



UNIVERSITÀ DEGLI STUDI DI BARI ALDO MORO

Dipartimento di Informatica

Corso di laurea "Informatica e Tecnologie per la Produzione del software"

DOCUMENTAZIONE PROGETTO

Corso di Integrazione e Test a.a. 2022/2023

Prof.ssa **Azzurra Ragone**

Team DoubleLF

- **Fabio Lacarbonara 737518**
- **Francesco Lisco 735640**

f.lacarbonara3@studenti.uniba.it

f.lisco7@studenti.uniba.it

Sommario

DOCUMENTAZIONE PROGETTO	1
Homework n°1	3
Testing Workflow For Specification-Based Testing (Merge Sort)	3
1) Comprensione dei requisiti:	3
2) Esplora cosa fa il programma, dati vari input:	3
3) Esplora input, output e identifica le partizioni	5
4) Identifica i casi limite (boundary cases)	5
5) Elaborare casi di test	6
6) Automatizza i casi di test	7
7) Aumenta i casi di test della suite con creatività ed esperienza	9
Homework n°2	10
Structural Testing and Code Coverage (Task 1)	10
Structural Testing and Code Coverage (Task 2)	15

Homework n°1

Testing Workflow For Specification-Based Testing (Merge Sort)

1) Comprensione dei requisiti:

- a. L'obiettivo del metodo è quello di riordinare una lista di numeri interi fornita in input, in ordine crescente.
- b. Il programma riceve 3 parametri in input:
 - i. **arr**, che rappresenta la lista di numeri da riordinare.
 - ii. **l**, che rappresenta la lista di numeri dal primo elemento fino al punto medio di "arr".
 - iii. **r**, che rappresenta la lista di numeri dal punto medio fino all'ultimo elemento di "arr".
- c. Il programma restituisce in output la lista di numeri "arr" ordinata.

2) Esplora cosa fa il programma, dati vari input:

a.

```
Given Array
0 0 0 0 0

Sorted array
0 0 0 0 0
```

b.

```
Given Array
0 1 2 3 4

Sorted array
0 1 2 3 4
```

c.

```
Given Array
4 3 2 1 0

Sorted array
0 1 2 3 4
```

d.

```
Given Array
-1 -4 -5 -6 -2

Sorted array
-6 -5 -4 -2 -1
```

e.

```
Given Array
-1 4 -5 6 -2

Sorted array
-5 -2 -1 4 6
```

- f. `Given Array`
`10 34 15 191 56`
`Sorted array`
`10 15 34 56 191`
- g. `Given Array`
`-110 1134 -5315 191 156`
`Sorted array`
`-5315 -110 156 191 1134`
- h. `Given Array`
`2147483647 -2147483648 0 5322 -12`
`Sorted array`
`-2147483648 -12 0 5322 2147483647`
- i. `Given Array`
`2147483647 -2147483648 2147483647 5322 -12`
`Sorted array`
`-2147483648 -12 5322 2147483647 2147483647`
- j. `Given Array`
`1`
`Sorted array`
`1`
- k. `Given Array`
`Null Array`
`Sorted array`
`Null Array`
- l. `Given Array`
`Sorted array`

3) Esplora input, output e identifica le partizioni

a. Input individuali:

Parametro arr:

- 1) Null
- 2) Array vuoto
- 3) Array di lunghezza = 1
- 4) Array di lunghezza > 1

Parametro l:

- 1) Null
- 2) Array vuoto
- 3) Array di lunghezza = 1
- 4) Array di lunghezza > 1

Parametro r:

- 1) Null
- 2) Array vuoto
- 3) Array di lunghezza = 1
- 4) Array di lunghezza > 1

b. Combinazioni input:

- 1) arr, l e r sono null
- 2) arr, l e r sono vuoti
- 3) arr è di lunghezza 1, l è vuoto ed r ha lunghezza 1
- 4) arr è di lunghezza > 1 e pari, l ed r hanno la stessa lunghezza
- 5) arr è di lunghezza > 1 e dispari, l è di lunghezza < di r

c. Classi di Output:

Array di numeri interi (Output)

- 1) Null
- 2) Array vuoto
- 3) Array di lunghezza = 1
- 4) Array di lunghezza > 1

4) Identifica i casi limite (boundary cases)

Analizzando il codice abbiamo individuato la presenza di boundary cases in corrispondenza della seguente condizione: `if (arr.length <= 1)`

- a. **1** è l'on point: il punto che si trova al limite (presente all'interno della condizione)
- b. **2** è l'off point: il punto più vicino al limite che appartiene alla partizione successiva (`arr.length > 1`) e poiché l'on point rende la condizione vera, l'off point sarà il primo punto che la rende falsa.
- c. **1** è l'in point: il punto che rende la condizione vera
- d. **2** è l'out point: il punto che rende la condizione falsa.

5) Elaborare casi di test

Abbiamo deciso di testare, come casi i seguenti:

Casi di test eccezionali:

- T1. arr è null
- T2. arr è vuoto
- T3. l è null
- T4. l è vuoto
- T5. r è null
- T6. r è vuoto
- T7. l è maggiore di r

Casi di test per arr di lunghezza = 1 (Boundary test)

- T8. arr ha un singolo elemento, l è vuoto ed r è uguale ad arr
- T9. arr ha un singolo elemento, l non è vuoto
- T10. arr ha un singolo elemento, r non è uguale ad arr
- T11. arr ha un singolo elemento, l non è vuoto ed r non è uguale ad arr

Casi di test per arr di lunghezza > 1:

Potevamo utilizzare invece dell'and, l'or, in questo modo potevamo non fare i test T9 e T10

- T12. arr ha dimensioni pari maggiori di 1, l è la prima metà ed r è la seconda metà
- T13. arr ha dimensioni dispari maggiori di 1, l contiene tutti i numeri prima del punto medio ed r contiene tutti i numeri dal punto medio in poi.

Casi di test di arr di lunghezza >1

arr ha dimensione di maggiore di 1, la somma delle lunghezze di l e r non corrispondono alla lunghezza di arr.

6) Automatizza i casi di test

T1, T2, T3, T4, T5, T6, T7:

```
//T1 e T2
no usages  📌 Fabio18 *
@Test
@DisplayName("arrIsNullorEmpty")
void arrIsNullorEmpty()
{
    assertNull(MergeSort.mergeSort(arr: null, new int[]{1}, new int[]{2,3}));
    assertEquals(new int[]{},MergeSort.mergeSort(new int[]{}, new int[]{}, new int[]{}));
}

//T3 e T4
no usages  📌 Fabio18
@Test
@DisplayName("lIsNullorEmpty")
void lIsNullorEmpty()
{
    assertNull(MergeSort.mergeSort(new int[]{3,5,7}, l: null, new int[]{5,7}));
    assertEquals(new int[]{3,5,7},MergeSort.mergeSort(new int[]{3,5,7}, new int[]{}, new int[]{5,7}));
}

//T5 e T6
no usages  📌 Fabio18 *
@Test
@DisplayName("rIsNullorEmpty")
void rIsNullorEmpty()
{
    assertNull(MergeSort.mergeSort(new int[]{3,5,7}, new int[]{3}, r: null));
    assertEquals(new int[]{3,5,7},MergeSort.mergeSort(new int[]{3,5,7}, new int[]{}, new int[]{}));
}

//T7
no usages  new *
@Test
@DisplayName("lIsMajorr")
void lIsMajorr()
{
    assertEquals(new int[]{3,5,7},MergeSort.mergeSort(new int[]{3,5,7}, new int[]{3}, new int[]{}));
}

//Boundary tests T8,T9,T10,T11
no usages  📌 Fabio18
@ParameterizedTest
@MethodSource("intArrayProvider")
@DisplayName("arrOfLengthMinorEqual1")
void arrOfLengthMinorEqual1(int[] arr, int[] l, int[] r)
{
    assertEquals(arr,MergeSort.mergeSort(arr, l, r));
}
```

T8, T9, T10, T11:

```
//Boundary tests T8,T9,T10,T11
no usages  👤 Fabio18
@ParameterizedTest
@MethodSource("intArrayProvider")
@DisplayName("arrOfLengthMinorEqual1")
void arrOfLengthMinorEqual1(int[] arr, int[] l , int [] r)
{
    assertEquals(arr, MergeSort.mergeSort(arr, l, r));
}

1 usage  👤 Fabio18 *
static Stream<Arguments> intArrayProvider()
{
    return Stream.of(
        arguments(new int[]{2}, new int[]{}, new int[]{2}), //T8
        arguments(new int[]{2}, new int[]{0}, new int[]{2}), //T9
        arguments(new int[]{1000}, new int[]{}, new int[]{1001}), //T10
        arguments(new int[]{1000}, new int[]{0}, new int[]{1001})); // T11
}
```

Durante l'elaborazione del test T9 in Junit , ci siamo resi conto della presenza di un controllo che non rispettava i requisiti:

```
if(arr.length == 1 && (l.length != 0 || r.length != 0))
```

poiché per “riordinare” un array composto da un solo elemento dovrebbe essere necessario restituire in output l’array di partenza, di conseguenza non basta che r sia diverso da zero ma sia uguale ad arr. Di seguito la modifica nel codice:

```
if(arr.length <= 1 && (l.length != 0 || !(Arrays.equals(r, arr))))
```

T12, T13:

```
//T12,T13
no usages  👤 Fabio18
@ParameterizedTest
@MethodSource("intArrayProvider2")
@DisplayName("arrOfLengthMajorThan1")
void arrOfLengthMajorThan1(int[] arr, int[] l , int [] r)
{
    assertEquals(arr, MergeSort.mergeSort(arr, l, r));
}

1 usage  👤 Fabio18 *
static Stream<Arguments> intArrayProvider2()
{
    return Stream.of(
        arguments(new int[]{4,5,10,7,1,2}, new int[]{4,5,10}, new int[]{7,1,2}), //T12
        arguments(new int[]{4,5,10,7,1,2,6}, new int[]{4,5,10}, new int[]{7,1,2,6})); //T13
}
```


7) Aumenta i casi di test della suite con creatività ed esperienza

T14. arr ha tutti gli elementi (più di uno) in ordine crescente

T15. arr ha più di un elemento, un elemento non è posizionato in ordine crescente

T16. arr sia con numeri positivi che negativi

T17. arr con soli valori negativi

T18. arr con elementi duplicati

T19. arr con elementi in ordine decrescente

T20. arr con elemento "maxvalue"

T21. arr con elemento "minvalue"

T22. arr sia con elemento "maxvalue" che "minvalue"

```
//T14,T15,T16,T17,T18,T19,T20,T21,T22
no usages  ▸ Fabio18
@ParameterizedTest
@MethodSource("intArrayProvider4")
@DisplayName("AugmentedTests")
void AugmentedTests(int[] arr, int[] l, int[] r)
{
    assertEquals(arr, MergeSort.mergeSort(arr, l, r));
}


1 usage  ▸ Fabio18 *
static Stream<Arguments> intArrayProvider4()
{
    return Stream.of(
        arguments(new int[]{1,5,10,27,30,35}, new int[]{1,5,10}, new int[]{27,30,35}),//T14
        arguments(new int[]{35,5,10,27,30}, new int[]{35,5}, new int[]{10,27,30}),//T15
        arguments(new int[]{-1,25,-100,-270,3000,15}, new int[]{-1,25,-100}, new int[]{-270,3000,15}),//T16
        arguments(new int[]{-1000,-10,-2034,-1,-30}, new int[]{-1000,-10}, new int[]{-2034,-1,-30}),//T17
        arguments(new int[]{1,5,10,5,30,40}, new int[]{1,5,10}, new int[]{5,30,40}),//T18
        arguments(new int[]{40,35,20,4,0}, new int[]{40,35}, new int[]{20,4,0}),//T19
        arguments(new int[]{Integer.MAX_VALUE,5,10}, new int[]{Integer.MAX_VALUE}, new int[]{5,10}),//T20
        arguments(new int[]{Integer.MIN_VALUE,5,10}, new int[]{Integer.MIN_VALUE}, new int[]{5,10}),//T21
        arguments(new int[]{Integer.MIN_VALUE,5,Integer.MAX_VALUE}, new int[]{Integer.MIN_VALUE}, new int[]{5,Integer.MAX_VALUE}));//T22
}
```

Homework n°2

Structural Testing and Code Coverage (Task 1)

Per completare la suite di test black-box dobbiamo effettuare gli structural testing e l'analisi di copertura del codice. Per quando riguarda il code coverage, ci siamo serviti del tool Jacoco per verificare quanto il nostro codice fosse coperto dai test.

MergeSort

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
mergeSort(int[], int[], int[])		100%		100%	0	16	0	32	0	1

```
13. public static int[] mergeSort(int[] arr, int [] l, int[] r){
14.
15.     if (arr == null || l == null || r == null) {
16.         System.out.println("Null Array");
17.         return null;
18.     }
19.     if(l.length > r.length)
20.     {
21.         System.out.println("Impossibile riordinare l'array, l > r: " + Arrays.toString(arr));
22.         return arr;
23.     }
24.     if(arr.length > 1 && l.length == 0) //input che potrebbero arrivare in maniera incorretta
25.     {
26.         System.out.println("Impossibile riordinare l'array, l o r = 0: " + Arrays.toString(arr));
27.         return arr;
28.     }
29.     if(arr.length <= 1 && (l.length != 0 || !(Arrays.equals(r, arr)))) //input che potrebbero arrivare in maniera incorretta
30.     {
31.         System.out.println("Errore di input per l o r, restituisco l'array fornito in input: " + Arrays.toString(arr));
32.         return arr;
33.     }
34.
35.     if (arr.length <= 1)
36.     {
37.         return arr;
38.     }
39.
40.     int newM = calcoloM(l);
41.     int[] newL = calcoloL(l,newM);
42.     int[] newR = calcoloR(l,newM);
43.
44.     mergeSort(l,newL,newR);
45.
46.     newM = calcoloM(r);
47.     newL = calcoloL(r,newM);
48.     newR = calcoloR(r,newM);
49.
50.     mergeSort(r,newL,newR);
51.
52.     //riordina gli elementi l'array
53.     int i = 0, j = 0, k = 0;
54.     while (i < l.length && j < r.length) {
55.         if (l[i] <= r[j]) {
56.             arr[k++] = l[i++];
57.         } else {
58.             arr[k++] = r[j++];
59.         }
60.     }
61.     while (i < l.length) {
62.         arr[k++] = l[i++];
63.     }
64.     while (j < r.length) {
65.         arr[k++] = r[j++];
66.     }
67.     return arr;
68. }
69.
```

Come si può evincere dall'immagine, **la line coverage** equivale al **100%** in quanto ogni riga del codice è stata testata almeno una volta, comprese le istruzioni come: if, while, for, ecc...

Per iniziare gli structural testing abbiamo deciso di utilizzare la **Rule of Thumb**:

- 1) Iniziare con il **Branch Coverage**
- 2) Per espressioni più complicate decidere se utilizzare la **Condition + Branch Coverage** oppure l'**MC/DC**.

Per quanto riguarda il Branch Coverage, abbiamo verificato che per ogni istruzione ogni branch sia stato coperto dai test:

```
if (arr == null || l == null || r == null) {  
    System.out.println("Null Array");  
    return null;  
}
```

$$\text{Branch Coverage} = \frac{\text{Branches covered}}{\text{Total number of branches}} = \frac{2}{2} = 100 \%$$

- T1: arr true, l false, r false
- T2: arr false, l false, r false

```
if(l.length > r.length)  
{  
    System.out.println("Impossibile riordinare l'array, l > r: " + Arrays.toString(arr));  
    return arr;  
}
```

$$\text{Branch Coverage} = \frac{\text{Branches covered}}{\text{Total number of branches}} = \frac{2}{2} = 100 \%$$

- T7: true
- T4: false

```
if(arr.length > 1 && l.length == 0) //input che potrebbero arrivare in maniera incorretta  
{  
    System.out.println("Impossibile riordinare l'array, l o r = 0: " + Arrays.toString(arr));  
    return arr;  
}
```

$$\text{Branch Coverage} = \frac{\text{Branches covered}}{\text{Total number of branches}} = \frac{2}{2} = 100 \%$$

- T6: arr true, l true
- T8: arr false, l true

```
if(arr.length <= 1 && (l.length != 0 || !(Arrays.equals(r, arr)))) //input che potrebbero arrivare in maniera incorretta  
{  
    System.out.println("Errore di input per l o r, restituisco l'array fornito in input: " + Arrays.toString(arr));  
    return arr;  
}
```

$$\text{Branch Coverage} = \frac{\text{Branches covered}}{\text{Total number of branches}} = \frac{2}{2} = 100 \%$$

- T9: arr true, l true, r true
- T8: arr true, l false, r true

```
if (arr.length <= 1)  
{  
    return arr;  
}
```

$$\text{Branch Coverage} = \frac{\text{Branches covered}}{\text{Total number of branches}} = \frac{2}{2} = 100 \%$$

- T8: arr true

- T7: arr false

```
while (i < l.length && j < r.length) {
    if (l[i] <= r[j]) {
        arr[k++] = l[i++];
    } else {
        arr[k++] = r[j++];
    }
}
```

Valutiamo il while:

$$\text{Branch Coverage} = \frac{\text{Branches covered}}{\text{Total number of branches}} = \frac{2}{2} = 100 \%$$

- T12: i true, j true
- T12: i false, j false (Utilizzando un test nel quale viene effettuato un ordinamento, senza casi particolari, la condizione nel while sarà vera finchè non avrà confrontato tutti gli elementi di l con quelli di r, perciò in quest'unico caso di test entrambi i branch vengono testati).

Valutiamo l'if:

$a = (l[i] \leq r[j])$

- T12: a true
- T12: a false

```
while (i < l.length) {
    arr[k++] = l[i++];
}
```

$$\text{Branch Coverage} = \frac{\text{Branches covered}}{\text{Total number of branches}} = \frac{2}{2} = 100 \%$$

- T12: false
- T12: true

```
while (j < r.length) {
    arr[k++] = r[j++];
}
```

$$\text{Branch Coverage} = \frac{\text{Branches covered}}{\text{Total number of branches}} = \frac{2}{2} = 100 \%$$

- T12: true
- T12: false

In conclusione, possiamo affermare che abbiamo raggiunto il 100% di **Branch Coverage** per ogni istruzione.

Proseguiamo con la Rule of Thumb adottando il principio di **code coverage MC/DC** per le espressioni più complesse e il **Condition + Branch** per quelle meno complesse:

MC/DC

```
if (arr == null || l == null || r == null) {
    System.out.println("Null Array");
    return null;
}
```

Test case	arr == null	l == null	r == null	Decision
T1	true	false	false	true
T2	false	false	false	false
T3	false	true	false	true
T5	false	false	true	true

Arr: {T1,T2}

L: {T2,T3}

R:{T2,T5}

I test che quindi prendiamo sono: **{T1, T2, T3, T5}**.

Condition + Branch:

```
if(arr.length > 1 && l.length == 0) //input che potrebbero arrivare in maniera incorretta
{
    System.out.println("Impossibile riordinare l'array, l o r = 0: " + Arrays.toString(arr));
    return arr;
}
```

$$C + B \text{ coverage} = \frac{\text{Branches covered} + \text{condition covered}}{\text{number of branches} + \text{number of conditions}} \times 100 = \frac{2+4}{2+4} = 100 \%$$

- T6: arr true, l true
- T9: arr false, l false

MC/DC

```
if(arr.length <= 1 && (l.length != 0 || !(Arrays.equals(r, arr)))) //input che potrebbero arrivare in maniera incorretta
{
    System.out.println("Errore di input per l o r, restituisco l'array fornito in input: " + Arrays.toString(arr));
    return arr;
}
```

Test case	arr.length ==1	l.length != 0	R != arr	decision
T1	false	true	true	false
T2	false	false	true	false
T4	false	False	false	false
T5	false	true	false	false
T8	true	false	false	false
T9	true	true	false	true
T10	true	false	true	true
T11	true	true	true	true

Arr:{T1,T11}{T2,T10}{T5,T9}

L:{T8,T9}

R:{T8,T10}

I test che quindi prendiamo sono {T5,T8,T9,T10}.

Condition + Branch (Condizione del while):

```
while (i < l.length && j < r.length) {
    if (l[i] <= r[j]) {
        arr[k++] = l[i++];
    } else {
        arr[k++] = r[j++];
    }
}
```

$$C + B \text{ coverage} = \frac{\text{Branches covered} + \text{condition covered}}{\text{number of branches} + \text{number of conditions}} \times 100 = \frac{2+4}{2+4} = 100 \%$$

- T12: i true, j true
- T12: i false, j false

I test che avevamo selezionato per la Branch Coverage, in questo caso, si sono rilevati sufficienti per coprire anche la Condition + Branch.

In conclusione, in seguito allo specification-based testing, abbiamo raggiunto il 100% di line coverage; perciò, abbiamo effettuato l'analisi di copertura del codice seguendo la Rule of Thumb, e abbiamo notato che anche per le espressioni più complesse non è stato necessario aggiungere alcun test white box.

Structural Testing and Code Coverage (Task 2)

Il codice che abbiamo selezionato è quello del Team GAF. Per il task 2 abbiamo seguito i seguenti passi:

1) Comprensione dei requisiti:

- a. L'obiettivo del programma è quello di calcolare il quadrato dei numeri pari e il cubo dei numeri dispari, presenti in un array di numeri interi positivi, forniti in input e poi ordinati in modo crescente. In maniera parallela, il programma ordina una lista di nomi, anch'essi forniti in input.
- b. Il programma riceve 4 parametri:
 - numeri che rappresenta la lista di numeri interi positivi
 - nomi che rappresenta la lista di stringhe da riordinare
 - m, rappresenta la dimensione della lista dei nomi
 - n, rappresenta la dimensione della lista dei numeri
- c. Il programma restituisce 2 array in output, il primo contenente la lista di numeri ordinati in ordine crescente, e il secondo contenente le stringhe ordinate in ordine alfabetico. Tutto questo viene eseguito se i quadrati non superano il valore 99 e i cubi il valore 199

2) Esplora cosa fa il programma per vari input:

```
Non e' possibile inserire stringhe nulle
```

```
0
```

```
Fabio
```

```
Non e' possibile inserire una grandezza di dimensione 0 o minore
```

```
1
```

```
4
```

```
Fabio
```

```
Giuseppe
```

```
Il numero di numeri e nomi inseriti non corrisponde
```

```
Il numero inserito e' troppo grande
```

```
Non e' possibile inserire un numero negativo
```

```
Fabio
```

```
Giuseppe
```

1

16

27

Fabio

Francesco

Giuseppe

3) Esplora gli input, output e identifica le partizioni:

a. Input individuali

Parametro numeri: 1) Null 2) Array vuoto 3) Array di lunghezza > 1 =	Parametro nomi: 1) Null 2) Array vuoto 3) Array di lunghezza > 1 =	Parametro m: 1) > 0 2) $< = 0$	Parametro n: 1) > 0 2) $< = 0$
---	---	--	--

b. Combinazioni input:

1) m,n sono diversi 2) numeri è null e m,n sono uguali 3) nomi è null e m,n sono uguali 4) numeri vuoto e m,n sono uguali 5) nomi vuoto e m,n sono uguali 6) numeri e nomi di lunghezza > 1 = e m,n sono uguali
--

c. Classi di Output:

Stringa

4) Identifica i casi limite (boundary cases):

Analizzando il codice abbiamo individuato la presenza di boundary cases in corrispondenza delle seguenti condizioni:

- if ($n \leq 0$) (questa condizione serve per verificare che n e di conseguenza m, poiché devono essere uguali, siano > 0)
 - **0** è l'on point: il punto che si trova al limite (presente all'interno della condizione)
 - **1** è l'off point: il punto più vicino al limite che appartiene alla partizione successiva ($n > 0$) e poiché l'on point rende la condizione vera, l'off point sarà il primo punto che la rende falsa.
 - **0** è l'in point: il punto che rende la condizione vera
 - **1** è l'out point: il punto che rende la condizione falsa.

- If (numeri[i] < 0)
 - **0** è l'on point: il punto che si trova al limite (presente all'interno della condizione)
 - **-1** è l'off point: il punto più vicino al limite che appartiene alla partizione successiva (numeri[i] < 0) e poiché l'on point rende la condizione falsa, l'off point sarà il primo punto che la rende vera.
 - **-1** è l'in point: il punto che rende la condizione vera
 - **0** è l'out point: il punto che rende la condizione falsa.

- if (numeri[i] > 99)
 - i. **99** è l'on point: il punto che si trova al limite (presente all'interno della condizione)
 - ii. **100** è l'off point: il punto più vicino al limite che appartiene alla partizione successiva (numeri[i] > 99) e poiché l'on point rende la condizione falsa, l'off point sarà il primo punto che la rende vera.
 - iii. **100** è l'in point: il punto che rende la condizione vera
 - iv. **99** è l'out point: il punto che rende la condizione falsa.

- if (quadrato < 100) (questa condizione serve per verificare che il quadrato dei numeri presenti nell'array "numeri" sia < 100 e di conseguenza possa essere accettato come valore)
 - **100** è l'on point: il punto che si trova al limite (presente all'interno della condizione)
 - **99** è l'off point: il punto più vicino al limite che appartiene alla partizione successiva (quadrato < 100) e poiché l'on point rende la condizione falsa, l'off point sarà il primo punto che la rende vera.
 - **99** è l'in point: il punto che rende la condizione vera
 - **100** è l'out point: il punto che rende la condizione falsa.

- if (cubo < 200) (questa condizione serve per verificare che il cubo dei numeri presenti nell'array "numeri" sia < 200 e di conseguenza possa essere accettato come valore)
 - **200** è l'on point: il punto che si trova al limite (presente all'interno della condizione)
 - **199** è l'off point: il punto più vicino al limite che appartiene alla partizione successiva (cubo < 200) e poiché l'on point rende la condizione falsa, l'off point sarà il primo punto che la rende vera.
 - **199** è l'in point: il punto che rende la condizione vera
 - **200** è l'out point: il punto che rende la condizione falsa.

5) Elaborare i casi di test

a. ***Casi di test eccezionali:***

- T1. n è diverso da m
- T2. numeri è null
- T3. nomi è null
- T4. numeri e nomi hanno lunghezze diverse
- T5. Almeno un elemento di nomi è null

b. ***Casi di test per soli numeri pari il cui quadrato è minore di 100:***

T6. numeri ha n elementi pari aventi i quadrati minori di 100, nomi ha m elementi, n e m sono uguali

c. ***Casi di test per soli numeri dispari il cui cubo è minore di 200:***

T7. numeri ha n elementi dispari aventi i cubi minori di 200, nomi ha m elementi, n e m sono uguali

d. **Boundary test**

- T8. n ed m sono uguali a 0
- T9. Almeno un elemento di numeri è negativo
- T10. Almeno un elemento di numeri è maggiore di 99
- T11. Almeno un quadrato di un elemento di numeri pari è maggiore di 100
- T12. Almeno un cubo di un elemento di numeri dispari è maggiore di 200

6) Automatizza i casi di test

T1, T2, T3, T4, T5:

```
//Casi di test eccezionali:
//T1
no usages  ▲ Fabio18 *
@Test
@DisplayName("nIsDifferentFromm")
void nIsDifferentFromm()
{
    output = Homework2.effettuaOperazioni( n: 1, m: 2, new String[]{"Leone"},new int[]{1,2});
    assertEquals( expected: "Il numero di numeri e nomi inseriti non corrisponde",output.messaggio);
}

//T2
no usages  ▲ Fabio18 *
@Test
@DisplayName("numeriIsNull")
void numeriIsNull()
{
    output = Homework2.effettuaOperazioni( n: 1, m: 1, new String[]{"Giuseppe"}, numeri: null);
    assertEquals( expected: "Una delle due o entrambe le liste sono nulle",output.messaggio);
}

//T3
no usages  ▲ Fabio18 *
@Test
@DisplayName("nomiIsNull")
void nomiIsNull()
{
    output = Homework2.effettuaOperazioni( n: 1, m: 1, nomi: null,new int[]{1});
    assertEquals( expected: "Una delle due o entrambe le liste sono nulle",output.messaggio);
}

//T4 Abbiamo deciso di non testare i casi in cui nomi o numeri sono vuoti poichè questo test include nella propria partizione anche quest'ultimi due casi
// permettendoci di effettuare due test in meno.
no usages  ▲ Fabio18 *
@Test
@DisplayName("numeriAndNomiDifferentLength")
void numeriAndNomiDifferentLength()
{
    output = Homework2.effettuaOperazioni( n: 1, m: 1, new String[]{},new int[]{5});
    assertEquals( expected: "La lunghezza di uno dei due o entrambe le liste non corrisponde alla dimensione fornita in input",output.messaggio);
    output2 = Homework2.effettuaOperazioni( n: 1, m: 1, new String[]{"Federico"},new int[]{});
    assertEquals( expected: "La lunghezza di uno dei due o entrambe le liste non corrisponde alla dimensione fornita in input", output2.messaggio);
}

//T5
no usages  ▲ Fabio18 *
@Test
@DisplayName("nomiHasElementNull")
void nomiHasElementNull()
{
    output = Homework2.effettuaOperazioni( n: 2, m: 2, new String[]{"Gianluca",null},new int[]{5,6});
    assertEquals( expected: "Non e' possibile inserire stringhe nulle",output.messaggio);
}
```

T6:

```
//Casi di test per soli numeri pari il cui quadrato è minore di 100:
//T6
no usages  Fabio18 *
@Test
@DisplayName("numeriHasEvenElementSquaredMinor100")
void numeriHasEvenElementSquaredMinor100()
{
    output = Homework2.effettuaOperazioni( n: 2, m: 2, new String[]{"Giovanni","Aldo"},new int[]{6,4});
    assertEquals(new String[]{"Aldo","Giovanni"},output.nomi);
    assertEquals(new int[]{16,36}, output.numeri.stream().mapToInt(Integer::intValue).toArray());
}
```

T7:

```
//Caso di test per soli numeri dispari il cui cubo è minore di 200:
//T7
no usages  new *
@Test
@DisplayName("numeriHasOddElementCubedMinor200")
void numeriHasOddElementCubedMinor200()
{
    output = Homework2.effettuaOperazioni( n: 2, m: 2, new String[]{"Giuseppe","Franco"},new int[]{3,5});
    assertEquals(new String[]{"Franco","Giuseppe"},output.nomi);
    assertEquals(new int[]{27,125}, output.numeri.stream().mapToInt(Integer::intValue).toArray());
}
```

T8, T9, T10, T11, T12:

```
//Boundary Test
//T8
no usages  new *
@Test
@DisplayName("nAndmAreZero")
void nAndmAreZero()
{
    output = Homework2.effettuaOperazioni( n: 0, m: 0, new String[]{},new int[]{});
    assertEquals( expected: "Non e' possibile inserire una grandezza di dimensione 0 o minore",output.messaggio);
}

//T9
no usages  new *
@Test
@DisplayName("NumeriHasNegative")
void NumeriHasNegative()
{
    output = Homework2.effettuaOperazioni( n: 2, m: 2, new String[]{"Marco","Andrea"},new int[]{5,-1});
    assertEquals( expected: "Non e' possibile inserire un numero negativo",output.messaggio);
}

//T10
no usages  new *
@Test
@DisplayName("NumeriHasNegative")
void NumeriMajor99()
{
    output = Homework2.effettuaOperazioni( n: 2, m: 2, new String[]{"Fabio","Antonietta"},new int[]{100,5});
    assertEquals( expected: "Il numero inserito e' troppo grande",output.messaggio);
}
```

```

//T11
no usages new *
@Test
@DisplayName("NumeriHasNegative")
void NumeriElementSquaredMajor100()
{
    output = Homework2.effettuaOperazioni( n: 2, m: 2, new String[]{"Beatrice","Gianmarco"},new int[]{12,4});
    assertEquals(new String[]{"Beatrice","Gianmarco"},output.nomi);
    assertEquals(new int[]{16}, output.numeri.stream().mapToInt(Integer::intValue).toArray());
}

//T12
no usages new *
@Test
@DisplayName("NumeriHasNegative")
void NumeriElementCubedMajor200()
{
    output = Homework2.effettuaOperazioni( n: 2, m: 2, new String[]{"Francesca","Chiara"},new int[]{11,1});
    assertEquals(new String[]{"Chiara","Francesca"},output.nomi);
    assertEquals(new int[]{1}, output.numeri.stream().mapToInt(Integer::intValue).toArray());
}

```

7) Aumenta i casi di test della suite con creatività ed esperienza

In seguito al completamento del 1.6 abbiamo raggiunto il 100% di Code Coverage effettuando 12 test (minor numero di test), rendendoci conto di avere la necessità di aggiungere due test nella prospettiva di analisi black box.

T13: nomi ha m elementi di cui almeno una stringa è vuota, numeri ha n elementi, n e m sono uguali

T14: numeri ha n elementi pari aventi i quadrati minori di 100 e dispari aventi i cubi minori di 200, nomi ha m elementi, n e m sono uguali

```

//Test Black Box
//T13
no usages new *
@Test
@DisplayName("NomiHasEmptyString")
void NomiHasEmptyString()
{
    output = Homework2.effettuaOperazioni( n: 2, m: 2, new String[]{"Ferdinando",""},new int[]{2,6});
    assertEquals(new String[]{"","Ferdinando"},output.nomi);
    assertEquals(new int[]{4,36}, output.numeri.stream().mapToInt(Integer::intValue).toArray());
}

//T14
no usages new *
@Test
@DisplayName("numeriHasEvenElementSquaredMinor100AndOddElementCubedMinor200")
void numeriHasEvenElementSquaredMinor100AndOddElementCubedMinor200()
{
    output = Homework2.effettuaOperazioni( n: 2, m: 2, new String[]{"Giuseppe","Franco"},new int[]{3,4});
    assertEquals(new String[]{"Franco","Giuseppe"},output.nomi);
    assertEquals(new int[]{16,27}, output.numeri.stream().mapToInt(Integer::intValue).toArray());
}

```

EffettuaOperazioni

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
● <code>effettuaOperazioni(int, int, String[], int[])</code>	<div></div>	100%	<div></div>	100%	0	15	0	33	0	1

```

39.     public static Output effettuaOperazioni(int n, int m, String[] nomi, int[] numeri) throws RuntimeException
40.     {
41.
42.         if (n != m) {
43.             return new Output("Il numero di numeri e nomi inseriti non corrisponde");
44.         }
45.         else
46.         {
47.             if (n <= 0) {
48.                 return new Output("Non e' possibile inserire una grandezza di dimensione 0 o minore");
49.             }
50.             else {
51.                 if (nomi == null || numeri == null) {
52.                     return new Output("Una delle due o entrambe le liste sono nulle");
53.                 }
54.
55.                 if (nomi.length != m || numeri.length != n) {
56.                     return new Output("La lunghezza di uno dei due o entrambe le liste non corrisponde alla dimensione fornita in input");
57.                 }
58.                 for (int i = 0; i <= n - 1; i++) {
59.                     if (numeri[i] < 0) {
60.                         return new Output("Non e' possibile inserire un numero negativo");
61.                     }
62.                     if (numeri[i] > 99) {
63.                         return new Output("Il numero inserito e' troppo grande");
64.                     }
65.                     if (nomi[i] == null) {
66.                         return new Output("Non e' possibile inserire stringhe nulle");
67.                     }
68.                 }
69.             }
70.         }
71.
72.         // Calcola il quadrato dei numeri pari e il cubo dei numeri dispari
73.         List<Integer> quadrati = new ArrayList<>();
74.         List<Integer> cubi = new ArrayList<>();
75.         for (int i = 0; i <= n - 1; i++) {
76.             if (numeri[i] % 2 == 0) {
77.                 int quadrato = numeri[i] * numeri[i];
78.                 if (quadrato < 100) {
79.                     quadrati.add(quadrato);
80.                 }
81.             } else {
82.                 int cubo = numeri[i] * numeri[i] * numeri[i];
83.                 if (cubo < 200) {
84.                     cubi.add(cubo);
85.                 }
86.             }
87.         }
88.
89.         // Unisci i quadrati e i cubi in un unico array e ordina il tutto
90.         ArrayList<Integer> numeriOrdinati = new ArrayList<>();
91.         numeriOrdinati.addAll(quadrati);
92.         numeriOrdinati.addAll(cubi);
93.         Collections.sort(numeriOrdinati);
94.
95.         Arrays.sort(nomi);
96.         Output output = new Output(nomi, numeriOrdinati);
97.         return output;
98.     }
99. }
100.
101. }

```

Homework n°3

Per la realizzazione del terzo Homework, abbiamo realizzato un codice che simula il funzionamento del gioco del “Superenalotto”. Il codice presenta i seguenti requisiti:

- 1) I numeri che compongono una scheda sono compresi tra 1 e 90.
- 2) Una scheda vince quando tutti i suoi numeri sono presenti all’interno della scheda vincente (non importa l’ordine).
- 3) Se almeno un numero nella scheda non è presente nella scheda vincente, allora la scheda perde.

Le proprietà sulla base dei requisiti sono:

- 1) **lose**, se almeno un numero compreso tra 1 (incluso) e 90 (incluso) non è presente nella scheda vincente, il programma restituisce false.
- 2) **win**, se tutti i numeri compresi tra 1 (incluso) e 90 (incluso) sono presenti nella scheda vincente, il programma restituisce true.
- 3) **Invalid**, se almeno un numero è minore di 1 (escluso) e maggiore di 90 (escluso), il programma genera un’*InvalidParameterException*.

Scriviamo il PBT relativo alla proprietà **lose**:

```
@Property(tries = 728999)
@Report(Reporting.GENERATED)
@StatisticsReport(format = Histogram.class)
void lose(@ForAll("Cards") int[] scheda) {
    int[] schedaVincente = Homework3.schedaVincente();
    Statistics.collect(Homework3.haveSameElements(schedaVincente, scheda));
    if(!(Homework3.haveSameElements(schedaVincente, scheda)))
    {
        assertFalse(Homework3.superenalotto(scheda, schedaVincente));
    }
}
```

Report sulle statistiche:

```
timestamp = 2023-06-28T16:29:04.854503300, [TestHomework3:lose] (728999) statistics =
# | label | count |
----|-----|-----|-----
0 | false | 728995 | #####
1 | true  | 4      |
```

In seguito alla raccolta dei dati statistici, su 728999 tentativi la possibilità di vincere è di circa $\frac{4}{728999}$.

Scriviamo il PBT relativo alla proprietà **win**:

```
@Property(tries = 728999)
@Report(Reporting.GENERATED)
@StatisticsReport(format = Histogram.class)
void win(@ForAll("Cards") int[] scheda) {
    int[] schedaVincente = Homework3.schedaVincente();
    Statistics.collect(Homework3.haveSameElements(schedaVincente, scheda));
    if(Homework3.haveSameElements(schedaVincente, scheda))
    {
        assertTrue(Homework3.superenalotto(scheda, schedaVincente));
    }
}

no usages
@Provide
private Arbitrary<int[]> Cards()
{
    return Arbitraries.integers().between(1,90).array(int[].class).ofSize(n);
}
```

Report sulle statistiche:

```
timestamp = 2023-06-28T16:16:40.469257100, [TestHomework3:win] (728999) statistics =
# | label | count |
----|-----|-----|
0 | false | 728995 | #####
1 | true  | 4      |
```

In seguito alla raccolta dei dati statistici, su 728999 tentativi la possibilità di vincere è di circa $\frac{4}{728999}$.

Nota: Per avere un calcolo più accurato sarebbe necessario effettuare un calcolo statistico più approfondito, ricordando che il gioco del Superenalotto è basato sul caso.

Scriviamo il PBT relativo alla proprietà **invalid**:

```
@Property(tries = 1000)
@Report(Reporting.GENERATED)
@StatisticsReport(format = Histogram.class)
void invalid(@ForAll("invalidInputs") int[] schedas) {
    int[] schedasVincenti = Homework3.schedasVincenti();
    for(int i=0; i<n; i++)
    {
        Statistics.collect( ...values: schedas[i] < 1);
    }
    assertThrows(InvalidParameterException.class, () -> {
        Homework3.superenalotto(schedas, schedasVincenti);
    });
}

no usages
@Provide
private Arbitrary<int[]> invalidInputs() {
    Arbitrary<Integer> invalidNumbers = Arbitraries.integers()
        .filter(number -> number < 1 || number > 90);

    return invalidNumbers.array(int[].class).ofSize(n);
}
```

Report sulle statistiche:

```
timestamp = 2023-06-28T17:12:00.329613700, [TestHomework3:invalid] (3000) statistics =
# | label | count |
-----|-----|-----|-----
0 | false | 1229 | #####
1 | true  | 1771 | #####
```

Il **false** rappresenta i numeri maggiori di 90, mentre il **true** rappresenta i numeri minori di 1.