Gruppo

LinAn

Integrazione e test di sistemi software

CDL I.T.P.S.

A.A. 2023/22024

Realizzato da

Andrea Greco – a.greco159@studenti.uniba.it (mat. 763808)

Antonio Lin - a.lin@studenti.uniba.it (mat. 716778)

Sommario

Homework 1

Specification-based testing

Progettazione e implementazione di una suite di test black-box (specification-based testing).

1-comprensione dei requisiti

Il frammento di codice preso in esame è un metodo scritto in Java che riceve in input due array di tipo *int* e li unisce creando un unico array ordinato in ordine crescente. Fatto ciò, restituisce un risultato

numerico pari al valore centrale dell'array se dispari oppure la somma dei due valori centrali diviso due se l'array risultante è pari.

2-Esplorazione del programma in base ai vari input

In questa fase abbiamo voluto esplorare il codice facendo due test come mostrato di seguito:

```
public class Main {
    public static void main(String[] args) {

        Mediana mediana = new Mediana();
        int[] array1 ={1,2};
        int[] array2={3,4};

        int [] array3 = {1,2};
        int [] array4 = {3};
        System.out.print("Prima prova -> ");
        System.out.println(mediana.findMedianSortedArrays(array1,array2));

        System.out.print("Seconda prova -> ");
        System.out.println(mediana.findMedianSortedArrays(array3,array4));
    }
}
```

Producendo il seguente output:

```
> Task :Main.main()
Prima prova -> 2.5
Seconda prova -> 2.0
```

3-Esplorare gli input, output e identificazione delle partizioni

In questa fase andiamo a fare alcune considerazioni sugli input, vedremo gli input validi e non validi.

Gli input considerati validi sono array contenenti elementi numerici di tipo *int*.

Gli input non validi sono i valori *null* e tutti gli altri tipi di array diversi dal tipo *int*. Il metodo preso in esame non può ricevere parametri nulli in quanto questo tipo di valore viene considerato un errore in fase di compilazione, con conseguente errore durante la fase di esecuzione.

L'output prevede la restituzione di un valore numerico, in questo caso è la mediana calcolata a partire dall'unione dei due array passati in input. Vengono considerati i seguenti risultati:

- Se l'array risultante avrà un valore di elementi **dispari**, l'output sarà un valore numerico di tipo *double* che corrisponderà al valore centrale dell'array.
- Se l'array risultante avrà un numero di elementi **pari**, l'output sarà un valore numerico di tipo *double* pari alla somma dei due elementi centrali diviso due.

Esaminiamo gli input individuali:

- ***Int [] nums1***
  - Array vuoto
  - Array con dimensione > 0
- ***Int [] nums2***
  - Array vuoto
  - Array con dimensione > 0

Combinazione degli input

- 1: Array1 vuoto, array2 vuoto
- 2: Array1 vuoto, array2 size >0
- 3: Array1 size >0, array2 size>0
- 4: Array1 size >0, array2 vuoto

Partizioni

- entrambi gli array vuoti
- un array vuoto e uno pieno
- entrambi gli array pieni

Possibili output

Mediana = 0

Mediana > 0

Mediana < 0

4-Identificazione dei casi limite

I casi limite in questo caso non riguardano i parametri *null* dato che vengono considerati errori in compilazione. Gli altri casi limite identificati sono i seguenti:

- Un array vuoto mentre il secondo array con un solo elemento;
- Entrambi gli array hanno 1 elemento;
- Entrambi gli array contengono lo stesso valore ripetuto;
- Entrambi gli array di dimensioni tali da far risultare l'array sul quale si andrà a calcolare la mediana, di dimensione pari;
- Entrambi gli array con più di un elemento e tali da far risultare l'array sul quale si andrà a calcolare la mediana, di dimensione dispari.

5-ideazione dei casi di test

In base ai casi limite e alle partizioni, si sono identificati i seguenti casi di test

- T1: Array1 vuoto, array2 vuoto
- T2: Array1 vuoto, array2 dimensione >0

- ~~T3: array1 dimensione >0, array2 dimensione >0~~
- T4: Array1 dimensione >0, array2 vuoto
- T5: Entrambi gli array contengono lo stesso valore ripetuto
- T6: entrambi gli array contengono un solo elemento
- T7: array1 di dimensione pari, array2 di dimensione dispari
- T8: array1 di dimensione dispari, array2 di dimensione dispari
- T9: array1 di dimensione pari, array2 di dimensione pari

6-Automazione dei test

```
@Test
@DisplayName("T1 - entrambi gli array vuoti")
public void testMethodWithTwoArrayEmpty(){
    int [] arrayVuoto1 = {};
    int [] arrayVuoto2 = {};
    assertEquals( expected: 0,m.findMedianSortedArrays(arrayVuoto1,arrayVuoto2));
}
```

```
@Test
@DisplayName("T2 - array1 vuoto, array2 dimensione >0")
public void testMethodArrayEmptyandArrayHaveSize(){
    int [] array1 = {};
    int [] array2 = {1,2,3};
    // T6 array1 vuoto array2 size>0
    assertEquals( expected: 2,m.findMedianSortedArrays(array1,array2));
}
```

```
@Test
@DisplayName("T4 array1 dimensione >0, array2 vuoto")
public void testMethodArrayHaveSizeArrayEmpty(){
    int [] array2 = {};
    int [] array1 = {1,2,3};
    assertEquals( expected: 2,m.findMedianSortedArrays(array1,array2));
}
```

```
@Test
@DisplayName("T5 Entrambi gli array contengono lo stesso valore ripetuto")
public void testT5(){
    int [] array1 = {1,3,3};
    int [] array2 = {2,3,3};
    // T9 array1 dispari array2 pari
    assertEquals( expected: 3,m.findMedianSortedArrays(array1,array2));
}
```

```java
@Test
@DisplayName("T6.1 entrambi gli array contengono un solo elemento")
public void testT6(){
    int [] array1 = {1};
    int [] array2 = {1};
    assertEquals( expected: 1,m.findMedianSortedArrays(array1,array2));
}

@Test
@DisplayName("T6.2 entrambi gli array contengono un solo elemento")
public void testT6_2(){
    int [] array1 = {Integer.MIN_VALUE};
    int [] array2 = {Integer.MIN_VALUE};
    double result = (array1[0] + array2[0])/2;
    assertEquals(result,m.findMedianSortedArrays(array1,array2));
}

@Test
@DisplayName("T6.3 entrambi gli array contengono un solo elemento")
public void testT6_3(){
    int [] array1 = {Integer.MAX_VALUE};
    int [] array2 = {Integer.MAX_VALUE};
    double result = (array1[0] + array2[0])/2;
    assertEquals(result,m.findMedianSortedArrays(array1,array2));
}
```

```java
@Test
@DisplayName("T7 array1 di dimensione pari, array2 di dimensione dispari")
public void testMethodArrayDispariArrayPari(){
    int [] array1pari = {1,2};
    int [] array2Dispari = {2,4,3};
    // T9 array1 dispari array2 pari
    assertEquals( expected: 2,m.findMedianSortedArrays(array1pari,array2Dispari));
}
```

```java
@Test
@DisplayName("T8 array1 di dimensione dispari, array2 di dimensione dispari")
public void testMethodArrayPariArrayDispari(){
    int [] array1 = {1,2,1};
    int [] array2 = {1,2,5};
    assertEquals( expected: 1.5,m.findMedianSortedArrays(array1,array2));
}
```

```java
@Test
@DisplayName("T9 array1 di dimensione pari, array2 di dimensione pari")
public void testT9(){
    int [] array1 = {1,2,1,6};
    int [] array2 = {1,2,5,4};
    assertEquals( expected: 2,m.findMedianSortedArrays(array1,array2));
}
```

7-Acccrescimento della suite di test con creatività ed esperienza

Sono stati pensati i seguenti ulteriori casi di test:

- T10: entrambi gli array non vuoti ed ordinati con valori in ordine decrescente
- T11: entrambi gli array di uguali dimensioni contenenti solamente valori identici
- T12: uno dei due array è un sottoinsieme dell'altro array
- T13: un array contiene valori identici mentre e l'altro array è vuoto

```
@Test
@DisplayName("T10 Array1 e array2 non vuoti e in ordine decrescente")
public void testMethodVuotiDecrescente(){
    int [] array1 = {9,7,5,3,1};
    int [] array2 = {8,6,4,2,0};
    assertEquals( expected: 4.5,m.findMedianSortedArrays(array1,array2));
}
```

```
@Test
@DisplayName("T11 stessi elementi e dimensioni")
public void testMethodEqualElementAndSize(){
    int [] array1 = {2,2,2};
    int [] array2 = {2,2,2};
    assertEquals( expected: 2,m.findMedianSortedArrays(array1,array2));
}
```

```
@Test
@DisplayName("T12.1 array1 è un sottoinsieme dell'array2 caso pari")
public void testMethodSottoinsiemeCasoPari(){
    int [] array1 = {2,3};
    int [] array2 = {1,2,3,4,5,6};
    assertEquals( expected: 3,m.findMedianSortedArrays(array1,array2));
}
```

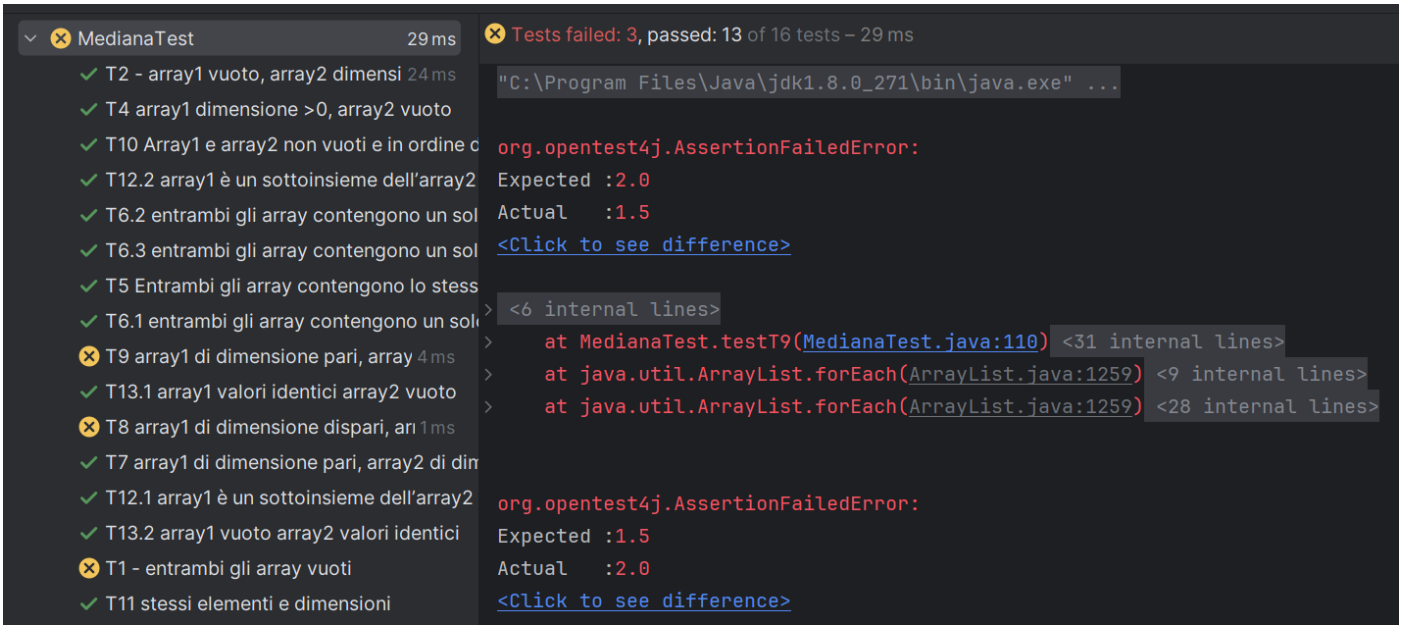
```java
@Test
@DisplayName("T12.2 array1 è un sottoinsieme dell'array2 caso dispari")
public void testMethodSottoinsiemeCasoDispari(){
    int [] array1 = {3,4};
    int [] array2 = {1,2,3,4,5};
    assertEquals( expected: 3,m.findMedianSortedArrays(array1,array2));
}
```

```java
@Test
@DisplayName("T13.1 array1 valori identici array2 vuoto")
public void testMethodIdenticiVuoto(){
    int [] array1 = {6,6,6,6,6};
    int [] array2 = {};
    assertEquals( expected: 6,m.findMedianSortedArrays(array1,array2));
}
```

```java
@Test
@DisplayName("T13.2 array1 vuoto array2 valori identici")
public void testMethodVuotoIdentici(){
    int [] array1 = {};
    int [] array2 = {6,6,6,6,6,6};
    assertEquals( expected: 6,m.findMedianSortedArrays(array1,array2));
}
```

Risultati

Dei 16 test, 3 hanno riportato un fallimento.



Analizzando caso per caso possiamo notare come il risultato atteso sia differente rispetto a quello attuale:

- Per T1

```
@Test
@DisplayName("T1 - entrambi gli array vuoti")
public void testMethodWithTwoArrayEmpty(){
    int [] arrayVuoto1 = {};
    int [] arrayVuoto2 = {};
    assertEquals( expected: 0,m.findMedianSortedArrays(arrayVuoto1,arrayVuoto2));
}
```

output

```
Expected :0.0
Actual   :-0.5
```

- Per T8 l'output è

```
@Test
@DisplayName("T8 array1 di dimensione dispari, array2 di dimensione dispari")
public void testMethodArrayPariArrayDispari(){
    int [] array1 = {1,2,1}; //produce bug
    int [] array2 = {1,2,5};
    assertEquals( expected: 1.5,m.findMedianSortedArrays(array1,array2));
}
```

output

```
Expected :1.5
Actual   :2.0
```

- Per T9

```
@Test
@DisplayName("T9 array1 di dimensione pari, array2 di dimensione pari")
public void testT9(){
    int [] array1 = {1,2,1,6}; //produce bug
    int [] array2 = {1,2,5,4};
    assertEquals( expected: 2,m.findMedianSortedArrays(array1,array2));
}
```

output

```
Expected :2.0
Actual   :1.5
```

Soluzioni

Per poter risolvere il primo fallimento causato dal test T1 si è proceduto modificando il codice presente nel metodo *findMedianSortedArrays()* della classe *Mediana* aggiungendo la seguente condizione (evidenziata in rosso):

```
public double findMedianSortedArrays(int[] nums1, int[] nums2) {  16 usages
    if (nums1.length > nums2.length) {
        return findMedianSortedArrays(nums2, nums1);
    }
    int m = nums1.length;
    int n = nums2.length;
    int totalLength = m + n;

    int left = 0;
    int right = m;//3
    if ((nums1.length !=0 || nums2.length!=0)) {
        while (left <= right) {

            int partitionNums1 = (left + right) / 2;
            int partitionNums2 = (totalLength + 1) / 2 - partitionNums1;
            int maxLeftNums1;
            if (partitionNums1 == 0) {
                maxLeftNums1 = Integer.MIN_VALUE;
            } else {
                maxLeftNums1 = nums1[partitionNums1 - 1];
            }

            int minRightNums1;
            if (partitionNums1 == m) {
                minRightNums1 = Integer.MAX_VALUE;
            } else {
                minRightNums1 = nums1[partitionNums1];
            }

            int maxLeftNums2;
            if (partitionNums2 == 0) {
                maxLeftNums2 = Integer.MIN_VALUE;
            } else {
                maxLeftNums2 = nums2[partitionNums2 - 1];
            }

            int minRightNums2;
            if (partitionNums2 == n) {
                minRightNums2 = Integer.MAX_VALUE;
            } else {
                minRightNums2 = nums2[partitionNums2];
            }

            if (maxLeftNums1 <= minRightNums2 && maxLeftNums2 <= minRightNums1) {
                if (totalLength % 2 == 0) {
                    return (double) (Math.max(maxLeftNums1, maxLeftNums2) + Math.min(minRightNums1, minRightNums2)) / 2.0;
                } else {
                    return (double) Math.max(maxLeftNums1, maxLeftNums2);
                }
            } else if (maxLeftNums1 > minRightNums2) {
                right = partitionNums1 - 1;
            } else {
                left = partitionNums1 + 1;
            }
        }
    }
    return 0;
}
```

In questo modo basta che uno dei due array contenga degli elementi per procedere con il calcolo della mediana. Inoltre, l'aggiunta di questa condizione rende inutile l'*if then else* che serviva per assegnare il valore alla variabile *maxLeftNums,* questo perché *PartitionNums2* non sarà mai uguale a zero. Dunque, il nuovo metodo per calcolare la mediana diventa:

```java
public double findMedianSortedArrays(int[] nums1, int[] nums2) {  18 usages
    if (nums1.length > nums2.length) {
        return findMedianSortedArrays(nums2, nums1);
    }
    int m = nums1.length;
    int n = nums2.length;
    int totalLength = m + n;

    int left = 0;
    int right = m;
    if ((nums1.length !=0 || nums2.length!=0)) {
        while (left <= right) {

            int partitionNums1 = (left + right) / 2;
            int partitionNums2 = (totalLength + 1) / 2 - partitionNums1;
            int maxLeftNums1;
            if (partitionNums1 == 0) {
                maxLeftNums1 = Integer.MIN_VALUE;
            } else {
                maxLeftNums1 = nums1[partitionNums1 - 1];
            }

            int minRightNums1;
            if (partitionNums1 == m) {
                minRightNums1 = Integer.MAX_VALUE;
            } else {
                minRightNums1 = nums1[partitionNums1];
            }

            int maxLeftNums2 = nums2[partitionNums2 - 1];

            int minRightNums2;
            if (partitionNums2 == n) {
                minRightNums2 = Integer.MAX_VALUE;
            } else {
                minRightNums2 = nums2[partitionNums2];
            }

            if (maxLeftNums1 <= minRightNums2 && maxLeftNums2 <= minRightNums1) {
                if (totalLength % 2 == 0) {
                    return (double) (Math.max(maxLeftNums1, maxLeftNums2) + Math.min(minRightNums1, minRightNums2)) / 2.0;
                } else {
                    return (double) Math.max(maxLeftNums1, maxLeftNums2);
                }
            } else if (maxLeftNums1 > minRightNums2) {
                right = partitionNums1 - 1;
            } else {
                left = partitionNums1 + 1;
            }
        }
    }
    return 0;
}
```

Per poter risolvere il fallimento causato dal test T8 e T9 si è analizzato il codice e si è scoperto che la causa dei fallimenti risiede nella logica del codice implementato. Al contrario di quanto detto nella fase 1, durante la *comprensione dei requisiti,* il codice implementato non unisce i due array passati in input ma funziona per partizionamenti e simula l'unione. Più specificamente, si crea una partizione sia per il primo parametro (il primo array), sia per il secondo parametro (il secondo array). Fatto

questo partizionamento, quindi diviso in due parti ciascun array, si prelevano due valori per ciascuna partizione: un valore dalla prima metà e un altro dalla seconda metà di ciascun array. I valori prelevati devono soddisfare una determinata condizione che è la seguente:

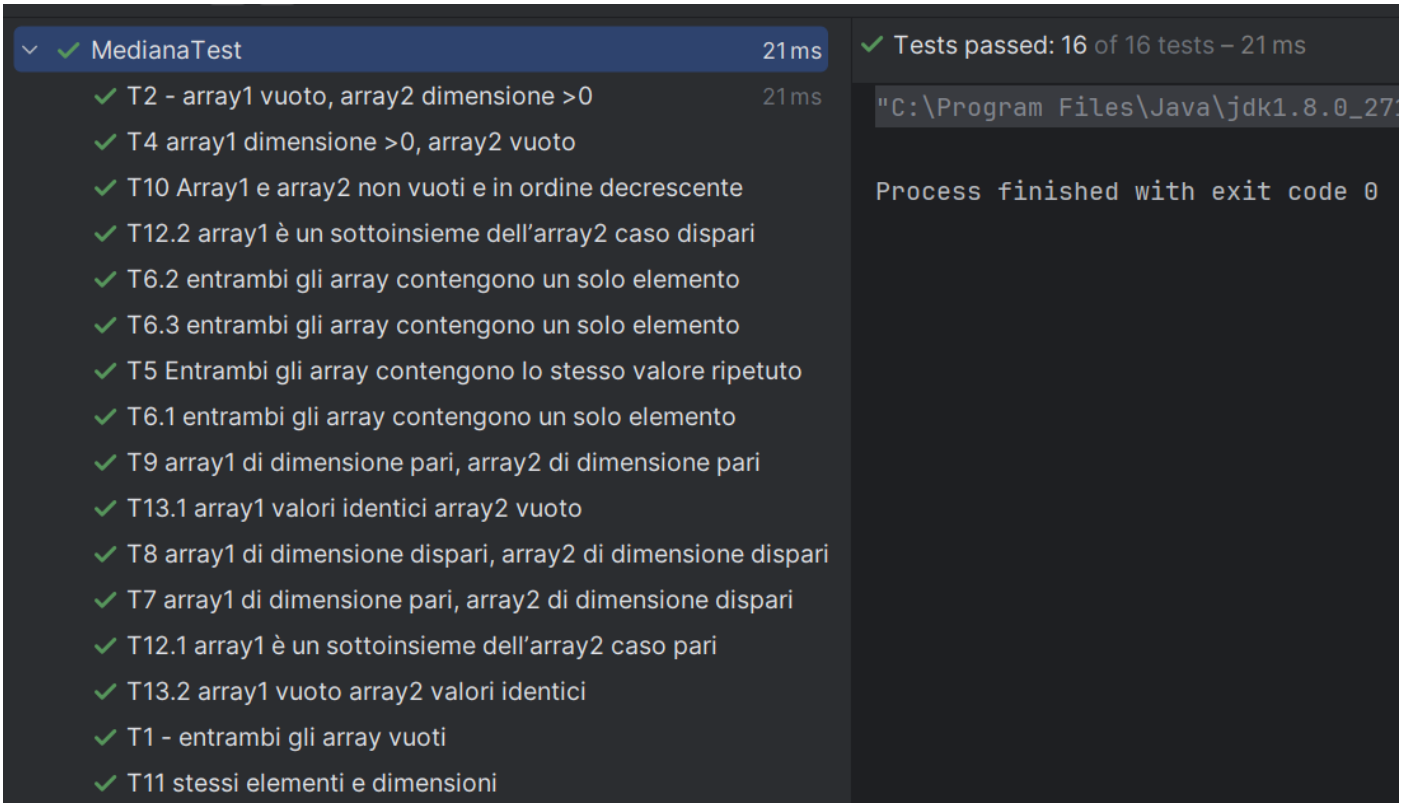
```
if (maxLeftNums1 <= minRightNums2 && maxLeftNums2 <= minRightNums1) {
```

Se questa condizione fosse vera allora le partizioni trovate sarebbero adeguate per procedere con il calcolo della mediana, altrimenti si dovrebbe spostare la partizione dei due array più a sinistra oppure più a destra. Questa particolare metodologia viene chiamata ricerca binaria e ha i suoi pro e contro. Uno dei requisiti di questa particolare metodologia è che richiede che gli array passati in input abbiano già un loro ordinamento altrimenti il risultato risulta non corretto. Lo scopo dei partizionamenti implica trovare una partizione che equilibri i due array come se fossero stati combinati e poi ordinati.

Nei casi specifici dei casi di test T8 e T9 i due array passati come parametri non godevano di un ordinamento, causando degli errori nel calcolo della mediana. Infatti, riordinando i valori, dotandoli di un ordinamento crescente, il calcolo della mediana risulta corretto. *Di seguito il risultato dei test con array ordinati:*

```
@Test
@DisplayName("T8 array1 di dimensione dispari, array2 di dimensione dispari")
public void testMethodArrayPariArrayDispari(){
    int [] array1 = {1,1,2};
    int [] array2 = {1,2,5};
    assertEquals( expected: 1.5,m.findMedianSortedArrays(array1,array2));
}
```

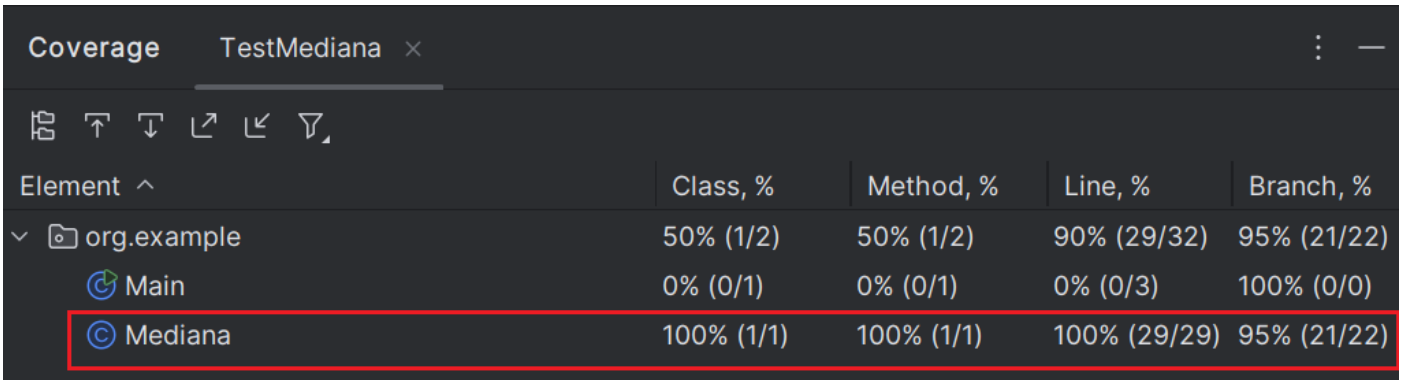
```
@Test
@DisplayName("T9 array1 di dimensione pari, array2 di dimensione pari")
public void testT9(){
    int [] array1 = {1,1,2,6};
    int [] array2 = {1,2,4,5};
    assertEquals( expected: 2,m.findMedianSortedArrays(array1,array2));
}
```



In questo modo tutti e 16 i test risultano passati con successo. In conclusione, il risultato errato riscontrato nei precedenti casi (T8 e T9) non dipende da una sbagliata implementazione del codice per il calcolo della mediana, bensì da una limitazione di questo tipo di approccio.

Test strutturali e code coverage

Dopo aver ideato i casi di test basati sulle specifiche si è voluto eseguire il codice con il tool di *coverage* integrato in **Intellij IDEA** per avere una panoramica riguardo il codice e le branche che si sono potute coprire grazie ai casi di test già ideati, producendo il seguente risultato.



Element	Class, %	Method, %	Line, %	Branch, %
org.example	50% (1/2)	50% (1/2)	90% (29/32)	95% (21/22)
Main	0% (0/1)	0% (0/1)	0% (0/3)	100% (0/0)
Mediana	100% (1/1)	100% (1/1)	100% (29/29)	95% (21/22)

Dunque, questi risultati ci dicono che le linee di codice coperte grazie ai test ideati è del 100% mentre per quanto riguarda le branch è del 95%.

A questo punto si è voluto analizzare il codice contenuto nel metodo *findMedianSortedArrays(int[] nums1, int[] nums2)* della classe Mediana per poter aumentare la copertura del codice.

```java
public class Mediana {  4 usages
    public double findMedianSortedArrays(int[] nums1, int[] nums2) {  18 usages
        if (nums1.length > nums2.length) {
            return findMedianSortedArrays(nums2, nums1);
        }
        int m = nums1.length;
        int n = nums2.length;
        int totalLength = m + n;

        int left = 0;
        int right = m;
        if ((nums1.length !=0 || nums2.length!=0)) {
            while (left <= right) {

                int partitionNums1 = (left + right) / 2;
                int partitionNums2 = (totalLength + 1) / 2 - partitionNums1;
                int maxLeftNums1;
                if (partitionNums1 == 0) {
                    maxLeftNums1 = Integer.MIN_VALUE;
                } else {
                    maxLeftNums1 = nums1[partitionNums1 - 1];
                }

                int minRightNums1;
                if (partitionNums1 == m) {
                    minRightNums1 = Integer.MAX_VALUE;
                } else {
                    minRightNums1 = nums1[partitionNums1];
                }

                int maxLeftNums2 = nums2[partitionNums2 - 1];

                int minRightNums2;
                if (partitionNums2 == n) {
                    minRightNums2 = Integer.MAX_VALUE;
                } else {
                    minRightNums2 = nums2[partitionNums2];
                }

                if (maxLeftNums1 <= minRightNums2 && maxLeftNums2 <= minRightNums1) {
                    if (totalLength % 2 == 0) {
                        return (double) (Math.max(maxLeftNums1, maxLeftNums2) + Math.min(minRightNums1, minRightNums2)) / 2.0;
                    } else {
                        return (double) Math.max(maxLeftNums1, maxLeftNums2);
                    }
                } else if (maxLeftNums1 > minRightNums2) {
                    right = partitionNums1 - 1;
                } else {
                    left = partitionNums1 + 1;
                }
            }
        }
        return 0;
    }
```

La riga 16 contenente il ciclo *while* risulta evidenziata di giallo e questo ci indica che non sono state coperte tutte le alternative che potrebbero essere state prodotte dalla condizione interna al *while*.

La variabile *left risulta essere sempre minore o uguale alla* variabile *right* al primo ciclo. La variabile *right,* essendo che rappresenta il numero di elementi di un array, non può che essere 0 o maggiore di zero; quindi, qualsiasi siano gli array passati come input la condizione ***left<=right*** sarà sempre vera e di conseguenza verrà sempre eseguito almeno una volta il ciclo *while*. Inoltre, la logica implementata nel metodo non permette che *right* diventi inferiore a *left* oppure che *left* superi il valore di *right*.

Questo è dovuto al fatto che il codice interno al ciclo, dopo *n* iterazioni, soddisfi sempre la condizione riportata di seguito, trovando dei valori all'interno delle partizioni degli array che soddisfano la condizione, oppure, se non ne trova, impostando dei valori predefiniti, assegnando alle dovute variabili il valore massimo o minimo accettato per il tipo *int* rendendo vera la condizione riportata sotto.

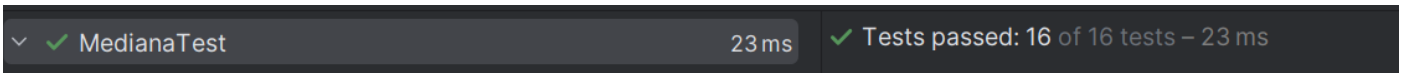
```
if (maxLeftNums1 <= minRightNums2 && maxLeftNums2 <= minRightNums1) {
    if (totalLength % 2 == 0) {
        return (double) (Math.max(maxLeftNums1, maxLeftNums2) + Math.min(minRightNums1, minRightNums2)) / 2.0;
    } else {
        return (double) Math.max(maxLeftNums1, maxLeftNums2);
    }
```

Conclusione

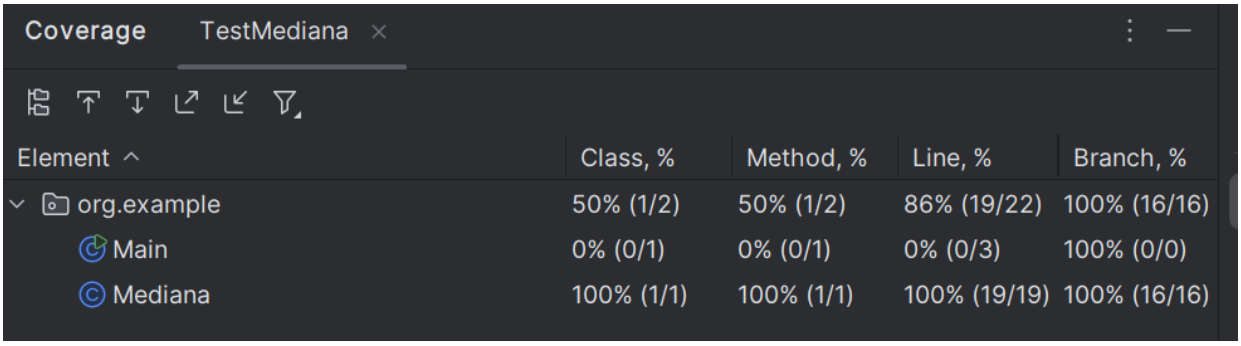
Fatte le dovute riflessioni, il team è arrivato alla conclusione che il codice analizzato e corretto, funziona per il calcolo della mediana ma con le dovute limitazioni. Per questo motivo si è voluto implementare un'altra logica per il calcolo della mediana rappresentata dal seguente metodo totalmente riscritto ma che rispetti sempre i requisiti iniziali.

```
public double findMedianSortedArraysRefactor(int [] nums1, int[] nums2){
    //1. controllare se gli array sono vuoti
    if (nums1.length == 0 && nums2.length == 0 ){
        return 0;
    }else{
        //2 unire i due array
        int [] combineArray = new int[nums1.length+nums2.length];

        for (int i = 0; i < nums1.length; i++) {
            combineArray[i] = nums1[i];
        }

        for (int i = 0; i < nums2.length; i++) {
            combineArray[nums1.length+i] = nums2[i];
        }

        //3. ordinare in ordine crescente l'array
        for (int i = 0; i < combineArray.length; i++) {
            for (int j = 1; j < (combineArray.length - i); j++) {
                if (combineArray[j - 1] > combineArray[j]) {
                    //  swap array's elements using temporary element
                    int temp = combineArray[j - 1];
                    combineArray[j - 1] = combineArray[j];
                    combineArray[j] = temp;
                }
            }
        }
        double result;
        if (combineArray.length % 2 == 0){//5. se l'array è pari prelevare i due elementi centrali e dividerli per 2
            int posizione = combineArray.length / 2;
            result =  (double) (combineArray[posizione - 1] + combineArray[posizione]) / 2;
        } else {//6. se l'array è dispari prelevare l'elemento di mezzo
            int posizione = (combineArray.length / 2);
            result = combineArray[posizione];
        }
        return result;
    }
}
```

Ripetendo i test già ideati, il nuovo metodo restituisce per tutti i test i risultati corretti passando come input anche array non ordinati



Arrivando ad una coverage del 100%




```java
public double findMedianSortedArrays(int [] nums1, int[] nums2){  15 usages
    //1. controllare se gli array sono vuoti
    if (nums1.length == 0 && nums2.length == 0 ){
        return 0;
    }else{
        //2 unire i due array
        int [] combineArray = new int[nums1.length+nums2.length];

        for (int i = 0; i < nums1.length; i++) {
            combineArray[i] = nums1[i];
        }

        for (int i = 0; i < nums2.length; i++) {
            combineArray[nums1.length+i] = nums2[i];
        }

        //3. ordinare in ordine crescente l'array
        for (int i = 0; i < combineArray.length; i++) {
            for (int j = 1; j < (combineArray.length - i); j++) {
                if (combineArray[j - 1] > combineArray[j]) {
                    //  swap array's elements using temporary element
                    int temp = combineArray[j - 1];
                    combineArray[j - 1] = combineArray[j];
                    combineArray[j] = temp;
                }
            }
        }
        double result;
        if (combineArray.length % 2 == 0){//5. se l'array è pari prelevare i due elementi cent
            int posizione = combineArray.length / 2;
            result =  (double) (combineArray[posizione - 1] + combineArray[posizione]) / 2;
        } else {//6. se l'array è dispari prelevare l'elemento di mezzo
            int posizione = (combineArray.length / 2);
            result = combineArray[posizione];
        }
        return result;
    }
}
```

Homework2

Introduzione

Il codice oggetto di verifica nell'ambito dell'attività di testing per homework2 riguarda il metodo `validaPassword(password, nome)`. Questo metodo mira a determinare la validità di una password fornita in input, restituendo un valore booleano corrispondente a *true* o *false*. Affinché una password sia considerata valida, deve soddisfare due criteri principali: il numero di caratteri deve essere superiore a quello del nome fornito come secondo parametro e deve contenere almeno un carattere numerico e letterale. È importante notare che i test eseguiti sono stati concepiti considerando un alfabeto latino.

```
public boolean validaPassword(String password, String nome) {  no usages
    if (password == null || nome == null){
        return false;
    } else{
        if (password.length() < nome.length()) {
            return false;
        }
    }
    // Controlla se la password contiene almeno un numero
    boolean checkNumInPw = false;
    for (char c : password.toCharArray()) {
        if (Character.isDigit(c)) {
            checkNumInPw = true;
            break;
        }
    }

    // Controlla se la password contiene almeno una lettera dell'alfabeto
    boolean checkAlphabetInPW = false;
    for(char c : password.toCharArray()){
        if (Character.isAlphabetic(c)){
            checkAlphabetInPW = true;
            break;
        }
    }
    // Restituisce true se la password ha almeno un numero e una lettera
    if (checkNumInPw && checkAlphabetInPW){
        return true;
    }else {
        return false;
    }
}
```

Ideare le partizioni

Le partizioni individuate sono le seguenti:

- **Password valida**, il primo parametro (password) contiene un numero di caratteri maggiori del secondo parametro (nome), inoltre contiene almeno un numero e una lettera o carattere speciale.
- **Password non valida,** il primo parametro (password) contiene un numero di caratteri inferiore al numero di caratteri presenti nel secondo parametro (nome) oppure contiene un numero di caratteri superiore del secondo parametro ma non contiene nessun numero oppure nessuna lettera.

1 PBT per "password non valida"

In primo luogo, sono stati creati due metodi generatori di stringhe arbitrarie, utilizzati per creare una vasta gamma di stringhe di input casuali per i test. In particolare, il metodo `stringaMista()` genera stringhe arbitrarie che includono lettere maiuscole e minuscole, numeri e vari simboli, con una lunghezza massima di 3 caratteri e minimo 0.

In seguito ai metodi di generazione di stringhe arbitrarie è stato creato il metodo di test per la verifica di password non valide *fail()*.

Successivamente, viene utilizzato il metodo *assertFalse()* per verificare che il valore di ritorno del metodo di test sia falso. Infine, viene impiegato il metodo *Statistics.label().collect()* per raccogliere ed etichettare i dati generati dal test.

```
@Property
@Report(Reporting.GENERATED)
@StatisticsReport(format = Histogram.class)
void fail(
            @ForAll ("stringaMista")  String password,
            @ForAll ("soloLettere") String nome){
    String risultato;
    if (password.matches( regex: "[a-zA-Z]+")){
        risultato = "pw con sole lettere";
    } else if (password.matches( regex: "\\d+")) {
        risultato = "pw solo numeri";

    } else {
        risultato = "pw mista";
    }
    assertFalse(credenziali.validaPassword(password,nome));
    Statistics.label("risultati").collect(risultato);
}


@Provide  no usages
Arbitrary<String> stringaMista() {
    return Arbitraries.strings()
            .withChars( charSequence: "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz1234567890!@#$%^&*(£)-_=+[]{}|;:'\",.<>?/`~\\")
            .ofMaxLength( i: 3)
            .ofMinLength( i: 0);
}

@Provide  no usages
Arbitrary<String> soloLettere() {
    return Arbitraries.strings()
            .withChars( charSequence: "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz'")
            .ofMinLength( i: 4)
            .ofMaxLength( i: 20);
}
```

1 Statistiche

Si è deciso di controllare dalle statistiche la distribuzione delle varie possibili combinazioni di password per assicurare che il metodo da testare abbia ricevuto in input i vari tipi di stringhe.

Dalle statistiche si può vedere che sono state generate i tipi di stringhe che ci si aspettava.

In questo caso sono state generate 1000 stringhe di test con la seguente distribuzione:

- 284 stringhe contenenti solo lettere,
- 689 stringhe alfanumeriche,
- 27 stringhe contenenti solo numeri.

```
timestamp = 2024-06-27T00:43:07.825, [CredenzialiTest:fail] (1000) risultati =
       # |               label | count |
    -----|---------------------|-------|-----------------------------------------------------------------------------------------
       0 | pw con sole lettere |   284 | ################################
       1 |            pw mista |   689 | ###################################################################################################
       2 |      pw solo numeri |    27 | ###

timestamp = 2024-06-27T00:43:07.831, CredenzialiTest:fail =
                             |-----------------------jqwik-----------------------
tries = 1000                 | # of calls to property
checks = 1000                | # of not rejected calls
generation = RANDOMIZED      | parameters are randomly generated
after-failure = SAMPLE_FIRST | try previously failed sample, then previous seed
when-fixed-seed = ALLOW      | fixing the random seed is allowed
edge-cases#mode = MIXIN      | edge cases are mixed in
edge-cases#total = 6         | # of all combined edge cases
edge-cases#tried = 6         | # of edge cases tried in current run
seed = -3276295653445385153  | random seed to reproduce generated values



Process finished with exit code 0
```

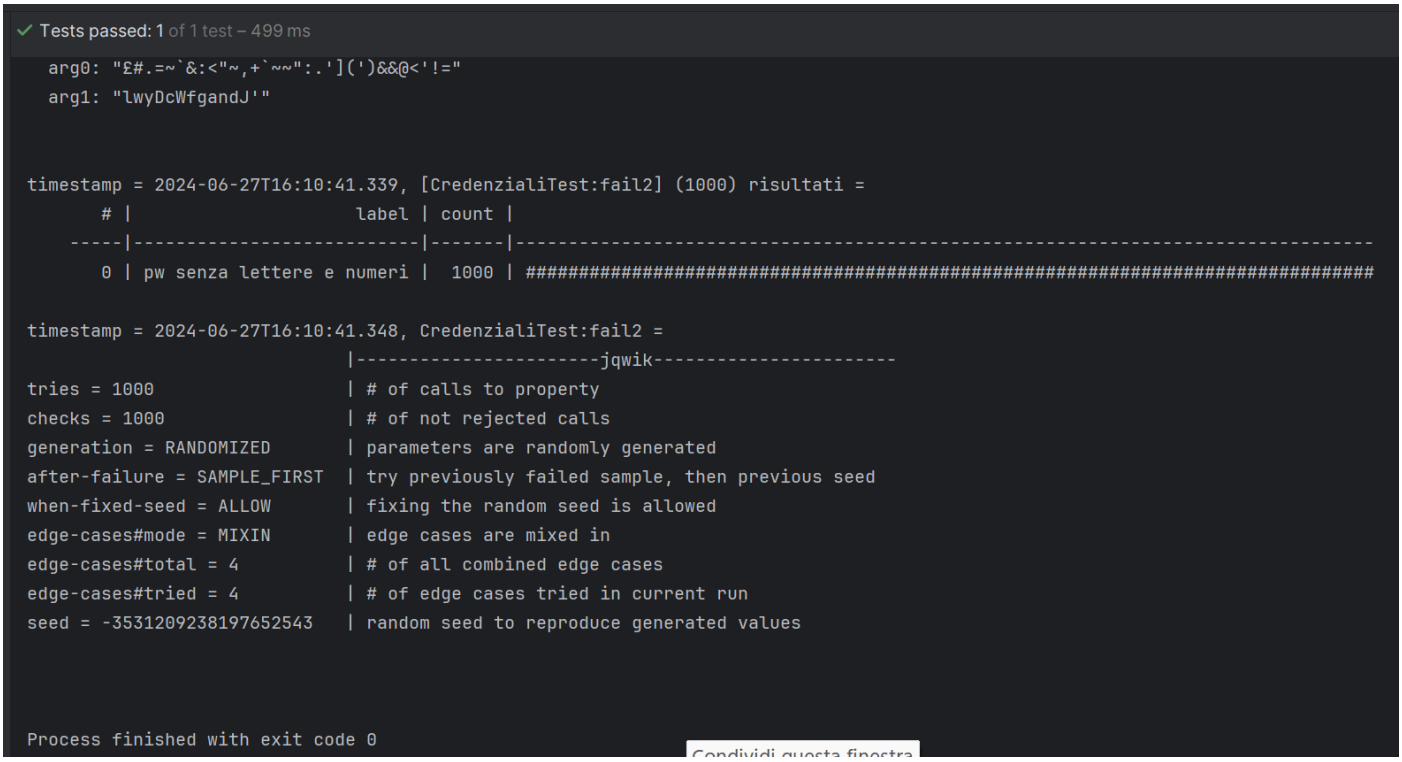
2 PBT “password non valida” (caratteri speciali)

Per questa proprietà si è voluto testare le stringhe di password non valide, che superano per numero di caratteri la stringa *nome* e che a causa dell’assenza di numeri e lettere dell’alfabeto risulta come password non valida.

```
@Property
@Report(Reporting.GENERATED)
@StatisticsReport(format = Histogram.class)
void fail2(
        @ForAll ("stringaMista2")  String password,
        @ForAll ("soloLettere2") String nome){
    String risultato;
    if (password.matches( regex: "[a-zA-Z]+")){
        risultato = "pw con sole lettere";
    } else if (password.matches( regex: "\\d+")) {
        risultato = "pw solo numeri";
    } else if (password.matches( regex: "[^a-zA-Z0-9]*")){
        risultato = "pw senza lettere e numeri";
    } else{
        risultato = "stringa mista";
    }

        assertFalse(credenziali.validaPassword(password,nome));
    Statistics.label("risultati").collect(risultato);
}


@Provide  no usages
Arbitrary<String> stringaMista2() {
    return Arbitraries.strings()
            .withChars( charSequence: "!@#$%^&*(£)-_=+[]{}|;:'\",.<>?/`~\\")
            .ofMinLength( i: 20);
}

@Provide  no usages
Arbitrary<String> soloLettere2() {
    return Arbitraries.strings()
            .withChars( charSequence: "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz'")
            .ofMinLength( i: 4)
            .ofMaxLength( i: 20);
}
```

2 Statistiche

Le password generate risultano al 100% stringhe senza né lettere né numeri.



3 PBT “password valida”

Il metodo *stringaMista1k()* è progettato per generare una stringa casuale di almeno 20 caratteri, combinando lettere maiuscole e minuscole, cifre e caratteri speciali.

In seguito ai metodi di generazione di stringhe, è stato creato il metodo di test per la verifica delle password valide *valid()*.

Successivamente, viene utilizzato il metodo *assertTrue()* per verificare che il valore di ritorno del metodo di test sia vero. Infine, viene impiegato il metodo *Statistics.collect()* per raccogliere i dati di interesse.

```
@Property
@StatisticsReport(format = Histogram.class)
void valid(
        @ForAll ("stringaMista1k") String password,
        @ForAll ("soloLettere1")String nome){
    String risultato;
    if (password.matches( regex: "[a-zA-Z]+")){
        risultato = "pw con sole lettere";
    } else if (password.matches( regex: "\\d+")) {
        risultato = "pw solo numeri";
    } else {
        risultato = "pw mista";
    }
    assertTrue(credenziali.validaPassword(password,nome));
    Statistics.collect(risultato);
}
@Provide no usages
Arbitrary<String> stringaMista1k() {
    Arbitrary<Character> letter = Arbitraries.of( ...values: 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z',
            'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z');
    Arbitrary<Character> digit = Arbitraries.of( ...values: '0', '1', '2', '3', '4', '5', '6', '7', '8', '9');

    Arbitrary<String> randomPart = Arbitraries.strings()
            .withChars( charSequence: "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz1234567890!@#$%^&*(£)-_=+[]{}|;:'\",.<>?/`~\\")
            .ofMinLength( i: 18); // Minimum length 18, as we add 2 more characters manually

    // combina per creare la stringa finale
    return Combinators.combine(letter, digit, randomPart) Combinator3<Character, Character, String>
            .as((l, d, r) -> shuffle( input: "" + l + d + r)) Arbitrary<String>
            .filter(s -> s.length() >= 20); // assicura che la lunghezza finale sia almeno 20
}
private String shuffle(String input) {  1 usage
    List<Character> characters = input.chars() IntStream
            .mapToObj(c -> (char) c) Stream<Character>
            .collect(Collectors.toList());
    Collections.shuffle(characters);
    StringBuilder result = new StringBuilder(characters.size());
    for (char c : characters) {
        result.append(c);
    }
    return result.toString();
}
```

3 Statistiche

Su un totale di 1000 stringhe di password sottoposte a test, la statistica ha mostrato che il 100% delle stringhe era composto da valori alfanumerici. Non sono state generate stringhe contenenti esclusivamente lettere o esclusivamente numeri, confermando così la corretta generazione delle stringhe attese.

```
✓ Tests passed: 1 of 1 test – 323 ms
"C:\Program Files\Java\jdk1.8.0_271\bin\java.exe" ...
timestamp = 2024-06-27T00:57:42.647, [CredenzialiTest:valid] (1000) statistics =
       # |    label | count |
    -----|----------|-------|-----------------------------------------------------------------------------
       0 | pw mista |  1000 | ################################################################################
```