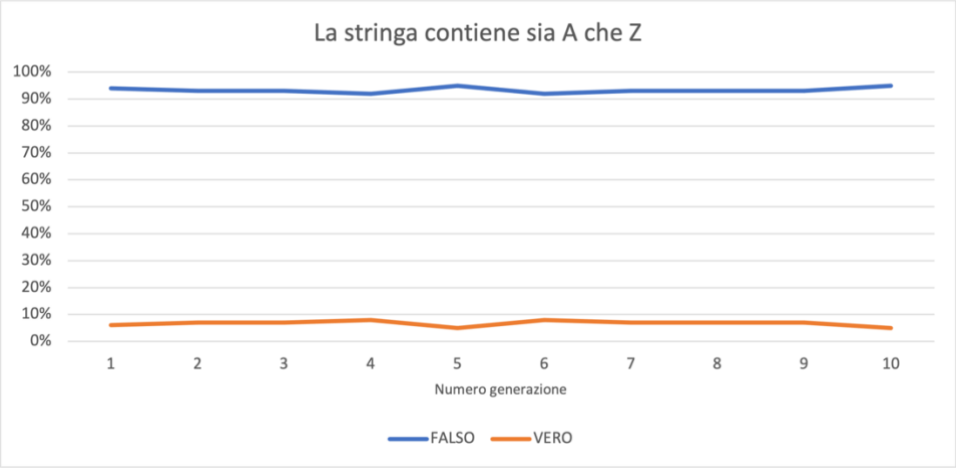
Per il caso invalid, poiché la generazione delle 1000 stringhe è stata affidata a Jqwik, abbiamo verificato che i casi di interesse fossero effettivamente testati. Per essere certi che i risultati ottenuto fossero veritieri abbiamo deciso di eseguire i test e raccogliere le statistiche per 10 volte, di seguito i dati raccolti.



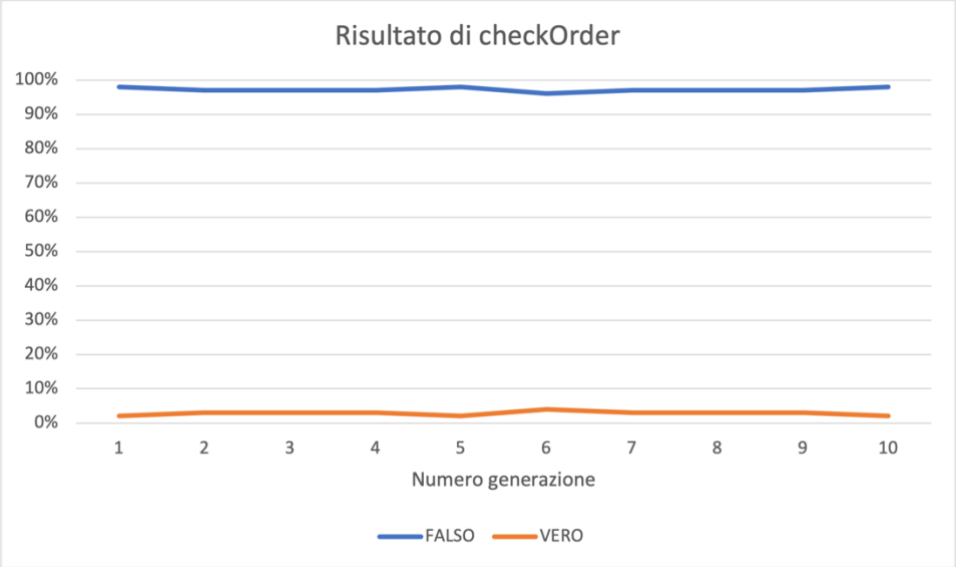
Notiamo come su una generazione di 1000 stringhe, Jqwik generi in automatico poco meno del 10% di stringhe vuote.



Il numero di stringhe con lunghezza 1 rimane invece costante intorno al 10%



Era poi necessario assicurarsi che la funzione identificasse correttamente le stringhe in cui comparivano sia `firstLetter` che `secondLetter`. Solo nel 7% dei casi la stringa casualmente generata conteneva entrambe le lettere.



Infine, abbiamo utilizzato le statistiche per verificare quante stringhe rispettassero la condizione stabilita e quindi in quanti casi la funzione restituiva `true` come risultato. La differenza rispetto ai casi precedenti è notevole, se le stringhe sono generate da Jqwik senza alcun vincolo (oltre la dimensione) il numero di esiti positivi si aggira intorno al 3%.

I risultati originali restituiti da Jqwik sono disponibili nel repositories:
https://github.com/ITSSuniba/1-appello-itss-aa2023-2024-software-chasers-me_Michele_Minervini.git
al percorso:
Relazione/risultati_jqwik

Pair programming

Nel corso del processo di testing, abbiamo adottato la tecnica del *pair programming*. Questa metodologia ha permesso una collaborazione sinergica, consentendo un approccio più completo all'implementazione dei test, garantendo più qualità e leggibilità.

Successivamente, abbiamo scambiato il codice di test con il gruppo "Pocket Coffees", al fine di identificare potenziali problematiche o inconsistenze nel nostro approccio. Durante questa fase di confronto, non sono state individuate significative problematiche nel nostro codice di test. Tuttavia, l'esperienza è risultata estremamente utile per confrontare metodologie, condividere best practices e migliorare la qualità complessiva del nostro processo di testing.
La pratica del pair programming e lo scambio di codice tra i gruppi si sono dimostrati efficaci strumenti collaborativi per garantire l'affidabilità e la robustezza delle test suite.

Conclusioni

La relazione offre una panoramica dettagliata sull'implementazione, il testing e l'analisi delle due classi, TwoSumProblem e LetterOrderChecker. L'impiego di metodologie come specification based testing e structural testing, affiancato dall'approccio dei property based testing, hanno dimostrato la loro efficacia nell'esplorare e coprire in modo completo una vasta gamma di scenari possibili.

Le correzioni apportate durante il processo di sviluppo hanno giocato un ruolo significativo nell'incrementare la robustezza del codice, affrontando eventuali vulnerabilità o errori precedentemente identificati. La code coverage è stata raggiunta in maniera esaustiva, aumentando la fiducia nella solidità complessiva del codice.

Per i property based testing le statistiche generate confermano l'adeguatezza e l'efficacia dei dati generati, sottolineando la qualità della suite di test e la sua capacità di identificare e correggere potenziali problemi.