

Documentație - Testare în Python

Barbarasa Maria-Cristina-311

Oproiu Matei-311

Problema noastră : Se testează un program care verifică cel mai mare divizor comun între două numere a și b modulo c . Mai precis, pentru trei variabile care vor îndeplini condiția să fie numere naturale strict pozitive se va calcula folosind algoritmul lui Euclid cu scăderi repetate între a și b , pentru un număr maxim de 1000 de iterații. Programul va produce un output care va indica rezultatul în urma calculului algoritmului modulo c .

1. Testarea funcțională clasei ModuloGCD

(a) Partiționare de echivalență

1. Domeniul de intrări: Pentru clasa compusa avem 3 intrări:

- a - valoare naturală strict pozitivă deci se disting două clase de echivalență:

$A_1 = \{ a \mid a \text{ număr natural strict pozitiv} \}$

$A_2 = \{ a \mid a \text{ nu este număr natural strict pozitiv} \}$

- b - valoare naturală strict pozitivă deci se disting două clase de echivalență:

$B_1 = \{ b \mid b \text{ număr natural strict pozitiv} \}$

$B_2 = \{ b \mid b \text{ nu este număr natural strict pozitiv} \}$

- c -valoare naturală strict pozitivă deci se disting două clase de echivalență:

$C_1 = \{ c \mid c \text{ număr natural strict pozitiv} \}$

$C_2 = \{ c \mid c \text{ nu este număr natural strict pozitiv} \}$

2. Domeniul de ieșire: Constă într-un număr natural $\text{result} \in [0, c)$ și este rezultatul procesării numerelor a, b, c . Acesta este folosit pentru a împărți domeniul de intrare în mai multe clase, în funcție de valorile a, b, c :

$X_1 = \{ \text{result} \mid \text{GCD}(a, b) > c \}$

$X_2 = \{ \text{result} \mid \text{GCD}(a, b) == c \}$

$X_3 = \{ \text{result} \mid \text{GCD}(a, b) < c \}$

Clasele de echivalențe globale pe care le avem:

$G_{111} = \{ (a,b,c) \mid a \in A_1 \ \& \ b \in B_1 \ \& \ c \in C_1 \}$

$G_{211} = \{ (a,b,c) \mid a \in A_2 \ \& \ b \in B_1 \ \& \ c \in C_1 \}$

$G_{121} = \{ (a,b,c) \mid a \in A_1 \ \& \ b \in B_2 \ \& \ c \in C_1 \}$

$G_{112} = \{ (a,b,c) \mid a \in A_1 \ \& \ b \in B_1 \ \& \ c \in C_2 \}$

$G_{122} = \{ (a,b,c) \mid a \in A_1 \ \& \ b \in B_2 \ \& \ c \in C_2 \}$

$G_{212} = \{ (a,b,c) \mid a \in A_2 \ \& \ b \in B_1 \ \& \ c \in C_2 \}$

$G_{221} = \{ (a,b,c) \mid a \in A_2 \ \& \ b \in B_2 \ \& \ c \in C_1 \}$

$G_{222} = \{ (a,b,c) \mid a \in A_2 \ \& \ b \in B_2 \ \& \ c \in C_2 \}$

Setul de date se alcătuiește prin alegerea unei valori de intrare pentru fiecare clasa. Exemplu:

$g_{111}: (10,5,2)$

$g_{211}: (-5,5,2)$

$g_{121}: (50,"5",4)$

$g_{112}: (43,5,-2)$

$g_{122}: (5,-5,-2)$

$g_{212}: (100.2,31,\text{none})$

$g_{221}: (-5,32.2,6)$

$g_{222}: (-5,15.2,"12")$

a	b	c	Rezultat afișat (expected)
10	5	2	1
15	10	4	1

8	12	5	4
-5	5	2	Ridică ValueError: "Valorile trebuie sa fie numere naturale pozitive"
50	"5"	4	Ridică ValueError: "Valorile trebuie sa fie numere naturale pozitive"
10	5	0	Ridică ValueError: "Valorile trebuie sa fie numere naturale pozitive"
5	-5	-2	Ridică ValueError: "Valorile trebuie sa fie numere naturale pozitive"
100.2	31	None	Ridică ValueError: "Valorile trebuie sa fie numere naturale pozitive"
-5	32.2	6	Ridică ValueError: "Valorile trebuie sa fie numere naturale pozitive"
-5	15.2	"12"	Ridică ValueError: "Valorile trebuie sa fie numere naturale pozitive"

```

"""Partitionare de echivalenta"""

def test_equivalence_classes(self):
    self.assertEqual(ModuloGCD(10,5,3).cmmdc(),2)
    self.assertEqual(ModuloGCD(15, 10,4).cmmdc(),1)
    self.assertEqual(ModuloGCD(8,12,5).cmmdc(),4)

    # G211 - Invalid a
    with self.assertRaises(ValueError):
        ModuloGCD(-5,5,2).cmmdc()
    # G121 - Invalid b
    with self.assertRaises(ValueError):
        ModuloGCD(50,'5',4).cmmdc()
    # G112 - Invalid c
    with self.assertRaises(ValueError):
        ModuloGCD(10,5,0).cmmdc()
    # G122 - Invalid b and c
    with self.assertRaises(ValueError):
        ModuloGCD(5,-5,-2).cmmdc()
    # G212 - Invalid a and c
    with self.assertRaises(ValueError):
        ModuloGCD(100.2,31,None).cmmdc()
    # G221 - Invalid a and b
    with self.assertRaises(ValueError):
        ModuloGCD(-5,32.2,6).cmmdc()
    # G222 - Invalid a, b, and c
    with self.assertRaises(ValueError):
        ModuloGCD(-5,15.2,'12').cmmdc()

```

"""1) Partitionare de echivalenta"""

```

def test_equivalence_classes(self):
    self.assertEqual(ModuloGCD(10,5,3).cmmdc(),2)
    self.assertEqual(ModuloGCD(15, 10,4).cmmdc(),1)
    self.assertEqual(ModuloGCD(8,12,5).cmmdc(),4)

```

```

# G211 - Invalid a
with self.assertRaises(ValueError):
    ModuloGCD(-5,5,2).cmmdc()
# G121 - Invalid b
with self.assertRaises(ValueError):
    ModuloGCD(50,'5',4).cmmdc()
# G112 - Invalid c
with self.assertRaises(ValueError):
    ModuloGCD(10,5,0).cmmdc()
# G122 - Invalid b and c
with self.assertRaises(ValueError):

```

```

ModuloGCD(5,-5,-2).cmmdc()
# G212 - Invalid a and c
with self.assertRaises(ValueError):
    ModuloGCD(100.2,31,None).cmmdc()
# G221 - Invalid a and b
with self.assertRaises(ValueError):
    ModuloGCD(-5,32.2,6).cmmdc()
# G222 - Invalid a, b, and c
with self.assertRaises(ValueError):
    ModuloGCD(-5,15.2,'12').cmmdc()

```

(b) Analiza valorilor de frontieră (boundary value analysis)

Analiza valorilor de frontieră este adesea utilizată împreună cu partiționarea în clase de echivalență. Această tehnică pune accentul pe testarea valorilor aflate la limita fiecărei clase, deoarece acestea reprezintă frecvent o sursă majoră de erori. În cazul exemplului nostru, odată ce clasele de echivalență au fost stabilite, identificarea valorilor de frontiera devine un proces intuitiv :

- valorile de frontieră sunt 0,1,2 pentru a, b, c

Se vor testa următoarele valori:

- N_1 : 0
- N_2 : 1
- N_3 : 2

Vom avea 9 date de test :

a	b	c	Rezultat afișat (expected)
0	5	2	Ridică ValueError: "Valorile trebuie sa fie numere naturale pozitive"

7	0	3	Ridică ValueError: "Valorile trebuie sa fie numere naturale pozitive"
7	21	0	Ridică ValueError: "Valorile trebuie sa fie numere naturale pozitive"
1	10	3	1
2	5	2	1
21	1	5	1
14	2	7	2
33	11	1	0
15	3	2	1

```

"""Analiza valorilor de frontiera"""
def test_boundary_values(self):
    # a=0
    with self.assertRaises(ValueError):
        ModuloGCD(0,5,2).cmmdc()
    # b=0
    with self.assertRaises(ValueError):
        ModuloGCD(7,0,3).cmmdc()
    # c=0
    with self.assertRaises(ValueError):
        ModuloGCD(7,21,0).cmmdc()

    # a=1
    self.assertEqual(ModuloGCD(1,10,3).cmmdc(),1)
    # a=2
    self.assertEqual(ModuloGCD(2,5,2).cmmdc(),1)
    # b=1
    self.assertEqual(ModuloGCD(21,1,5).cmmdc(),1)
    # b=2
    self.assertEqual(ModuloGCD(14,2,7).cmmdc(),2)
    # c=1
    self.assertEqual(ModuloGCD(33,11,1).cmmdc(),0)
    # c=2
    self.assertEqual(ModuloGCD(15,3,2).cmmdc(),1)

```

"""2) Analiza valorilor de frontiera"""

```
def test_boundary_values(self):
```

```
# a=0
```

```
with self.assertRaises(ValueError):
```

```
    ModuloGCD(0,5,2).cmmdc()
```

```
# b=0
```

```
with self.assertRaises(ValueError):
```

```
    ModuloGCD(7,0,3).cmmdc()
```

```

# c=0

with self.assertRaises(ValueError):

    ModuloGCD(7,21,0).cmmdc()

# a=1
self.assertEqual(ModuloGCD(1,10,3).cmmdc(),1)
# a=2
self.assertEqual(ModuloGCD(2,5,2).cmmdc(),1)
# b=1
self.assertEqual(ModuloGCD(21,1,5).cmmdc(),1)
# b=2
self.assertEqual(ModuloGCD(14,2,7).cmmdc(),2)
# c=1
self.assertEqual(ModuloGCD(33,11,1).cmmdc(),0)
# c=2
self.assertEqual(ModuloGCD(15,3,2).cmmdc(),1)

```

2. Testarea structurală

1. Graful de flux de control

Am ales datele de test astfel încât să parcurgă toate elementele noastre introduse în cod.

```

12     def cmmdc(self):
13         a=self.validate_integer(self.a)
14         b=self.validate_integer(self.b)
15         c=self.validate_integer(self.c)
16
17         max_iterations=1000
18         iterations=0
19
20         if c<=0:
21             raise ValueError('Modulo trebuie sa fie pozitiv')
22
23         if a==b:
24             return a%c
25
26         while a!=b:
27             if iterations>max_iterations:
28                 raise RuntimeError('Prea multe iteratii (posibil loop infinit)')
29             iterations+=1
30
31             if a>b:
32                 a=a-b
33             else:
34                 b=b-a
35
36         return a%c

```

def cmmdc(self):

```

a=self.validate_integer(self.a)
b=self.validate_integer(self.b)
c=self.validate_integer(self.c)
max_iterations=1000
iterations=0
if c<=0:
    raise ValueError('Modulo trebuie sa fie pozitiv')
if a==b
return a%c

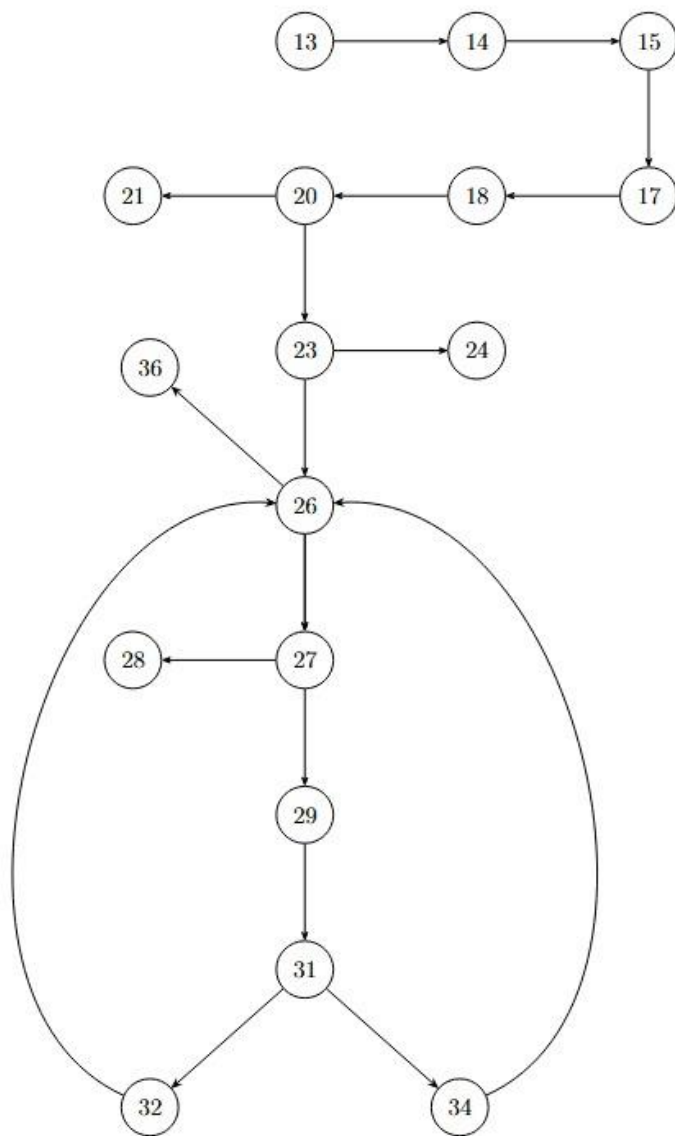
while a!=b:
    if iterations>max_iterations:
        raise RuntimeError('Prea multe iteratii (posibil loop infinit)')
    iterations+=1
    if a>b:
        a=a-b
    else:
        b=b-a
return a%c

```

Nodul 13 verifică a, nodul 14 verifică b si nodul 15 verifică c. Nodul 17 ne inițializează numărul maxim de iterații, nodul 18 ne inițializează numărul de iterații, nodul 20 verifică dacă c este valid în caz contrar programul se încheie, nodul 23 verifică dacă a=b atunci iese din program. Nodul 26 intră în loop-ul while cât timp a!=b, nodul 27 verifică dacă am depășit nr maxim de iterații, nodul 29 crește indexarea iterațiilor, nodul 31 verifică dacă este mai mare decât în cazul acesta nodul 32 schimbă valoarea lui a conform a=a-b

altfel nodul 34 schimbă valoarea lui b conform $b=b-a$. Iar nodul 36 este nodul nostru de ieșire.

De asemenea, pentru nodurile 13,14,15 este necesară trecerea de către condiția `if not isinstance(value,int) or value < 0` pentru a se putea forma.



(a) Acoperirea la nivel de instrucțiune :

Ca să avem ca rezultat o acoperire la nivel de instrucțiune testul este luat astfel încât să parcurgă fiecare nod al grafului.

(b) Acoperire la nivel de decizie (decision coverage)

	Decizii
(1)	if not isinstance(value,int) or value <0:
(2)	while a!=b:
(3)	if a>b:

a	b	c	Rezultat afișat	Decizii acoperite
10	10	12	10	(1)-A,(2)-F
14	10	5	2	(1)-A, (2)-A,(3)-A
6	9	7	3	(1)-A, (2)-A,(3)-F

```
"""Acoperire la nivel de decizie"""  
  
def test_decision_coverage(self):  
    # Loop skipped  
    self.assertEqual(ModuloGCD(10,10,12).cmmdc(),10)  
    # Loop true branch  
    self.assertEqual(ModuloGCD(14,10,5).cmmdc(),2)  
    # Else branch  
    self.assertEqual(ModuloGCD(6,9,7).cmmdc(),3)  
  
    # Invalid input  
    with self.assertRaises(ValueError):  
        ModuloGCD('10',5,2).cmmdc()
```

"""4) Acoperire la nivel de decizie"""

```
def test_decision_coverage(self): # Loop skipped  
self.assertEqual(ModuloGCD(10,10,12).cmmdc(),10) # Loop true branch  
self.assertEqual(ModuloGCD(14,10,5).cmmdc(),2) # Else branch  
self.assertEqual(ModuloGCD(6,9,7).cmmdc(),3)  
  
# Invalid input  
with self.assertRaises(ValueError):  
    ModuloGCD('10',5,2).cmmdc()
```

(c) Acoperire la nivel de condiție (condition coverage)

Decizii	Condiții individuale
if not isinstance(value,int) or value <=0 :	isinstance(value,int), value <=0
while a!= b:	a!=b
if a>b:	a>b

a	b	c	Rezultat afișat	Condiții/Decizii individuale acoperite
"9"	4	3	Ridică ValueError: "Valorile trebuie sa fie numere naturale pozitive"	isinstance(value,int)- F
10	-2	3	Ridică ValueError: "Valorile trebuie sa fie numere naturale pozitive"	value <=0-F
4	6	6	2	isinstance(value,int)- A, value <=0-A, a!=b-A, a>b-A
3	3	6	3	isinstance(value,int)- A, value <=0-A, a!=b-F
4	1	5	1	isinstance(value,int)- A, value <=0-A, a!=b-A, a>b-A
1	3	10	1	isinstance(value,int)- A, value <=0-A, a!=b-A, a>b-F

```

"""Acoperire la nivel de conditie"""

def test_condition_coverage(self):
    # Condition - isinstance fail
    with self.assertRaises(ValueError):
        ModuloGCD('9',4,3).cmmdc()
    # Condition value check fail
    with self.assertRaises(ValueError):
        ModuloGCD(10,-2,3).cmmdc()

    # While condition true
    self.assertEqual(ModuloGCD(4,6,6).cmmdc(),2)
    # While condition false
    self.assertEqual(ModuloGCD(3,3,6).cmmdc(),3)
    #If condition true
    self.assertEqual(ModuloGCD(4,1,5).cmmdc(),1)
    #If condition false
    self.assertEqual(ModuloGCD(1,3,10).cmmdc(),1)

```

"""Acoperire la nivel de conditie"""

def test_condition_coverage(self): # Condition - isinstance fail

with self.assertRaises(ValueError):

ModuloGCD('9',4,3).cmmdc()

Condition value check fail

with self.assertRaises(ValueError):

ModuloGCD(10,-2,3).cmmdc()

While condition true

self.assertEqual(ModuloGCD(4,6,6).cmmdc(),2)

While condition false

self.assertEqual(ModuloGCD(3,3,6).cmmdc(),3)

#If condition true

self.assertEqual(ModuloGCD(4,1,5).cmmdc(),1)

#If condition false

self.assertEqual(ModuloGCD(1,3,10).cmmdc(),1)

(d) Testarea circuitelor independente:

Numărul minim de circuite independente pentru a obține o acoperire completă a ramurilor este data de formula lui McCabe. $V(G)=e-n+2p$, unde:

e = numărul de muchii ale graficului = 18

n = numărul de noduri ale graficului = 17

p = numărul de componente conectate = 1

Deci, în cazul nostru $V(G) = 18 - 17 + 2 \cdot 1 = 3$

- Circuitele independente:

a) 13,14,15,17,18,20,23,26 = fals -> 36

b) 13,14,15,17,18,20,23,26 = adevărat , repetăm 27,29,31,32 -> 36

c) 13,14,15,17,18,20,23,26 = adevărat repetam 27,29,31,34 ->36

```
"""Testarea circuitelor independente"""  
  
def test_independent_paths(self):  
    # Exist of isinstance fail  
    with self.assertRaises(ValueError):  
        ModuloGCD('10',3,6).cmmdc()  
    # Exist of value check fail  
    with self.assertRaises(ValueError):  
        ModuloGCD(10,3.3,6).cmmdc()  
    # Exist on both checks failing  
    with self.assertRaises(ValueError):  
        ModuloGCD('10',3.3,6).cmmdc()  
  
    # Exist when a==b  
    self.assertEqual(ModuloGCD(10,10,12).cmmdc(),10)  
    # Loop executed  
    self.assertEqual(ModuloGCD(55,10,12).cmmdc(),5)
```

"""6) Testarea circuitelor independente"""

```
def test_independent_paths(self):
```

```
    # Exist of isinstance fail
```

```
    with self.assertRaises(ValueError):
```

```
        ModuloGCD('10',3,6).cmmdc()
```

```
    # Exist of value check fail
```

```
    with self.assertRaises(ValueError):
```

```
        ModuloGCD(10,3.3,6).cmmdc()
```

```
# Exist on both checks failing

with self.assertRaises(ValueError):

    ModuloGCD('10',3.3,6).cmmdc()

# Exist when a==b
self.assertEqual(ModuloGCD(10,10,12).cmmdc(),10)
# Loop executed
self.assertEqual(ModuloGCD(55,10,12).cmmdc(),5)
```

3.Mutanti

Testarea mutanților este considerată o testare a calității testelor prin adăugarea unor modificări și a unor mutanți în codul de bază.

Mutation= modificare foarte mică

Evaluarea testează dacă se detectează mutanții.

Ideea este de a evalua cât de bune sunt testele făcute.

Pentru a evalua toti mutantii, i-am generat automat folosind mutpy.

Programul nostru :

```

1 class ModuloGCD:
2     def __init__(self, a, b, c):
3         self.a = a
4         self.b = b
5         self.c = c
6
7     def validate_integer(self, value):
8         if not isinstance(value, int) or value <= 0:
9             raise ValueError('Valorile trebuie sa fie numere naturale pozitive')
10        return value
11
12    def cmmdc(self):
13        a = self.validate_integer(self.a)
14        b = self.validate_integer(self.b)
15        c = self.validate_integer(self.c)
16
17        max_iterations = 1000
18        iterations = 0
19
20        if c <= 0:
21            raise ValueError('Modulo trebuie sa fie pozitiv')
22
23        if a == b:
24            return a % c
25
26        while a != b:
27            if iterations > max_iterations:
28                raise RuntimeError('Prea multe iteratii (posibil loop infinit)')
29            iterations += 1
30
31            if a > b:
32                a = a - b
33            else:
34                b = b - a
35
36        return a % c

```

În urma primului set de teste am obținut aceste rezultate:

```

[0.00100 s] survived
[*] Mutation score [0.60074 s]: 72.7%
- all: 23
- killed: 16 (69.6%)
- survived: 6 (26.1%)
- incompetent: 1 (4.3%)
- timeout: 0 (0.0%)

```

Iar acestia sunt mutanții care au supraviețuit:

```

- [# 1] AOR ModuloGCD:
-----
20:         if c <= 0:
21:             raise ValueError('Modulo trebuie sa fie pozitiv')
22:
23:         if a == b:
- 24:             return a % c
+ 24:             return a * c
25:
26:         while a != b:
27:             if iterations > max_iterations:
28:                 raise RuntimeError('Prea multe iteratii (posibil loop infinit)')
-----
[0.00204 s] survived
- [# 13] LCR ModuloGCD:
-----
4:         self.b = b
5:         self.c = c
6:
7:     def validate_integer(self, value):
- 8:         if (not (isinstance(value, int))) or value <= 0):
+ 8:         if (not (isinstance(value, int))) and value <= 0):
9:             raise ValueError('Valorile trebuie sa fie numere naturale pozitive')
10:         return value
11:
12:     def cmmdc(self):
-----
[0.00100 s] incompetent

```



```

- [# 15] ROR ModuloGCD:
-----
4:         self.b = b
5:         self.c = c
6:
7:         def validate_integer(self, value):
- 8:             if (not (isinstance(value, int)) or value <= 0):
+ 8:             if (not (isinstance(value, int)) or value < 0):
9:                 raise ValueError('Valorile trebuie sa fie numere naturale pozitive')
10:             return value
11:
12:         def cmmdc(self):
-----
[0.00101 s] survived

- [# 17] ROR ModuloGCD:
-----
16:
17:         max_iterations = 1000
18:         iterations = 0
19:
- 20:         if c <= 0:
+ 20:         if c < 0:
21:             raise ValueError('Modulo trebuie sa fie pozitiv')
22:
23:         if a == b:
24:             return a % c
-----
[0.00099 s] survived

- [# 21] ROR ModuloGCD:
-----
23:         if a == b:
24:             return a % c
25:
26:         while a != b:
- 27:             if iterations > max_iterations:
+ 27:             if iterations >= max_iterations:
28:                 raise RuntimeError('Prea multe iteratii (posibil loop infinit)')
29:                 iterations += 1
30:
31:             if a > b:
-----
[0.00105 s] survived

- [# 23] ROR ModuloGCD:
-----
27:         if iterations > max_iterations:
28:             raise RuntimeError('Prea multe iteratii (posibil loop infinit)')
29:             iterations += 1
30:
- 31:         if a > b:
+ 31:         if a >= b:
32:             a = a - b
33:         else:
34:             b = b - a
35:
-----
[0.00100 s] survived

```

Codul testat:

```

1  import unittest
2  from ModuloGCD import ModuloGCD
3
4  class ModuloGCDTest(unittest.TestCase):
5      def test_valid_inputs(self):
6          self.assertEqual(ModuloGCD(10, 5, 3).cmmdc(), 2)
7          self.assertEqual(ModuloGCD(15, 10, 4).cmmdc(), 1)
8          self.assertEqual(ModuloGCD(8, 12, 5).cmmdc(), 4)
9
10     def test_invalid_inputs(self):
11         with self.assertRaises(ValueError):
12             ModuloGCD(-5, 5, 2).cmmdc()
13         with self.assertRaises(ValueError):
14             ModuloGCD(50, '5', 4).cmmdc()
15         with self.assertRaises(ValueError):
16             ModuloGCD(5, -5, -2).cmmdc()
17
18     def test_boundary_values(self):
19         self.assertEqual(ModuloGCD(1, 10, 3).cmmdc(), 1)
20         self.assertEqual(ModuloGCD(21, 1, 5).cmmdc(), 1)
21         self.assertEqual(ModuloGCD(33, 11, 1).cmmdc(), 0)
22         self.assertEqual(ModuloGCD(15, 3, 2).cmmdc(), 1)
23
24     def test_instruction_coverage(self):
25         self.assertEqual(ModuloGCD(43, 17, 2).cmmdc(), 1)
26
27     def test_decision_coverage(self):
28         self.assertEqual(ModuloGCD(14, 10, 5).cmmdc(), 2)
29
30     def test_condition_coverage(self):
31         with self.assertRaises(ValueError):
32             ModuloGCD('9', 4, 3).cmmdc()
33         self.assertEqual(ModuloGCD(4, 1, 5).cmmdc(), 1)
34
35         self.assertEqual(ModuloGCD(4, 1, 5).cmmdc(), 1)
36
37     def test_independent_paths(self):
38         with self.assertRaises(ValueError):
39             ModuloGCD('10', 3, 6).cmmdc()
40         self.assertEqual(ModuloGCD(55, 10, 12).cmmdc(), 5)
41
42 if __name__ == '__main__':
43     unittest.main()

```

Astfel pentru a rezolva problema întâmpinată, am adăugat noi teste pentru a acoperii toate ramurile și condițiile noastre.

Mutanti obtinuti în urma codului nou:

```

[*] Start mutation process:
- targets: ModuloGCD
- tests: test_nou
[*] 7 tests passed:
- test_nou [0.00100 s]
[*] Start mutants generation and execution:
- [# 1] AOR ModuloGCD:
-----
20:         if c <= 0:
21:             raise ValueError('Modulo trebuie sa fie pozitiv')
22:
23:         if a == b:
- 24:             return a % c
+ 24:             return a * c
25:
26:         while a != b:
27:             if iterations > max_iterations:
28:                 raise RuntimeError('Prea multe iteratii (posibil loop infinit)')
-----
[0.04130 s] killed by test_condition_coverage (test_nou.ModuloGCDTest.test_condition_coverage)

```

```

- [# 2] AOR ModuloGCD:
-----
28:         raise RuntimeError('Prea multe iteratii (posibil loop infinit)')
29:         iterations += 1
30:
31:         if a > b:
- 32:             a = a - b
+ 32:             a = a + b
33:         else:
34:             b = b - a
35:
36:         return a % c
-----
[0.00404 s] killed by test_boundary_values (test_nou.ModuloGCDTest.test_boundary_values)

```

```

- [# 3] AOR ModuloGCD:
-----
30:
31:         if a > b:
32:             a = a - b
33:         else:
- 34:             b = b - a
+ 34:             b = b + a
35:
36:         return a % c
-----
[0.00100 s] killed by test_boundary_values (test_nou.ModuloGCDTest.test_boundary_values)

```

```

- [# 4] AOR ModuloGCD:
-----
32:         a = a - b
33:         else:
34:             b = b - a
35:
- 36:         return a % c
+ 36:         return a * c
-----
[0.00099 s] killed by test_boundary_values (test_nou.ModuloGCDTest.test_boundary_values)

```

```

- [# 5] ASR ModuloGCD:
-----
25:
26:         while a != b:
27:             if iterations > max_iterations:
28:                 raise RuntimeError('Prea multe iteratii (posibil loop infinit)')
- 29:             iterations += 1
+ 29:             iterations -= 1
30:
31:             if a > b:
32:                 a = a - b
33:             else:
-----
[0.00102 s] survived

```

```

- [# 6] COD ModuloGCD:
-----
4:         self.b = b
5:         self.c = c
6:
7:         def validate_integer(self, value):
- 8:             if (not isinstance(value, int)) or value <= 0):
+ 8:             if (isinstance(value, int) or value <= 0):
9:                 raise ValueError('Valorile trebuie sa fie numere naturale pozitive')
10:             return value
11:
12:         def cmmdc(self):
-----
[0.00500 s] killed by test_boundary_values (test_nou.ModuloGCDTest.test_boundary_values)

```

```

- [# 7] COI ModuloGCD:
-----
4:         self.b = b
5:         self.c = c
6:
7:         def validate_integer(self, value):
- 8:             if (not isinstance(value, int)) or value <= 0):
+ 8:             if not ((not isinstance(value, int)) or value <= 0)):
9:                 raise ValueError('Valorile trebuie sa fie numere naturale pozitive')
10:             return value
11:
12:         def cmmdc(self):
-----
[0.00429 s] killed by test_boundary_values (test_nou.ModuloGCDTest.test_boundary_values)

```

```

- [# 8] COI ModuloGCD:
-----
16:
17:     max_iterations = 1000
18:     iterations = 0
19:
- 20:     if c <= 0:
+ 20:     if not (c <= 0):
21:         raise ValueError('Modulo trebuie sa fie pozitiv')
22:
23:     if a == b:
24:         return a % c
-----
[0.00101 s] killed by test_boundary_values (test_nou.ModuloGCDTest.test_boundary_values)

```

```

- [# 9] COI ModuloGCD:
-----
19:
20:     if c <= 0:
21:         raise ValueError('Modulo trebuie sa fie pozitiv')
22:
- 23:     if a == b:
+ 23:     if not (a == b):
24:         return a % c
25:
26:     while a != b:
27:         if iterations > max_iterations:
-----
[0.00201 s] killed by test_boundary_values (test_nou.ModuloGCDTest.test_boundary_values)

```

```

- [# 10] COI ModuloGCD:
-----
22:
23:     if a == b:
24:         return a % c
25:
- 26:     while a != b:
+ 26:     while not (a != b):
27:         if iterations > max_iterations:
28:             raise RuntimeError('Prea multe iteratii (posibil loop infinit)')
29:             iterations += 1
30:
-----
[0.00100 s] killed by test_boundary_values (test_nou.ModuloGCDTest.test_boundary_values)

```

```

- [# 11] COI ModuloGCD:
-----
23:     if a == b:
24:         return a % c
25:
26:     while a != b:
- 27:         if iterations > max_iterations:
+ 27:         if not (iterations > max_iterations):
28:             raise RuntimeError('Prea multe iteratii (posibil loop infinit)')
29:             iterations += 1
30:
31:     if a > b:
-----
[0.00099 s] killed by test_boundary_values (test_nou.ModuloGCDTest.test_boundary_values)

```

```

- [# 12] COI ModuloGCD:
-----
27:     if iterations > max_iterations:
28:         raise RuntimeError('Prea multe iteratii (posibil loop infinit)')
29:         iterations += 1
30:
- 31:     if a > b:
+ 31:     if not (a > b):
32:         a = a - b
33:     else:
34:         b = b - a
35:
-----
[0.00200 s] killed by test_boundary_values (test_nou.ModuloGCDTest.test_boundary_values)

```

```

- [# 13] LCR ModuloGCD:
-----
4:     self.b = b
5:     self.c = c
6:
7:     def validate_integer(self, value):
- 8:         if not (isinstance(value, int)) or value <= 0:
+ 8:         if not (isinstance(value, int)) and value <= 0:
9:             raise ValueError('Valorile trebuie sa fie numere naturale pozitive')
10:         return value
11:
12:     def cmmdc(self):
-----
[0.00201 s] killed by test_boundary_values (test_nou.ModuloGCDTest.test_boundary_values)

```

```

- [# 14] ROR ModuloGCD:
-----
4:         self.b = b
5:         self.c = c
6:
7:     def validate_integer(self, value):
- 8:         if (not (isinstance(value, int)) or value <= 0):
+ 8:         if (not (isinstance(value, int)) or value >= 0):
9:             raise ValueError('Valorile trebuie sa fie numere naturale pozitive')
10:         return value
11:
12:     def cmmdc(self):
-----
[0.00300 s] killed by test_boundary_values (test_nou.ModuloGCDTest.test_boundary_values)

```

```

- [# 15] ROR ModuloGCD:
-----
4:         self.b = b
5:         self.c = c
6:
7:     def validate_integer(self, value):
- 8:         if (not (isinstance(value, int)) or value <= 0):
+ 8:         if (not (isinstance(value, int)) or value < 0):
9:             raise ValueError('Valorile trebuie sa fie numere naturale pozitive')
10:         return value
11:
12:     def cmmdc(self):
-----
[0.00201 s] killed by test_boundary_values (test_nou.ModuloGCDTest.test_boundary_values)

```

```

- [# 16] ROR ModuloGCD:
-----
16:
17:         max_iterations = 1000
18:         iterations = 0
19:
- 20:         if c <= 0:
+ 20:         if c >= 0:
21:             raise ValueError('Modulo trebuie sa fie pozitiv')
22:
23:         if a == b:
24:             return a % c
-----
[0.00131 s] killed by test_boundary_values (test_nou.ModuloGCDTest.test_boundary_values)

```

```

- [# 17] ROR ModuloGCD:
-----
16:
17:         max_iterations = 1000
18:         iterations = 0
19:
- 20:         if c <= 0:
+ 20:         if c < 0:
21:             raise ValueError('Modulo trebuie sa fie pozitiv')
22:
23:         if a == b:
24:             return a % c
-----
[0.00151 s] survived

```

```

- [# 18] ROR ModuloGCD:
-----
19:
20:     if c <= 0:
21:         raise ValueError('Modulo trebuie sa fie pozitiv')
22:
- 23:     if a == b:
+ 23:     if a != b:
24:         return a % c
25:
26:     while a != b:
27:         if iterations > max_iterations:
-----
[0.00000 s] killed by test_boundary_values (test_nou.ModuloGCDTest.test_boundary_values)
- [# 19] ROR ModuloGCD:
-----
22:
23:     if a == b:
24:         return a % c
25:
- 26:     while a != b:
+ 26:     while a == b:
27:         if iterations > max_iterations:
28:             raise RuntimeError('Prea multe iteratii (posibil loop infinit)')
29:             iterations += 1
30:
-----
[0.00100 s] killed by test_boundary_values (test_nou.ModuloGCDTest.test_boundary_values)
- [# 20] ROR ModuloGCD:
-----
23:     if a == b:
24:         return a % c
25:
26:     while a != b:
- 27:         if iterations > max_iterations:
+ 27:         if iterations < max_iterations:
28:             raise RuntimeError('Prea multe iteratii (posibil loop infinit)')
29:             iterations += 1
30:
31:     if a > b:
-----
[0.00200 s] killed by test_boundary_values (test_nou.ModuloGCDTest.test_boundary_values)
- [# 21] ROR ModuloGCD:
-----
23:     if a == b:
24:         return a % c
25:
26:     while a != b:
- 27:         if iterations > max_iterations:
+ 27:         if iterations >= max_iterations:
28:             raise RuntimeError('Prea multe iteratii (posibil loop infinit)')
29:             iterations += 1
30:
31:     if a > b:
-----
[0.00100 s] survived

```

```

- [# 22] ROR ModuloGCD:
-----
27:         if iterations > max_iterations:
28:             raise RuntimeError('Prea multe iteratii (posibil loop infinit)')
29:             iterations += 1
30:
- 31:         if a > b:
+ 31:         if a < b:
32:             a = a - b
33:         else:
34:             b = b - a
35:
-----
[0.00206 s] killed by test_boundary_values (test_nou.ModuloGCDTest.test_boundary_values)

- [# 23] ROR ModuloGCD:
-----
27:         if iterations > max_iterations:
28:             raise RuntimeError('Prea multe iteratii (posibil loop infinit)')
29:             iterations += 1
30:
- 31:         if a > b:
+ 31:         if a >= b:
32:             a = a - b
33:         else:
34:             b = b - a
35:
-----
[0.00000 s] survived
[*] Mutation score [0.47773 s]: 82.6%
- all: 23
- killed: 19 (82.6%)
- survived: 4 (17.4%)
- incompetent: 0 (0.0%)
- timeout: 0 (0.0%)

```

Codul nou pe care am testat:

```

1  import unittest
2  from ModuloGCD import ModuloGCD
3
4  class ModuloGCDTest(unittest.TestCase):
5      """Partitionare de echivalenta"""
6
7      def test_valid_inputs(self):
8          self.assertEqual(ModuloGCD(10,5,3).cmmdc(),2)
9          self.assertEqual(ModuloGCD(15, 10,4).cmmdc(),1)
10         self.assertEqual(ModuloGCD(8,12,5).cmmdc(),4)
11
12         def test_invalid_inputs(self):
13             with self.assertRaises(ValueError):
14                 ModuloGCD(-5,5,2).cmmdc()
15             with self.assertRaises(ValueError):
16                 ModuloGCD(50,'5',4).cmmdc()
17             with self.assertRaises(ValueError):
18                 ModuloGCD(10,5,0).cmmdc()
19             with self.assertRaises(ValueError):
20                 ModuloGCD(5,-5,-2).cmmdc()
21             with self.assertRaises(ValueError):
22                 ModuloGCD(100.2,31,None).cmmdc()
23             with self.assertRaises(ValueError):
24                 ModuloGCD(-5,32.2,6).cmmdc()
25             with self.assertRaises(ValueError):
26                 ModuloGCD(-5,15.2,'12').cmmdc()
27
28         """Analiza valorilor de frontiera"""
29
30         def test_boundary_values(self):
31             with self.assertRaises(ValueError):
32                 ModuloGCD(0,5,2).cmmdc()
33             with self.assertRaises(ValueError):

```



```

34         ModuloGCD(7,0,3).cmmdc()
35         with self.assertRaises(ValueError):
36             ModuloGCD(7,21,0).cmmdc()
37
38
39         self.assertEqual(ModuloGCD(1,10,3).cmmdc(),1)
40         self.assertEqual(ModuloGCD(2,5,2).cmmdc(),1)
41         self.assertEqual(ModuloGCD(21,1,5).cmmdc(),1)
42         self.assertEqual(ModuloGCD(14,2,7).cmmdc(),2)
43         self.assertEqual(ModuloGCD(33,11,1).cmmdc(),0)
44         self.assertEqual(ModuloGCD(15,3,2).cmmdc(),1)
45
46         """Acoperire la nivel de instructiune"""
47
48         def test_instruction_coverage(self):
49             self.assertEqual(ModuloGCD(43,17,2).cmmdc(),1)
50             self.assertEqual(ModuloGCD(7,19,3).cmmdc(),1)
51             self.assertEqual(ModuloGCD(14,7,4).cmmdc(),3)
52
53         """Acoperire la nivel de decizie"""
54
55         def test_decision_coverage(self):
56             self.assertEqual(ModuloGCD(10,10,12).cmmdc(),10)
57             self.assertEqual(ModuloGCD(14,10,5).cmmdc(),2)
58             self.assertEqual(ModuloGCD(6,9,7).cmmdc(),3)
59
60         with self.assertRaises(ValueError):
61             ModuloGCD('10',5,2).cmmdc()

```

```

62
63         """Acoperire la nivel de conditie"""
64
65         def test_condition_coverage(self):
66             with self.assertRaises(ValueError):
67                 ModuloGCD('9',4,3).cmmdc()
68             with self.assertRaises(ValueError):
69                 ModuloGCD(10,-2,3).cmmdc()
70
71
72             self.assertEqual(ModuloGCD(4,6,6).cmmdc(),2)
73             self.assertEqual(ModuloGCD(3,3,6).cmmdc(),3)
74             self.assertEqual(ModuloGCD(4,1,5).cmmdc(),1)
75             self.assertEqual(ModuloGCD(1,3,10).cmmdc(),1)
76
77         """Testarea circuitelor independente"""
78
79         def test_independent_paths(self):
80             with self.assertRaises(ValueError):
81                 ModuloGCD('10',3,6).cmmdc()
82             with self.assertRaises(ValueError):
83                 ModuloGCD(10,3.3,6).cmmdc()
84             with self.assertRaises(ValueError):
85                 ModuloGCD('10',3.3,6).cmmdc()
86
87             self.assertEqual(ModuloGCD(10,10,12).cmmdc(),10)
88             self.assertEqual(ModuloGCD(55,10,12).cmmdc(),5)
89
90         if __name__ == '__main__':
91             unittest.main()

```

AOR – ARITHMETIC OPERATOR REPLACEMENT

ASR – ASSIGEMNT OPERATOR REPLACEMENT

ROR – RELATIONAL OPERATOR REPLACEMENT

COR – CONDITIONAL OPERATOR REPLACEMENT

COI – CONDITION INVERSION

LCR – LOGICAL CONNECOR REPLACEMENT

4) Compararea cu AI

Pentru comparația cu AI codul a fost generat de ChatGPT 4o

a) Partiționare in clase de echivalenta

Putem observa ca în urma unei discuții codul generat este bun și tratează majoritatea cazurilor posibile.

```
# Valid input: distinct positive integers
def test_valid_distinct_inputs(self):
    m = ModuloGCD(48, 18, 7)
    self.assertEqual(m.cmmdc(), 6 % 7) # GCD(48,18)=6

# Valid input: a == b
def test_valid_equal_inputs(self):
    m = ModuloGCD(20, 20, 6)
    self.assertEqual(m.cmmdc(), 20 % 6) # Direct return

# Invalid input: non-integer values
def test_non_integer_input(self):
    with self.assertRaises(ValueError):
        ModuloGCD(10.5, 5, 3).cmmdc()
    with self.assertRaises(ValueError):
        ModuloGCD("10", 5, 3).cmmdc()

# Invalid input: zero or negative numbers
def test_zero_or_negative_input(self):
    with self.assertRaises(ValueError):
        ModuloGCD(0, 10, 5).cmmdc()
    with self.assertRaises(ValueError):
        ModuloGCD(-10, 20, 5).cmmdc()

# Invalid input: c <= 0 (modulo must be positive)
def test_invalid_modulo(self):
    with self.assertRaises(ValueError):
        ModuloGCD(15, 10, 0).cmmdc()
    with self.assertRaises(ValueError):
        ModuloGCD(15, 10, -3).cmmdc()
```

Valid input: distinct positive integers

def test_valid_distinct_inputs(self):

 m = ModuloGCD(48, 18, 7)

 self.assertEqual(m.cmmdc(), 6 % 7) # GCD(48,18)=6

Valid input: a == b

def test_valid_equal_inputs(self):

 m = ModuloGCD(20, 20, 6)

 self.assertEqual(m.cmmdc(), 20 % 6) # Direct return

Invalid input: non-integer values

```

def test_non_integer_input(self):
    with self.assertRaises(ValueError):
        ModuloGCD(10.5, 5, 3).cmmdc()
    with self.assertRaises(ValueError):
        ModuloGCD("10", 5, 3).cmmdc()

# Invalid input: zero or negative numbers
def test_zero_or_negative_input(self):
    with self.assertRaises(ValueError):
        ModuloGCD(0, 10, 5).cmmdc()
    with self.assertRaises(ValueError):
        ModuloGCD(-10, 20, 5).cmmdc()

# Invalid input: c <= 0 (modulo must be positive)
def test_invalid_modulo(self):
    with self.assertRaises(ValueError):
        ModuloGCD(15, 10, 0).cmmdc()
    with self.assertRaises(ValueError):
        ModuloGCD(15, 10, -3).cmmdc()

```

Pentru a face o comparație am întrebat dacă varianta generată de noi este mai indicată sau cea degerată de el, am primit următorul răspuns:

- Your version is excellent in terms of *coverage* and *structured thinking*.
- Mine is better in terms of *unit test structure* and *readability during failure*.
- The ideal is to merge your detailed input combinations with a more modular test structure.

b) Analiza valorilor the frontiera

Condul primit este corect si tratează toate cazurile posibile, codul verifică chiar și daca codul intră în loop infinit.

```

# Valid minimum boundary values
def test_min_valid_values(self):
    m = ModuloGCD(1, 1, 1)
    self.assertEqual(m.cmmdc(), 1 % 1) # 1 % 1 = 0

# a = 1, b just above 1
def test_a_one_b_two(self):
    m = ModuloGCD(1, 2, 5)
    self.assertEqual(m.cmmdc(), 1 % 5) # GCD(1,2)=1

# Very large values close to each other
def test_large_close_values(self):
    m = ModuloGCD(10**6, 10**6 - 1, 100)
    self.assertEqual(m.cmmdc(), 1 % 100) # GCD(1000000,999999)=1

# Edge: max allowed iterations hit (simulate infinite loop)
def test_potential_infinite_loop(self):
    with self.assertRaises(RuntimeError):
        # Constructing inputs that alternate without reducing quickly
        m = ModuloGCD(999_999, 1, 3)
        m.cmmdc()

# Invalid: zero values (lower than minimum valid boundary)
def test_zero_input(self):
    with self.assertRaises(ValueError):
        ModuloGCD(0, 5, 3).cmmdc()
    with self.assertRaises(ValueError):
        ModuloGCD(5, 0, 3).cmmdc()
    with self.assertRaises(ValueError):
        ModuloGCD(5, 3, 0).cmmdc()

# Invalid: negative values
def test_negative_input(self):
    with self.assertRaises(ValueError):
        ModuloGCD(-1, 5, 2).cmmdc()
    with self.assertRaises(ValueError):
        ModuloGCD(5, -1, 2).cmmdc()
    with self.assertRaises(ValueError):
        ModuloGCD(5, 2, -1).cmmdc()

```

Valid minimum boundary values

```
def test_min_valid_values(self):
```

```
    m = ModuloGCD(1, 1, 1)
```

```
    self.assertEqual(m.cmmdc(), 1 % 1) # 1 % 1 = 0
```

a = 1, b just above 1

```
def test_a_one_b_two(self):
```

```
    m = ModuloGCD(1, 2, 5)
```

```
    self.assertEqual(m.cmmdc(), 1 % 5) # GCD(1,2)=1
```

Very large values close to each other

```
def test_large_close_values(self):
```

```
    m = ModuloGCD(10**6, 10**6 - 1, 100)
```

```

self.assertEqual(m.cmmdc(), 1 % 100) # GCD(1000000,999999)=1

# Edge: max allowed iterations hit (simulate infinite loop)
def test_potential_infinite_loop(self):
    with self.assertRaises(RuntimeError):
        # Constructing inputs that alternate without reducing quickly
        m = ModuloGCD(999_999, 1, 3)
        m.cmmdc()

# Invalid: zero values (lower than minimum valid boundary)
def test_zero_input(self):
    with self.assertRaises(ValueError):
        ModuloGCD(0, 5, 3).cmmdc()
    with self.assertRaises(ValueError):
        ModuloGCD(5, 0, 3).cmmdc()
    with self.assertRaises(ValueError):
        ModuloGCD(5, 3, 0).cmmdc()

# Invalid: negative values
def test_negative_input(self):
    with self.assertRaises(ValueError):
        ModuloGCD(-1, 5, 2).cmmdc()
    with self.assertRaises(ValueError):
        ModuloGCD(5, -1, 2).cmmdc()
    with self.assertRaises(ValueError):
        ModuloGCD(5, 2, -1).cmmdc()

```

Am intrebat care implementare este mai corectă și am primit următorul răspuns:

- Your test method is very good and efficient for core boundary values.
- My version adds more structure and performance edge coverage.
- The best approach is a hybrid: keep your test values, add labeling (e.g., via subTest), and optionally include high-end boundaries.

c) Acoperirea la nivel de instructiune

```
def test_valid_shortcut_equal_values(self):
    # Covers the: if a == b: return a % c
    m = ModuloGCD(12, 12, 5)
    self.assertEqual(m.cmmmc(), 12 % 5)

def test_valid_distinct_values(self):
    # Covers full while loop (a > b and b > a cases)
    m = ModuloGCD(9, 6, 4) # GCD(9,6)=3
    self.assertEqual(m.cmmmc(), 3 % 4)

def test_valid_reverse_order(self):
    # Covers loop path where a < b initially
    m = ModuloGCD(6, 9, 5) # GCD(6,9)=3
    self.assertEqual(m.cmmmc(), 3 % 5)

def test_invalid_a_type(self):
    # Triggers validate_integer type check
    with self.assertRaises(ValueError):
        ModuloGCD("10", 5, 3).cmmmc()

def test_invalid_b_value(self):
    # Triggers validate_integer value <= 0
    with self.assertRaises(ValueError):
        ModuloGCD(10, 0, 3).cmmmc()

def test_invalid_c_value(self):
    # Covers: if c <= 0: raise ValueError
    with self.assertRaises(ValueError):
        ModuloGCD(10, 5, -1).cmmmc()

def test_loop_max_iterations_exceeded(self):
    # Forces too many iterations -> RuntimeError
    with self.assertRaises(RuntimeError):
        ModuloGCD(10**6, 1, 3).cmmmc()
```

```
def test_valid_shortcut_equal_values(self):
```

```
# Covers the: if a == b: return a % c
```

```
m = ModuloGCD(12, 12, 5) self.assertEqual(m.cmmmc(), 12 % 5)
```

```
def test_valid_distinct_values(self):
```

```
# Covers full while loop (a > b and b > a cases)
```

```
m = ModuloGCD(9, 6, 4) # GCD(9,6)=3
```

```

self.assertEqual(m.cmmdc(), 3 % 4)

def test_valid_reverse_order(self):
    # Covers loop path where a < b initially
    m = ModuloGCD(6, 9, 5) # GCD(6,9)=3
    self.assertEqual(m.cmmdc(), 3 % 5)

def test_invalid_a_type(self):
    # Triggers validate_integer type check
    with self.assertRaises(ValueError):
        ModuloGCD("10", 5, 3).cmmdc()

def test_invalid_b_value(self):
    # Triggers validate_integer value <= 0
    with self.assertRaises(ValueError):
        ModuloGCD(10, 0, 3).cmmdc()

def test_invalid_c_value(self):
    # Covers: if c <= 0: raise ValueError
    with self.assertRaises(ValueError):
        ModuloGCD(10, 5, -1).cmmdc()

def test_loop_max_iterations_exceeded(self):
    # Forces too many iterations -> RuntimeError
    with self.assertRaises(RuntimeError):
        ModuloGCD(10**6, 1, 3).cmmdc()

```

Am intrebat care implementare este mai corecta si am primit urmatorul raspuns:

- Your test does partial instruction coverage: it ensures that the loop and modulo return logic are exercised.
- My version is more comprehensive and designed specifically to hit 100% of the executable statements in the method.

d) Acoperire la nivel de conditie

```
# 1. validate_integer: isinstance is False
def test_invalid_type(self):
    with self.assertRaises(ValueError):
        ModuloGCD("5", 3, 2).cmmdc()

# 2. validate_integer: value <= 0
def test_invalid_value_zero(self):
    with self.assertRaises(ValueError):
        ModuloGCD(0, 3, 2).cmmdc()

# 3. c <= 0
def test_invalid_modulo(self):
    with self.assertRaises(ValueError):
        ModuloGCD(10, 5, -1).cmmdc()

# 4. a == b → True (shortcut path)
def test_equal_values(self):
    m = ModuloGCD(12, 12, 7)
    self.assertEqual(m.cmmdc(), 12 % 7)

# 5. a != b → True (loop runs)
def test_distinct_values(self):
    m = ModuloGCD(8, 5, 3) # GCD=1
    self.assertEqual(m.cmmdc(), 1 % 3)

# 6. a > b inside loop
def test_a_greater_than_b(self):
    m = ModuloGCD(20, 5, 6)
    self.assertEqual(m.cmmdc(), 5 % 6)

# 7. a < b inside loop
def test_a_less_than_b(self):
    m = ModuloGCD(6, 9, 5)
    self.assertEqual(m.cmmdc(), 3 % 5)

# 8. iterations > max_iterations
def test_max_iterations_exceeded(self):
    with self.assertRaises(RuntimeError):
        ModuloGCD(10**6, 1, 3).cmmdc()
```

```
# 1.validate_integer: isinstance is False
def test_invalid_type(self):
    with self.assertRaises(ValueError):
        ModuloGCD("5", 3, 2).cmmdc()
```

```
# 2. validate_integer: value <= 0

def test_invalid_value_zero(self):
    with self.assertRaises(ValueError):
        ModuloGCD(0, 3, 2).cmmdc()
```

3. $c \leq 0$

```
def test_invalid_modulo(self):  
    with self.assertRaises(ValueError):  
        ModuloGCD(10, 5, -1).cmmdc()
```

4. $a == b \rightarrow \text{True}$ (shortcut path)

```
def test_equal_values(self):  
    m = ModuloGCD(12, 12, 7)  
    self.assertEqual(m.cmmdc(), 12 % 7)
```

5. $a \neq b \rightarrow \text{True}$ (loop runs)

```
def test_distinct_values(self):  
    m = ModuloGCD(8, 5, 3) # GCD=1  
    self.assertEqual(m.cmmdc(), 1 % 3)
```

6. $a > b$ inside loop

```
def test_a_greater_than_b(self):  
    m = ModuloGCD(20, 5, 6)  
    self.assertEqual(m.cmmdc(), 5 % 6)
```

7. $a < b$ inside loop

```
def test_a_less_than_b(self):  
    m = ModuloGCD(6, 9, 5)  
    self.assertEqual(m.cmmdc(), 3 % 5)
```

8. iterations > max_iterations

```
def test_max_iterations_exceeded(self):  
    with self.assertRaises(RuntimeError):  
        ModuloGCD(10**6, 1, 3).cmmdc()
```

Am intrebat care implementare este mai corecta si am primit urmatorul raspuns:

- Your version is very solid and covers almost all key branches, with well-labeled, focused test cases.
- To reach full condition-level coverage, you just need to add two cases:
 1. Invalid $c \leq 0$
 2. Exceeded iteration limit

e) Testarea circuitelor independente

```
# Path 1: All valid, a == b → returns a % c without loop
def test_path_equal_values(self):
    m = ModuloGCD(10, 10, 4)
    self.assertEqual(m.cmmdc(), 10 % 4)

# Path 2: Loop runs, a > b on first iteration
def test_path_a_greater_than_b(self):
    m = ModuloGCD(15, 5, 6) # GCD = 5
    self.assertEqual(m.cmmdc(), 5 % 6)

# Path 3: Loop runs, a < b on first iteration
def test_path_a_less_than_b(self):
    m = ModuloGCD(4, 10, 5) # GCD = 2
    self.assertEqual(m.cmmdc(), 2 % 5)

# Path 4: Invalid c (c <= 0)
def test_path_invalid_c(self):
    with self.assertRaises(ValueError):
        ModuloGCD(10, 5, 0).cmmdc()

# Path 5: Invalid input type or <= 0 value
def test_path_invalid_type_or_value(self):
    with self.assertRaises(ValueError):
        ModuloGCD("10", 5, 3).cmmdc()
    with self.assertRaises(ValueError):
        ModuloGCD(10, -5, 3).cmmdc()

# Path 6: Loop runs too many iterations → RuntimeError
def test_path_max_iterations_exceeded(self):
    with self.assertRaises(RuntimeError):
        ModuloGCD(999999, 1, 3).cmmdc()
```

Path 1: All valid, $a == b \rightarrow$ returns $a \% c$ without loop

```
def test_path_equal_values(self):
    m = ModuloGCD(10, 10, 4)
    self.assertEqual(m.cmmdc(), 10 % 4)
```

Path 2: Loop runs, $a > b$ on first iteration

```
def test_path_a_greater_than_b(self):
    m = ModuloGCD(15, 5, 6) # GCD = 5
```

```

self.assertEqual(m.cmmmdc(), 5 % 6)

# Path 3: Loop runs, a < b on first iteration
def test_path_a_less_than_b(self):
    m = ModuloGCD(4, 10, 5) # GCD = 2
    self.assertEqual(m.cmmmdc(), 2 % 5)

# Path 4: Invalid c (c <= 0)
def test_path_invalid_c(self):
    with self.assertRaises(ValueError):
        ModuloGCD(10, 5, 0).cmmmdc()

# Path 5: Invalid input type or <= 0 value
def test_path_invalid_type_or_value(self):
    with self.assertRaises(ValueError):
        ModuloGCD("10", 5, 3).cmmmdc()
    with self.assertRaises(ValueError):
        ModuloGCD(10, -5, 3).cmmmdc()

# Path 6: Loop runs too many iterations → RuntimeError
def test_path_max_iterations_exceeded(self):
    with self.assertRaises(RuntimeError):
        ModuloGCD(999999, 1, 3).cmmmdc()

```

Am intrebat care implementare este mai corecta si am primit urmatorul raspuns:

Your version covers about 4 out of 6 key independent paths. It's well-written and clear, but to fully meet the basis path testing goal, you just need to:

1. Add a test for $a < b$ at loop start
2. Add a test for invalid $c \leq 0$
3. Add a test to trigger `RuntimeError` (long loop)

MUTANȚI

De asemenea, în urma codului primit am testat mutanții și am obținut acest scor:

```
[*] Mutation score [0.68952 s]: 86.4%  
  - all: 23  
  - killed: 19 (82.6%)  
  - survived: 3 (13.0%)  
  - incompetent: 1 (4.3%)  
  - timeout: 0 (0.0%)  
  
C:\Users\maria>
```