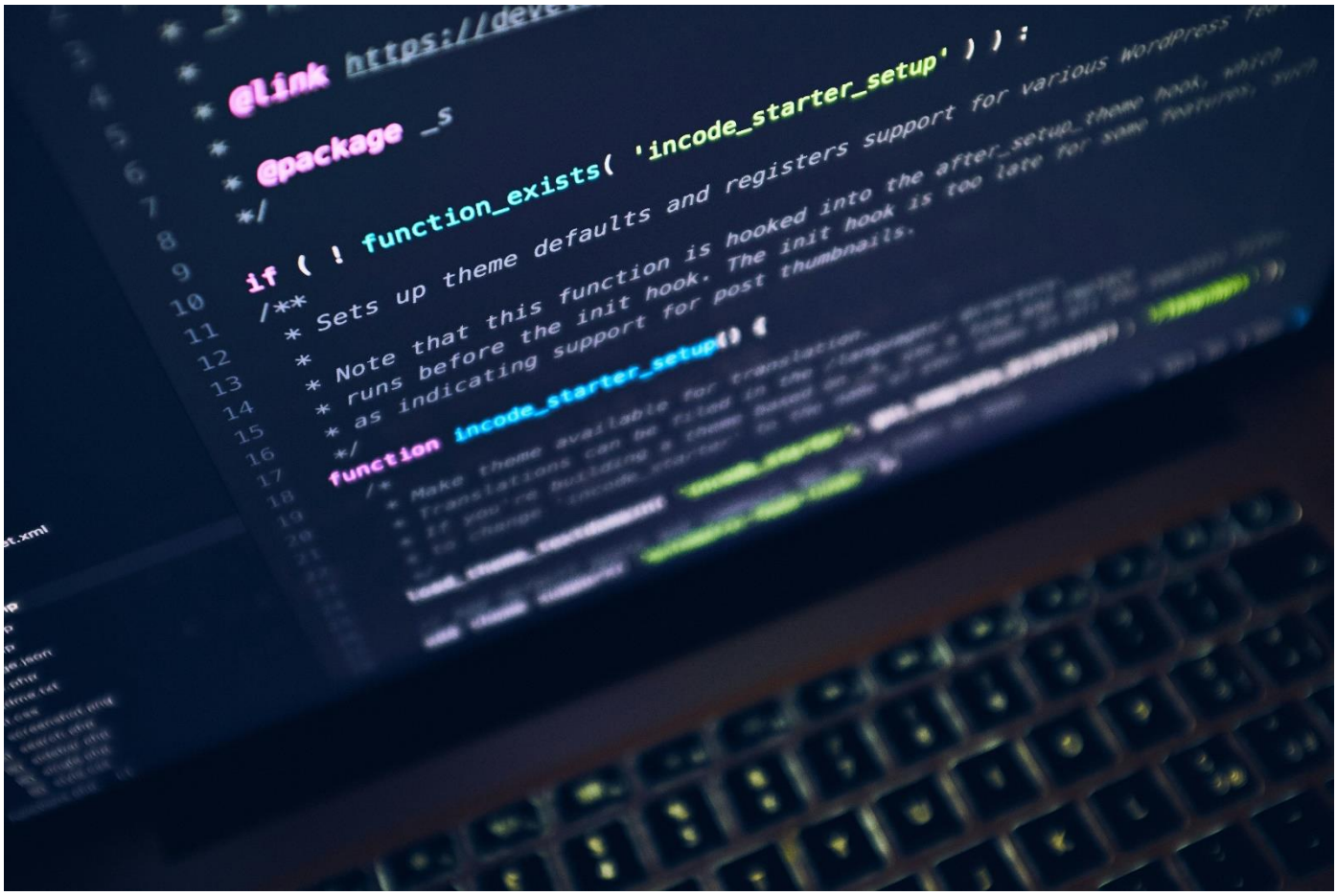


SAFR – ITSS23/24

Fractions java class methods from Apache Commons Lang Library

Componenti:

- Samuel Giovanni Paparella: s.paparella24@studenti.uniba.it, 764513;
- Francesco Pappagallo: f.pappagallo5@studenti.uniba.it, 758237.

Indice

1 Introduzione

In un mondo sempre più digitale e interconnesso, la qualità del software è diventata un elemento critico per il successo delle aziende e per la soddisfazione degli utenti finali. Effettuare software testing rappresenta è fondamentale per garantire che il software sia affidabile, sicuro e in particolare conforme ai requisiti specificati.

Esploreremo i diversi tipi di test, le tecniche di progettazione dei test, gli strumenti di automazione e le best practice per la gestione dei processi di test.

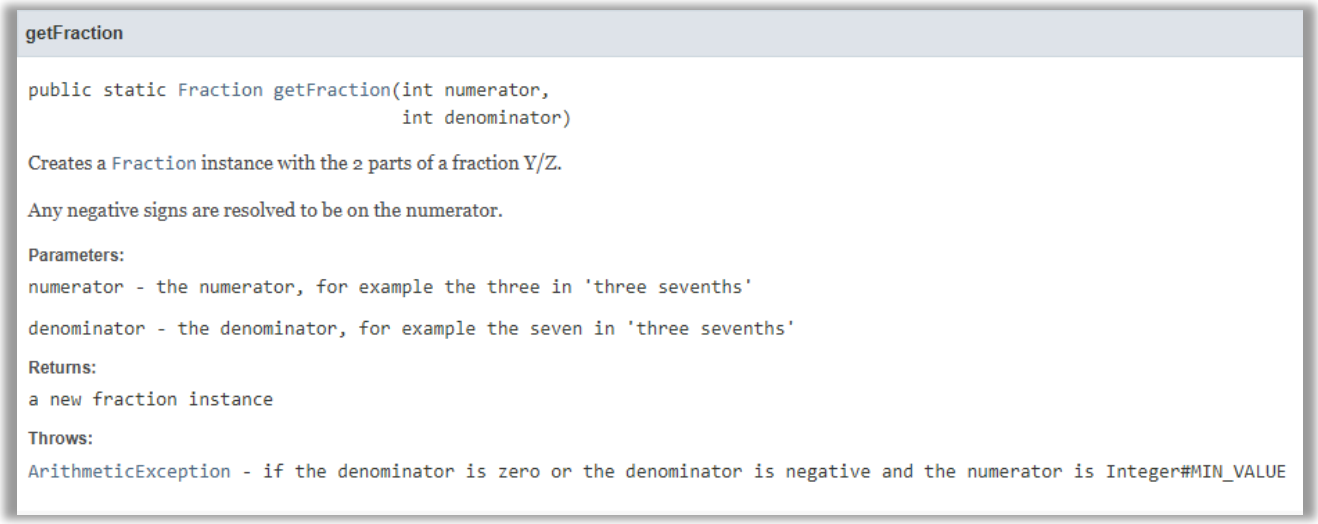
2 Capitolo 1 (HW1)

Nell'ambito dell'Homework 1, abbiamo effettuato specification – based testing sul costruttore della classe Fraction, getReducedFraction e greatestCommonDivisor. Questi metodi riguardano operazioni come calcoli numerici di base e altre funzionalità.

Il nostro obiettivo sarà comprendere e testare attentamente il comportamento di questi metodi rispetto alle specifiche fornite al fine di garantire che operino correttamente in una varietà di scenari.

Step 1: Understanding the requirements

- **Fraction constructor**:



getFraction

```
public static Fraction getFraction(int numerator,
                                   int denominator)
```

Creates a Fraction instance with the 2 parts of a fraction Y/Z.

Any negative signs are resolved to be on the numerator.

Parameters:

numerator - the numerator, for example the three in 'three sevenths'

denominator - the denominator, for example the seven in 'three sevenths'

Returns:

a new fraction instance

Throws:

ArithmeticException - if the denominator is zero or the denominator is negative and the numerator is Integer#MIN_VALUE

Il costruttore della classe Fraction accetta un numeratore e un denominatore come input e inizializza un **oggetto Fraction** con tali valori. Prima di procedere all'inizializzazione, il costruttore effettua controlli per garantire che il denominatore non sia zero e gestisce casi speciali come denominatori negativi.

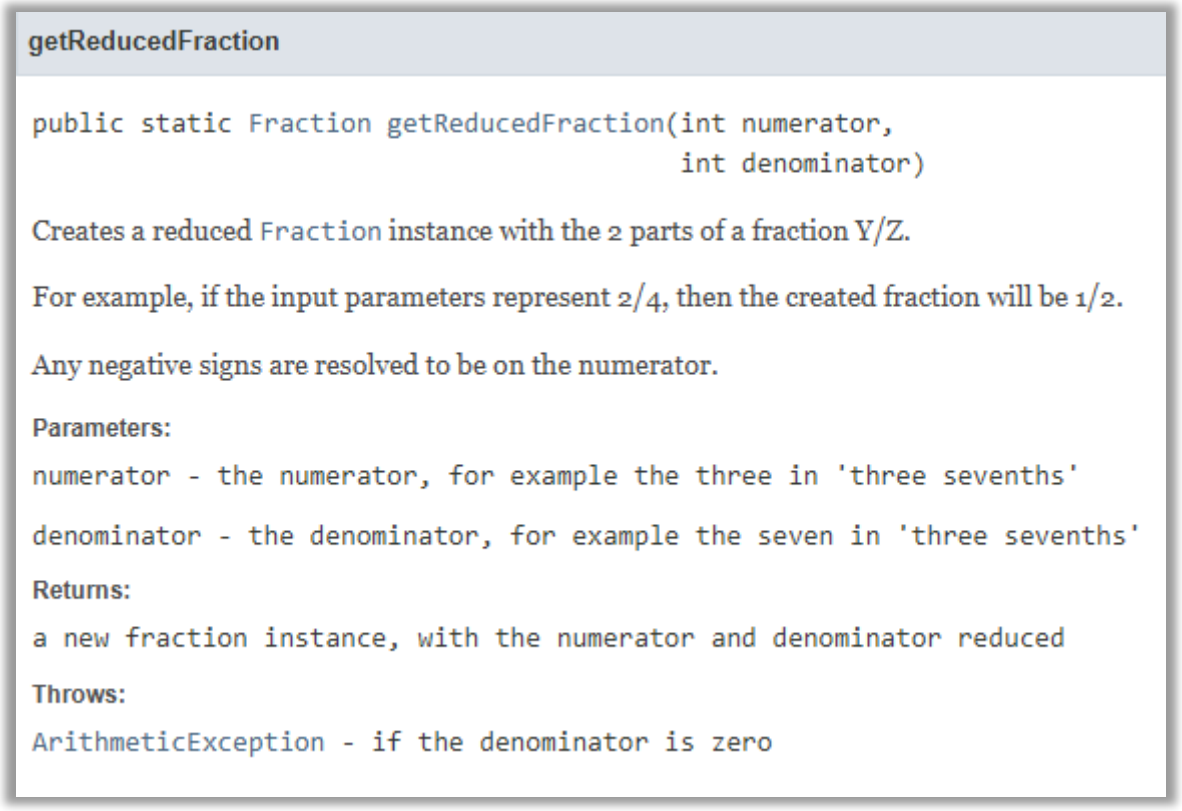
- Se il denominatore è zero, viene lanciata un'eccezione;
- Se il numeratore è zero, la frazione viene normalizzata a 0/1;
- Se il denominatore è negativo, nega anche il numeratore per mantenere la coerenza.

Infine, vengono impostati il numeratore e il denominatore con i valori forniti. In questo modo, il costruttore garantisce che l'oggetto **Fraction** sia correttamente inizializzato e rappresenti una frazione valida.

Nota: precisiamo che in questa fase ci siamo accorti che il costruttore della classe Fraction appartenente al codice sorgente di partenza, ovvero il codice preso direttamente dalla libreria Apache Commons Lang (Fraction (Apache Commons Lang 3.14.0 API)), non effettuava assolutamente nessuno dei controlli soprammenzionati. In questo modo era possibile generare una frazione che avesse valori invalidi come numeratore zero, denominatore zero, entrambi pari a zero e così via.

Pertanto, abbiamo corretto il codice sorgente di partenza inserendo gli opportuni controlli degli input nel costruttore.

- **getReducedFraction:**



getReducedFraction

```
public static Fraction getReducedFraction(int numerator,
                                          int denominator)
```

Creates a reduced `Fraction` instance with the 2 parts of a fraction Y/Z.

For example, if the input parameters represent 2/4, then the created fraction will be 1/2.

Any negative signs are resolved to be on the numerator.

Parameters:

`numerator` - the numerator, for example the three in 'three sevenths'

`denominator` - the denominator, for example the seven in 'three sevenths'

Returns:

a new fraction instance, with the numerator and denominator reduced

Throws:

`ArithmeticException` - if the denominator is zero

Il metodo accetta un numeratore e un denominatore come input e restituisce una **frazione ridotta ai minimi termini**.

Inizialmente, il metodo esegue controlli per assicurarsi che il denominatore non sia zero e gestisce casi speciali come denominatori negativi e numeratori nulli.

Successivamente, semplifica la frazione calcolando il Massimo Comune Divisore tra il numeratore e il denominatore e dividendo entrambi per questo valore.

Infine, restituisce un oggetto **Fraction** che rappresenta la frazione ridotta ai minimi termini.

- **greatestCommonDivisor:**

Calcola il Massimo Comune Divisore del valore assoluto di due numeri, utilizzando il *binary gcd* che evita operazioni di divisione e modulo. Come input il metodo prende due valori interi u e v, che sono due numeri diversi da zero, e come output restituisce il **Massimo Comune Divisore** che non è mai zero.

Step 2: Explore what the program does for various inputs

- **Fraction constructor:**

Abbiamo eseguito una suite di test utilizzando una varietà di input al fine di coprire un'ampia gamma di scenari possibili. Verifichiamo l'affidabilità del metodo utilizzando valori che creano frazioni valide e identificando le situazioni in cui tali frazioni non possano essere considerate valide.

```java
public class FractionTest
{
    @Nested
    class FractionConstructorTests
    {
        @ParameterizedTest
        @MethodSource("validFractionProvider")
        void shouldReturnAValidFraction(Fraction fraction, int expectedNumerator, int expectedDenominator) // T1
        {
            assertAll(
                    () -> assertEquals(expectedNumerator, fraction.getNumerator()),
                    () -> assertEquals(expectedDenominator, fraction.getDenominator())
            );
        }

        private static Stream<Arguments> validFractionProvider()
        {
            return Stream.of(
                    Arguments.of(new Fraction(numerator:1, denominator:2), 1, 2), // T1.1
                    Arguments.of(new Fraction(numerator:1, -2), -1, 2), // T1.2
                    Arguments.of(new Fraction(-1, denominator:2), -1, 2), // T1.3
                    Arguments.of(new Fraction(-1, -2), 1, 2) // T1.4
            );
        }

        @Test
        void zeroDenominatorShouldThrowArithmeticException() // T2
        {
            assertThrows(expectedType:ArithmeticException.class, () -> new Fraction(numerator:1, denominator:0));

            // Other cases:
            // first case: numerator is Integer.MAX_VALUE
            assertThrows(expectedType:ArithmeticException.class, () -> new Fraction(Integer.MAX_VALUE, denominator:0));

            // second case: numerator is Integer.MIN_VALUE
            assertThrows(expectedType:ArithmeticException.class, () -> new Fraction(Integer.MIN_VALUE, denominator:0));
        }
```

- **getReducedFraction:**

Abbiamo eseguito una suite di test utilizzando una varietà di input al fine di coprire un'ampia gamma di scenari possibili. Verifichiamo che il metodo sia in grado di restituire una frazione ridotta ai minimi termini valida utilizzando valori interi positivi e negativi generati manualmente. In particolare, verifichiamo che il metodo restituisca una frazione ridotta ai minimi termini corretta anche con input negativi.

```java
@Nested
class GetReducedFractionTests
{
    @ParameterizedTest
    @MethodSource("reducedFractionProvider")
    void shouldReturnAValidReducedFraction(Fraction expectedFraction, int numerator, int denominator) // T6
    {
        assertEquals(expectedFraction, Fraction.getReducedFraction(numerator, denominator));
    }

    private static Stream<Arguments> reducedFractionProvider()
    {
        return Stream.of(
                // T6.1, 2/4 should return 1/2
                Arguments.of(new Fraction(numerator:1, denominator:2), 2, 4),

                // T6.2, 2/-6 should return -1/3
                Arguments.of(new Fraction(-1, denominator:3), 2, -6),

                // T6.3, -4/6 should return -2/3
                Arguments.of(new Fraction(-2, denominator:3), -4, 6),

                // T6.4, -6/-4 should return 3/2
                Arguments.of(new Fraction(numerator:3, denominator:2), -6, -4),

                // T6.5, 2/2 should return 1/1 or 1
                Arguments.of(new Fraction(numerator:1, denominator:1), 2, 2),
                Arguments.of(new Fraction(-1, denominator:1), -2, 2),
                Arguments.of(new Fraction(-1, denominator:1), 2, -2),
                Arguments.of(new Fraction(numerator:1, denominator:1), -2, -2),

                // T6.6, odd numerator and odd denominator
                // This test also verifies the method behavior with prime numbers.
                // 1 and 3 are indeed the first two prime numbers
                Arguments.of(new Fraction(numerator:1, denominator:3), 1, 3),
                Arguments.of(new Fraction(-1, denominator:3), 1, -3),
                Arguments.of(new Fraction(-1, denominator:3), -1, 3),
                Arguments.of(new Fraction(numerator:1, denominator:3), -1, -3),

                // T6.7, odd numerator and even denominator
                Arguments.of(new Fraction(numerator:3, denominator:4), 3, 4),
                Arguments.of(new Fraction(-3, denominator:4), 3, -4),
                Arguments.of(new Fraction(-3, denominator:4), -3, 4),
                Arguments.of(new Fraction(numerator:3, denominator:4), -3, -4),
```

- **greatestCommonDivisor:**

Abbiamo eseguito una suite di test utilizzando una varietà di input al fine di coprire un'ampia gamma di scenari possibili. Verifichiamo la correttezza del calcolo del Massimo Comune Divisore (GCD) utilizzando valori sia positivi che negativi. Inoltre, identifichiamo i casi in cui il GCD risulta essere invalido.

```java
@Nested
class GreatestCommonDivisorTests
{
    @Test
    void shouldReturnCorrectGCD() // T12
    {
        // Classic GCD calculation
        assertEquals(expected:4, Fraction.greatestCommonDivisor(u:4, v:8));
        assertEquals(expected:4, Fraction.greatestCommonDivisor(-4, v:8));
        assertEquals(expected:4, Fraction.greatestCommonDivisor(u:8, -4));
        assertEquals(expected:4, Fraction.greatestCommonDivisor(-8, -4));

        // The GCD of two prime numbers that are not the same is 1
        assertEquals(expected:1, Fraction.greatestCommonDivisor(u:3, v:5));
        assertEquals(expected:1, Fraction.greatestCommonDivisor(-3, v:5));
        assertEquals(expected:1, Fraction.greatestCommonDivisor(u:3, -5));
        assertEquals(expected:1, Fraction.greatestCommonDivisor(-3, -5));

        // GCD calculation between an even number and an odd number
        // First case: u is even and v is odd
        assertEquals(expected:1, Fraction.greatestCommonDivisor(u:2, v:3));
        assertEquals(expected:1, Fraction.greatestCommonDivisor(-2, v:3));
        assertEquals(expected:1, Fraction.greatestCommonDivisor(u:2, -3));
        assertEquals(expected:1, Fraction.greatestCommonDivisor(-2, -3));

        // Second case: u is odd and v is even
        assertEquals(expected:1, Fraction.greatestCommonDivisor(u:3, v:4));
        assertEquals(expected:1, Fraction.greatestCommonDivisor(-3, v:4));
        assertEquals(expected:1, Fraction.greatestCommonDivisor(u:3, -4));
        assertEquals(expected:1, Fraction.greatestCommonDivisor(-3, -4));
    }
```

Step 3: Explore inputs, outputs and identify partitions

- **Fraction constructor**:

Individual inputs

Il costruttore della classe Fraction accetta due numeri interi in input così classificabili: tutti gli input sono di tipo **intero** in base alle specifiche di Apache e vanno in un range da -2^31 (Integer.MIN_VALUE) a 2^31 (Integer.MAX_VALUE).

- Numerator: diverso da zero, zero, Integer.MAX_VALUE, Integer.MIN_VALUE;
- Denominator: diverso da zero, zero, Integer.MAX_VALUE, Integer.MIN_VALUE.

Combinations of inputs

Miriamo a testare una vasta gamma di scenari per garantire una copertura completa dei possibili input. Questi scenari includono:

- Numeratore e denominatore diversi da zero;
- Numeratore diverso da zero e denominatore = 0;
- Numeratore diverso da zero e denominatore = Integer.MAX_VALUE;

- Numeratore diverso da zero e denominatore = Integer.MIN_VALUE;
- Numeratore = 0 e denominatore diverso da zero;
- Numeratore = 0 e denominatore = 0;
- Numeratore = 0 e denominatore = Integer.MAX_VALUE;
- Numeratore = 0 e denominatore = Integer.MIN_VALUE;
- Numeratore = Integer.MAX_VALUE e denominatore diverso da zero;
- Numeratore = Integer.MAX_VALUE e denominatore = 0;
- Numeratore = Integer.MAX_VALUE e denominatore = Integer.MAX_VALUE;
- Numeratore = Integer.MAX_VALUE e denominatore = Integer.MIN_VALUE;
- Numeratore = Integer.MIN_VALUE e denominatore diverso da zero;
- Numeratore = Integer.MIN_VALUE e denominatore = 0;
- Numeratore = Integer.MIN_VALUE e denominatore = Integer.MAX_VALUE;
- Numeratore = Integer.MIN_VALUE e denominatore = Integer.MIN_VALUE.

In questo modo, possiamo garantire che il metodo sia in grado di gestire diversi casi di input in modo accurato.

Classes of (expected) outputs

Tutti gli output validi sono frazioni **intere** in base alle specifiche di Apache e vanno in un range da -2^31 a 2^31.

- **getReducedFraction:**

Individual inputs

Anche in questo caso il metodo accetta due interi, quindi gli individual inputs sono: tutti gli input sono di tipo **intero** in base alle specifiche di Apache e vanno in un range da -2^31 (Integer.MIN_VALUE) a 2^31 (Integer.MAX_VALUE).

- Numerator: diverso da zero, zero, Integer.MAX_VALUE, Integer.MIN_VALUE;
- Denominator: diverso da zero, zero, Integer.MAX_VALUE, Integer.MIN_VALUE.

Combinations of inputs

Miriamo a testare una vasta gamma di inputs al fine di ottenere una copertura completa dei possibili scenari di input. In particolare, testiamo i casi:

- Numeratore e denominatore diversi da zero;
- Numeratore diverso da zero e denominatore = 0;
- Numeratore diverso da zero e denominatore = Integer.MAX_VALUE;
- Numeratore diverso da zero e denominatore = Integer.MIN_VALUE;
- Numeratore = 0 e denominatore diverso da zero;
- Numeratore = 0 e denominatore = 0;
- Numeratore = 0 e denominatore = Integer.MAX_VALUE;
- Numeratore = 0 e denominatore = Integer.MIN_VALUE;
- Numeratore = Integer.MAX_VALUE e denominatore diverso da zero;
- Numeratore = Integer.MAX_VALUE e denominatore zero;
- Numeratore = Integer.MAX_VALUE e denominatore = Integer.MAX_VALUE;
- Numeratore = Integer.MAX_VALUE e denominatore = Integer.MIN_VALUE;
- Numeratore = Integer.MIN_VALUE e denominatore diverso da zero;
- Numeratore = Integer.MIN_VALUE e denominatore = 0;
- Numeratore = Integer.MIN_VALUE e denominatore = Integer.MAX_VALUE;
- Numeratore = Integer.MIN_VALUE e denominatore = Integer.MIN_VALUE.

In questo modo, possiamo garantire che il metodo funzioni in modo robusto e in grado di gestire diversi casi di input in modo accurato.

Classes of (expected) outputs

Il nostro obiettivo è garantire che il metodo restituisca una frazione ridotta ai minimi termini, rispettando regole matematiche precise per evitare errori.

Tutti gli output validi sono frazioni **intere** in base alle specifiche di Apache e vanno in un range da -2^31 a 2^31.

- **greatestCommonDivisor:**

Individual inputs

In questo caso, questo metodo accetta due interi, quindi andiamo ad individuare gli individual inputs:

PARAMETRO u
u = 0
u = 1
u positivo
u primo
u negativo

PARAMETRO v
v = 0
v = 1
v positivo
v primo
v negativo

Combinations of inputs

Miriamo a testare una vasta gamma di inputs al fine di ottenere una copertura completa dei possibili scenari di input:

- u 0 e v 0;
- u 0 e v positivo;
- u 0 e v negativo;
- u positivo e v 0;
- u negativo e v 0;
- u 0 e v MIN_VALUE;
- u MIN_VALUE e v 0;
- u MIN_VALUE e v MIN_VALUE;
- u 1 e v positivo;
- u -1 e v positivo;
- u -1 e v negativo;
- u positivo e v 1;
- u negativo e v 1;
- u negativo e v -1;
- u 1 e v 1;
- u -1 e v 1;
- u -1 e v -1;
- u 1 e v MIN_VALUE;
- u MIN_VALUE e v 1;
- u -1 e v MIN_VALUE;
- u MIN_VALUE e v -1;

- u 1 e v MAX_VALUE;
- u MAX_VALUE e v 1;
- u -1 e v MAX_VALUE;
- u MAX_VALUE e v -1;
- u MIN_VALUE e v MAX_VALUE;
- u MAX_VALUE e v MIN_VALUE;
- u MAX_VALUE e v MAX_VALUE;
- u positivo e v MIN_VALUE;
- u MIN_VALUE e v positivo;
- u negativo e v MIN_VALUE;
- u MIN_VALUE e v negativo;
- u positivo e v MAX_VALUE;
- u MAX_VALUE e v positivo;
- u negativo e v MAX_VALUE;
- u MAX_VALUE e v negativo;
- u positivo e v MIN_VALUE;
- u MIN_VALUE e v positivo;
- u negativo e v MIN_VALUE;
- u MIN_VALUE e v negativo;
- u positivo e v MAX_VALUE;
- u MAX_VALUE e v positivo;
- u negativo e v MAX_VALUE;
- u MAX_VALUE e v negativo;
- u MIN_VALUE + 1 v MIN_VALUE + 1;
- u MIN_VALUE + 1 v -1;
- u MIN_VALUE + 1 v 1;
- u MIN_VALUE + 1 v MAX_VALUE - 1;
- u MAX_VALUE - 1 v MIN_VALUE + 1;
- u MAX_VALUE - 1 v -1;
- u MAX_VALUE - 1 v 1;
- u MAX_VALUE - 1 v MAX_VALUE - 1;

Classes of (expected) outputs

Tutti gli output validi sono numeri **interi positivi.**

Step 4: Identify boundary cases (aka corner case)

- **Fraction constructor:**

Basandoci sulla definizione di frazione, che coinvolge due numeri interi assegnati rispettivamente a denominatore e numeratore, abbiamo individuato situazioni significative che richiedono una gestione appropriata:

1. **Il denominatore è prossimo allo zero**: se il denominatore fosse zero, porterebbe ad un valore indefinito. In base ai requisiti di Apache *lancia un'eccezione di tipo ArithmeticException*. Se il denominatore fosse un valore prossimo allo zero (-1 e 1), il metodo funziona correttamente;

2. **Il numeratore è zero e il denominatore è un valore limite o prossimo a quest' ultimo**: se il numeratore è uguale a zero, il metodo restituisce il valore costante Fraction.ZERO;

3. **Valori limite numeratore/denominatore**: se il numeratore o il denominatore o entrambi fossero uguali a Intger.MAX_VALUE oppure Integer.MIN_VALUE non dovrebbe sollevare eccezioni;

4. **Overflow/Underflow on point ed off point**: se il numeratore o il denominatore o entrambi fossero prossimi a Integer.MAX_VALUE oppure Integer.MIN_VALUE non dovrebbe sollevare eccezioni (Integer.MAX_VALUE – 1 e Integer.MIN_VALUE + 1);

- **GetReducedFraction:**

Basandoci sulla definizione di frazione, che coinvolge due numeri interi assegnati rispettivamente a denominatore e numeratore, abbiamo individuato situazioni significative che riguardano la riduzione delle frazioni:

1. **Il denominatore è prossimo zero**: se il denominatore fosse zero, porterebbe ad un valore indefinito. In base ai requisiti di Apache *lancia un'eccezione di tipo ArithmeticException*. Se il denominatore fosse un valore prossimo allo zero (-1 e 1), il metodo funziona correttamente;

2. **Il numeratore è zero e il denominatore è un valore limite o prossimo a quest' ultimo**: se il numeratore è uguale a zero, il metodo restituisce il valore costante Fraction.ZERO;

3. **Valori limite numeratore/denominatore**: se il numeratore o il denominatore o entrambi fossero uguali a Intger.MAX_VALUE oppure Integer.MIN_VALUE non dovrebbe sollevare eccezioni;

4. **Overflow/Underflow on point ed off point**: se il numeratore o il denominatore o entrambi fossero prossimi a Integer.MAX_VALUE oppure Integer.MIN_VALUE non dovrebbe sollevare eccezionin (Integer.MAX_VALUE – 1 e Integer.MIN_VALUE + 1);

5. **Numeratore è un numero pari molto grande o molto piccolo, Il denominatore è MIN_VALUE**: Se il numeratore e il denominatore sono numeri pari il metodo divide entrambi i termini per 2;

- **GreatestCommonDivisor:**

Basandoci sulla definizione di Massimo comune divisore (GCD), che coinvolge due numeri interi assegnati a due parametri **u** e **v**, ci siamo soffermati sulle situazioni fondamentali gestite da questo metodo:

1. Calcolo del GCD tra 1, -1 e MIN_VALUE;
2. Calcolo del GCD tra 1, -1 e MAX_VALUE;
3. Calcolo del GCD tra 1, -1 e MIN_VALUE + 1;
4. Calcolo del GCD tra 1, -1 e MAX_VALUE - 1;
5. Calcolo del GCD tra 2, -2 e MIN_VALUE;
6. Calcolo del GCD tra MIN_VALUE e MAX_VALUE e viceversa;
7. Calcolo del GCD tra MIN_VALUE + 1 e MAX_VALUE - 1 e viceversa;
8. Calcolo del GCD tra numero pari (positivo e negativo) e MIN_VALUE;
9. Calcolo del GCD tra numero pari (positivo e negativo) e MAX_VALUE;
10. Calcolo del GCD tra numero dispari (positivo e negativo) e MIN_VALUE;
11. Calcolo del GCD tra numero dispari (positivo e negativo) e MAX_VALUE;

Step 5: Devise test cases

In questo step andiamo a verificare per ogni metodo, quali parametri devono essere combinati tra loro e quali no:

- **Fraction**:

T1: Numeratore positivo e denominatore positivo;
Numeratore positivo e denominatore negativo;
Numeratore negativo e denominatore positivo;
Numeratore negativo e denominatore negativo;

T2: Numeratore 1 e denominatore 0;
Numeratore MAX_VALUE e denominatore 0;
Numeratore MIN_VALUE e denominatore 0;

T3: Numeratore 0 e denominatore 1;
Numeratore 0 e denominatore negativo;
Numeratore 0 e denominatore MAX_VALUE;
Numeratore 0 e denominatore MIN_VALUE;
Numeratore 0 e denominatore MAX_VALUE -1;
Numeratore 0 e denominatore MIN_VALUE +1;

T4: Numeratore MIN_VALUE e denominatore -1;
Numeratore 1 e denominatore MIN_VALUE;
Numeratore MIN_VALUE e denominatore MIN_VALUE;
Numeratore MAX_VALUE e denominatore MIN_VALUE;

T5: Numeratore MIN_VALUE e denominatore 1;
Numeratore MIN_VALUE e denominatore MAX_VALUE;
Numeratore MAX_VALUE e denominatore -1;
Numeratore MAX_VALUE e denominatore 1;
Numeratore MAX_VALUE e denominatore MAX_VALUE;
Numeratore MIN_VALUE + 1 e denominatore MIN_VALUE + 1;
Numeratore MIN_VALUE + 1 e denominatore -1;
Numeratore MIN_VALUE + 1 e denominatore 1;
Numeratore MIN_VALUE + 1 e denominatore MAX_VALUE - 1;
Numeratore MAX_VALUE - 1e denominatore MIN_VALUE + 1;
Numeratore MAX_VALUE - 1 e denominatore -1;
Numeratore MAX_VALUE - 1 e denominatore 1;
Numeratore MAX_VALUE - 1 e denominatore MAX_VALUE - 1;

- **GetReducedFraction:**

T6: Numeratore positivo e denominatore positivo;
Numeratore positivo e denominatore negativo;
Numeratore negativo e denominatore positivo;
Numeratore negativo e denominatore negativo;
Numeratore positivo e denominatore positivo;
Numeratore negativo e denominatore positivo;
Numeratore positivo e denominatore negativo;
Numeratore negativo e denominatore negativo;

Numeratore positivo primo e denominatore positivo primo;
Numeratore positivo primo e denominatore negativo primo;
Numeratore negativo primo e denominatore positivo primo;
Numeratore negativo primo e denominatore positivo primo;

Numeratore positivo dispari e denominatore positivo pari;
Numeratore positivo dispari e denominatore negativo pari;
Numeratore negativo dispari e denominatore positivo pari;
Numeratore negativo dispari e denominatore positivo pari;

Numeratore positivo pari e denominatore positivo dispari;
Numeratore positivo pari e denominatore negativo dispari;
Numeratore negativo pari e denominatore positivo dispari;
Numeratore negativo pari e denominatore positivo dispari;

T7: Numeratore 1 e denominatore 0;
Numeratore MAX_VALUE e denominatore 0;
Numeratore MIN_VALUE e denominatore 0;

T8: Numeratore 0 e denominatore 1;
Numeratore 0 e denominatore MAX_VALUE;
Numeratore 0 e denominatore MIN_VALUE;
Numeratore 0 e denominatore MAX_VALUE - 1;
Numeratore 0 e denominatore MIN_VALUE + 1;

T9: Numeratore 2 e denominatore MIN VALUE;
Numeratore -2 e denominatore MIN VALUE;

Numeratore MAX_VALUE - 1 e denominatore MIN VALUE;
Numeratore MIN_VALUE + 2 e denominatore MIN VALUE;

T10: Numeratore MIN_VALUE e denominatore -1;
Numeratore 1 e denominatore MIN_VALUE;
Numeratore MAX_VALUE e denominatore MIN_VALUE;

T11: Numeratore MIN_VALUE e denominatore 1;
Numeratore MIN_VALUE e denominatore MAX_VALUE;
Numeratore MAX_VALUE e denominatore -1;
Numeratore MAX_VALUE e denominatore 1;
Numeratore MAX_VALUE e denominatore MAX_VALUE;
Numeratore MIN_VALUE + 1 e denominatore MIN_VALUE + 1;
Numeratore MIN_VALUE + 1 e denominatore -1;
Numeratore MIN_VALUE + 1 e denominatore 1;
Numeratore MIN_VALUE + 1 e denominatore MAX_VALUE - 1;
Numeratore MAX_VALUE - 1 e denominatore MIN_VALUE + 1;
Numeratore MAX_VALUE - 1 e denominatore -1;
Numeratore MAX_VALUE - 1 e denominatore 1;
Numeratore MAX_VALUE - 1 e denominatore MAX_VALUE - 1;

- **GreatestCommonDivisor:**

T12: Calcolo classico del GCD

u positivo e v positivo;
u negativo e v positivo;
u positivo e v negativo;
u negativo e v negativo;

Calcolo del GCD tra numeri primi non uguali tra loro

u positivo e v positivo;
u negativo e v positivo;
u positivo e v negativo;
u negativo e v negativo;

Calcolo del GCD tra numero pari e numero dispari

u positivo e v positivo;
u negativo e v positivo;
u positivo e v negativo;
u negativo e v negativo;

Calcolo del GCD tra numero dispari e numero pari

u positivo e v positivo;
u negativo e v positivo;
u positivo e v negativo;
u negativo e v negativo;

T13: u 0 e v 0;

T14: u 0 e v positivo;
u 0 e v negativo;
u positivo e v 0;
u negativo e v 0;

T15: u 0 e v MIN_VALUE;
u MIN_VALUE e v 0;
u MIN_VALUE e v MIN_VALUE;

T16: Calcolo del GCD tra 1 e numero intero

u 1 e v positivo;
u -1 e v positivo;
u -1 e v negativo;

Calcolo del GCD tra numero intero e 1

u positivo e v 1;
u negativo e v 1;
u negativo e v -1;

Calcolo del GCD con entrambi i parametri uguali a 1

u 1 e v 1;
u -1 e v 1;
u -1 e v -1;

T17: Calcolo del GCD tra 1 e MIN_VALUE

u 1 e v MIN_VALUE;

u MIN_VALUE e v 1;
u -1 e v MIN_VALUE;
u MIN_VALUE e v -1;

Calcolo del GCD tra 1 e MAX_VALUE

u 1 e v MAX_VALUE;
u MAX_VALUE e v 1;
u -1 e v MAX_VALUE;
u MAX_VALUE e v -1;

Calcolo del GCD tra MIN_VALUE e MAX_VALUE

u MIN_VALUE e v MAX_VALUE;
u MAX_VALUE e v MIN_VALUE;
u MAX_VALUE e v MAX_VALUE;

Calcolo del GCD tra numero pari e MIN_VALUE

u positivo e v MIN_VALUE;

u MIN_VALUE e v positivo;
u negativo e v MIN_VALUE;
u MIN_VALUE e v negativo;

Calcolo del GCD tra numero pari e MAX_VALUE

u positivo e v MAX_VALUE;

u MAX_VALUE e v positivo;
u negativo e v MAX_VALUE;
u MAX_VALUE e v negativo;

Calcolo del GCD tra numero dispari e MIN_VALUE

u positivo e v MIN_VALUE;

u MIN_VALUE e v positivo;
u negativo e v MIN_VALUE;
u MIN_VALUE e v negativo;

Calcolo del GCD tra numero dispari e MAX_VALUE

u positivo e v MAX_VALUE;

u MAX_VALUE e v positivo;
u negativo e v MAX_VALUE;

u MAX_VALUE e v negativo;

Calcolo del GCD con MIN_VALUE + 1

u MIN_VALUE + 1 v MIN_VALUE + 1;
u MIN_VALUE + 1 v -1;
u MIN_VALUE + 1 v 1;
u MIN_VALUE + 1 v MAX_VALUE - 1;

Calcolo del GCD con MAX_VALUE - 1

u MAX_VALUE - 1 v MIN_VALUE + 1;
u MAX_VALUE - 1 v -1;
u MAX_VALUE - 1 v 1;
u MAX_VALUE - 1 v MAX_VALUE - 1;

Step 6: Automate test cases

- **Fraction:** Automatizziamo i test nel codice esplicitati nello step 5.

```java
public class FractionTest
{
    @Nested
    class FractionConstructorTests
    {
        @ParameterizedTest
        @MethodSource("validFractionProvider")
        void shouldReturnAValidFraction(Fraction fraction, int expectedNumerator, int expectedDenominator) // T1
        {
            assertAll(
                    () -> assertEquals(expectedNumerator, fraction.getNumerator()),
                    () -> assertEquals(expectedDenominator, fraction.getDenominator())
            );
        }

        private static Stream<Arguments> validFractionProvider()
        {
            return Stream.of(
                    Arguments.of(new Fraction(numerator:1, denominator:2), 1, 2), // T1.1
                    Arguments.of(new Fraction(numerator:1, -2), -1, 2), // T1.2
                    Arguments.of(new Fraction(-1, denominator:2), -1, 2), // T1.3
                    Arguments.of(new Fraction(-1, -2), 1, 2) // T1.4
            );
        }

        @Test
        void zeroDenominatorShouldThrowArithmeticException() // T2
        {
            assertThrows(expectedType:ArithmeticException.class, () -> new Fraction(numerator:1, denominator:0));

            // Other cases:
            // first case: numerator is Integer.MAX_VALUE
            assertThrows(expectedType:ArithmeticException.class, () -> new Fraction(Integer.MAX_VALUE, denominator:0));

            // second case: numerator is Integer.MIN_VALUE
            assertThrows(expectedType:ArithmeticException.class, () -> new Fraction(Integer.MIN_VALUE, denominator:0));
        }
```

```java
@Test
void zeroNumeratorShouldReturnZero() // T3
{
    // First case: the denominator is one
    assertEquals(Fraction.ZERO, new Fraction(numerator:0, denominator:1));

    // Second case: the denominator is minus two
    assertEquals(Fraction.ZERO, new Fraction(numerator:0, -2));

    // Other cases:
    // first case: denominator is Integer.MAX_VALUE
    assertEquals(Fraction.ZERO, new Fraction(numerator:0, Integer.MAX_VALUE));

    // second case: denominator is Integer.MIN_VALUE
    assertEquals(Fraction.ZERO, new Fraction(numerator:0, Integer.MIN_VALUE));

    // third case: denominator is Integer.MAX_VALUE - 1
    assertEquals(Fraction.ZERO, new Fraction(numerator:0, Integer.MAX_VALUE - 1));

    // fourth case: denominator is Integer.MIN_VALUE + 1
    assertEquals(Fraction.ZERO, new Fraction(numerator:0, Integer.MIN_VALUE + 1));
}

@Test
void negativeDenominatorShouldThrowArithmeticException() // T4
{
    // First case: the numerator is Integer.MIN_VALUE
    assertThrows(expectedType:ArithmeticException.class, () -> new Fraction(Integer.MIN_VALUE, -1));

    // Second case: the denominator is Integer.MIN_VALUE
    assertThrows(expectedType:ArithmeticException.class, () -> new Fraction(numerator:1, Integer.MIN_VALUE));

    // Other cases: first both numerator and denominator are Integer.MIN_VALUE;
    // second: the numerator is Integer.MAX_VALUE and the denominator is Integer.MIN_VALUE
    assertThrows(expectedType:ArithmeticException.class, () -> new Fraction(Integer.MIN_VALUE, Integer.MIN_VALUE));
    assertThrows(expectedType:ArithmeticException.class, () -> new Fraction(Integer.MAX_VALUE, Integer.MIN_VALUE));
}
```

```java
    // The range values of the numerator is [Integer.MIN_VALUE, Integer.MAX_VALUE],
    // while the range values of the denominator is [Integer.MIN_VALUE, -1] U [1, Integer.MAX_VALUE]
    @ParameterizedTest
    @MethodSource("fractionBoundaryValuesProvider")
    void fractionBoundaryValues(Fraction fraction, int expectedNumerator, int expectedDenominator) // T5
    {
        assertAll(
                () -> assertEquals(expectedNumerator, fraction.getNumerator()),
                () -> assertEquals(expectedDenominator, fraction.getDenominator())
        );
    }

    private static Stream<Arguments> fractionBoundaryValuesProvider()
    {
        return Stream.of(
                // First case: the numerator is Integer.MIN_VALUE
                Arguments.of(new Fraction(Integer.MIN_VALUE, denominator:1), Integer.MIN_VALUE, 1),
                Arguments.of(new Fraction(Integer.MIN_VALUE, Integer.MAX_VALUE), Integer.MIN_VALUE, Integer.MAX_VALUE),

                // Second case: the numerator is Integer.MAX_VALUE
                Arguments.of(new Fraction(Integer.MAX_VALUE, -1), -Integer.MAX_VALUE, 1),
                Arguments.of(new Fraction(Integer.MAX_VALUE, denominator:1), Integer.MAX_VALUE, 1),
                Arguments.of(new Fraction(Integer.MAX_VALUE, Integer.MAX_VALUE), Integer.MAX_VALUE, Integer.MAX_VALUE),

                // Third case: the numerator is Integer.MIN_VALUE + 1
                Arguments.of(new Fraction(Integer.MIN_VALUE + 1, Integer.MIN_VALUE + 1), -(Integer.MIN_VALUE + 1), -(Integer.MIN_VALUE + 1)),
                Arguments.of(new Fraction(Integer.MIN_VALUE + 1, -1), -(Integer.MIN_VALUE + 1), 1),
                Arguments.of(new Fraction(Integer.MIN_VALUE + 1, denominator:1), Integer.MIN_VALUE + 1, 1),
                Arguments.of(new Fraction(Integer.MIN_VALUE + 1, Integer.MAX_VALUE - 1), Integer.MIN_VALUE + 1, Integer.MAX_VALUE - 1),

                // Fourth case: the numerator is Integer.MAX_VALUE - 1
                Arguments.of(new Fraction(Integer.MAX_VALUE - 1, Integer.MIN_VALUE + 1), -(Integer.MAX_VALUE - 1), -(Integer.MIN_VALUE + 1)),
                Arguments.of(new Fraction(Integer.MAX_VALUE - 1, -1), -(Integer.MAX_VALUE - 1), 1),
                Arguments.of(new Fraction(Integer.MAX_VALUE - 1, denominator:1), Integer.MAX_VALUE - 1, 1),
                Arguments.of(new Fraction(Integer.MAX_VALUE - 1, Integer.MAX_VALUE - 1), Integer.MAX_VALUE - 1, Integer.MAX_VALUE - 1)
        );
    }
}
```

- **GetReducedFraction:** Automatizziamo i test nel codice esplicitati nello step 5.

```java
@Nested
class GetReducedFractionTests
{
    @ParameterizedTest
    @MethodSource("reducedFractionProvider")
    void shouldReturnAValidReducedFraction(Fraction expectedFraction, int numerator, int denominator) // T6
    {
        assertEquals(expectedFraction, Fraction.getReducedFraction(numerator, denominator));
    }

    private static Stream<Arguments> reducedFractionProvider()
    {
        return Stream.of(
                // T6.1, 2/4 should return 1/2
                Arguments.of(new Fraction(numerator:1, denominator:2), 2, 4),

                // T6.2, 2/-6 should return -1/3
                Arguments.of(new Fraction(-1, denominator:3), 2, -6),

                // T6.3, -4/6 should return -2/3
                Arguments.of(new Fraction(-2, denominator:3), -4, 6),

                // T6.4, -6/-4 should return 3/2
                Arguments.of(new Fraction(numerator:3, denominator:2), -6, -4),

                // T6.5, 2/2 should return 1/1 or 1
                Arguments.of(new Fraction(numerator:1, denominator:1), 2, 2),
                Arguments.of(new Fraction(-1, denominator:1), -2, 2),
                Arguments.of(new Fraction(-1, denominator:1), 2, -2),
                Arguments.of(new Fraction(numerator:1, denominator:1), -2, -2),

                // T6.6, odd numerator and odd denominator
                // This test also verifies the method behavior with prime numbers.
                // 1 and 3 are indeed the first two prime numbers
                Arguments.of(new Fraction(numerator:1, denominator:3), 1, 3),
                Arguments.of(new Fraction(-1, denominator:3), 1, -3),
                Arguments.of(new Fraction(-1, denominator:3), -1, 3),
                Arguments.of(new Fraction(numerator:1, denominator:3), -1, -3),

                // T6.7, odd numerator and even denominator
                Arguments.of(new Fraction(numerator:3, denominator:4), 3, 4),
                Arguments.of(new Fraction(-3, denominator:4), 3, -4),
                Arguments.of(new Fraction(-3, denominator:4), -3, 4),
                Arguments.of(new Fraction(numerator:3, denominator:4), -3, -4),
```

```
            // T6.8, even numerator and odd denominator
            Arguments.of(new Fraction(numerator:2, denominator:3), 2, 3),
            Arguments.of(new Fraction(-2, denominator:3), 2, -3),
            Arguments.of(new Fraction(-2, denominator:3), -2, 3),
            Arguments.of(new Fraction(numerator:2, denominator:3), -2, -3)

    );
}

@Test
void zeroDenominatorShouldThrowArithmeticException() // T7
{
    assertThrows(expectedType:ArithmeticException.class, () -> Fraction.getReducedFraction(numerator:1, denominator:0));

    // Other cases:
    // first case: numerator is Integer.MAX_VALUE
    assertThrows(expectedType:ArithmeticException.class, () -> Fraction.getReducedFraction(Integer.MAX_VALUE, denominator:0));

    // second case: numerator is Integer.MIN_VALUE
    assertThrows(expectedType:ArithmeticException.class, () -> Fraction.getReducedFraction(Integer.MIN_VALUE, denominator:0));
}

@Test
void zeroNumeratorShouldReturnZEROConstant() // T8
{
    assertEquals(Fraction.ZERO, Fraction.getReducedFraction(numerator:0, denominator:1));

    // Other cases:
    // first case: denominator is Integer.MAX_VALUE
    assertEquals(Fraction.ZERO, Fraction.getReducedFraction(numerator:0, Integer.MAX_VALUE));

    // second case: denominator is Integer.MIN_VALUE
    assertEquals(Fraction.ZERO, Fraction.getReducedFraction(numerator:0, Integer.MIN_VALUE));

    // third case: denominator is Integer.MAX_VALUE - 1
    assertEquals(Fraction.ZERO, Fraction.getReducedFraction(numerator:0, Integer.MAX_VALUE - 1));

    // fourth case: denominator is Integer.MIN_VALUE + 1
    assertEquals(Fraction.ZERO, Fraction.getReducedFraction(numerator:0, Integer.MIN_VALUE + 1));
}
```

```
@ParameterizedTest
@MethodSource("validEvenReducedFractionProvider")
void validEvenNumeratorMIN_VALUE_DenominatorReducedFraction(int numerator, int denominator) // T9
{
    assertEquals(new Fraction(numerator / 2, denominator / 2), Fraction.getReducedFraction(numerator, denominator));
}

static Stream<Arguments> validEvenReducedFractionProvider()
{
    return Stream.of(
            // First case: the numerator is a small even number, positive and negative
            Arguments.of(...arguments:2, Integer.MIN_VALUE), // T9.1
            Arguments.of(-2, Integer.MIN_VALUE) ,// T9.2

            // Second case: the numerator is a large even number
            Arguments.of(Integer.MAX_VALUE - 1, Integer.MIN_VALUE), // T9.3

            // Third case: the numerator is a very small number
            Arguments.of(Integer.MIN_VALUE + 2, Integer.MIN_VALUE) // T9.4
    );
}

@Test
void negativeDenominatorShouldThrowArithmeticException() // T10, same test as T4
{
    // First case: the numerator is Integer.MIN_VALUE
    assertThrows(expectedType:ArithmeticException.class, () -> Fraction.getReducedFraction(Integer.MIN_VALUE, -1));

    // Second case: the denominator is Integer.MIN_VALUE
    assertThrows(expectedType:ArithmeticException.class, () -> Fraction.getReducedFraction(numerator:1, Integer.MIN_VALUE));

    // Other case
    assertThrows(expectedType:ArithmeticException.class, () -> Fraction.getReducedFraction(Integer.MAX_VALUE, Integer.MIN_VALUE));
}
```

```
    // The range values of the numerator is [Integer.MIN_VALUE, Integer.MAX_VALUE],
    // while the range values of the denominator is [Integer.MIN_VALUE, -1] U [1, Integer.MAX_VALUE]
    @Test
    void reducedFractionBoundaryValues() // T11
    {
        assertAll(
        // First case: the numerator is Integer.MIN_VALUE
        () -> assertEquals(new Fraction(Integer.MIN_VALUE, denominator:1), Fraction.getReducedFraction(Integer.MIN_VALUE, denominator:1)),
        () -> assertEquals(new Fraction(Integer.MIN_VALUE, Integer.MAX_VALUE), Fraction.getReducedFraction(Integer.MIN_VALUE, Integer.MAX_VALUE)),

        // Second case: the numerator is Integer.MAX_VALUE
        () -> assertEquals(new Fraction(-Integer.MAX_VALUE, denominator:1), Fraction.getReducedFraction(Integer.MAX_VALUE, -1)),
        () -> assertEquals(new Fraction(Integer.MAX_VALUE, denominator:1), Fraction.getReducedFraction(Integer.MAX_VALUE, denominator:1)),
        () -> assertEquals(new Fraction(numerator:1, denominator:1), Fraction.getReducedFraction(Integer.MAX_VALUE, Integer.MAX_VALUE)),

        // Third case: the numerator is Integer.MIN_VALUE + 1
        () -> assertEquals(new Fraction(numerator:1, denominator:1), Fraction.getReducedFraction(Integer.MIN_VALUE + 1, Integer.MIN_VALUE + 1)),
        () -> assertEquals(new Fraction(-(Integer.MIN_VALUE + 1), denominator:1), Fraction.getReducedFraction(Integer.MIN_VALUE + 1, -1)),
        () -> assertEquals(new Fraction(Integer.MIN_VALUE + 1, denominator:1), Fraction.getReducedFraction(Integer.MIN_VALUE + 1, denominator:1)),
        () -> assertEquals(new Fraction(Integer.MIN_VALUE + 1, Integer.MAX_VALUE - 1), Fraction.getReducedFraction(Integer.MIN_VALUE + 1, Integer.MAX_VALUE - 1)),

        // Fourth case: the numerator is Integer.MAX_VALUE - 1
        () -> assertEquals(new Fraction(Integer.MAX_VALUE - 1, Integer.MIN_VALUE + 1), Fraction.getReducedFraction(Integer.MAX_VALUE - 1, Integer.MIN_VALUE + 1)),
        () -> assertEquals(new Fraction(-(Integer.MAX_VALUE - 1), denominator:1), Fraction.getReducedFraction(Integer.MAX_VALUE - 1, -1)),
        () -> assertEquals(new Fraction(Integer.MAX_VALUE - 1, denominator:1), Fraction.getReducedFraction(Integer.MAX_VALUE - 1, denominator:1)),
        () -> assertEquals(new Fraction(numerator:1, denominator:1), Fraction.getReducedFraction(Integer.MAX_VALUE - 1, Integer.MAX_VALUE - 1))
        );
    }
}
```

- **GreatestCommonDivisor:** Automatizziamo i test nel codice esplicitati nello step 5.

```
@Nested
class GreatestCommonDivisorTests
{
    @Test
    void shouldReturnCorrectGCD() // T12
    {
        // Classic GCD calculation
        assertEquals(expected:4, Fraction.greatestCommonDivisor(u:4, v:8));
        assertEquals(expected:4, Fraction.greatestCommonDivisor(-4, v:8));
        assertEquals(expected:4, Fraction.greatestCommonDivisor(u:8, -4));
        assertEquals(expected:4, Fraction.greatestCommonDivisor(-8, -4));

        // The GCD of two prime numbers that are not the same is 1
        assertEquals(expected:1, Fraction.greatestCommonDivisor(u:3, v:5));
        assertEquals(expected:1, Fraction.greatestCommonDivisor(-3, v:5));
        assertEquals(expected:1, Fraction.greatestCommonDivisor(u:3, -5));
        assertEquals(expected:1, Fraction.greatestCommonDivisor(-3, -5));

        // GCD calculation between an even number and an odd number
        // First case: u is even and v is odd
        assertEquals(expected:1, Fraction.greatestCommonDivisor(u:2, v:3));
        assertEquals(expected:1, Fraction.greatestCommonDivisor(-2, v:3));
        assertEquals(expected:1, Fraction.greatestCommonDivisor(u:2, -3));
        assertEquals(expected:1, Fraction.greatestCommonDivisor(-2, -3));

        // Second case: u is odd and v is even
        assertEquals(expected:1, Fraction.greatestCommonDivisor(u:3, v:4));
        assertEquals(expected:1, Fraction.greatestCommonDivisor(-3, v:4));
        assertEquals(expected:1, Fraction.greatestCommonDivisor(u:3, -4));
        assertEquals(expected:1, Fraction.greatestCommonDivisor(-3, -4));
    }

    @Test
    void zeroOperandsShouldThrowArithmeticException() // T13
    {
        assertThrows(expectedType:ArithmeticException.class, () -> Fraction.greatestCommonDivisor(u:0, v:0));
    }
```

```java
@Test
void gcdBetweenZeroAndIntNumber() // T14
{
    assertAll(
        // First case: u is zero and v is an int number
        () -> assertEquals(expected:2, Fraction.greatestCommonDivisor(u:0, v:2)), // T14.1
        () -> assertEquals(expected:2, Fraction.greatestCommonDivisor(u:0, -2)), // T14.2

        // Second case: u is an int number and v is zero
        () -> assertEquals(expected:2, Fraction.greatestCommonDivisor(u:2, v:0)), // T14.3
        () -> assertEquals(expected:2, Fraction.greatestCommonDivisor(-2, v:0)) // T14.4
    );
}

@Test
void testGCDOverflow() // T15
{
    // First case: u is zero and v is Integer.MIN_VALUE
    assertThrows(expectedType:ArithmeticException.class, () -> {
        Fraction.greatestCommonDivisor(u:0, Integer.MIN_VALUE);
    });

    // Second case: u is Integer.MIN_VALUE and v is zero
    assertThrows(expectedType:ArithmeticException.class, () -> {
        Fraction.greatestCommonDivisor(Integer.MIN_VALUE, v:0);
    });

    // Third case: both u and v are Integer.MIN_VALUE.
    // This triggers the condition at line 171
    assertThrows(expectedType:ArithmeticException.class, () -> {
        Fraction.greatestCommonDivisor(Integer.MIN_VALUE, Integer.MIN_VALUE);
    });
}
```

```java
@Test
void gdcBetweenOneAndIntNumber() // T16
{
    assertAll(
        // First case: u is one and v is an int number
        () -> assertEquals(expected:1, Fraction.greatestCommonDivisor(u:1, v:2)), // T16.1
        () -> assertEquals(expected:1, Fraction.greatestCommonDivisor(-1, v:2)), // T16.2
        () -> assertEquals(expected:1, Fraction.greatestCommonDivisor(-1, -2)), // T16.3

        // Second case: u is an int number and v is one
        () -> assertEquals(expected:1, Fraction.greatestCommonDivisor(u:2, v:1)), // T16.4
        () -> assertEquals(expected:1, Fraction.greatestCommonDivisor(-2, v:1)), // T16.5
        () -> assertEquals(expected:1, Fraction.greatestCommonDivisor(-2, -1)), // T16.6

        // Third case: both u and v are one
        () -> assertEquals(expected:1, Fraction.greatestCommonDivisor(u:1, v:1)), // T16.7
        () -> assertEquals(expected:1, Fraction.greatestCommonDivisor(-1, v:1)), // T16.8
        () -> assertEquals(expected:1, Fraction.greatestCommonDivisor(-1, -1)) // T16.9
    );
}
```

```java
@ParameterizedTest
@MethodSource("gcdBoundaryValuesProvider")
void testGCDBoundaryValues(int expectedGCD, int u, int v) // T17
{
    assertEquals(expectedGCD, Fraction.greatestCommonDivisor(u, v));
}

static Stream<Arguments> gcdBoundaryValuesProvider()
{
    return Stream.of(
            // GCD calculation between one and Integer.MIN_VALUE
            Arguments.of(...arguments:1, 1, Integer.MIN_VALUE),
            Arguments.of(...arguments:1, Integer.MIN_VALUE, 1),
            Arguments.of(...arguments:1, -1, Integer.MIN_VALUE),
            Arguments.of(...arguments:1, Integer.MIN_VALUE, -1),

            // GCD calculation between one and Integer.MAX_VALUE
            Arguments.of(...arguments:1, 1, Integer.MAX_VALUE),
            Arguments.of(...arguments:1, Integer.MAX_VALUE, 1),
            Arguments.of(...arguments:1, -1, Integer.MAX_VALUE),
            Arguments.of(...arguments:1, Integer.MAX_VALUE, -1),

            // GCD calculation between Integer.MIN_VALUE and Integer.MAX_VALUE
            Arguments.of(...arguments:1, Integer.MIN_VALUE, Integer.MAX_VALUE),
            Arguments.of(...arguments:1, Integer.MAX_VALUE, Integer.MIN_VALUE),
            Arguments.of(Integer.MAX_VALUE, Integer.MAX_VALUE, Integer.MAX_VALUE),

            // Test GCD with an even number and Integer.MIN_VALUE
            Arguments.of(...arguments:2, 2, Integer.MIN_VALUE),
            Arguments.of(...arguments:2, Integer.MIN_VALUE, 2),
            Arguments.of(...arguments:2, -2, Integer.MIN_VALUE),
            Arguments.of(...arguments:2, Integer.MIN_VALUE, -2),

            // Test GCD with an even number and Integer.MAX_VALUE
            Arguments.of(...arguments:1, 2, Integer.MAX_VALUE),
            Arguments.of(...arguments:1, Integer.MAX_VALUE, 2),
            Arguments.of(...arguments:1, -2, Integer.MAX_VALUE),
            Arguments.of(...arguments:1, Integer.MAX_VALUE, -2),

            // Test GCD with an odd number and Integer.MIN_VALUE
            Arguments.of(...arguments:1, 3, Integer.MIN_VALUE),
            Arguments.of(...arguments:1, Integer.MIN_VALUE, 3),
            Arguments.of(...arguments:1, -3, Integer.MIN_VALUE),
            Arguments.of(...arguments:1, Integer.MIN_VALUE, -3),
```

Step 7: Augment the test suite with creativity and experience

- **Fraction:** Abbiamo lavorato sul seguente metodo in maniera tale da testare più situazioni:

    - Nel **T2**, abbiamo verificato che venga lanciata un'eccezione quando si assegna il valore 0 al denominatore;
    - Nel **T3**, abbiamo assegnato il valore 0 al numeratore per confermare che venga restituita la costante Fraction.ZERO;
    - Nel **T4**, in particolare nell'ultimo caso, abbiamo testato una combinazione aggiuntiva di valori per i due parametri selezionati: abbiamo combinato il valore minimo e il valore massimo per generare nuovamente un'eccezione;

```java
public class FractionTest
{
    @Nested
    class FractionConstructorTests
    {
        @ParameterizedTest
        @MethodSource("validFractionProvider")
        void shouldReturnAValidFraction(Fraction fraction, int expectedNumerator, int expectedDenominator) // T1
        {
            assertAll(
                    () -> assertEquals(expectedNumerator, fraction.getNumerator()),
                    () -> assertEquals(expectedDenominator, fraction.getDenominator())
            );
        }

        private static Stream<Arguments> validFractionProvider()
        {
            return Stream.of(
                    Arguments.of(new Fraction(numerator:1, denominator:2), 1, 2), // T1.1
                    Arguments.of(new Fraction(numerator:1, -2), -1, 2), // T1.2
                    Arguments.of(new Fraction(-1, denominator:2), -1, 2), // T1.3
                    Arguments.of(new Fraction(-1, -2), 1, 2) // T1.4
            );
        }

        @Test
        void zeroDenominatorShouldThrowArithmeticException() // T2
        {
            assertThrows(expectedType:ArithmeticException.class, () -> new Fraction(numerator:1, denominator:0));

            // Other cases:
            // first case: numerator is Integer.MAX_VALUE
            assertThrows(expectedType:ArithmeticException.class, () -> new Fraction(Integer.MAX_VALUE, denominator:0));

            // second case: numerator is Integer.MIN_VALUE
            assertThrows(expectedType:ArithmeticException.class, () -> new Fraction(Integer.MIN_VALUE, denominator:0));
        }
```

```java
@Test
void zeroNumeratorShouldReturnZero() // T3
{
    // First case: the denominator is one
    assertEquals(Fraction.ZERO, new Fraction(numerator:0, denominator:1));

    // Second case: the denominator is minus two
    assertEquals(Fraction.ZERO, new Fraction(numerator:0, -2));

    // Other cases:
    // first case: denominator is Integer.MAX_VALUE
    assertEquals(Fraction.ZERO, new Fraction(numerator:0, Integer.MAX_VALUE));

    // second case: denominator is Integer.MIN_VALUE
    assertEquals(Fraction.ZERO, new Fraction(numerator:0, Integer.MIN_VALUE));

    // third case: denominator is Integer.MAX_VALUE - 1
    assertEquals(Fraction.ZERO, new Fraction(numerator:0, Integer.MAX_VALUE - 1));

    // fourth case: denominator is Integer.MIN_VALUE + 1
    assertEquals(Fraction.ZERO, new Fraction(numerator:0, Integer.MIN_VALUE + 1));
}

@Test
void negativeDenominatorShouldThrowArithmeticException() // T4
{
    // First case: the numerator is Integer.MIN_VALUE
    assertThrows(expectedType:ArithmeticException.class, () -> new Fraction(Integer.MIN_VALUE, -1));

    // Second case: the denominator is Integer.MIN_VALUE
    assertThrows(expectedType:ArithmeticException.class, () -> new Fraction(numerator:1, Integer.MIN_VALUE));

    // Other cases: first both numerator and denominator are Integer.MIN_VALUE;
    // second: the numerator is Integer.MAX_VALUE and the denominator is Integer.MIN_VALUE
    assertThrows(expectedType:ArithmeticException.class, () -> new Fraction(Integer.MIN_VALUE, Integer.MIN_VALUE));
    assertThrows(expectedType:ArithmeticException.class, () -> new Fraction(Integer.MAX_VALUE, Integer.MIN_VALUE));
}
```

- **GetReducedFraction:** Abbiamo lavorato su questo metodo per verificare nel **T10**, in particolare nella sua ultima condizione, un'altra combinazione per i nostri due parametri. Abbiamo assegnato loro i valori MIN VALUE e MAX VALUE, in modo da poter generare una nuova eccezione.
  Abbiamo verificato l'efficacia del nostro metodo testandolo su valori pari, dispari e numeri primi, per assicurarci che le frazioni fossero ridotte correttamente.

```java
@Nested
class GetReducedFractionTests
{
    @ParameterizedTest
    @MethodSource("reducedFractionProvider")
    void shouldReturnAValidReducedFraction(Fraction expectedFraction, int numerator, int denominator) // T6
    {
        assertEquals(expectedFraction, Fraction.getReducedFraction(numerator, denominator));
    }

    private static Stream<Arguments> reducedFractionProvider()
    {
        return Stream.of(
                // T6.1, 2/4 should return 1/2
                Arguments.of(new Fraction(numerator:1, denominator:2), 2, 4),

                // T6.2, 2/-6 should return -1/3
                Arguments.of(new Fraction(-1, denominator:3), 2, -6),

                // T6.3, -4/6 should return -2/3
                Arguments.of(new Fraction(-2, denominator:3), -4, 6),

                // T6.4, -6/-4 should return 3/2
                Arguments.of(new Fraction(numerator:3, denominator:2), -6, -4),

                // T6.5, 2/2 should return 1/1 or 1
                Arguments.of(new Fraction(numerator:1, denominator:1), 2, 2),
                Arguments.of(new Fraction(-1, denominator:1), -2, 2),
                Arguments.of(new Fraction(-1, denominator:1), 2, -2),
                Arguments.of(new Fraction(numerator:1, denominator:1), -2, -2),

                // T6.6, odd numerator and odd denominator
                // This test also verifies the method behavior with prime numbers.
                // 1 and 3 are indeed the first two prime numbers
                Arguments.of(new Fraction(numerator:1, denominator:3), 1, 3),
                Arguments.of(new Fraction(-1, denominator:3), 1, -3),
                Arguments.of(new Fraction(-1, denominator:3), -1, 3),
                Arguments.of(new Fraction(numerator:1, denominator:3), -1, -3),

                // T6.7, odd numerator and even denominator
                Arguments.of(new Fraction(numerator:3, denominator:4), 3, 4),
                Arguments.of(new Fraction(-3, denominator:4), 3, -4),
                Arguments.of(new Fraction(-3, denominator:4), -3, 4),
                Arguments.of(new Fraction(numerator:3, denominator:4), -3, -4),
```

```java
    // The range values of the numerator is [Integer.MIN_VALUE, Integer.MAX_VALUE],
    // while the range values of the denominator is [Integer.MIN_VALUE, -1] U [1, Integer.MAX_VALUE]
    @Test
    void reducedFractionBoundaryValues() // T11
    {
        assertAll(
        // First case: the numerator is Integer.MIN_VALUE
        () -> assertEquals(new Fraction(Integer.MIN_VALUE, denominator:1), Fraction.getReducedFraction(Integer.MIN_VALUE, denominator:1)),
        () -> assertEquals(new Fraction(Integer.MIN_VALUE, Integer.MAX_VALUE), Fraction.getReducedFraction(Integer.MIN_VALUE, Integer.MAX_VALUE)),

        // Second case: the numerator is Integer.MAX_VALUE
        () -> assertEquals(new Fraction(-Integer.MAX_VALUE, denominator:1), Fraction.getReducedFraction(Integer.MAX_VALUE, -1)),
        () -> assertEquals(new Fraction(Integer.MAX_VALUE, denominator:1), Fraction.getReducedFraction(Integer.MAX_VALUE, denominator:1)),
        () -> assertEquals(new Fraction(numerator:1, denominator:1), Fraction.getReducedFraction(Integer.MAX_VALUE, Integer.MAX_VALUE)),

        // Third case: the numerator is Integer.MIN_VALUE + 1
        () -> assertEquals(new Fraction(numerator:1, denominator:1), Fraction.getReducedFraction(Integer.MIN_VALUE + 1, Integer.MIN_VALUE + 1)),
        () -> assertEquals(new Fraction(-(Integer.MIN_VALUE + 1), denominator:1), Fraction.getReducedFraction(Integer.MIN_VALUE + 1, -1)),
        () -> assertEquals(new Fraction(Integer.MIN_VALUE + 1, denominator:1), Fraction.getReducedFraction(Integer.MIN_VALUE + 1, denominator:1)),
        () -> assertEquals(new Fraction(Integer.MIN_VALUE + 1, Integer.MAX_VALUE - 1), Fraction.getReducedFraction(Integer.MIN_VALUE + 1, Integer.MAX_VALUE - 1)),

        // Fourth case: the numerator is Integer.MAX_VALUE - 1
        () -> assertEquals(new Fraction(Integer.MAX_VALUE - 1, Integer.MIN_VALUE + 1), Fraction.getReducedFraction(Integer.MAX_VALUE - 1, Integer.MIN_VALUE + 1)),
        () -> assertEquals(new Fraction(-(Integer.MAX_VALUE - 1), denominator:1), Fraction.getReducedFraction(Integer.MAX_VALUE - 1, -1)),
        () -> assertEquals(new Fraction(Integer.MAX_VALUE - 1, denominator:1), Fraction.getReducedFraction(Integer.MAX_VALUE - 1, denominator:1)),
        () -> assertEquals(new Fraction(numerator:1, denominator:1), Fraction.getReducedFraction(Integer.MAX_VALUE - 1, Integer.MAX_VALUE - 1))
        );
    }
}
```

- **GreatestCommonDivisor:** Abbiamo lavorato sul seguente metodo in maniera tale da testare più situazioni:

    - Nel **T13**, dopo aver modificato la condizione originale del metodo, abbiamo verificato la possibilità che entrambi i parametri u e v siano uguali a 0, generando di conseguenza un'eccezione;
    - Nel **T15**, abbiamo sperimentato diverse combinazioni per i due parametri u e v, assegnando valori come 0 e MIN VALUE, per assicurare che venga nuovamente generata un'eccezione;

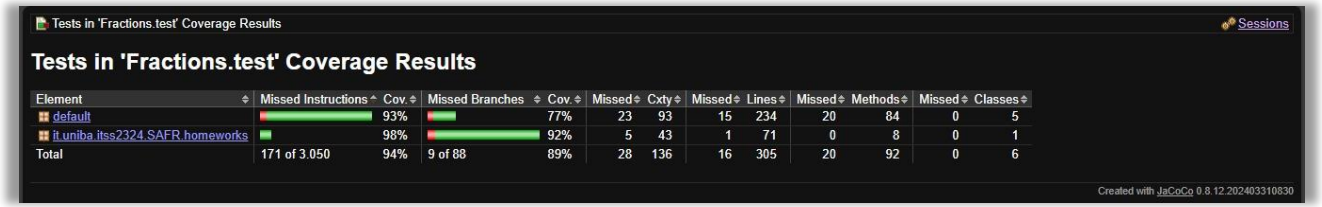
```java
@Test
void gcdBetweenZeroAndIntNumber() // T14
{
    assertAll(
        // First case: u is zero and v is an int number
        () -> assertEquals(expected:2, Fraction.greatestCommonDivisor(u:0, v:2)), // T14.1
        () -> assertEquals(expected:2, Fraction.greatestCommonDivisor(u:0, -2)), // T14.2

        // Second case: u is an int number and v is zero
        () -> assertEquals(expected:2, Fraction.greatestCommonDivisor(u:2, v:0)), // T14.3
        () -> assertEquals(expected:2, Fraction.greatestCommonDivisor(-2, v:0)) // T14.4
    );
}

@Test
void testGCDOverflow() // T15
{
    // First case: u is zero and v is Integer.MIN_VALUE
    assertThrows(expectedType:ArithmeticException.class, () -> {
        Fraction.greatestCommonDivisor(u:0, Integer.MIN_VALUE);
    });

    // Second case: u is Integer.MIN_VALUE and v is zero
    assertThrows(expectedType:ArithmeticException.class, () -> {
        Fraction.greatestCommonDivisor(Integer.MIN_VALUE, v:0);
    });

    // Third case: both u and v are Integer.MIN_VALUE.
    // This triggers the condition at line 171
    assertThrows(expectedType:ArithmeticException.class, () -> {
        Fraction.greatestCommonDivisor(Integer.MIN_VALUE, Integer.MIN_VALUE);
    });
}
```

Structural testing

Dopo aver completato il testing basato sulle specifiche, passiamo al testing strutturale. Abbiamo esaminato l'implementazione del codice ed eseguito la suite di test utilizzando uno strumento di code coverage, per identificare quali linee di codice non erano coperte dai test. Abbiamo utilizzato il **Condition + branch**, che considera non solo i possibili rami, ma anche ogni condizione di ciascuna istruzione di ramo. La suite di test dovrebbe verificare:

- che ciascuna di queste condizioni venga valutata almeno una volta come vera e come falsa
- che l'intera istruzione di ramo sia vera e falsa almeno una volta.

Di seguito riportiamo il risultato mostrato dal tool di coverage JaCoCo.


Tests in 'Fractions.test' Coverage Results

Sessions

Tests in 'Fractions.test' Coverage Results

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
default		93%		77%	23	93	15	234	20	84	0	5
it.uniba.itss2324.SAFR.homeworks		98%		92%	5	43	1	71	0	8	0	1
Total	171 of 3.050	94%	9 of 88	89%	28	136	16	305	20	92	0	6

Created with JaCoCo 0.8.12.202403310830

- **Fraction:** Grazie al lavoro di testing svolto, tutte le linee di codice di questo metodo sono state coperte.

- **GetReducedFraction:** Grazie al lavoro di testing svolto, tutte le linee di codice di questo metodo sono state coperte.

- **GreatestCommonDivisor:** Per quanto riguarda questo metodo, non siamo riusciti a coprire completamente una singola riga di codice (riga 182). Anche se abbiamo fornito in input valori molto grandi, la copertura della riga risulta parziale.

3 Capitolo 2 (HW2)

Introduzione

L'obiettivo del nostro homework 2 è effettuare il property-based testing su un metodo della classe Fraction di Apache Commons Lang.

Il property-based testing è una tecnica che permette di generare automaticamente una vasta gamma di input casuali, ma in un intervallo specifico, per verificare che le proprietà specificate del software siano sempre soddisfatte.

Il metodo che abbiamo scelto per questo testing è getReducedFraction.

Property-Based Testing

Abbiamo quindi individuato per il nostro metodo, le proprietà principali che deve soddisfare:

- **PBT1**: Andiamo a testare l'invalidità del denominatore nel momento in cui questo risulta essere uguale a 0;
- **PBT2**: Andiamo a testare la validità della frazione quando numeratore e denominatore sono diversi da 0;
- **PBT3**: Andiamo a testare l'invalidità della frazione nel momento in cui al numeratore viene assegnato un valore dispari e il denominatore risulta essere uguale a MIN VALUE;
- **PBT4**: Andiamo a testare l'invalidità della frazione nel momento in cui il numeratore risulta essere uguale a MIN VALUE e al denominatore viene assegnato un valore negativo dispari;
- **PBT5**: Andiamo a testare la validità della frazione in più situazioni:
    1. Quando numeratore è diverso da 0 e non è uguale a MIN VALUE;
    2. Quando denominatore è diverso da 0 e non è uguale a MIN VALUE;

- ## PBT1:

Nel primo test, verifichiamo una condizione essenziale per le frazioni: la validità del denominatore. In particolare, controlliamo che il denominatore sia considerato invalido quando gli viene assegnato il valore 0.

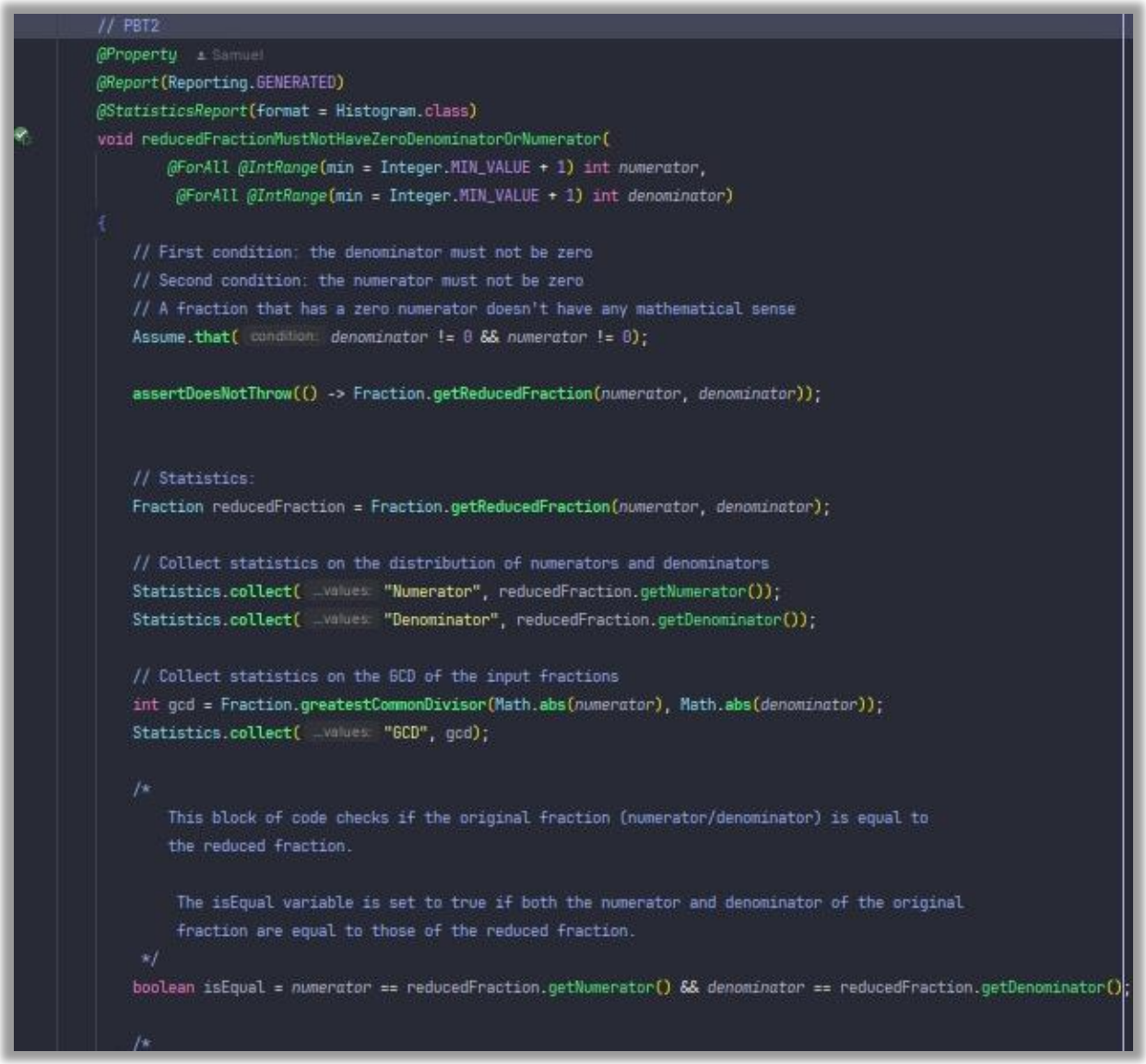
```java
// PBT1
@Property  ± Samuel
@Report(Reporting.GENERATED)
void testInvalidFractionZeroDenominator(
        @ForAll @IntRange(min = Integer.MIN_VALUE) int numerator,
         @ForAll @IntRange(max = 0) int denominator)
{
    // If the denominator is zero throw new arithmetic exception
    assertThrows(ArithmeticException.class, () -> Fraction.getReducedFraction(numerator, denominator));

    //Statistics:
    try {
        Fraction.getReducedFraction(numerator, denominator);
        Statistics.collect( ...values: "Exception Cases", "No Exception");
    } catch (ArithmeticException e) {
        Statistics.collect( ...values: "Exception Cases", "ArithmeticException");
    }
}
```

Per ogni test effettuato, abbiamo deciso di raccogliere delle statistiche per confermare e visualizzare su schermo i dati attesi. In questo caso, le statistiche mostrano che la condizione da noi verificata è sempre risultata vera (100%).

```
timestamp = 2024-05-23T12:28:48.059313300, [FractionPropertyBasedTest:testInvalidFractionZeroDenominator] (1000) statistics =
    Exception Cases ArithmeticException (1000) : 100 %

timestamp = 2024-05-23T12:28:48.076313900, FractionPropertyBasedTest:testInvalidFractionZeroDenominator =
                          |----------------------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 = 9      | # of all combined edge cases
edge-cases#tried = 9      | # of edge cases tried in current run
seed = -8124227158389866065 | random seed to reproduce generated values
```

- **PBT2:**

Nel secondo test, verifichiamo che, quando sia il denominatore sia il numeratore sono diversi da 0, la frazione risulti sempre valida, ovvero venga sempre ridotta ai minimi termini.



```
// PBT2
@Property  ± Samuel
@Report(Reporting.GENERATED)
@StatisticsReport(format = Histogram.class)
void reducedFractionMustNotHaveZeroDenominatorOrNumerator(
        @ForAll @IntRange(min = Integer.MIN_VALUE + 1) int numerator,
        @ForAll @IntRange(min = Integer.MIN_VALUE + 1) int denominator)
{
    // First condition: the denominator must not be zero
    // Second condition: the numerator must not be zero
    // A fraction that has a zero numerator doesn't have any mathematical sense
    Assume.that( condition: denominator != 0 && numerator != 0);

    assertDoesNotThrow(() -> Fraction.getReducedFraction(numerator, denominator));


    // Statistics:
    Fraction reducedFraction = Fraction.getReducedFraction(numerator, denominator);

    // Collect statistics on the distribution of numerators and denominators
    Statistics.collect( ...values: "Numerator", reducedFraction.getNumerator());
    Statistics.collect( ...values: "Denominator", reducedFraction.getDenominator());

    // Collect statistics on the GCD of the input fractions
    int gcd = Fraction.greatestCommonDivisor(Math.abs(numerator), Math.abs(denominator));
    Statistics.collect( ...values: "GCD", gcd);

    /*
        This block of code checks if the original fraction (numerator/denominator) is equal to
        the reduced fraction.

        The isEqual variable is set to true if both the numerator and denominator of the original
        fraction are equal to those of the reduced fraction.
     */
    boolean isEqual = numerator == reducedFraction.getNumerator() && denominator == reducedFraction.getDenominator();

    /*
```

Anche in questo caso, vogliamo analizzare le statistiche. Possiamo osservare che per 968 dei casi esaminati nel test, la nostra condizione risulta essere sempre valida.

In aggiunta analizziamo la distribuzione dei valori del numeratore e del denominatore, i valori del MCD (Massimo Comun Divisore) e la probabilità che la frazione iniziale non venga ridotta (ossia quando il MCD è uguale a 1).

Abbiamo osservato che in 662 casi la frazione è stata ridotta ai minimi termini, mentre in 306 casi il metodo non è riuscito a ridurre la frazione poiché i termini avevano un MCD pari a 1.

684	Fraction Equality false	662	##
685	Fraction Equality true	306	################################
686	GCD 1	618	##
687	GCD 2	146	################
688	GCD 3	40	####
689	GCD 4	33	###
690	GCD 5	23	##
691	GCD 7	16	#
692	GCD 6	14	#
693	GCD 9	9	#
694	GCD 8	9	#
695	GCD 11	6	
696	GCD 16	5	
697	GCD 10	4	
698	GCD 22	4	
699	GCD 15	4	
700	GCD 2147483647	4	
701	GCD 2147483646	4	
702	GCD 18	3	
703	GCD 12	2	
704	GCD 13	2	
705	GCD 17	2	
706	GCD 27	2	
707	GCD 23	1	
708	GCD 19	1	
709	GCD 662	1	
710	GCD 72	1	
711	GCD 14	1	
712	GCD 107	1	
713	GCD 268	1	
714	GCD 45	1	
715	GCD 38	1	
716	GCD 211	1	
717	GCD 20	1	
718	GCD 36	1	
719	GCD 53	1	
720	GCD 43	1	
721	GCD 24	1	

```
timestamp = 2024-05-23T12:28:50.427317100, FractionPropertyBasedTest:reducedFractionMustNotHaveZeroDenominatorOrNumerator =
                            |----------------------jqwik----------------------
tries = 1000                | # of calls to property
checks = 968                | # 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 = 81       | # of all combined edge cases
edge-cases#tried = 81       | # of edge cases tried in current run
seed = 7314910000840193824  | random seed to reproduce generated values
```

- **PBT3:**

Nel terzo test, verifichiamo l'invalidità della frazione quando il numeratore è un numero dispari e il denominatore è il valore minimo possibile (MIN VALUE).

In questo caso, ci aspettiamo che venga lanciata un'eccezione, poiché la frazione non sarà mai valida.

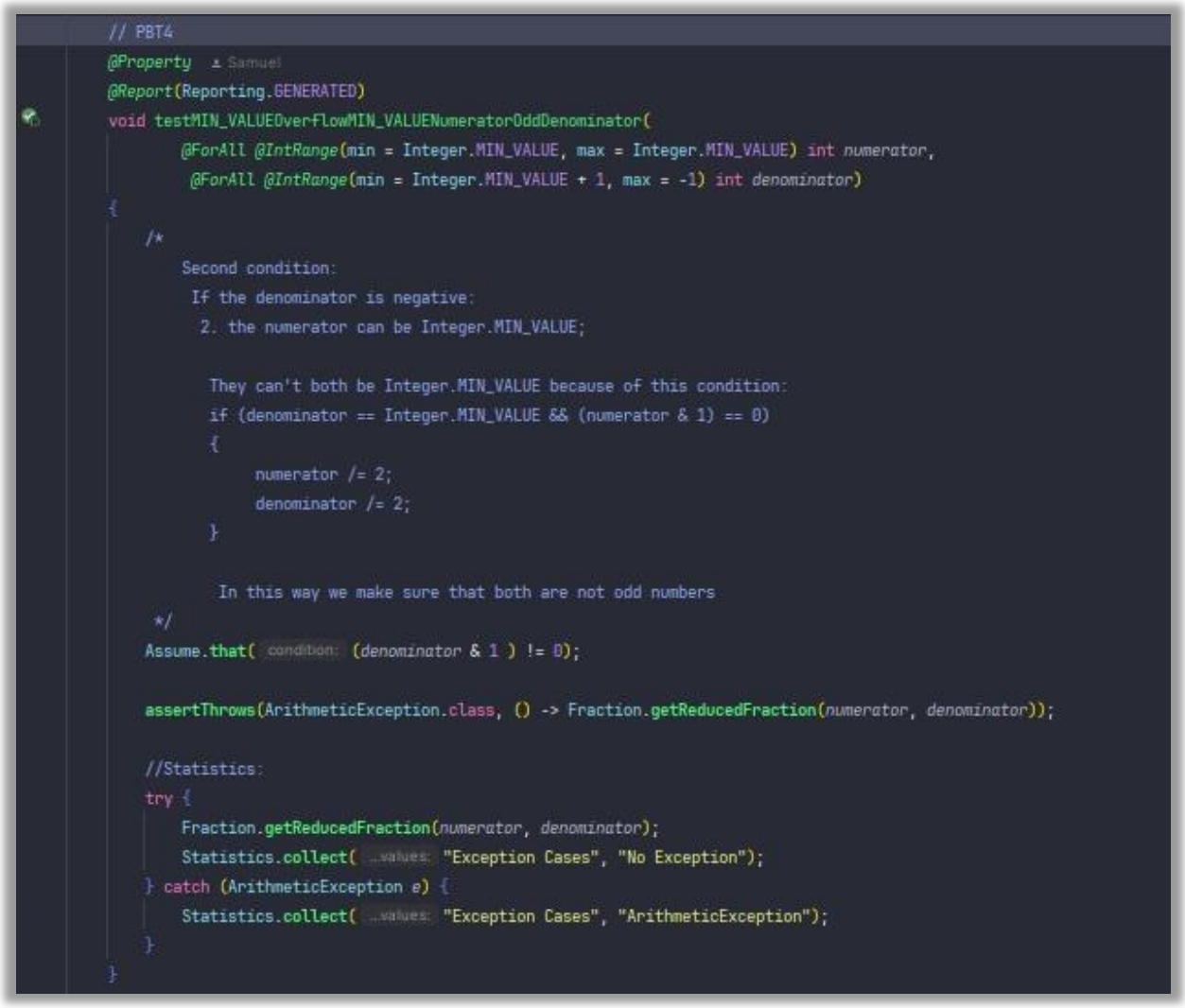
```java
//PBT3
@Property()  ▲ Samuel
@Report(Reporting.GENERATED)
void testMIN_VALUEOverflowOddNumeratorMIN_VALUEDenominator(
        @ForAll @IntRange(min = Integer.MIN_VALUE + 1) int numerator,
        @ForAll @IntRange(min = Integer.MIN_VALUE, max = Integer.MIN_VALUE) int denominator)
{
    /*
        First condition:
         If the denominator is negative:
          1. it can be Integer.MIN_VALUE;

           They can't both be Integer.MIN_VALUE because of this condition:
           if (denominator == Integer.MIN_VALUE && (numerator & 1) == 0)
           {
                numerator /= 2;
                denominator /= 2;
           }

            In this way we make sure that both are not odd numbers
     */
    Assume.that ( condition: (numerator & 1) != 0);

    assertThrows(ArithmeticException.class, () -> Fraction.getReducedFraction(numerator, denominator));

    //Statistics:
    try {
        Fraction.getReducedFraction(numerator, denominator);
        Statistics.collect( ...values: "Exception Cases", "No Exception");
    } catch (ArithmeticException e) {
        Statistics.collect( ...values: "Exception Cases", "ArithmeticException");
    }
}
```

Analizzando le statistiche di questo test, notiamo che sono state generate solo 504 combinazioni che soddisfano i criteri dichiarati nel test, ovvero i parametri per cui la nostra frazione risulta non valida al 100%.

```
timestamp = 2024-05-23T12:28:51.090317700, [FractionPropertyBasedTest:testMIN VALUEOverflowOddNumeratorMIN VALUEDenominator] (504) statistics =
    Exception Cases ArithmeticException (504) : 100 %

timestamp = 2024-05-23T12:28:51.090317700, FractionPropertyBasedTest:testMIN VALUEOverflowOddNumeratorMIN VALUEDenominator =
                            |-----------------------jqwik-----------------------
tries = 1000                | # of calls to property
checks = 504                | # 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 = 9        | # of all combined edge cases
edge-cases#tried = 9        | # of edge cases tried in current run
seed = 3352250921469578215  | random seed to reproduce generated values
```

- **PBT4:**

  Nel quarto test, verifichiamo l'invalidità della frazione quando il numeratore è il valore minimo possibile (MIN VALUE) e il denominatore è un numero dispari negativo.
  In questo caso, ci aspettiamo che venga lanciata un'eccezione, poiché la frazione non sarà mai valida.



```
// PBT4
@Property  ± Samuel
@Report(Reporting.GENERATED)
void testMIN_VALUEOverflowMIN_VALUENumeratorOddDenominator(
        @ForAll @IntRange(min = Integer.MIN_VALUE, max = Integer.MIN_VALUE) int numerator,
        @ForAll @IntRange(min = Integer.MIN_VALUE + 1, max = -1) int denominator)
{
    /*
        Second condition:
        If the denominator is negative:
         2. the numerator can be Integer.MIN_VALUE;

          They can't both be Integer.MIN_VALUE because of this condition:
          if (denominator == Integer.MIN_VALUE && (numerator & 1) == 0)
          {
              numerator /= 2;
              denominator /= 2;
          }

          In this way we make sure that both are not odd numbers
     */
    Assume.that( condition: (denominator & 1 ) != 0);

    assertThrows(ArithmeticException.class, () -> Fraction.getReducedFraction(numerator, denominator));

    //Statistics:
    try {
        Fraction.getReducedFraction(numerator, denominator);
        Statistics.collect( ...values: "Exception Cases", "No Exception");
    } catch (ArithmeticException e) {
        Statistics.collect( ...values: "Exception Cases", "ArithmeticException");
    }
}
```

Analogamente al test precedente, le statistiche mostrano che solo 498 combinazioni di parametri generate risultano sempre non valide al 100%. Questo accade perché il range impostato permette al test di generare anche valori pari. Tuttavia, quando il test verifica quali combinazioni rispettano la condizione, considera solo quelle in cui il parametro scelto è dispari.

```
timestamp = 2024-05-23T12:28:51.516315400, [FractionPropertyBasedTest:testMIN VALUEOverflowMIN VALUENumeratorOddDenominator] (498) statistics =
    Exception Cases ArithmeticException (498) : 100 %

timestamp = 2024-05-23T12:28:51.516315400, FractionPropertyBasedTest:testMIN VALUEOverflowMIN VALUENumeratorOddDenominator =
                              |----------------------jqwik----------------------
tries = 1000                  | # of calls to property
checks = 498                  | # 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 = 4          | # of all combined edge cases
edge-cases#tried = 4          | # of edge cases tried in current run
seed = 6562592843737227058    | random seed to reproduce generated values
```

- **PBT5:**

  Nel quinto e ultimo test, verifichiamo più situazioni per cui la nostra frazione risulti valida:

  - La frazione è valida quando il numeratore ha valori diversi da MIN VALUE e 0;
  - La frazione è valida quando il denominatore è negativo e ha valori diversi da 0 e MIN VALUE.

```java
// PBT5
@Property  ▲ Samuel
@Report(Reporting.GENERATED)
@StatisticsReport(format = Histogram.class)
void testValidNegativeReducedFraction(
        @ForAll @IntRange(min = Integer.MIN_VALUE + 1) int numerator,
         @ForAll @IntRange(min = Integer.MIN_VALUE + 1, max = -1) int denominator)
{
    /*
        Third condition: if the denominator is negative:
           1.1 it must not be Integer.MIN_VALUE;
            1.2 the numerator must not be Integer.MIN_VALUE;
             1.3 the numerator must not be zero.
    */
    Assume.that( condition: numerator != 0);
    assertDoesNotThrow(() -> Fraction.getReducedFraction(numerator, denominator));

    //Statistics:
    Fraction reducedFraction = Fraction.getReducedFraction(numerator, denominator);

    // Collect statistics on the distribution of numerators and denominators
    Statistics.collect( ...values: "Numerator", reducedFraction.getNumerator());
    Statistics.collect( ...values: "Denominator", reducedFraction.getDenominator());

    // Collect statistics on the GCD of the input fractions
    int gcd = Fraction.greatestCommonDivisor(Math.abs(numerator), Math.abs(denominator));
    Statistics.collect( ...values: "GCD", gcd);

    /*
        This block of code checks if the original fraction (numerator/denominator) is equal to
        the reduced fraction.

         The isEqual variable is set to true if both the numerator and denominator of the original
         fraction are equal to those of the reduced fraction.
     */
    boolean isEqual = numerator == reducedFraction.getNumerator() && denominator == reducedFraction.getDenominator();
```

In questo caso, le statistiche mostrano che la condizione da noi verificata è sempre risultata vera (100%) nel 990 dei casi presi in considerazione dal test.

Anche in questo caso andiamo ad analizzare la distribuzione dei valori del numeratore e del denominatore, i valori del MCD (Massimo Comun Divisore) e la probabilità che la frazione iniziale non venga ridotta (ossia quando il MCD è uguale a 1).

Al contrario del PBT2, possiamo vedere come in tutti i casi verificati la frazione sia stata ridotta ai minimi termini.

```
728 | Fraction Equality false |   990 | ##################################################################################
729 |                   GCD 1 |   619 | ###########################################
730 |                   GCD 2 |   148 | ##########
731 |                   GCD 3 |    60 | ####
732 |                   GCD 4 |    42 | ###
733 |                   GCD 5 |    21 | #
734 |                   GCD 6 |    12 |
735 |                   GCD 7 |    11 |
736 |                   GCD 9 |     8 |
737 |                  GCD 10 |     5 |
738 |                   GCD 8 |     5 |
739 |                  GCD 22 |     4 |
740 |                  GCD 11 |     4 |
741 |                  GCD 13 |     4 |
742 |                  GCD 20 |     3 |
743 |                  GCD 14 |     3 |
744 |                  GCD 18 |     3 |
745 |          GCD 2147483647 |     3 |
746 |                  GCD 26 |     3 |
747 |                  GCD 23 |     2 |
748 |                  GCD 15 |     2 |
749 |                  GCD 37 |     2 |
750 |                  GCD 39 |     2 |
751 |                  GCD 25 |     2 |
752 |                  GCD 28 |     2 |
753 |          GCD 2147483646 |     2 |
754 |                  GCD 12 |     2 |
755 |                  GCD 45 |     1 |
756 |                 GCD 144 |     1 |
757 |                  GCD 17 |     1 |
758 |                  GCD 48 |     1 |
759 |                  GCD 24 |     1 |
760 |                  GCD 27 |     1 |
761 |                  GCD 63 |     1 |
762 |                  GCD 66 |     1 |
763 |                  GCD 19 |     1 |
764 |                  GCD 42 |     1 |
765 |                  GCD 71 |     1 |
```

```
timestamp = 2024-05-23T12:28:49.565316700, FractionPropertyBasedTest:testValidNegativeReducedFraction =
                              |----------------------jqwik----------------------
tries = 1000                  | # of calls to property
checks = 990                  | # 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 = 36         | # of all combined edge cases
edge-cases#tried = 36         | # of edge cases tried in current run
seed = -6011010209249568368   | random seed to reproduce generated values
```