Lab Report

By: Jahmil Ally (501045419)

Observations and Analysis

Q1:

Truth Table for Clauses in *p*

a	b	c	¬b	¬b ∨ c	p = a ∧ (¬b ∨ c)
F	F	F	T	T	F
F	F	T	T	T	F
F	T	F	F	F	F
F	T	T	F	T	F
T	F	F	T	T	T
T	F	T	T	T	T
T	T	F	F	F	F
T	T	T	F	T	T

Conditions for Clause Determination

A clause determines p when flipping the clause value changes the value of p, while keeping other variables constant.

- Clause a:
  - a determines p when ¬b∨c is true, Since p=a∧(¬b∨c).
  - Key rows: When ¬b∨c=1 p changes as a changes.
  - Conditions: a determines p when ¬b∨c=1.
- Clause b:
  - b appears as ¬b, so its effect is analyzed in ¬b∨c.
  - Key rows: When a=1 and c=0, changing b affects p.
  - Conditions: b determines p when a=1,c=0.
- Clause c:
  - c directly affects ¬b∨c influencing p.
  - Key rows: When a=1 and b=1, changing c affects p.
  - Conditions: c determines p when a = 1, b = 1.

Output:

```
Truth Table:
a       | b      | c       | p
--------------------------
0       | 0       | 0       | False
0       | 0       | 1       | False
0       | 1       | 0       | False
0       | 1       | 1       | False
1       | 0       | 0       | True
1       | 0       | 1       | True
1       | 1       | 0       | False
1       | 1       | 1       | True
```

```
Received test ids from temp file.
test_clauses (Q1_test.TestQ1.test_clauses) ... ok
test_coverage_criteria (Q1_test.TestQ1.test_coverage_criteria) ... ok
test_evaluate_predicate (Q1_test.TestQ1.test_evaluate_predicate) ... ok
test_generate_truth_table (Q1_test.TestQ1.test_generate_truth_table) ... ok

----------------------------------------------------------------------
Ran 4 tests in 0.000s

OK
Finished running tests!
```

Truth Table:

Row#	a	b	c	P	Pa	Pb	Pc
1	T	T	T	T	T		T
2	T	T				T	T
3	T		T	T	T		
4	T			T	T	T	
5		T	T		T		
6		T					
7			T		T		
8					T		

The following result for GACC is based on the truth table on the right:

Major Clause	Set of possible tests
a	(1,5), (1,7), (1,8), (3,5), (3,7), (3,8), (4,5), (4,7), (4,8)
b	(2,4)
c	(1,2)

The following result for CACC is based on the truth table on the right:

Major Clause	Set of possible tests
a	(1,5), (1,7), (1,8), (3,5), (3,7), (3,8), (4,5), (4,7), (4,8)
b	(2,4)
c	(1,2)

The following result for RACC is based on the truth table on the right:

Major Clause	Set of possible tests
a	(1,5), (3,7), (4,8)
b	(2,4)
c	(1,2)

The following result for GICC is based on the truth table on the right:

Major Clause	Set of possible tests	
a	No feasible pairs for P = T	P = F: (2,6)
b	P = T: (1,3)	P = F: (5,7), (5,8), (6,7), (6,8)
c	P = T: (3,4)	P = F: (5,6), (5,8), (7,6), (7,8)

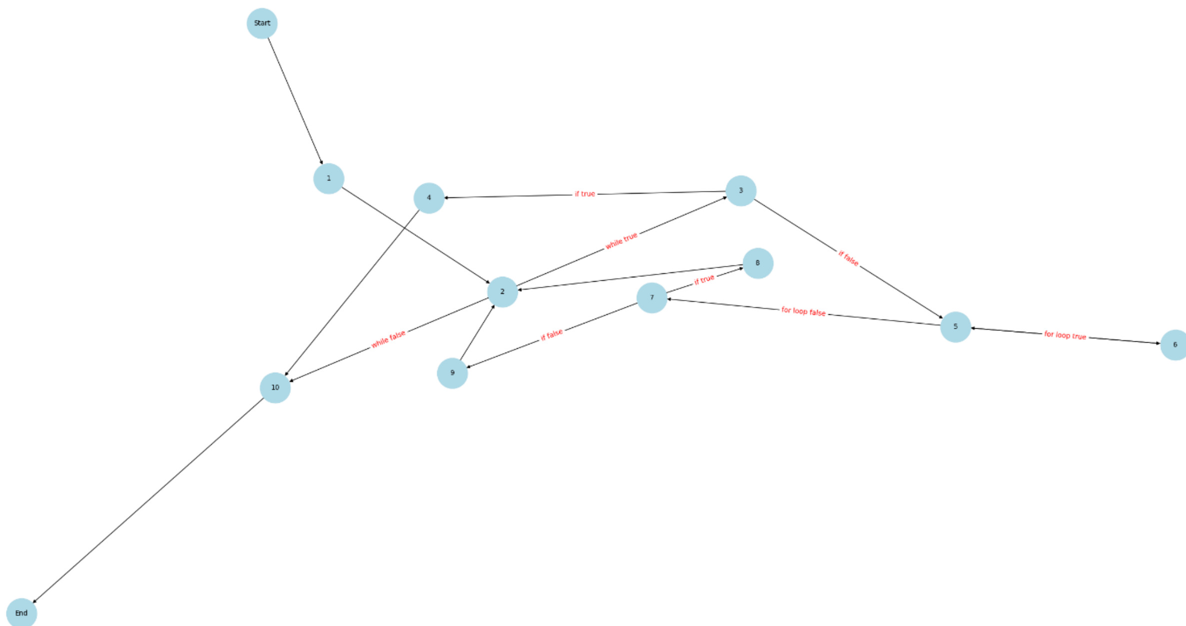
Major Clause	Set of possible tests	
a	No feasible pairs for P = T	P = F: (2,6)
b	P = T: (1,3)	P = F: (5,7), (6,8)
c	P = T: (3,4)	P = F: (5,6), (7,8)

Q2:

Data Flow Graph:

Node	Lines
1	1
2	2
3	3
4	4, 5

5	6
6	7
7	8
8	9
9	10,11
10	12



Def(n)/ Use(n)

Node	Def(n)	Use(n)
Start	y	
1	x	y
2		x
3		x, y
4	x	x
5	z	x, z
6	x	x, z
7		x
8	y	y
9	y	y
10		x, y
End		

DU Pairs:

Variables	Defined	Used
X	1, 4, 6	2, 3, 4, 5, 6, 7, 10
Y	Start,8, 9	1, 3, 8, 9, 10
Z	5	5, 6

No infeasible paths

Output:

```
PS C:\Users\jazza\Documents\Coding Projects\COE891Labs\lab3> & C:/Users/jazza/anacon
2, 3
Visited Nodes: ['Start', '1', '2', '3', '5', '7', '9', '2', '3', '4', '10', 'End']
```

Q3:

Reachability Predicates

Predicate	Condition	Effect on Reachability
P1	(s1 <= 0) or (s2 <= 0) or (s3 <= 0)	If True, return INVALID, function exits.
P2	(s1 + s2 <= s3) or (s2 + s3 <= s1) or (s1 + s3 <= s2)	If True, return INVALID, function exits.
P3	s1 == s2 == s3	If True, return EQUILATERAL, function exits.
P4	(s1 == s2) or (s2 == s3) or (s1 == s3)	If True, return ISOSCELES, function exits.
P5	Default case: None of the above predicates are true (scalene triangle).	True: All sides are different.

Test Requirements (PC)

Test Case ID	Input (s1, s2, s3)	Expected Output	Predicate Coverage Satisfied
TC1	-80, 2, 3	Triangle.INVALID	P1 = True, P2 = False
TC2	100, 2, 3	Triangle.INVALID	P1 = False, P2 = True
TC3	3, 3, 3	Triangle.EQUILATERAL	P1 = False, P2 = False, P3 = True
TC4	3, 3, 5	Triangle.ISOSCELES	P1 = False, P2 = False, P3 = False, P4 = True
TC5	3, 4, 5	Triangle.SCALENE	P1 = False, P2 = False, P3 = False, P4 = False, P5 = True

Test Requirements (CC)

Predicates and Atomic Conditions:

1. P1: Non-positive sides:
   a. Atomic Conditions:
      - C1.1: s1 <= 0
      - C1.2: s2 <= 0

- C1.3: s3 <= 0

b. Predicate:

- C1 = C1.1 OR C1.2 OR C1.3

2. P2: Triangle inequality

a. Atomic Conditions:

- C2.1: s1 + s2 <= s3
- C2.2: s2 + s3 <= s1
- C2.3: s1 + s3 <= s2

b. Predicate:

- C2 = C2.1 OR C2.2 OR C2.3

3. P3: Equilateral triangle

a. Atomic Conditions:

- C3.1: s1 == s2
- C3.2: s2 == s3

b. Predicate:

- C3 = C3.1 AND C3.2

4. P4: Isosceles triangle

a. Atomic Conditions:

- C4.1: s1 == s2
- C4.2: s2 == s3
- C4.3: s1 == s3

b. Predicate: C4 = C4.1 OR C4.2 OR C4.3

5. P5: Scalene

a. Predicate: !C1 OR !C2 OR!C3 OR !C4

Predicate	Atomic Conditions	TRs (True/False for each condition)
P1	C1.1, C1.2, C1.3	{[True, False, False], [False, True, False], [False, False, True]}
P2	C2.1, C2.2, C2.3	{[True, False, False], [False, True, False], [False, False, True]}
P3	C3.1, C3.2	{[True, True] , [True, False], [False, True], [False, False]}
P4	C4.1, C4.2, C4.3	{[True, False, False], [False, True, False], [False, False, True]}
P5	C1, C2, C3, C4	{[True, False, False, False], [False, True, False, False], [False, False, True, False], [False, False, False, True]}

Test Case ID	Input (s1, s2, s3)	Expected Output	Conditions Satisfied

TC1	0, 2, 3	Triangle.INVALID	C1.1 = True, C1.2 = False, C1.3 = False
TC2	1, 0, 3	Triangle.INVALID	C1.1 = False, C1.2 = True, C1.3 = False
TC3	1, 2, 0	Triangle.INVALID	C1.1 = False, C1.2 = False, C1.3 = True
TC4	1, 2, 3	Triangle.INVALID	C2.1 = True, C2.2 = False, C2.3 = False
TC5	3, 1, 2	Triangle.INVALID	C2.1 = False, C2.2 = True, C2.3 = False
TC6	2, 3, 1	Triangle.INVALID	C2.1 = False, C2.2 = False, C2.3 = True
TC7	3, 3, 3	Triangle.EQUILATERAL	C3.1 = True, C3.2 = True
TC8	3, 3, 5	Triangle.ISOSCELES	C4.1 = True, C4.2 = False, C4.3 = False
TC9	3, 5, 3	Triangle.ISOSCELES	C4.1 = False, C4.2 = True, C4.3 = False
TC10	5, 3, 3	Triangle.ISOSCELES	C4.1 = False, C4.2 = False, C4.3 = True
TC11	3, 4, 5	Triangle.SCALENE	All conditions False

Determination predicates

Predicate	Simplified Expression	Meaning
P1	C1	At least one side is non-positive.
P2	C2	Triangle inequality is violated.
P3	C3	All three sides are equal (equilateral triangle).
P4	C4	At least two sides are equal (isosceles triangle).
P5	!C1 OR !C2 OR !C3 OR !C4	None of the above predicates are true (scalene triangle).

Test Requirements (CACC/RACC)

P1:

Major Clause	Other Clauses	Predicate Outcome	Test Case Input (s1, s2, s3)
C1.1	C1.2 = False, C1.3 = False	True	0, 2, 3
C1.1	C1.2 = False, C1.3 = False	False	1, 2, 3
C1.2	C1.1 = False, C1.3 = False	True	1, 0, 3
C1.2	C1.1 = False, C1.3 = False	False	1, 2, 3
C1.3	C1.1 = False, C1.2 = False	True	1, 2, 0
C1.3	C1.1 = False, C1.2 = False	False	1, 2, 3

P2:

Major Clause	Other Clauses	Predicate Outcome	Test Case Input (s1, s2, s3)
C2.1	C2.2 = False, C2.3 = False	True	1, 2, 3
C2.1	C2.2 = False, C2.3 = False	False	3, 4, 5
C2.2	C2.1 = False, C2.3 = False	True	3, 1, 2
C2.2	C2.1 = False, C2.3 = False	False	3, 4, 5
C2.3	C2.1 = False, C2.2 = False	True	2, 3, 1
C2.3	C2.1 = False, C2.2 = False	False	3, 4, 5

P3:

Major Clause	Other Clauses	Predicate Outcome	Test Case Input (s1, s2, s3)
C3.1	C3.2 = True	True	3, 3, 3
C3.1	C3.2 = True	False	3, 3, 5
C3.2	C3.1 = True	True	3, 3, 3
C3.2	C3.1 = True	False	3, 5, 3

P4:

Major Clause	Other Clauses	Predicate Outcome	Test Case Input (s1, s2, s3)
C4.1	C4.2 = False, C4.3 = False	True	3, 3, 5
C4.1	C4.2 = False, C4.3 = False	False	3, 4, 5
C4.2	C4.1 = False, C4.3 = False	True	3, 5, 3
C4.2	C4.1 = False, C4.3 = False	False	3, 4, 5
C4.3	C4.1 = False, C4.2 = False	True	5, 3, 3
C4.3	C4.1 = False, C4.2 = False	False	3, 4, 5

Infeasible Requirements

1. P3 (Equilateral Triangle) and P4 (Isosceles Triangle): If P3 (all sides are equal) is true, then P4 (at least two sides are equal) is also true. However, the reverse is not always true. Testing for P3 being false while P4 is true is feasible, but testing for P3 being true while P4 is false is infeasible.
2. P1 (Non-positive sides) and P2 (Triangle Inequality): If P1 is true (one or more sides are non-positive), then P2 (triangle inequality) is irrelevant because the triangle is already invalid. Testing for P1 being true while P2 is false is infeasible.
3. P5 (Scalene Triangle) and P3/P4: If P5 (all sides are different) is true, then both P3 (all sides equal) and P4 (at least two sides equal) must be false. Testing for P5 being true while P3 or P4 is true is infeasible.

Appendix:

Q1:

Q1.py > ...

```python
from itertools import product

# Predicate: p = a ∧ (~ b ∨ c)
def evaluate_predicate(a, b, c):
    return a and (not b or c)

# Generate truth table for all combinations of (a, b, c)
def generate_truth_table():
    return [(a, b, c, evaluate_predicate(a, b, c)) for a, b, c in product([False, True], repeat=3)]

# Print the truth table
def print_truth_table():
    truth_table = generate_truth_table()
    print("Truth Table:")
    print("a      | b     | c       | p")
    print("--------------------------")
    for row in truth_table:
        print(f"{row[0]:<6} | {row[1]:<6} | {row[2]:<6} | {row[3]}")

# Call the function to print the truth table
if __name__ == "__main__":
    print_truth_table()
```

Q1_test.py > ...

```python
import unittest
from Q1 import evaluate_predicate, generate_truth_table
class TestQ1(unittest.TestCase):
    # Test the evaluate_predicate function
    def test_evaluate_predicate(self):
        self.assertTrue(evaluate_predicate(True, False, False))  # a=True, b=False, c=False
        self.assertFalse(evaluate_predicate(False, False, False))  # a=False, b=False, c=False
        self.assertTrue(evaluate_predicate(True, True, True))  # a=True, b=True, c=True
        self.assertFalse(evaluate_predicate(True, True, False))  # a=True, b=True, c=False

    # Test the truth table generation
    def test_generate_truth_table(self):
        truth_table = generate_truth_table()
        expected_table = [
            (False, False, False, False),
            (False, False, True, False),
            (False, True, False, False),
            (False, True, True, False),
            (True, False, False, True),
            (True, False, True, True),
            (True, True, False, False),
            (True, True, True, True)
        ]
        self.assertEqual(truth_table, expected_table)


    # Test Conditional Clauses
    def test_clauses(self):
        # Clause a: ((not b) V c) = 1
        ##False
        self.assertFalse(evaluate_predicate(False, False, False))
        self.assertFalse(evaluate_predicate(False, False, True))
        self.assertFalse(evaluate_predicate(False, True, True))
        ##True
        self.assertTrue(evaluate_predicate(True, False, False))
        self.assertTrue(evaluate_predicate(True, False, True))
        self.assertTrue(evaluate_predicate(True, True, True))

        # Clause b: a = 1, c = 0
        ##False
        self.assertFalse(evaluate_predicate(True, True, False))
        ##True
        self.assertTrue(evaluate_predicate(True, False, False))

        # Clause c: a = 1, b = 1
        ##False
        self.assertFalse(evaluate_predicate(True, True, False))
        ##True
        self.assertTrue(evaluate_predicate(True, True, True))
```

```python
        # Test Major Clause Coverage
    def test_coverage_criteria(self):
        # GACC (General Active Clause Coverage) == CACC (Correlated Active Clause Coverage)
        #Clause a
        self.assertNotEqual(evaluate_predicate(True, True, True), evaluate_predicate(False, True, True))
        self.assertNotEqual(evaluate_predicate(True, True, True), evaluate_predicate(False, False,True))
        self.assertNotEqual(evaluate_predicate(True, True, True), evaluate_predicate(False, False, False))
        self.assertNotEqual(evaluate_predicate(True, False, True), evaluate_predicate(False, True, True))
        self.assertNotEqual(evaluate_predicate(True, False, True), evaluate_predicate(False, False, True))
        self.assertNotEqual(evaluate_predicate(True, False, True), evaluate_predicate(False, False, False))
        self.assertNotEqual(evaluate_predicate(True, False, False), evaluate_predicate(False, True, True))
        self.assertNotEqual(evaluate_predicate(True, False, False), evaluate_predicate(False, False, True))
        self.assertNotEqual(evaluate_predicate(True, False, False), evaluate_predicate(False, False, False))
        #Clause b
        self.assertNotEqual(evaluate_predicate(True, True, False), evaluate_predicate(True, False, False))
        #Clause c
        self.assertNotEqual(evaluate_predicate(True, True, True), evaluate_predicate(True, True, False))

        #  RACC (Restricted Active Clause Coverage)
        #Clause a
        self.assertNotEqual(evaluate_predicate(True, True, True), evaluate_predicate(False, True, True))
        self.assertNotEqual(evaluate_predicate(True, False, True), evaluate_predicate(False, False, True))
        self.assertNotEqual(evaluate_predicate(True, False, False), evaluate_predicate(False, False, False))
        #Clause b
        self.assertNotEqual(evaluate_predicate(True, True, False), evaluate_predicate(True, False, False))
        #Clause c
        self.assertNotEqual(evaluate_predicate(True, True, True), evaluate_predicate(True, True, False))

        # GICC (General Inactive Clause Coverage) contains RICC (Restricted Inactive Clause Coverage)
        #Clause a
        ##False
        self.assertEqual(evaluate_predicate(False, True, False),evaluate_predicate(True,True, False))
        #Clause b
        ##True
        self.assertEqual(evaluate_predicate(True, True, True),evaluate_predicate(True, False, True))
        ##False
        self.assertEqual(evaluate_predicate(False, True, True),evaluate_predicate(False, False, True))
        self.assertEqual(evaluate_predicate(False, True, True),evaluate_predicate(False, False, False))
        self.assertEqual(evaluate_predicate(False, True, False),evaluate_predicate(False, False, True))
        self.assertEqual(evaluate_predicate(False, True, False),evaluate_predicate(False, False, False))
        #Clause c
        ##True
        self.assertEqual(evaluate_predicate(True, False, True),evaluate_predicate(True, False, False))
        ##False
        self.assertEqual(evaluate_predicate(False, True, True),evaluate_predicate(False, True, False))
        self.assertEqual(evaluate_predicate(False, True, True),evaluate_predicate(False, False, False))
        self.assertEqual(evaluate_predicate(False, True, False),evaluate_predicate(False, False, True))
        self.assertEqual(evaluate_predicate(False,False, True),evaluate_predicate(False, False, False))
if __name__ == "__main__":
    unittest.main()
```

Q2:

Q2.py > ...

```python
import networkx as nx
import matplotlib.pyplot as plt

# Define the control flow graph (CFG) for the given program
# Simulate the program execution and track visited nodes
def simulate_program(y):
    visited_nodes = []
    visited_nodes.append("Start")  # Start of the program

    # Node 1: int x = y;
    visited_nodes.append("1")
    x = y

    # Node 2: check while condition (x < 100)
    visited_nodes.append("2")
    while x < 100:
        # Node 3: Enter While loop
        visited_nodes.append("3")
        if x < y:
            # Node 4: x <  y is true
            visited_nodes.append("4")
            x += 1
            break

        # Node 5: Check for loop condition (int z = 1; z < x; z++)
        visited_nodes.append("5")
        for z in range(1, x):
            # Node 6: for loop
            visited_nodes.append("6")
            x += z
            # Node 5 (Loop back): Check for loop condition (int z = 1; z < x; z++)
            visited_nodes.append("5")

        # Node 7:check if condition (x > 5)
        visited_nodes.append("7")
        if x > 5:
            # Node 8: x > 5 is true
            visited_nodes.append("8")
            y += 1
        else:
            # Node 9: x > 5 is false
            visited_nodes.append("9")
            y += 2

        # Node 2: check while condition (x < 100)
        visited_nodes.append("2")

    # Node 12: System.out.println(x + ',' + y);
    visited_nodes.append("10")
    print(f"{x}, {y}")

    # End of the program
    visited_nodes.append("End")
    print(f"Visited Nodes: {visited_nodes}")
    return visited_nodes
```

```python
def create_cfg():
    """
    Generates a simplified Control Flow Graph (CFG) for the power function.
    """
    cfg = nx.DiGraph()

    # Nodes representing different parts of the program
    cfg.add_nodes_from([
        "Start","1", "2", "3", "4", "5", "6", "7", "8", "9", "10", "End"
    ])

    # Edges representing control flow
    edges = [
        ("Start","1"),
        ("1", "2"),
        ("2", "3"),  # while true
        ("2", "10"),  # while false
        ("3", "4"),  # if true
        ("3", "5"),  # if false
        ("4", "10"), # break
        ("5", "6"),  # for loop true
        ("5", "7"),  # for loop false
        ("6", "5"),  # Loop back
        ("7", "8"),  # if true
        ("7", "9"),  # if false
        ("8", "2"),  # Loop back
        ("9", "2"),  # Loop back
        ("10", "End")
    ]

    cfg.add_edges_from(edges)
    return cfg

def draw_cfg(cfg):
    """
    Draws the generated CFG using NetworkX and Matplotlib.
    """
    pos = nx.spring_layout(cfg)  # Position nodes
    plt.figure(figsize=(10, 6))
    nx.draw(cfg, pos, with_labels=True, node_color='lightblue', edge_color='black', font_size=10, node_size=2000)

    # Define edge labels
    edge_labels = {
        ("2", "3"): "while true",
        ("2", "10"): "while false",
        ("3", "4"): "if true",
        ("3", "5"): "if false",
        ("5", "6"): "for loop true",
        ("5", "7"): "for loop false",
        ("7", "8"): "if true",
        ("7", "9"): "if false"
    }

    # Draw edge labels
    nx.draw_networkx_edge_labels(cfg, pos, edge_labels=edge_labels, font_color='red')

    plt.title("Control Flow Graph")
    plt.show()

# Main function to print necessary outputs
if __name__ == "__main__":
    draw_cfg(create_cfg())  # Draw the CFG for the program
    simulate_program(1)  # Simulate the program execution with y=1
```

Q2_test.py > ...

```python
import unittest
from Q2 import create_cfg, simulate_program


class TestQ2(unittest.TestCase):
    test_cfg (Not yet run). ol flow graph (CFG)
    def test_cfg(self):
        G = create_cfg()
        self.assertTrue(G.has_edge("Start", "1"))
        self.assertTrue(G.has_edge("10", "End"))
        self.assertFalse(G.has_edge("1", "End"))

    def test_all_def_coverage(self):
        # Initialize a set to collect all unique visited nodes
        unique_visited_nodes = set()

        # Run the program with different inputs and collect visited nodes
        for y in [1,10]:
            visited_nodes = simulate_program(y)
            unique_visited_nodes.update(visited_nodes)  # Add visited nodes to the set

        # Check if all required nodes are in the unique visited nodes
        required_nodes = {"1", "4", "5", "6", "8", "9"}
        self.assertTrue(required_nodes.issubset(unique_visited_nodes),
                        f"Missing nodes: {required_nodes - unique_visited_nodes}")

    def test_all_use_coverage(self):
        # Initialize a set to collect all unique visited nodes
        unique_visited_nodes = set()

        # Run the program with different inputs and collect visited nodes
        for y in [1,10]:
            visited_nodes = simulate_program(y)
            unique_visited_nodes.update(visited_nodes)  # Add visited nodes to the set

        # Check if all required nodes are in the unique visited nodes
        required_nodes = {"1", "2", "3", "4", "5", "6", "7", "8", "9", "10"}
        self.assertTrue(required_nodes.issubset(unique_visited_nodes),
                        f"Missing nodes: {required_nodes - unique_visited_nodes}")


    def test_all_du_paths_coverage(self):
        # Initialize a set to collect all unique transitions (edges)
        unique_transitions = set()

        # Run the program with different inputs and collect visited transitions
        for y in [1, 10]:
            visited_nodes = simulate_program(y)
            # Collect transitions as pairs of consecutive nodes
            transitions = [(visited_nodes[i], visited_nodes[i + 1]) for i in range(len(visited_nodes) - 1)]
            unique_transitions.update(transitions)  # Add transitions to the set

        # Define the required transitions (edges) for all-DU-paths coverage
        required_transitions = {
            ("Start", "1"), ("1", "2"), ("2", "3"), ("2", "10"), ("3", "4"), ("3", "5"),
            ("4", "10"), ("5", "6"), ("5", "7"), ("6", "5"), ("7", "8"), ("7", "9"),
            ("8", "2"), ("9", "2"), ("10", "End")
        }

        # Check if all required transitions are in the unique transitions
        self.assertTrue(required_transitions.issubset(unique_transitions),
                        f"Missing transitions: {required_transitions - unique_transitions}")

if __name__ == "__main__":
    unittest.main()
```

Q3:

Q3.py > ...

```python
from enum import Enum

class Triangle(Enum):
    SCALENE = "SCALENE"
    ISOSCELES = "ISOSCELES"
    EQUILATERAL = "EQUILATERAL"
    INVALID = "INVALID"

def triangle_type(s1, s2, s3):
    # Reachability predicates
    if s1 <= 0 or s2 <= 0 or s3 <= 0:  # P1: Non-positive sides
        return Triangle.INVALID
    if s1 + s2 <= s3 or s2 + s3 <= s1 or s1 + s3 <= s2:  # P2: Triangle inequality
        return Triangle.INVALID
    if s1 == s2 == s3:  # P3: Equilateral triangle
        return Triangle.EQUILATERAL
    if s1 == s2 or s2 == s3 or s1 == s3:  # P4: Isosceles triangle
        return Triangle.ISOSCELES
    return Triangle.SCALENE  # P5: Scalene triangle
```

Q3_test.py > ...

```python
import unittest
from Q3 import triangle_type, Triangle

class TestTriangleType(unittest.TestCase):
    # Test cases for Predicate Coverage (PC)
    def test_predicate_coverage(self):
        # P1: Non-positive side
        self.assertEqual(triangle_type(-80, 2, 3), Triangle.INVALID)  # P1 = True
        # P2: Triangle inequality
        self.assertEqual(triangle_type(100, 2, 3), Triangle.INVALID)  # P2 = True
        # P3: Equilateral triangle
        self.assertEqual(triangle_type(3, 3, 3), Triangle.EQUILATERAL)  # P3 = True
        # P4: Isosceles triangle
        self.assertEqual(triangle_type(3, 3, 5), Triangle.ISOSCELES)  # P4 = True
        # P5: Scalene triangle
        self.assertEqual(triangle_type(3, 4, 5), Triangle.SCALENE)  # P5 = True

    # Test cases for Complete Condition Coverage (CC)
    def test_complete_condition_coverage(self):
        # P1: Non-positive side
        self.assertEqual(triangle_type(0, 2, 3), Triangle.INVALID)  # C1.1 = True
        self.assertEqual(triangle_type(1, 0, 3), Triangle.INVALID)  # C1.2 = True
        self.assertEqual(triangle_type(1, 2, 0), Triangle.INVALID)  # C1.3 = True

        # P2: Triangle inequality
        self.assertEqual(triangle_type(1, 2, 3), Triangle.INVALID)  # C2.1 = True
        self.assertEqual(triangle_type(3, 1, 2), Triangle.INVALID)  # C2.2 = True
        self.assertEqual(triangle_type(2, 3, 1), Triangle.INVALID)  # C2.3 = True

        # P3: Equilateral triangle
        self.assertNotEqual(triangle_type(3, 3, 5), Triangle.EQUILATERAL)  # C3.1 = True
        self.assertNotEqual(triangle_type(5, 3, 3), Triangle.EQUILATERAL) # C3.2 = True

        # P4: Isosceles triangle
        self.assertEqual(triangle_type(3, 3, 5), Triangle.ISOSCELES)  # C4.1 = True
        self.assertEqual(triangle_type(5, 3, 3), Triangle.ISOSCELES)  # C4.2 = True
        self.assertEqual(triangle_type(3, 5, 3), Triangle.ISOSCELES)  # C4.3 = True

        # P5: Scalene triangle
        self.assertEqual(triangle_type(3, 4, 5), Triangle.SCALENE)  # All conditions False

    # Test cases for Correlated Active Clause Coverage (CACC)
    def test_cacc(self):
        # P1: Non-positive side
        self.assertEqual(triangle_type(0, 2, 3), Triangle.INVALID)  # C1.1 = True
        self.assertEqual(triangle_type(1, 2, 3), Triangle.INVALID)  # C1.1 = False
        self.assertEqual(triangle_type(1, 0, 3), Triangle.INVALID)  # C1.2 = True
        self.assertEqual(triangle_type(1, 2, 0), Triangle.INVALID)  # C1.3 = True

        # P2: Triangle inequality
        self.assertEqual(triangle_type(1, 2, 3), Triangle.INVALID)  # C2.1 = True
        self.assertEqual(triangle_type(3, 4, 5), Triangle.SCALENE)  # C2.1 = False
        self.assertEqual(triangle_type(3, 1, 2), Triangle.INVALID)  # C2.2 = True
        self.assertEqual(triangle_type(2, 3, 1), Triangle.INVALID)  # C2.3 = True

        # P3: Equilateral triangle
        self.assertEqual(triangle_type(3, 3, 3), Triangle.EQUILATERAL)  # C3.1 = True, C3.2 = True
        self.assertNotEqual(triangle_type(3, 3, 5), Triangle.EQUILATERAL)  # C3.1 = True, C3.2 = False

        # P4: Isosceles triangle
        self.assertEqual(triangle_type(3, 3, 5), Triangle.ISOSCELES)  # C4.1 = True
        self.assertEqual(triangle_type(3, 5, 3), Triangle.ISOSCELES)  # C4.2 = True
        self.assertEqual(triangle_type(5, 3, 3), Triangle.ISOSCELES)  # C4.3 = True


if __name__ == "__main__":
    unittest.main()
```