UNTITLED WARE

Integrazione e Test di Sistemi Software a.a. 2022 - 2023

Realizzato da:

Napoletano Alberto 738558 ITPS – a.napoletano32@studenti.uniba.it
Peragine Antonio 746886 ITPS – a.peragine14@studenti.uniba.it
Petrov Igor 735287 ITPS – i.petrov@studenti.uniba.it

Indice

metodo — getBestPrice()

HOMEWORK 1 **3**
Specification-based testing 3
1. COMPRENDERE I REQUISITI 3
2. ESPLORARE COSA FA IL PROGRAMMA PER DIVERSI INPUT 4
3. INPUT, OUTPUT E PARTIZIONI 5
4. CASI LIMITE 6
5. DEFINIRE CASI DI TEST 7
6. AUTOMATIZZARE CASI DI TEST 9
7. ARRICCHIRE LA SUITE DI TEST 19

HOMEWORK 2 **21**
TASK 1 21
Structural testing 21
- COMPRENDERE IL CODICE 22
- CODE COVERAGE 23
- INDIVIDUARE CASI DI TEST MANCANTI 24

Casi di Test falliti 26
Mutation Testing 28

metodo — calculateAverage()

TASK 2 30
CODE COVERAGE TOTALE CON IL MINOR NUMERO DI TEST 30
È SUFFICIENTE? 33
Specification-based testing 34
1. COMPRENDERE I REQUISITI 34
2. ESPLORARE COSA FA IL PROGRAMMA PER DIVERSI INPUT 35
3. INPUT, OUTPUT E PARTIZIONI 36
4. CASI LIMITE 37
5. DEFINIRE CASI DI TEST 38
6. AUTOMATIZZARE CASI DI TEST 39

HOMEWORK 3 **40**
Property based testing 40
Proprietà 1 : shouldCalculateAverage 41
RACCOLTA DATI 42
Proprietà 2 : invalidIndexes 43
RACCOLTA DATI 44
Casi non coperti 45

HOMEWORK 1

Specification-based testing

1. COMPRENDERE I REQUISITI

Il metodo ***getBestPrice*** viene utilizzato affinché le persone possano trovare i posti disponibili più economici possibili per un treno e ritorna il prezzo totale.
Se ci sono meno posti di quanti richiesti, il metodo restituisce il prezzo dei soli posti disponibili.
Se il prezzo totale è maggiore di 100 euro, viene applicato uno sconto di 5.

Il programma riceve tre parametri:

- *prices* – vettore contenente il prezzo (*double*) per ogni posto
- *taken* – vettore che indica per ogni posto se questo è già stato prenotato (**false** → libero ; **true** → occupato)
- *numberOfSeats* – indica il numero di posti richiesti dall'utente (*integer*). Un utente deve richiedere almeno un posto.

I due vettori *prices* e *taken* non possono essere null e devono avere lo stesso numero di elementi.

Il programma ritorna il prezzo totale per i posti richiesti.

2. ESPLORARE COSA FA IL PROGRAMMA PER DIVERSI INPUT

Per capire meglio il funzionamento del programma abbiamo deciso di testare cosa fa il programma dati questi input:

1. Vettore *taken* con 5 elementi, di cui 3 ***false***, e numberOfSeats pari a 2

```
@Test
void BuyTwoSeats(){
    assertEquals( expected: 12.98, FindBestSeat.getBestPrice(
            new double[]{7.99,4.99,9.90,10,8}, new boolean[]{false, false, true, true, false},  numberOfSeats: 2));
}
```

2. *numberOfSeats* pari al numero di elementi false del vettore *taken*

```
@Test
void BuyAllAvailableSeats(){
    assertEquals( expected: 25.89, FindBestSeat.getBestPrice(
            new double[]{7.99,4.99,9.90,10,8}, new boolean[]{false, true, false, true, false},  numberOfSeats: 3));
}
```

3. Vettore *taken* con un singolo elemento uguale a ***false*** e *numberOfSeats* superiore a 1

```
@Test
void RequestedSeatsMoreThanAvailable(){
    assertEquals( expected: 4.99, FindBestSeat.getBestPrice(
            new double[]{7.99,4.99,9.90,10,8}, new boolean[]{true, false, true, true, true},  numberOfSeats: 4));
}
```

4. Vettore *prices* con 2 elementi più piccoli di pari valore, vettore *taken* con elementi false e *numberOfSeats* superiore a 2

```
@Test
void MultipleSeatsWithSamePrice(){
    assertEquals( expected: 17.97, FindBestSeat.getBestPrice(
            new double[]{7.99,4.99,9.90,10,4.99}, new boolean[]{false, false, false, false, false},  numberOfSeats: 3));
}
```

3. INPUT, OUTPUT E PARTIZIONI

a) Classi di input

prices (array of doubles):

- Null
- Empty
- Non-empty (>=1)

price (array element):

- Positive (> 0)
- Negative (<= 0)

taken (array of booleans):

- Null
- Empty
- Non-empty (>=1)

taken (array element):

- True
- False

numberOfSeats:

- >= 1
- < 1

b) Combinazioni di input

(*prices*, *taken*):

- Same size
- Different size

(*taken*, *numberOfSeats*):

- *numberOfSeats* > size(*taken*)
- *numberOfSeats* <= available seats
- *numberOfSeats* > available seats

L'aggiunta del parametro *prices*,
non genera nuove combinazioni di input,
per questo è stato deciso di non considerarlo.

c) Classi di output attesi

totalPrice (output):

- > 100 (- 5)
- <= 100

4. CASI LIMITE

Boundaries (on Point):

- size(*prices*) = 1
- size(*taken*) = 1
- price (*prices* element) = 0
- *numberOfSeats* = 1
- size(*prices*) = size(*taken*)
- *numberOfSeats* = size(*taken*)
- *numberOfSeats* = available seats

Boundaries (off Point):

- size(*prices*) = 0
- size(*taken*) = 0
- price (*prices* element) = 0 + Ɛ
- *numberOfSeats* = 0
- size(*prices*) = size(*taken*)-1
- size(*prices*) = size(*taken*)+1
- *numberOfSeats* = size(*taken*)+1
- *numberOfSeats* = available seats+1

Consideriamo inoltre casi boundary per le classi di output attesi

Boundaries (**output**):

- *totalPrice* = 100 (on Point)
- *totalPrice* = 100 + Ɛ (off Point)

Ɛ sarà un numero piccolo maggiore di 0.
Ma quanto vale Ɛ? Dato il contesto d'uso (denaro), aggiunto al fatto che non è diversamente specificato nella documentazione, assumiamo che questo prenda valore ***0.01***

5. DEFINIRE CASI DI TEST

a) Test dei casi eccezionali:

T1: vettore *prices* null
T2: vettore *taken* null
T3: vettori *prices* e *taken* null

T4: *numberOfSeats* < 1

b) Test per i vettori di dimensione diversa da zero:

T5: uno o più elementi di *prices* sono negativi

T6: tutti gli elementi di *taken* sono true (nessun posto disponibile)

T7: i vettori *prices* e *taken* hanno dimensioni diverse

c) Combinazione di input:

prices size = taken size
numberOfSeats >= 1

T8: *numberOfSeats* supera la dimensione del vettore *taken*
In base ai requisiti, il test T8 dovrebbe reagire allo stesso modo del test T9. Per sicurezza e completezza, tuttavia, consideriamo due casi separati.

T9: *numberOfSeats* supera il numero di posti disponibili
T10: *numberOfSeats* è inferiore al numero di posti disponibili

(Ricordarsi che un posto è disponibile
quando il relativo elemento di *taken* è uguale a *false*)

d) Output attesi:

T11: la somma dei prezzi più bassi supera 100
Il caso opposto, in cui il prezzo non supera 100, è implicitamente coperto da altri casi di test. Di conseguenza, appare ridondante pensare a un test dedicato.

e) Boundaries:

condizione → (*size(prices) = size(taken) >= 1*)
Non ha senso considerare singolarmente le condizioni su *prices* e *taken*, perché rientriamo nel caso in cui i due vettori sono di grandezza diversa.

T12: i vettori *prices* e *taken* hanno dimensione 1

T13: i vettori *prices* e *taken* hanno dimensione 0
In base ai requisiti, il test T13 dovrebbe reagire allo stesso modo del test T9. Per sicurezza e completezza, tuttavia, consideriamo due casi separati.

condizione → (*prices' elements > 0*)
T14: uno o più elementi di *prices* sono pari a 0
T15: uno o più elementi di *prices* sono pari a 0 + Ɛ (Ɛ = 0.01)

condizione → (*numberOfSeats >= 1*)
T16: *numberOfSeats* è pari a 1
T17: *numberOfSeats* è pari a 0

condizione → (*size(prices) = size(taken*))
Il caso in cui i due vettori hanno la stessa dimensione è implicitamente coperto da altri casi di test. Di conseguenza, appare ridondante pensare a un test dedicato.

T18: Il vettore *prices* ha un elemento in meno del vettore *taken*
T19: Il vettore *prices* ha un elemento in più del vettore *taken*

condizione → (*numberOfSeats <= available seats*)
T20: *numberOfSeats* è pari alla dimensione del vettore *taken*
T21: *numberOfSeats* è pari alla dimensione del vettore *taken* + 1

condizione → (*numberOfSeats <= available seats*)
T22: *numberOfSeats* è pari al numero di posti disponibili
T23: *numberOfSeats* è pari al numero di posti disponibili + 1

f) Boundaries output generati:

T24: la somma dei prezzi più bassi è pari a 100
T25: la somma dei prezzi più bassi è pari a 100 + Ɛ (Ɛ = 0.01)

6. AUTOMATIZZARE CASI DI TEST

T1: vettore *prices* null
T2: vettore *taken* null
T3: vettori *prices* e *taken* null

```
@ParameterizedTest
@MethodSource("vectorsNullProvider")
@DisplayName("Check what method does if Vectors are null")
void VectorsNull(double[] prices, boolean[] taken){
    assertThrows(IllegalArgumentException.class,()->FindBestSeat.getBestPrice(prices, taken,  numberOfSeats: 3));
}

1 usage   Igor Petrov *
static Stream<Arguments> vectorsNullProvider(){
    return Stream.of(
            arguments(null, new boolean[]{true, true, false}), // prices null       T1
            arguments(new double[]{5.0, 3.0, 5.5, 4.2}, null), // taken null        T2
            arguments(null, null) // both null                                      T3
    );
}
```

T4: *numberOfSeats* < 1
T17: *numberOfSeats* è pari a 0

```
@ParameterizedTest
@ValueSource(ints = {-48 /* T4 */, 0 /* T17 */})
@DisplayName("Check what method does if the number of seats is zero (boundary off-point) or below")
void NumberOfSeatsZeroOrBelow(int numberOfSeats){
    assertThrows(IllegalArgumentException.class,
            () -> FindBestSeat.getBestPrice(new double[]{5.0, 3.0}, new boolean[]{true, false}, numberOfSeats));
}
```

T5: uno o più elementi di *prices* sono negativi
T14: uno o più elementi di *prices* sono pari a 0
T15: uno o più elementi di *prices* sono pari a 0 + Ɛ (Ɛ = 0.01)

```
@Test
@DisplayName("Buying tickets with zero or negative prices. *Includes boundary cases*")
void ZeroOrNegativePrices(){
    Assertions.assertAll(
            () -> assertThrows(IllegalArgumentException.class, () -> FindBestSeat.getBestPrice(
                    new double[]{2.0,3.0,-3.0,2.0,-4.5,-3.5},
                    new boolean[]{false,false,false,false,true,true},
                     numberOfSeats: 3)),                                                    // T5
            () -> assertThrows(IllegalArgumentException.class, () -> FindBestSeat.getBestPrice(
                    new double[]{2.0,3.0,0.0,12.0,0.0,13.5},
                    new boolean[]{false,false,false,false,true,true},
                     numberOfSeats: 3)),                                                    // T14
            () -> assertEquals( expected: 4.01, FindBestSeat.getBestPrice(
                    new double[]{2.0,3.0,0.01,2.0,4.5,3.5},
                    new boolean[]{false,false,false,false,true,true},
                     numberOfSeats: 3))                                                     // T15
    );
}
```

T6: tutti gli elementi di *taken* sono true (nessun posto disponibile)
T8: *numberOfSeats* supera la dimensione del vettore *taken*
T9: *numberOfSeats* supera il numero di posti disponibili
T12: i vettori *prices* e *taken* hanno dimensione 1
T13: i vettori *prices* e *taken* hanno dimensione 0

```java
@Test
@DisplayName("Check what method does if the number of seats is above availability. *Includes boundary cases*")
void NumberOfSeatsAboveAvailable(){
    Assertions.assertAll(
            () -> assertEquals( expected: 0.0, FindBestSeat.getBestPrice(
                    new double[]{5.0, 3.0, 9.0}, new boolean[]{true, true, true}, numberOfSeats: 3)),              // T6
            () -> assertEquals( expected: 7.0, FindBestSeat.getBestPrice(
                    new double[]{5.0, 3.0, 9.0, 4.0}, new boolean[]{true, false, true, false}, numberOfSeats: 10)), // T8
            () -> assertEquals( expected: 14.0, FindBestSeat.getBestPrice(
                    new double[]{5.0, 3.0, 9.0, 8.0}, new boolean[]{false, true, false, true}, numberOfSeats: 4)),  // T9
            ()-> assertEquals( expected: 7.0, FindBestSeat.getBestPrice(
                    new double[]{7.0}, new boolean[]{false}, numberOfSeats: 3)),                                    // T12
            () -> assertEquals( expected: 0.0, FindBestSeat.getBestPrice(
                    new double[]{}, new boolean[]{}, numberOfSeats: 3))                                             // T13
    );
}
```

T7: i vettori *prices* e *taken* hanno dimensioni diverse
T18: Il vettore *prices* ha un elemento in meno del vettore *taken*
T19: Il vettore *prices* ha un elemento in più del vettore *taken*

```java
@ParameterizedTest
@MethodSource("differentSizeVectorsProvider")
@DisplayName("Check what method does if prices vector's length is not equal taken vector's length")
void PriceVectorNotEqualTakenVector(double[] prices, boolean[] taken){
    assertThrows(IllegalArgumentException.class,
            () -> FindBestSeat.getBestPrice(prices, taken, numberOfSeats: 3));
}

1 usage  new *
static Stream<Arguments> differentSizeVectorsProvider(){
    return Stream.of(
            arguments(new double[]{5.0, 3.0}, new boolean[]{true, true, false, true, false}),   // T7
            arguments(new double[]{5.0, 3.0}, new boolean[]{true, true, false}),                // T18
            arguments(new double[]{5.0, 3.0, 7.0}, new boolean[]{true, true})                   // T19
    );
}
```

T10: *numberOfSeats* è inferiore al numero di posti disponibili

```java
@Test
@DisplayName("Check what method does if the number of seats is below availability")
void NumberOfSeatsBelowAvailable(){
    assertEquals( expected: 7.0, FindBestSeat.getBestPrice(
            new double[]{2.0,3.0,3.0,2.0,4.5,3.5},
            new boolean[]{false,false,false,false,true,true},
            numberOfSeats: 3));                                     // T10
}
```

T11: la somma dei prezzi più bassi supera 100
T24: la somma dei prezzi più bassi è pari a 100
T25: la somma dei prezzi più bassi è pari a 100 + Ɛ (Ɛ = 0.01)

```
@Test
@DisplayName("Buying tickets for a total price greater than 100 for having a discount. *Includes boundary cases*")
void TotalPriceGreaterThanOneHundred(){
    Assertions.assertAll(
            ()-> assertEquals( expected: 118.5 - 5, FindBestSeat.getBestPrice(
                    new double[]{50.5, 30, 49.5, 69}, new boolean[]{true, true, false, false}, numberOfSeats: 2)),     // T11
            ()-> assertEquals( expected: 100, FindBestSeat.getBestPrice(
                    new double[]{50.5, 30, 49.5, 69}, new boolean[]{false, true, false, false}, numberOfSeats: 2)),    // T24
            ()-> assertEquals( expected: 100.01 - 5, FindBestSeat.getBestPrice(
                    new double[]{50.5, 30, 49.51, 69}, new boolean[]{false, true, false, false}, numberOfSeats: 2),    // T25
                    2)
    );
}
```

T16: *numberOfSeats* è pari a 1

```
@Test
@DisplayName("Check what method does if only 1 seat is requested")
void OneRequestedSeatBoundaryCase(){
    assertEquals( expected: 2.0, FindBestSeat.getBestPrice(
            new double[]{2.0,3.0,3.0,2.0,4.5,3.5},
            new boolean[]{false,false,false,false,true,true},
            numberOfSeats: 1));                                                    // T16
}
```

T20: *numberOfSeats* è pari alla dimensione del vettore *taken*
T21: *numberOfSeats* è pari alla dimensione del vettore *taken* + 1
T22: *numberOfSeats* è pari al numero di posti disponibili
T23: *numberOfSeats* è pari al numero di posti disponibili + 1

```
@Test
@DisplayName("Check what method does if the number of seats equals availability")
void NumberOfSeatsEqualsAvailable(){
    Assertions.assertAll(
            ()-> assertEquals( expected: 10.0, FindBestSeat.getBestPrice(
                    new double[]{2.0,4.5,3.5},
                    new boolean[]{false,false,false},
                    numberOfSeats: 3)),                                    // T20
            ()-> assertEquals( expected: 10.0, FindBestSeat.getBestPrice(
                    new double[]{2.0,4.5,3.5},
                    new boolean[]{false,false,false},
                    numberOfSeats: 4)),                                    // T21
            ()-> assertEquals( expected: 7.0, FindBestSeat.getBestPrice(
                    new double[]{2.0,3.0,3.0,2.0,4.5,3.5},
                    new boolean[]{false,false,true,false,true,true},
                    numberOfSeats: 3)),                                    // T22
            ()-> assertEquals( expected: 5.5, FindBestSeat.getBestPrice(
                    new double[]{2.0,4.5,3.5},
                    new boolean[]{false,true,false},
                    numberOfSeats: 3))                                     // T23
    );
}
```

Method Name	getBestPrice
Pre-conditions	- Gli array di input *prices* e *taken* non devono essere *null* - Gli array di input *prices* e *taken* devono avere la stessa dimensione - *numberOfSeats* non deve essere più piccolo di 1
Post-conditions	- Ritorna il prezzo totale per i posti più economici disponibili

No.	**Test Condition**	**Test Steps**	**Test Input**	**Expected Output**	**Actual Output**	**Result**
T1	Verificare il comportamento del metodo quando l'array di input *prices* è null	1. Il metodo riceve l'array *prices* **null** 2. Verificare se la pre-condizione è rispettata	prices = null --- taken = { true , true, false } --- numberOfSeats = 3	java.lang.IllegalArgumentException	java.lang.IllegalArgumentException	PASSED

T2	Verificare il comportamento del metodo quando l'array di input *taken* è null	1. Il metodo riceve l'array *taken* **null** 2. Verificare se la pre-condizione è rispettata	prices = { 5.0 , 3.0 , 5.5 , 4.2 } --- taken = null --- numberOfSeats = 3	java.lang.IllegalArgumentException	java.lang.IllegalArgumentException	PASSED
T3	Verificare il comportamento del metodo quando gli array di input sono null	1.Il metodo riceve gli array *prices* e *taken* uguali a **null** 2. Verificare se la pre-condizione è rispettata	prices = null --- taken = null --- numberOfSeats = 3	java.lang.IllegalArgumentException	java.lang.IllegalArgumentException	PASSED
T4	Verificare il comportamento del metodo quando il numero di posti richiesti è inferiore a 1	1. Il metodo riceve il parametro *numberOfSeats* inferiore a 1 2. Verificare se la pre-condizione è rispettata	prices = { 5.0 , 3.0 } --- taken = { true , false } --- numberOfSeats = -48	java.lang.IllegalArgumentException	java.lang.IllegalArgumentException	PASSED
T5	Verificare il comportamento del metodo quando l'array di input *prices* contiene elementi negativi	1. Il metodo riceve l'array *prices* con elementi negativi 2. Verificare se l'array *prices* non contiene valori negativi	prices = { 2.0 , 3.0 , -3.0 , 2.0 , -4.5, -3.5 } --- taken = { false , false , false , false , true , true } --- numberOfSeats = 3	java.lang.IllegalArgumentException	-2.0	FAILED
T6	Verificare il comportamento del metodo quando non ci sono posti disponibili	1. il metodo riceve l'array *taken* con tutti gli elementi posti a **true** 2. Il metodo calcola la somma degli elementi 3. Verificare se il metodo restituisce il valore corretto	prices = { 5.0 , 3.0 , 9.0 } --- taken = { true , true , true } --- numberOfSeats = 3	0.0	0.0	PASSED

T7	Verificare il comportamento del metodo quando i due array hanno dimensioni diverse	1. il metodo riceve i due array *prices* e *taken* con dimensioni diverse 2. Verificare se la pre-condizione è rispettata	prices = { 5.0 , 3.0 } --- taken = { true , true , false , true , false } --- numberOfSeats = 3	java.lang.IllegalArgumentException	java.lang.IllegalArgumentException	PASSED
T8	Verificare il comportamento del metodo quando il numero di posti richiesti supera le dimensioni dell'array *taken*	1. Il metodo riceve il parametro *numberOfSeats* con un valore maggiore della dimensione dell'array *taken* 2. Il metodo fa la somma dei prezzi dei posti disponibili 3. Verificare se il metodo restituisce il valore corretto	prices = { 5.0 , 3.0 , 9.0 , 4.0 } --- taken = { true , false , true , false } --- numberOfSeats = 10	7.0	7.0	PASSED
T9	Verificare il comportamento del metodo quando il numero di posti richiesti supera il numero di posti disponibili	1. Il metodo riceve il parametro *numberOfSeats* con un valore maggiore del numero di elementi **false** di *taken* 2. Il metodo fa la somma dei prezzi dei posti disponibili 3. Verificare se il metodo restituisce il valore corretto	prices = { 5.0 , 3.0 , 9.0 , 4.0 } --- taken = { false , true , false , true } --- numberOfSeats = 4	14.0	14.0	PASSED
T10	Verificare il comportamento del metodo quando il numero di posti richiesti è inferiore al numero di posti disponibili	1.. Il metodo riceve il parametro *numberOfSeats* con un valore minore del numero di elementi **false** di *taken* 2.Il metodo fa la somma dei prezzi dei posti disponibili 3. Verificare se il metodo restituisce il valore corretto	prices = { 2.0 , 3.0 , 3.0 , 2.0 , 4.5 , 3.5 } --- taken = { false , false , false , false , true , true } --- numberOfSeats = 3	7.0	7.0	PASSED

T11	Verificare il comportamento del metodo quando la somma dei prezzi è più grande di 100	1. Il metodo riceve i due array *prices* e *taken* e la variabile *numberOfSeats* 2. Il metodo fa la somma dei prezzi dei posti disponibili 3. La somma è maggiore di 100, quindi viene applicato uno sconto di 5 4. Verificare se il metodo restituisce il valore corretto	prices = { 50.5 , 30 , 49.5 , 69 } taken = { true , true , false , false } numberOfSeats = 2	118.5 - 5.0 (***113.5***)	113.5	PASSED
T12	Verificare il comportamento del metodo quando i due array hanno dimensione 1	1. Il metodo riceve i due array *prices* e *taken*, con dimensione 1 2. Il metodo fa la somma dei prezzi dei posti disponibili 3. Verificare se il metodo restituisce il valore corretto	prices = { 7.0 } taken = { false } numberOfSeats = 3	7.0	7.0	PASSED
T13	Verificare il comportamento del metodo quando i due array sono vuoti (dimensione 0)	1.Il metodo riceve i due array *prices* e *taken* vuoti 2. Il metodo fa la somma dei prezzi dei posti disponibili 3. Verificare se il metodo restituisce il valore corretto	prices = { } taken = { } numberOfSeats = 3	0.0	0.0	PASSED
T14	Verificare il comportamento del metodo quando l'array di input *prices* contiene elementi pari a 0.0	1. Il metodo riceve l'array *prices* con elementi pari a 0 2. Verificare se l'array *prices* non contiene valori pari a 0	prices = { 2.0 , 3.0 , 0.0 , 2.0 , 0.0 , 13.5 } taken = { false , false , false , false , true , true } numberOfSeats = 3	java.lang.IlleglArgumentException	5.0	FAILED

T15	Verificare il comportamento del metodo quando l'array di input *prices* contiene elementi pari a 0 + Ɛ (Ɛ = 0.01)	1.Il metodo riceve l'array *prices* contente un elemento di valore 0.01 2. Il metodo fa la somma dei prezzi dei posti disponibili 3. Verificare se il metodo restituisce il valore corretto	prices = { 2.0 , 3.0 , 0.01 , 2.0 , 4.5 , 3.5 } --- taken = { false , false , false , false , true , true } --- numberOfSeats = 3	4.01	4.01	PASSED
T16	Verificare il comportamento del metodo quando viene richiesto un posto	1. Il metodo riceve il parametro *numberOfSeats* pari a 1 2. Il metodo fa la somma dei prezzi dei posti disponibili 3. Verificare se il metodo restituisce il valore corretto	prices = { 2.0 , 3.0 , 3.0 , 2.0 , 4.5 , 3.5 } --- taken = { false, false , false , false , true , true } --- numberOfSeats = 1	2.0	2.0	PASSED
T17	Verificare il comportamento del metodo quando vengono richiesti 0 posti	1. Il metodo riceve il parametro *numberOfSeats* pari a 0 2. Verificare se la pre-condizione è rispettata	prices = { 5.0 , 3.0 } --- taken = { true , false } --- numberOfSeats = 0	java.lang.IllegalArgumentException	java.lang.IllegalArgumentException	PASSED
T18	Verificare il comportamento del metodo quando l'array *prices* ha un elemento in meno dell'array *taken*	1. Il metodo riceve l'array *prices* con 1 elemento in meno di *taken* 2. Verificare se la pre-condizione è rispettata	prices = { 5.0 , 3.0 } --- taken = { true , true , false } --- numberOfSeats = 3	java.lang.IllegalArgumentException	java.lang.IllegalArgumentException	PASSED
T19	Verificare il comportamento del metodo quando l'array *prices* ha un elemento in più dell'array *taken*	1. Il metodo riceve l'array *prices* con 1 elemento in meno di *taken* 2. Verificare se la pre-condizione è rispettata	prices = { 5.0 , 3.0 , 7.0 } --- taken = { true , true } --- numberOfSeats = 3	java.lang.IllegalArgumentException	java.lang.IllegalArgumentException	PASSED

T20	Verificare il comportamento del metodo quando il numero di posti richiesti è pari alla dimensione dell'array *taken*	1. Il metodo riceve il parametro *numberOfSeats* con valore uguale alla dimensione di *taken* 2. Il metodo fa la somma dei prezzi dei posti disponibili 3. Verificare se il metodo restituisce il valore corretto	prices = { 2.0 , 4.5 , 3.5 } --- taken = { false, false , false } --- numberOfSeats = 3	10.0	10.0	PASSED
T21	Verificare il comportamento del metodo quando il numero di posti richiesti è superiore di 1 rispetto alla dimensione dell'array *taken*	1. Il metodo riceve il parametro *numberOfSeats* con il valore maggiore di 1 rispetto alla dimensione di *taken*. 2. Il metodo fa la somma dei prezzi dei posti disponibili 3. Verificare se il metodo restituisce il valore corretto	prices = { 2.0 , 4.5 , 3.5 } --- taken = { false, false , false } --- numberOfSeats = 4	10.0	10.0	PASSED
T22	Verificare il comportamento del metodo quando il numero di posti richiesti è pari al numero di posti disponibili	1. Il metodo riceve il parametro *numberOfSeats* con valore uguale al numeri di posti disponibili 2. Il metodo fa la somma dei prezzi dei posti disponibili 3. Verificare se il metodo restituisce il valore corretto	prices = { 2.0 , 3.0 , 3.0 , 2.0 , 4.5 , 3.5 } --- taken = { false, false , true , false , true , true } --- numberOfSeats = 3	7.0	7.0	PASSED
T23	Verificare il comportamento del metodo quando il numero di posti richiesti è superiore di 1 rispetto al numero di posti disponibili	1. Il metodo riceve il parametro *numberOfSeats* con valore superiore di 1 rispetto al numero degli elementi di *taken* uguali a **false** 2. Il metodo fa la somma dei prezzi dei posti disponibili 3. Verificare se il metodo restituisce il valore corretto	prices = { 2.0 , 4.5 , 3.5 } --- taken = { false, true , false } --- numberOfSeats = 3	5.5	5.5	PASSED

T24	Verificare il comportamento del metodo quando la somma dei prezzi è pari a 100	1. Il metodo riceve i due array *prices* e *taken* e la variabile *numberOfSeats* 2. Il metodo fa la somma dei prezzi dei posti disponibili 3. Verificare se il metodo restituisce il valore corretto	prices = { 50.5 , 30 , 49.5 , 69 } --- taken = { false , true , false , false } --- numberOfSeats = 2	100	100	PASSED
T25	Verificare il comportamento del metodo quando la somma dei prezzi è pari a 100 + Ɛ (Ɛ = 0.01)	1. Il metodo riceve i due array *prices* e *taken* e la variabile *numberOfSeats* 2. Il metodo fa la somma dei prezzi dei posti disponibili 3. La somma è maggiore di 100, quindi viene applicato uno sconto di 5 4. Verificare se il metodo restituisce il valore corretto	prices = { 50.5 , 30 , 49.51 , 69 } --- taken = { false , true , false , false } --- numberOfSeats = 2	100.01 - 5 (***95.01***)	95.01	PASSED

Tutti i test passano con successo ad eccezione dei test T5 e T14, che falliscono: il metodo non restituisce l'output atteso, ma termina restituendo una somma negativa o pari a 0.

Di conseguenza, si rende necessario informare il team di sviluppo del problema, attraverso un report che lo descriva e ne presenti i passi per riprodurlo.
A questo punto, il team dovrebbe risolvere il problema e il test dovrebbe di conseguenza passare.

Per risolvere il problema si dovrebbero controllare tutti gli elementi del vettore *prices*. Un possibile codice di controllo potrebbe essere il seguente:

```
for (i = 0 ; i < prices.length ; i++) {
        if (prices[i] <= 0) {
                throw new IllegalArgumentException("message");
        }
}
```

Si deve inoltre documentare la pre-condizione al metodo, specificando che l'array di input *prices* deve contenere valori positivi maggiori di 0.

Qualora il metodo sia invece stato progettato per avere questo comportamento, il team dovrebbe spiegare chiaramente il motivo e fornire una documentazione adeguata. A quel punto, sarebbe nostro compito quello di modificare il metodo di test affinché si adatti al comportamento atteso.

7. ARRICCHIRE LA SUITE DI TEST

Un caso aggiuntivo a cui potremmo pensare è relativo all'inserimento di input double con molte cifre decimali. Come si comporterà il metodo? Questi numeri saranno approssimati alle prime due cifre decimali, per rappresentare il denaro? Oppure non ci sarà alcuna approssimazione?

Possiamo scrivere il caso di test come:

T26: uno o più elementi di *prices* hanno più di due cifre decimali

Considerato che non è espressamente specificato nulla nella documentazione, possiamo logicamente pensare che il metodo potrebbe approssimare i numeri alle prime due cifre decimali

```
no usages  new *
@Test
void MoreDecimalDigits(){
    assertEquals( expected: 60.70, FindBestSeat.getBestPrice(
            new double[]{50, 3.4721, 90.2345, 7.231}, new boolean[]{false, false, false, false}, numberOfSeats: 3));    // T26
}
```

Risultato:

```
expected: <60.7> but was: <60.7031>
Expected :60.7
Actual   :60.7031
<Click to see difference>
```

Method Name	getBestPrice
Pre-conditions	- Gli array di input *prices* e *taken* non devono essere *null* - Gli array di input *prices* e *taken* devono avere la stessa dimensione - *numberOfSeats* non deve essere più piccolo di 1
Post-conditions	- Ritorna il prezzo totale per i posti più economici disponibili

No.	**Test Condition**	**Test Steps**	**Test Input**	**Expected Output**	**Actual Output**	**Result**
T26	Verificare il comportamento del metodo quando uno o più elementi di *prices* hanno più di due cifre decimali	1. Il metodo riceve i due array *prices* e *taken* e la variabile *numberOfSeats* 2. Il metodo approssima gli elementi di *prices* alle prime due cifre decimali 3. Il metodo fa la somma dei prezzi dei posti disponibili 4. Verificare se il metodo restituisce il valore corretto	prices = { 50 , 3.4721 , 90.2345 , 7.231 } --- taken = { false , false, false , false } --- numberOfSeats = 3	60.70	60.7031	FAILED

Come si nota il metodo non approssima in alcun modo i numeri alle prime due cifre decimali, ci ritroviamo quindi con 4 cifre decimali.

Se tale comportamento fosse previsto, dovrebbe essere opportunamente documentato.
In caso contrario sarebbe necessario correggere il problema, tramite una pre-condizione, più o meno restrittiva.
Una possibile soluzione, che prevederebbe una pre-condizione più debole (accettare qualsiasi valore), sarebbe appunto quella di approssimare il risultato finale oppure tutti gli elementi del vettore *prices*.

Un possibile codice per correggere il problema approssimando gli elementi del vettore potrebbe essere il seguente:

```
for (i = 0 ; i < prices.length ; i++) {
        BigDecimal price = BigDecimal.valueOf(prices[i]);
        price = price.setScale(2, RoundingMode.HALF_UP);
        prices[i] = price.doubleValue();
}
```

Non abbiamo individuato ulteriori casi di test.

HOMEWORK 2

TASK 1

Structural testing

WHITE-BOX

```java
public static double getBestPrice(double[] prices, boolean[] taken, int numberOfSeats) {
    if (prices == null || taken == null) {
        throw new IllegalArgumentException("prices and taken arrays cannot be null");
    }
    if (prices.length != taken.length) {
        throw new IllegalArgumentException("prices and taken arrays do not have the same length");
    }
    if (numberOfSeats <= 0) {
        throw new IllegalArgumentException("The number of seats has to be at least 1");
    }

    int[] seats = IntStream.range(0, prices.length) IntStream
            .boxed() Stream<Integer>
            .sorted(Comparator.comparingDouble(s -> prices[s]))
            .mapToInt(Integer::intValue) IntStream
            .toArray();

    int numberOfTickets = 0;
    double totalPrice = 0;
    for (int i = 0; i < seats.length; i++) {
        int seat = seats[i];
        if (!taken[seat]) {
            totalPrice += prices[seat];
            numberOfTickets++;
        }
        if (numberOfTickets == numberOfSeats) {
            break;
        }
    }

    if (totalPrice > 100.00) {
        return totalPrice - 5.00;
    }
    return totalPrice;
}
```

- COMPRENDERE IL CODICE

All’inizio il metodo verifica 3 condizioni per validare gli input.
Verifica innanzitutto se il vettore *prices* e/o il vettore *taken* sono ***null***.
Poi verifica se la lunghezza dei due vettori non coincide.
Infine verifica se il numero di posti richiesti (*numberOfSeats*) è inferiore a 1.
In tutti i casi viene lanciata un’eccezione se la condizione risulta vera.

Dopo aver validato gli input viene istanziato un nuovo vettore di interi denominato seats, che include una mappa dei posti ordinati in base al prezzo.

Prosegue istanziando e inizializzando un int *numberOfTickets* e un double *totalPrice* a 0.
In un ciclo for, per ogni elemento del vettore *seats*, viene verificato se il posto è libero e in caso positivo viene aggiunto il prezzo al totale e incrementato *numberOfTickets*, che in caso raggiunga il numero di posti richiesti, interrompe il ciclo for.

Infine verifica se il prezzo totale supera 100, e quindi ritorna il prezzo totale a cui viene sottratto 5. In caso negativo, invece, ritorna semplicemente il prezzo totale.

- CODE COVERAGE

```
    public static double getBestPrice(double[] prices, boolean[] taken, int numberOfSeats) {
        if (prices == null || taken == null) {
            throw new IllegalArgumentException("prices and taken arrays cannot be null");
        }
        if (prices.length != taken.length) {
            throw new IllegalArgumentException("prices and taken arrays do not have the same length");
        }
        if (numberOfSeats <= 0) {
            throw new IllegalArgumentException("The number of seats has to be at least 1");
        }

        int[] seats = IntStream.range(0, prices.length)
                .boxed()
                .sorted(Comparator.comparingDouble(s -> prices[s]))
                .mapToInt(Integer::intValue)
                .toArray();

        int numberOfTickets = 0;
        double totalPrice = 0;
        for (int i = 0; i < seats.length; i++) {
            int seat = seats[i];
            if (!taken[seat]) {
                totalPrice += prices[seat];
                numberOfTickets++;
            }
            if (numberOfTickets == numberOfSeats) {
                break;
            }
        }

        if (totalPrice > 100.00) {
            return totalPrice - 5.00;
        }
        return totalPrice;
    }
```

Line Coverage — 100%
Branch + Condition Coverage — 100%

La suite di test identificata nella fase di testing black-box basato sulle specifiche ci ha portato ad avere una code coverage totale per Line e Branch+Condition.

- INDIVIDUARE CASI DI TEST MANCANTI

Proseguiamo comunque con il testing white-box per identificare eventuali casi di test mancati.

```
if (prices == null || taken == null) {
    throw new IllegalArgumentException("prices and taken arrays cannot be null");
}
```

I test T1, T2 e T3 coprono già i casi in cui una o entrambe le condizioni sono vere. Tutti gli altri test coprono, anche se implicitamente, il caso in cui le condizioni sono entrambe false. Risultano quindi coperte tutte le possibili combinazioni:

{false, false}, *{false, true}*, *{true, false}*, *{true, true}*

Non sono identificabili altri casi.

```
if (prices.length != taken.length) {
    throw new IllegalArgumentException("prices and taken arrays do not have the same length");
}
if (numberOfSeats <= 0) {
    throw new IllegalArgumentException("The number of seats has to be at least 1");
}
```

Anche in questi due casi non sono identificabili altri test.
Il test T7 copre già il caso in cui la condizione del primo if è vera.
Il test T4 copre già il caso in cui la condizione del secondo è vera.
Gli altri test coprono implicitamente il caso in cui le due condizioni sono false.

```
int[] seats = IntStream.range(0, prices.length) IntStream
        .boxed() Stream<Integer>
        .sorted(Comparator.comparingDouble(s -> prices[s]))
        .mapToInt(Integer::intValue) IntStream
        .toArray();
```

```
int numberOfTickets = 0;
double totalPrice = 0;
for (int i = 0; i < seats.length; i++) {
    int seat = seats[i];
    if (!taken[seat]) {
        totalPrice += prices[seat];
        numberOfTickets++;
    }
    if (numberOfTickets == numberOfSeats) {
        break;
    }
}
```

Per soddisfare il criterio di **Loops boundary adequacy** è necessario testare il caso in cui il loop viene iterato ***zero*** volte, uno in cui viene iterato ***una*** volta e un altro in cui viene iterato ***molteplici*** volte.

1. Zero volte: questo caso è già coperto dal test T13
2. Una volta: già coperto dai test T12 e T16
3. Molteplici volte: coperto implicitamente da altri test

Per quanto riguarda i due if interni, questi sono coperti implicitamente da molteplici casi.
In particolare il caso in cui la condizione del secondo if è falsa è coperto esplicitamente dai test T6, T9, T21 e T23, per cui la condizione resta falsa per tutte le iterazioni.

```
if (totalPrice > 100.00) {
    return totalPrice - 5.00;
}
return totalPrice;
```

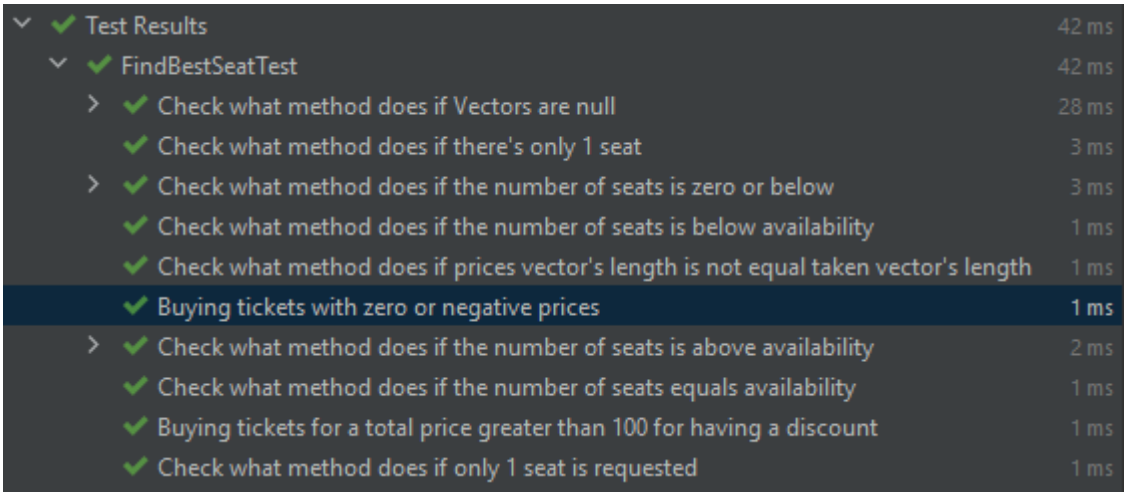
Quest'ultimo pezzo di codice è già coperto da diversi test.
I test T11 e T25 coprono il caso in cui la condizione è vera.
Il test T24 invece, copre il caso in cui la condizione non è soddisfatta, che è coperto implicitamente anche da altri test.

Casi di Test falliti

Ricordiamo che i test T5 e T14 falliscono e notiamo che nel codice non c'è alcun controllo sugli elementi del vettore prices.
Come scritto in precedenza, è possibile risolvere il problema aggiungendo un codice di controllo, come quello di esempio da noi individuato.

```java
public static double getBestPrice(double[] prices, boolean[] taken, int numberOfSeats) {
    if (prices == null || taken == null) {
        throw new IllegalArgumentException("prices and taken arrays cannot be null");
    }
    if (prices.length != taken.length) {
        throw new IllegalArgumentException("prices and taken arrays do not have the same length");
    }
    if (numberOfSeats <= 0) {
        throw new IllegalArgumentException("The number of seats has to be at least 1");
    }
    for (int i = 0; i < prices.length; i++) {
        if (prices[i] <= 0) {
            throw new IllegalArgumentException("Price cannot be 0 or less");
        }
    }
```

Questa potrebbe quindi essere una possibile implementazione, che come si può osservare fa superare i test.



Lo stesso si può dire per il test T26 che fallisce per mancanza di approssimazione.
Il problema anche in questo caso può essere risolto, ad esempio, approssimando gli elementi del vettore, come nell'esempio individuato.

```java
public static double getBestPrice(double[] prices, boolean[] taken, int numberOfSeats) {
    if (prices == null || taken == null) {
        throw new IllegalArgumentException("prices and taken arrays cannot be null");
    }
    if (prices.length != taken.length) {
        throw new IllegalArgumentException("prices and taken arrays do not have the same length");
    }
    if (numberOfSeats <= 0) {
        throw new IllegalArgumentException("The number of seats has to be at least 1");
    }
    for (int i = 0; i < prices.length; i++) {
        if (prices[i] <= 0) {
            throw new IllegalArgumentException("Price cannot be 0 or less");
        }

        BigDecimal price = BigDecimal.valueOf(prices[i]);
        price = price.setScale( newScale: 2, RoundingMode.HALF_UP);
        prices[i] = price.doubleValue();
    }
```

Con questa possibile implementazione, come nel caso precedente, il test viene superato.

Con queste due implementazioni la copertura del codice rimane del 100% sia per ***line*** che per ***branch+condition***.

```java
    public static double getBestPrice(double[] prices, boolean[] taken, int numberOfSeats) {
        if (prices == null || taken == null) {
            throw new IllegalArgumentException("prices and taken arrays cannot be null");
        }
        if (prices.length != taken.length) {
            throw new IllegalArgumentException("prices and taken arrays do not have the same length");
        }
        if (numberOfSeats <= 0) {
            throw new IllegalArgumentException("The number of seats has to be at least 1");
        }
        for (int i = 0; i < prices.length; i++) {
            if (prices[i] <= 0) {
                throw new IllegalArgumentException("Price cannot be 0 or less");
            }

            BigDecimal price = BigDecimal.valueOf(prices[i]);
            price = price.setScale(2, RoundingMode.HALF_UP);
            prices[i] = price.doubleValue();
        }

        int[] seats = IntStream.range(0, prices.length)
                .boxed()
                .sorted(Comparator.comparingDouble(s -> prices[s]))
```

Mutation Testing

Per avere una maggiore sicurezza sull'effettiva correttezza e completezza della nostra suite di test, abbiamo usato la tecnica di **Mutation testing**, in modo tale da vedere come la suite si comporta con l'introduzione di bug (mutanti). Per fare ciò abbiamo utilizzato un **tool automatizzato** di nome ***PiTest***.

```
11      /**
12       * Finds the cheapest seats and returns the total price for those seats.
13       *
14       * If there are fewer seats available than requested,
15       * only the prices for the ones that are available are counted.
16       *
17       * When the total price of the seats is larger than 100,
18       * a discount of 5 euros is applied.
19       *
20       * The prices and taken arrays cannot be null and need to have the same size.
21       * The number of seats requested should be at least 1.
22       *
23       * @param prices an array indicating the price of each seat
24       * @param taken an array indicating whether a seat has been taken already
25       * @param numberOfSeats the number of seats requested
26       * @throws IllegalArgumentException when any of the arguments do not
27       *                                  adhere to their requirements
28       * @return total price for the seats
29       */
30      public static double getBestPrice(double[] prices, boolean[] taken, int numberOfSeats) {
31 2        if (prices == null || taken == null) {
32              throw new IllegalArgumentException("prices and taken arrays cannot be null");
33          }
34 1        if (prices.length != taken.length) {
35              throw new IllegalArgumentException("prices and taken arrays do not have the same length");
36          }
37 2        if (numberOfSeats <= 0) {
38              throw new IllegalArgumentException("The number of seats has to be at least 1");
39          }
40 2        for (int i = 0; i < prices.length; i++) {
41 2            if (prices[i] <= 0) {
42                  throw new IllegalArgumentException("Price cannot be 0 or less");
43              }
44
45              BigDecimal price = BigDecimal.valueOf(prices[i]);
46              price = price.setScale(2, RoundingMode.HALF_UP);
47              prices[i] = price.doubleValue();
48          }
49
50          int[] seats = IntStream.range(0, prices.length)
51                  .boxed()
52 1                .sorted(Comparator.comparingDouble(s -> prices[s]))
53                  .mapToInt(Integer::intValue)
54                  .toArray();
55
56          int numberOfTickets = 0;
57          double totalPrice = 0;
58 2        for (int i = 0; i < seats.length; i++) {
59              int seat = seats[i];
60 1            if (!taken[seat]) {
61 1                totalPrice += prices[seat];
62 1                numberOfTickets++;
63              }
64 1            if (numberOfTickets == numberOfSeats) {
65                  break;
66              }
67          }
68
69 2        if (totalPrice > 100.00) {
70 2            return totalPrice - 5.00;
71          }
72 1        return totalPrice;
73      }
74  }
```

Mutations

31	1. negated conditional → KILLED 2. negated conditional → KILLED
34	1. negated conditional → KILLED
37	1. changed conditional boundary → KILLED 2. negated conditional → KILLED
40	1. changed conditional boundary → KILLED 2. negated conditional → KILLED
41	1. changed conditional boundary → KILLED 2. negated conditional → KILLED
52	1. replaced double return with 0.0d for org/example/FindBestSeat::lambda$getBestPrice$0 → KILLED
58	1. changed conditional boundary → KILLED 2. negated conditional → KILLED
60	1. negated conditional → KILLED
61	1. Replaced double addition with subtraction → KILLED
62	1. Changed increment from 1 to -1 → KILLED
64	1. negated conditional → KILLED
69	1. changed conditional boundary → KILLED 2. negated conditional → KILLED
70	1. Replaced double subtraction with addition → KILLED 2. replaced double return with 0.0d for org/example/FindBestSeat::getBestPrice → KILLED
72	1. replaced double return with 0.0d for org/example/FindBestSeat::getBestPrice → KILLED

Active mutators

- CONDITIONALS_BOUNDARY
- EMPTY_RETURNS
- FALSE_RETURNS
- INCREMENTS
- INVERT_NEGS
- MATH
- NEGATE_CONDITIONALS
- NULL_RETURNS
- PRIMITIVE_RETURNS
- TRUE_RETURNS
- VOID_METHOD_CALLS

Un’insieme di tipologie di mutanti che sono stati introdotti nel codice, come ad esempio: negazione delle condizioni, ritorno dei valori nulli, rimozione di chiamate alle funzioni, ecc.

Tests examined

- org.example.FindBestSeatTest.[engine:junit-jupiter]/[class:org.example.FindBestSeatTest]/[method:NumberOfSeatsAboveAvailable()] (2 ms)
- org.example.FindBestSeatTest.[engine:junit-jupiter]/[class:org.example.FindBestSeatTest]/[method:OneRequestedSeatBoundaryCase()] (1 ms)
- org.example.FindBestSeatTest.[engine:junit-jupiter]/[class:org.example.FindBestSeatTest]/[method:NumberOfSeatsBelowAvailable()] (0 ms)
- org.example.FindBestSeatTest.[engine:junit-jupiter]/[class:org.example.FindBestSeatTest]/[method:MoreDecimalDigits()] (2 ms)
- org.example.FindBestSeatTest.[engine:junit-jupiter]/[class:org.example.FindBestSeatTest]/[method:TotalPriceGreaterThanOneHundred()] (2 ms)
- org.example.FindBestSeatTest.[engine:junit-jupiter]/[class:org.example.FindBestSeatTest]/[test-template:PriceVectorNotEqualTakenVector(%5BD, %5BZ)]/[test-template-invocation:#3] (0 ms)
- org.example.FindBestSeatTest.[engine:junit-jupiter]/[class:org.example.FindBestSeatTest]/[test-template:PriceVectorNotEqualTakenVector(%5BD, %5BZ)]/[test-template-invocation:#2] (0 ms)
- org.example.FindBestSeatTest.[engine:junit-jupiter]/[class:org.example.FindBestSeatTest]/[method:ZeroOrNegativePrices()] (1 ms)
- org.example.FindBestSeatTest.[engine:junit-jupiter]/[class:org.example.FindBestSeatTest]/[test-template:NumberOfSeatsZeroOrBelow(int)]/[test-template-invocation:#2] (0 ms)
- org.example.FindBestSeatTest.[engine:junit-jupiter]/[class:org.example.FindBestSeatTest]/[test-template:VectorsNull(%5BD, %5BZ)]/[test-template-invocation:#1] (2 ms)
- org.example.FindBestSeatTest.[engine:junit-jupiter]/[class:org.example.FindBestSeatTest]/[test-template:VectorsNull(%5BD, %5BZ)]/[test-template-invocation:#3] (1 ms)
- org.example.FindBestSeatTest.[engine:junit-jupiter]/[class:org.example.FindBestSeatTest]/[test-template:PriceVectorNotEqualTakenVector(%5BD, %5BZ)]/[test-template-invocation:#1] (1 ms)
- org.example.FindBestSeatTest.[engine:junit-jupiter]/[class:org.example.FindBestSeatTest]/[test-template:NumberOfSeatsZeroOrBelow(int)]/[test-template-invocation:#1] (1 ms)
- org.example.FindBestSeatTest.[engine:junit-jupiter]/[class:org.example.FindBestSeatTest]/[method:NumberOfSeatsEqualsAvailable()] (2 ms)
- org.example.FindBestSeatTest.[engine:junit-jupiter]/[class:org.example.FindBestSeatTest]/[test-template:VectorsNull(%5BD, %5BZ)]/[test-template-invocation:#2] (1 ms)

Il tool introduce una serie di mutazioni all’interno del codice, per vedere se la suite di test si rompe (caso positivo), nel caso contrario il mutante sopravvive e ciò significa che dobbiamo riscrivere i test.

Come possiamo vedere dagli screenshot, i test reagiscono bene ai bug introdotti, ovvero: il tool ha generato 21 mutanti, che sono stati tutti “uccisi” dalla suite di test, ovvero che la suite non ha accettato i cambiamenti introdotti.

TASK 2

```java
public static double calculateAverage(double[] arr, int startIndex, int endIndex) {
    if (arr == null) {
        throw new IllegalArgumentException("L'array di input non può essere null.");
    }
    if (startIndex < 0 || endIndex >= arr.length) {
        throw new IllegalArgumentException("Gli indici di inizio e fine " +
                "devono essere compresi tra 0 e " + (arr.length - 1));
    }
    if (startIndex > endIndex) {
        throw new IllegalArgumentException("L'indice di inizio " +
                "deve essere minore o uguale all'indice di fine.");
    }

    double sum = 0;
    for (int i = startIndex; i <= endIndex; i++) {
        sum += arr[i];
    }

    return sum / (endIndex - startIndex + 1);
}
```

CODE COVERAGE TOTALE CON IL MINOR NUMERO DI TEST

Per coprire tutti e 2 i branch del primo if è sufficiente eseguire **1** solo test, coprendo il caso in cui la condizione è vera. Gli altri test copriranno implicitamente il secondo branch.

T1: vettore *arr* null

Per coprire il secondo if, seguendo il criterio Condition + Branch, sono necessari 3 test, che verifichino rispettivamente:

1. Prima condizione vera, seconda falsa
2. Prima condizione falsa, seconda vera
3. Condizioni entrambe false

Il terzo caso sarà implicitamente coperto dai successivi test, pertanto finiamo con l'avere esplicitati solo **2** test.

T2: *startindex* < 0

T3: *endIndex* supera (*size(arr)* - 1)

Anche per coprire il terzo if è sufficiente **1** solo test, che copra il caso in cui la condizione è vera, lasciando che i successivi test coprano implicitamente il caso in cui la condizione è invece falsa.

T4: *startIndex* è più grande di *endIndex*

Per coprire il ciclo for basta **1** solo test, che copre il caso in cui viene iterato una sola volta.

T5: *size(arr)* >= 1 e *startIndex* è pari a *endIndex*

In definitiva, ci troviamo una suite composta da soli **5 test** che ci garantisce una copertura del codice totale sia per ***line*** che per ***branch+condition***.

```
AntonPera
@ParameterizedTest
@MethodSource("illegalTestParameters")
void IllegalArgumentTest(double[] arr, int start, int end) {
    assertThrows(IllegalArgumentException.class, () ->
            ArraySubsetAverage.calculateAverage(arr, start, end));
}

1 usage  AntonPera *
static Stream<Arguments> illegalTestParameters() {
    return Stream.of(
            arguments(null, 0, 10), // array null                                                    T1
            arguments(new double[]{5.0, 3.0, 5.5, 4.2}, -10, 3), // startIndex lower than 0         T2
            arguments(new double[]{5.0, 3.0, 5.5, 4.2}, 0, 4), // endIndex greater than (arr length - 1)  T3
            arguments(new double[]{5.0, 3.0, 5.5, 4.2}, 8, 3) // startIndex greater than endIndex      T4
    );
}

AntonPera *
@Test
void LoopTest() {
    assertEquals( expected: 5.5, ArraySubsetAverage.calculateAverage(
            new double[]{5.0, 3.0, 5.5, 4.2}, startIndex: 2, endIndex: 2)); // one iteration      T5
}
```

```
    public static double calculateAverage(double[] arr, int startIndex, int endIndex) {
        if (arr == null) {
            throw new IllegalArgumentException("L'array di input non può essere null.");
        }
        if (startIndex < 0 || endIndex >= arr.length) {
            throw new IllegalArgumentException("Gli indici di inizio e fine " +
                    "devono essere compresi tra 0 e " + (arr.length - 1));
        }
        if (startIndex > endIndex) {
            throw new IllegalArgumentException("L'indice di inizio " +
                    "deve essere minore o uguale all'indice di fine.");
        }

        double sum = 0;
        for (int i = startIndex; i <= endIndex; i++) {
            sum += arr[i];
        }

        return sum / (endIndex - startIndex + 1);
    }
```

Method Name	calculateAverage
Pre-conditions	- L'array di input *arr* non deve essere null - Gli indici devono essere nel range di lunghezza dell'array - L'indice di inizio deve essere al più uguale all'indice di fine
Post-conditions	- Ritorna la media dei valori nel range degli indici.

No.	**Test Condition**	**Test Steps**	**Test Input**	**Expected Output**	**Actual Output**	**Result**
T1	Verificare il comportamento del metodo quando l'array di input *arr* è null	1. Il metodo riceve un array null 2. Verificare se la pre-condizione è rispettata	arr = null startIndex = 0 endIndex = 10	java.lang.IllegalArgumentException	java.lang.IllegalArgumentException	PASSED
T2	Verificare il comportamento del metodo quando l'indice *startIndex* è negativo (<0)	1. Il metodo riceve *startIndex* negativo 2. Verificare se la pre-condizione è rispettata	arr = { 5.0 , 3.0 , 5.5 , 4.2 } startIndex = -10 endIndex = 3	java.lang.IllegalArgumentException	java.lang.IllegalArgumentException	PASSED
T3	Verificare il comportamento del metodo quando l'indice *endIndex* supera *size(arr)*-1	1. Il metodo riceve *endIndex* pari o superiore alla dimensione di *arr* 2. Verificare se la pre-condizione è rispettata	arr = { 5.0 , 3.0 , 5.5 , 4.2 } startIndex = 0 endIndex = 4	java.lang.IllegalArgumentException	java.lang.IllegalArgumentException	PASSED
T4	Verificare il comportamento del metodo quando l'indice *startIndex* è più grande di *endIndex*	1. Il metodo riceve startIndex più grande di endIndex 2. Verificare se la pre-condizione è rispettata	arr = { 5.0 , 3.0 , 5.5 , 4.2 } startIndex = 8 endIndex = 3	java.lang.IllegalArgumentException	java.lang.IllegalArgumentException	PASSED
T5	Verificare il comportamento del metodo quando l'array di input *arr* ha almeno un elemento e i due indici sono uguali	1. Il metodo riceve gli input indicati 2. Il metodo calcola la media degli elementi nell'array 3. Verificare se il metodo restituisce il valore corretto	arr = { 5.0 , 3.0 , 5.5 , 4.2 } startIndex = 2 endIndex = 2	5.5	5.5	PASSED

È SUFFICIENTE?

Ma questi test sono sufficienti per coprire tutti i casi più importanti?

ArraySubsetAverage.java

```
1   package org.example;
2
3   public class ArraySubsetAverage {
4
5       public static double calculateAverage(double[] arr, int startIndex, int endIndex) {
6   1       if (arr == null) {
7               throw new IllegalArgumentException("L'array di input non puÃ² essere null.");
8           }
9   4       if (startIndex < 0 || endIndex >= arr.length) {
10  1           throw new IllegalArgumentException("Gli indici di inizio e fine " +
11                      "devono essere compresi tra 0 e " + (arr.length - 1));
12          }
13  2       if (startIndex > endIndex) {
14              throw new IllegalArgumentException("L'indice di inizio " +
15                      "deve essere minore o uguale all'indice di fine.");
16          }
17
18          double sum = 0;
19  2       for (int i = startIndex; i <= endIndex; i++) {
20  1           sum += arr[i];
21          }
22
23  4       return sum / (endIndex - startIndex + 1);
24      }
25  }
```

Mutations

```
6   1. negated conditional → KILLED
9   1. changed conditional boundary → SURVIVED
    2. changed conditional boundary → KILLED
    3. negated conditional → KILLED
    4. negated conditional → KILLED
10  1. Replaced integer subtraction with addition → SURVIVED
13  1. changed conditional boundary → KILLED
    2. negated conditional → KILLED
19  1. changed conditional boundary → KILLED
    2. negated conditional → KILLED
20  1. Replaced double addition with subtraction → KILLED
23  1. Replaced integer subtraction with addition → KILLED
    2. Replaced integer addition with subtraction → KILLED
    3. Replaced double division with multiplication → SURVIVED
    4. replaced double return with 0.0d for org/example/ArraySubsetAverage::calculateAverage → KILLED
```

Partendo anche solo dal **Mutation Testing** si nota come la nostra suite non reagisca propriamente all'introduzione di mutanti:

1. manca la copertura per i casi boundary nel secondo if (riga 9). Una modifica errata ai boundary delle due condizioni passerebbe con molta probabilità inosservata.

2. al calcolo della media, alla fine del metodo, la sostituzione della divisione con la moltiplicazione non scatena una reazione, in quanto l'unico test a coprire quella parte di codice prevede un risultato che, al variare dell'operazione, non varia.
   In questo ultimo caso, se per coprire il ciclo for avessimo arbitrariamente scelto il caso in cui questo viene iterato molteplici volte, il mutante non sarebbe sopravvissuto.

Si noti, tuttavia, come il secondo mutante sopravvissuto (riga 10) non sarebbe neutralizzato neanche se avessimo una suite di test più robusta, in quanto l'operazione eseguita serve per la visualizzazione corretta del messaggio di errore. Questo significa che **il mutante non altera il comportamento funzionale del programma**, ma solo un dettaglio di presentazione.

In conclusione, sarebbe quindi importante eseguire anche test sui casi boundary del secondo if e su molteplici iterazioni del ciclo for.
Più nel dettaglio, l'aggiunta dei seguenti 3 test renderebbe la nostra suite più completa:

T6 : *startIndex* = 0
T7 : *endIndex* = *size(arr)* - 1
T8 : iterazioni multiple

Specification-based testing

Come dimostrato quindi anche solo dal Mutation Testing, **il 100% di code coverage non garantisce la copertura di tutti i casi di test**; di conseguenza dobbiamo seguire i passi del testing specification-based per trovare ulteriori casi di test.

1. COMPRENDERE I REQUISITI

Capiamo innanzitutto cosa fa il programma, e come gli input vengono convertiti in output.
Il metodo *calculateAverage* calcola la media dei valori in un sottoinsieme di un array di tipo double. Il metodo riceve tre parametri di input:

- *arr* – vettore di tipo double, per cui calcolare la media di un sottoinsieme.
- *startIndex* e *endIndex* – I due indici (inizio e fine) di tipo int che specificano il range del sottoinsieme dell'array.

Il metodo restituisce la media del sottoinsieme dell'array.

2. ESPLORARE COSA FA IL PROGRAMMA PER DIVERSI INPUT

Per avere una maggiore consapevolezza dell funzionamento del programma, abbiamo deciso di testare cosa fa dati questi input:

a. Caso semplice - array composto da 5 elementi, con indice di inizio pari a 1 e indice di fine pari a 3.

```
@Test
void SimpleCase() {
    assertEquals( expected: 2, ArraySubsetAverage.calculateAverage(new double[]{3,5,1,0,9}, startIndex: 1, endIndex: 3));
}
```

b. Caso con i numeri negativi

```
@Test
void NegativeNumberCase() {
    double delta = 0.01;
    assertEquals( expected: -1.33, ArraySubsetAverage.calculateAverage(new double[]{3,-5,1,0,-9}, startIndex: 1, endIndex: 3), delta);
}
```

In questo caso in quanto stiamo lavorando con i numeri in virgola mobile, come valore d'uscita può capitare un numero periodico, di conseguenza per far sì che il test passi dobbiamo specificare il grado di tolleranza per il test.

c. Array vuoto — Abbiamo visto cosa il programma fa quando l'array è null, ma cosa succede nel caso in cui l'array è vuoto? Questo caso non è apparso dall'analisi white box, né è stato identificato con il mutation testing.
Ma come reagisce il metodo?

```
@Test
void EmptyArray() {
    assertThrows(IllegalArgumentException.class,
            () -> ArraySubsetAverage.calculateAverage(new double[]{}, startIndex: 0, endIndex: 0));
}
```

Il metodo reagisce correttamente, restituendo una IllegalArgumentException. m Manca tuttavia un messaggio di errore appropriato che evidenzi il reale problema.

Tutti i casi di test elencati sopra sono andati a buon fine, e il programma ha avuto il comportamento che ci eravamo aspettati.

3. INPUT, OUTPUT E PARTIZIONI

Individuiamo ora le classi di input, output e possibili partizioni

a. Classi di input

arr (array of doubles):

- Null
- Empty
- Non-empty (>= 1)

number (array element):

- Positive (>= 0)
- Negative (< 0)

startIndex:

- Positive (>= 0)
- Negative (< 0)

endIndex:

- Positive (>= 0)
- Negative (< 0)

b. Combinazioni di input

(*startIndex*, *endIndex*):

- startIndex <= endIndex
- startIndex > endIndex

(*arr*, *startIndex*):

- *startIndex* <= size(*arr*)-1
- startIndex > size(*arr*)-1

(*arr*, *endIndex*):

- endIndex <= size(*arr*)-1
- endIndex > size(*arr*)-1

c. Classi di output attesi

average (output):

- Positive (>= 0)
- Negative (< 0)

4. CASI LIMITE

Boundaries (on point):

- size(*arr*) = 1
- *startIndex* = 0
- *startIndex* = size(arr)-1
- *endIndex = 0*
- *endIndex = size(arr)-1*
- *startIndex = endIndex*

Boundaries (off point):

- size(*arr*) = 0
- startIndex = -1
- *startIndex* = size(arr)
- *endIndex = -1*
- *endIndex = size(arr)*
- *startIndex = endIndex + 1*

5. DEFINIRE CASI DI TEST

Quali casi di test riusciamo a individuare partendo dall'analisi black box?

a) Test dei casi eccezionali:

T1: vettore *arr* null
T2: *startindex* < 0
T3: *endIndex* < 0
T4: *startIndex* < 0 and *endIndex* < 0

b) Prima combinazione di Input:

size(arr) > 1
endIndex != (size(arr) - 1)

T5: *endIndex* supera (*size(arr)* - 1)
T6: *endIndex* è inferiore a (*size(arr)* - 1)

c) Seconda combinazione di Input:

size(arr) > 1
startIndex != endIndex

T7: *startIndex* supera *endIndex*
T8: *startIndex* è inferiore a *endIndex* (iterazioni multiple)

d) Boundaries:

condizione → (*size(arr) >= 1*)
T9: *size(arr)* = 1 e *startIndex* è pari a *endIndex* che è pari a 0
T10: *size(arr)* = 0

condizione → (*startIndex >= 0*)
T11: *size(arr)* > 1 e *startIndex* è pari a 0
T12: *size(arr)* > 1 e *startIndex* è pari a -1

condizione → (*startIndex <= size(arr)-1*)
I test boundary su questa condizione non sono necessari, in quanto già coperti:

- dal test case T7, qualora endIndex dovesse essere più piccolo
- dai test boundary sulla condizione (*endIndex <= size(arr)-1*) qualora endIndex dovesse essere maggiore o uguale

condizione → (*endIndex >= 0*)
Per lo stesso motivo di prima, i test boundary su questa condizione non sono necessari, in quanto coperti:

- dal test T7, qualora startIndex dovesse essere più grande
- dai test boundary T11 e T12, qualora startIndex dovesse essere uguale o più piccolo

condizione → (*endIndex <= size(arr)-1*)
T13: *size(arr)* > 1 e *endIndex* è pari a (*size(arr)* - 1)
T14: *size(arr)* > 1 e *endIndex* è pari a *size(arr)*

condizione → (*startIndex <= endIndex*)
T15: *size(arr)* > 1 e *startIndex* è pari a *endIndex*
T16: *size(arr)* > 1 e *startIndex* è pari a *endIndex* + 1

6. AUTOMATIZZARE CASI DI TEST

I casi di test T1, T2, T5, T7 e T15 sono quelli identificati inizialmente, e che ci hanno garantito una copertura totale del codice.

Invece i casi di test T8, T11 e T13 sono stati identificati in seguito al *Mutation Testing,* ma non sono stati implementati in JUnit.

Di conseguenza, abbiamo trovato 8 ulteriori casi di test che rendono la nostra suite di test sicuramente più completa:

T3: *endIndex* < 0
T4: *startIndex* < 0 and *endIndex* < 0
T6: *endIndex* è inferiore a (*size(arr)* - 1)
T9: *size(arr)* = 1 e *startIndex* è pari a *endIndex* che è pari a 0
T10: *size(arr)* = 0
T12: *size(arr)* > 1 e *startIndex* è pari a -1
T14: *size(arr)* > 1 e *endIndex* è pari a *size(arr)*
T16: *size(arr)* > 1 e *startIndex* è pari a *endIndex* + 1

HOMEWORK 3

Property based testing

```java
public static double calculateAverage(double[] arr, int startIndex, int endIndex) {
    if (arr == null) {
        throw new IllegalArgumentException("L'array di input non può essere null.");
    }
    if (startIndex < 0 || endIndex >= arr.length) {
        throw new IllegalArgumentException("Gli indici di inizio e fine " +
                "devono essere compresi tra 0 e " + (arr.length - 1));
    }
    if (startIndex > endIndex) {
        throw new IllegalArgumentException("L'indice di inizio " +
                "deve essere minore o uguale all'indice di fine.");
    }

    double sum = 0;
    for (int i = startIndex; i <= endIndex; i++) {
        sum += arr[i];
    }

    return sum / (endIndex - startIndex + 1);
}
```

Un altro modo per arricchire la nostra suite di test per questo metodo è quello di integrare dei test property-based, andando quindi a definire le proprietà, lasciando al framework la scelta degli esempi.
Il framework utilizzato per il PBT (Property Based Testing) è jqwik.

Basandoci sui requisiti, abbiamo identificato le seguenti proprietà:

1. ***shouldCalculateAverage*** – Per ogni partizione dell'array, con startIndex ed endIndex correttamente definiti, dove quindi i due indici sono nel range della dimensione dell'array e startIndex è minore di endIndex, il metodo dovrebbe calcolare la media dei valori in quella partizione.

2. ***invalidIndexes*** – Per ogni partizione non valida, dove quindi i due indici sono fuori dal range della dimensione dell'array e/o startIndex è più grande di endIndex, il metodo dovrebbe lanciare un'eccezione.

Possiamo quindi passare a scrivere i PBT per le due proprietà.

Proprietà 1 : shouldCalculateAverage

Il seguente PBT seleziona una partizione casuale di un array di 100 elementi casuali. In quella partizione, il metodo *calculateAverage()* calcola la media degli elementi. Di conseguenza, il test si aspetta che il metodo ritorni la media corretta, che sia il risultato della somma diviso il numero di elementi.

Per raggiungere l'obiettivo, il PBT ha bisogno di 2 parametri:

- *numbers* – Una lista di Double casuali, di dimensione 100, i cui valori sono nel range [-1000, 1000], scelto casualmente. Questa sarà successivamente convertita in array.

- *indexes* – un array di due elementi di tipo intero, di cui il primo elemento sarà startIndex e il secondo endIndex. I due elementi sono nel range [0,99] e il secondo è più grande del primo.

```
@Property
@Report(Reporting.GENERATED)
@StatisticsReport(format = Histogram.class)
void shouldCalculateAverage(
        /* We generate a list with 100 numbers, ranging from -1000 to 1000. Range chosen randomly. */
        @ForAll @Size(value = 100) List<@DoubleRange(min = -1000, max = 1000) Double> numbers,
        /* We randomly pick a subset to search in */
        @ForAll("subsetIndexes") int[] indexes
        ) {
```

Per il parametro indexes, gli input sono forniti dal metodo *subsetIndexes()*, che li genera secondo i criteri da noi stabiliti.

```
@Provide
/unused/
Arbitrary<int[]> subsetIndexes() {
    Arbitrary<Integer> indexes = Arbitraries.integers().between(0, 99);
    return Combinators.combine(indexes, indexes)
            .filter((start, end) -> start <= end)
            .as((start, end) -> new int[] {start, end});
}
```

Il metodo crea un Arbitrary di valori interi nel range [0, 99] e combina questi valori a coppie, andando a scartare le coppie in cui il primo elemento è più grande.

Il PBT si aspetta che il metodo ritorni la media corretta, che sia il risultato della somma diviso il numero di elementi.

```
/* we convert the list to an array, as expected by the method */
double[] arr = listToArray(numbers);
/* we take the two indexes from the indexes array */
int startIndex = indexes[0];
int endIndex = indexes[1];

/* we expect the method to return average, calculated as
 * sum of the elements divided by the number of elements */
double sum = 0;
for (int i = startIndex; i <= endIndex; i++) {
    sum += arr[i];
}

double expectedAverage = sum / (endIndex - startIndex + 1);
assertEquals(expectedAverage, ArraySubsetAverage.calculateAverage(arr, startIndex, endIndex));
```

```
private double[] listToArray(List<Double> list) { return list.stream().mapToDouble(x -> x).toArray(); }
```

RACCOLTA DATI

In seguito andiamo a raccogliere i dati, per poter infine generare le statistiche.

```
    String elements = (startIndex == endIndex ? "one element" :
            ((startIndex == 0 && endIndex == arr.length-1) ? "whole array" : "more elements")) + " --";
    String average = "average: " + (expectedAverage == 0 ? "zero" : expectedAverage < 0 ? "negative" : "positive");

    Statistics.collect(elements, average);
}
```

I dati che ci interessa raccogliere sono sul numero di elementi e la media risultante.
Nello specifico, ci interessa se abbiamo uno o più elementi e se la media è negativa, positiva o pari a zero.

Raccolti tutti i dati, viene generato il report delle statistiche per il PBT in formato istogramma.

```
timestamp = 2023-08-20T21:10:42.354163900, [ArraySubsetAveragePBTest:shouldCalculateAverage] (1000) statistics =
      # |                                label | count |
   -----|--------------------------------------|-------|-----------------------------------------------------------------------------------------
      0 | more elements -- average: negative |   473 | ##############################################################################
      1 | more elements -- average: positive |   437 | ######################################################################
      2 |     more elements -- average: zero |    19 | ###
      3 |   one element -- average: negative |    35 | #####
      4 |   one element -- average: positive |    22 | ###
      5 |       one element -- average: zero |     6 | #
      6 |   whole array -- average: negative |     4 |
      7 |   whole array -- average: positive |     3 |
      8 |       whole array -- average: zero |     1 |
```

Notiamo come con altissima frequenza sono generati valori diversi per startIndex ed endIndex. Con più bassa frequenza invece ci troviamo nel caso limite in cui i due indici assumono lo stesso valore, quindi il metodo calcola la media di un solo valore, che sarà pari quindi al valore stesso. Sia che ci troviamo nel primo che nel secondo caso, la media sarà positiva o negativa con simile frequenza.
Con frequenza molto bassa invece, la media nel primo caso, o il valore nel secondo, sarà pari a 0.

Inoltre, con bassissima frequenza, ci troviamo nel caso limite in cui il sottoinsieme selezionato è l'array intero:

(startIndex = 0 ; endIndex = size(arr)-1)

Proprietà 2 : invalidIndexes

Il seguente PBT seleziona una partizione casuale di un array di 100 elementi casuali. Quella partizione non è valida. Di conseguenza, il test si aspetta che il metodo lanci un'eccezione di tipo *IllegalArgumentException*.

Per raggiungere l'obiettivo, il PBT ha bisogno di 2 parametri:

- *numbers* – Una lista di Double casuali di dimensione 100 i cui valori sono nel range [-1000, 1000], scelto casualmente. Questa sarà successivamente convertita in array.

- *indexes* – un array di due elementi di tipo intero, di cui il primo elemento sarà startIndex e il secondo endIndex. I due elementi sono nel range [-199,199]. Possono quindi essere fuori dal range valido, e inoltre il secondo potrebbe essere più piccolo del primo.

```
@Property
@Report(Reporting.GENERATED)
@StatisticsReport(format = Histogram.class)
void invalidIndexes(
        /* We generate a list with 100 numbers, ranging from -1000 to 1000. Range chosen randomly. */
        @ForAll @Size(value = 100) List<@DoubleRange(min = -1000, max = 1000) Double> numbers,
        /* We randomly pick a wrong subset */
        @ForAll("wrongIndexes") int[] indexes
) {
```

Per il parametro indexes, gli input sono forniti dal metodo *wrongIndexes()*, che li genera secondo i criteri da noi stabiliti.

```
@Provide
/unused/
Arbitrary<int[]> wrongIndexes() {
    Arbitrary<Integer> indexes = Arbitraries.integers().between(-199, 199);
    return Combinators.combine(indexes, indexes)
            .filter((start, end) -> start > end || start < 0 || end > 99)
            .as((start, end) -> new int[] {start, end});
}
```

Il metodo crea un Arbitrary di valori interi nel range [-199, 199] e combina questi valori a coppie, andando a scartare le coppie "valide". Prenderà quindi solo le coppie in cui il primo (start) è più grande del secondo (end), oppure il primo è negativo, oppure il secondo è superiore a 99.

Il PBT si aspetta che il metodo lanci una IllegalArgumentException.

```
/* we convert the list to an array, as expected by the method */
double[] arr = listToArray(numbers);
/* we take the two indexes from the indexes array */
int startIndex = indexes[0];
int endIndex = indexes[1];

/* we expect the method to throw an exception */
assertThrows(IllegalArgumentException.class, () -> ArraySubsetAverage.calculateAverage(arr, startIndex, endIndex));
```

```
private double[] listToArray(List<Double> list) { return list.stream().mapToDouble(x -> x).toArray(); }
```

RACCOLTA DATI

In seguito andiamo a raccogliere i dati, per poter infine generare le statistiche.

```
    String greaterStart = (startIndex > endIndex ? "start > end" : "start <= end") + " --";
    String negativeStart = (startIndex < 0 ? "negative start" : "positive start") + " --";
    String biggerEnd = (endIndex >= arr.length ? "bigger end" : "lower end");

    Statistics.label("Reason").collect(greaterStart, negativeStart , biggerEnd);
}
```

Ci interessa raccogliere i dati sui motivi per cui il metodo lancia l'eccezione.

Raccolti tutti i dati, viene generato il report delle statistiche per il PBT in formato istogramma.

```
timestamp = 2023-06-07T21:23:46.729148100, [ArraySubsetAveragePBTest:failOnIndexes] (1000) Reason =
      # |                                        label | count |
    ----|----------------------------------------------|-------|----------------------------------------------------------------------------------------------------
      0 |  start <= end -- negative start -- lower end |   347 | ##################################################################
      1 | start <= end -- positive start -- bigger end |    60 | ###########
      2 | start <= end -- negative start -- bigger end |    56 | ##########
      3 |   start > end -- positive start -- lower end |   417 | ###############################################################################
      4 |   start > end -- negative start -- lower end |   114 | ####################
      5 |  start > end -- positive start -- bigger end |     6 | #
```

Notiamo innanzitutto come non ci troviamo mai nel caso in cui tutte e tre le condizioni portano al fallimento. Come è facile intuire, questo è vero perché, se l'indice di inizio è più grande di quello di fine ed è al tempo stesso negativo, l'indice di fine, essendo più piccolo, quindi anch'esso negativo, non potrà essere al tempo stesso superiore a 99.

Casi non coperti

Con queste due proprietà andiamo a garantire la copertura di moltissimi casi di test.
Quello che non andiamo invece a coprire sono i casi che potremmo teoricamente racchiudere in una proprietà "***invalidArray***". Questo risulta tuttavia sconveniente, in quanto gli esempi possibili sono solo due: array vuoto e array null.
Questi due casi sono quindi coperti dai test example-based.

```
@Test
void InvalidArray() {
    Assertions.assertAll(
            () -> assertThrows(IllegalArgumentException.class,
                    () -> ArraySubsetAverage.calculateAverage( arr: null,  startIndex: 0,  endIndex: 10)), // null
            () -> assertThrows(IllegalArgumentException.class,
                    () -> ArraySubsetAverage.calculateAverage(new double[]{},  startIndex: 0,  endIndex: 0)) // empty
    );
}
```