IT313 - Software Engineering
Lab - 9 Mutation Testing

Name: Harsh Rajwani
ID : 202201027

Q.1. The code below is part of a method in the ConvexHull class in the VMAP system. The following is a small fragment of a method in the ConvexHull class. For the purposes of this exercise, you do not need to know the intended function of the method. The parameter p is a Vector of Point objects, p.size() is the size of the vector p, (p.get(i)).x is the x component of the ith point appearing in p, similarly for (p.get(i)).y.

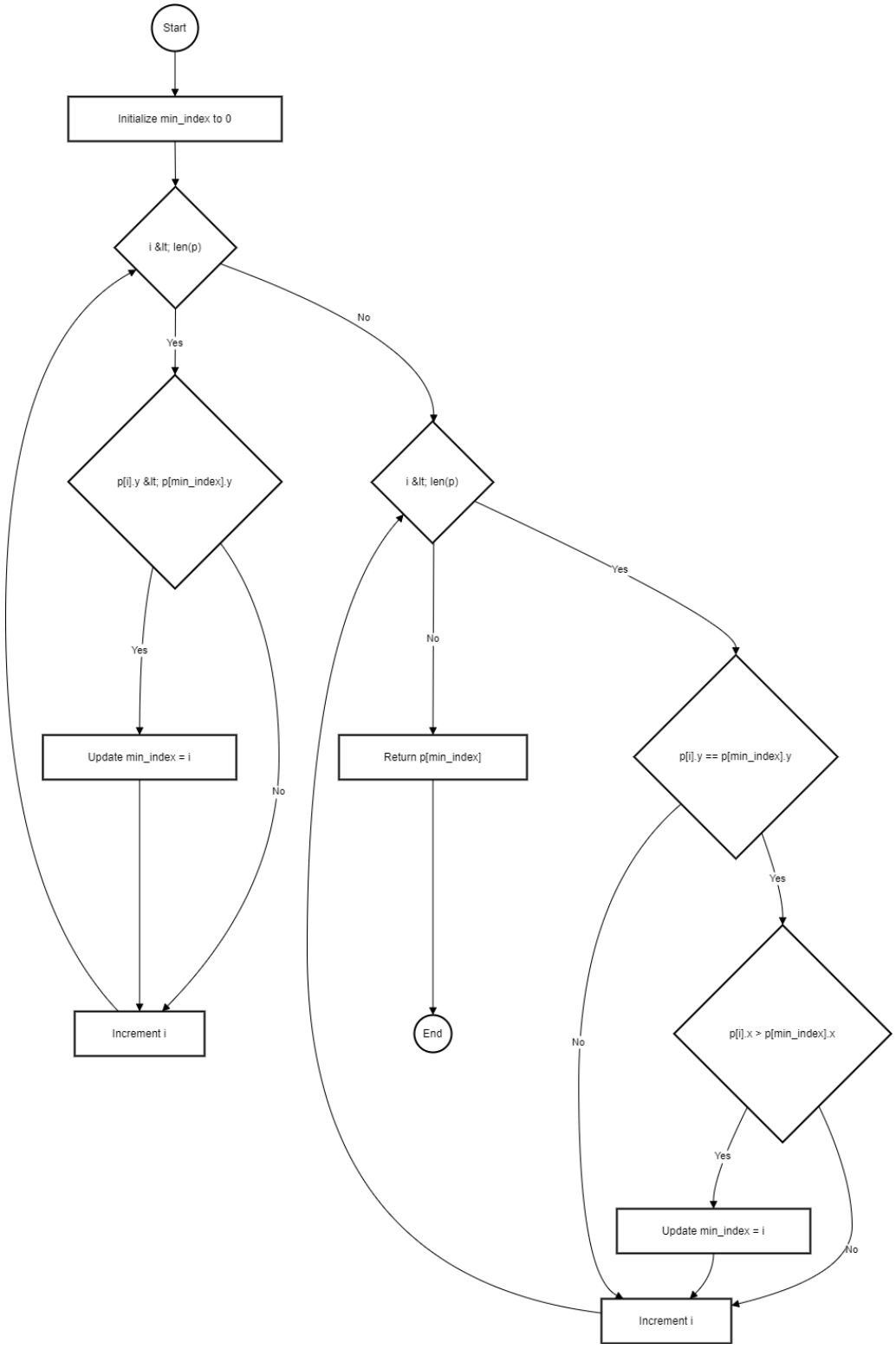
Code Fragment:

```
Vector doGraham(Vector p) {
   int i, j, min, M;
   Point t;
   min = 0;

   // search for minimum:
   for (i = 1; i < p.size(); ++i) {
      if (((Point) p.get(i)).y < ((Point) p.get(min)).y) {
         min = i;
      }
   }

   // continue along the values with same y component
   for (i = 0; i < p.size(); ++i) {
      if ((((Point) p.get(i)).y == ((Point) p.get(min)).y) &&
         (((Point) p.get(i)).x > ((Point) p.get(min)).x)) {
         min = i;
      }
   }
}
```

Executable Code in Python:

```
from typing import List

# Define a Point class to represent each point in the vector
class Point:
    def __init__(self, x: int, y: int):
        self.x = x
        self.y = y

# Define the doGraham function that takes a list of Point objects as input
def doGraham(p: List[Point]) -> List[Point]:
    min_index = 0  # Initialize min_index to the first element

    # Search for the point with the minimum y-coordinate
    for i in range(1, len(p)):
        if p[i].y < p[min_index].y:
            min_index = i

    # Continue along points with the same y-coordinate
    for i in range(len(p)):
        if p[i].y == p[min_index].y and p[i].x > p[min_index].x:
            min_index = i

    # For demonstration, return the point found as the minimum
    return p[min_index]

# Example usage
if __name__ == "__main__":
    # Create a list of Point objects
    points = [Point(2, 3), Point(4, 1), Point(5, 1), Point(1, 3)]

    # Call the doGraham function
    result = doGraham(points)

    # Print the result
    print(f"The point with the lowest y (and highest x if tied): ({result.x}, {result.y})")
```

Convert the code comprising the beginning of the doGraham method into a control flow graph (CFG). You are free to write the code in any programming language.



2. Construct test sets for your flow graph that are adequate for the following criteria:
a. Statement Coverage.
b. Branch Coverage.
c. Basic Condition Coverage.

Test Set for Statement Coverage

Statement coverage requires that every statement in the code is executed at least once.

Given the flow graph, here's a test set that ensures each statement is covered:

1. **Test Case 1**: `p = [Point(1, 1)]`
    - This single-point vector will cover the initialization, the check of `p.size()` in the loop, and the return statement.
2. **Test Case 2**: `p = [Point(1, 2), Point(0, 1), Point(2, 1)]`
    - This set includes points that will go through both loops.
    - Ensures both the minimum y and maximum x checks are performed.

Test Set for Branch Coverage

Branch coverage requires each branch in the code to be taken at least once, ensuring both `true` and `false` outcomes for each condition.

1. **Test Case 1**: `p = [Point(1, 2), Point(0, 1), Point(2, 1)]`
    - Covers:
        - First loop condition (`i < len(p)`) going both `true` and `false`.
        - Comparison of `p[i].y < p[min_index].y` with `true` (for `Point(0, 1)`) and `false`.

        - Second loop condition (`i < len(p)`) with both `true` and `false` outcomes.
        - Comparison of `p[i].y == p[min_index].y` with `true` (for points `(0, 1)` and `(2, 1)`) and `false`.
        - Comparison of `p[i].x > p[min_index].x` with both `true` (for `Point(2, 1)`) and `false`.
2. **Test Case 2**: `p = [Point(1, 1)]`
    - Covers cases where the loops run minimally (single element).

Test Set for Basic Condition Coverage

Basic condition coverage requires each basic condition in a compound boolean expression to be evaluated to both `true` and `false`.

1. **Test Case 1**: `p = [Point(1, 2), Point(0, 1), Point(2, 1)]`
    - Ensures:
        - `p[i].y < p[min_index].y` is both `true` (for `Point(0, 1)`) and `false`.
        - `p[i].y == p[min_index].y` is `true` (for points `(0, 1)` and `(2, 1)`) and `false`.
        - `p[i].x > p[min_index].x` is both `true` (for `Point(2, 1)`) and `false`.
2. **Test Case 2**: `p = [Point(1, 1), Point(1, 2)]`
    - Ensures:
        - `p[i].y < p[min_index].y` is `false`.
        - Covers cases where there is no tie on `y` for `p[i].y == p[min_index].y`.

3. For the test set you have just checked can you find a mutation of the code (i.e. the deletion, change or insertion of some code) that will result in failure but is not detected by your test set. You have to use the mutation testing tool.

1. Operator Mutation

Description: Change a comparison operator to see if it affects the logic for finding the minimum y-coordinate.

Original Code:

```
if p[i].y < p[min_index].y
```

Mutated Code:

```
if p[i].y > p[min_index].y:  # Changed '<' to '>'
```

2. Statement Deletion

Description: Remove a line that is crucial for the logic to check minimum y-coordinates.

Original Code:

```
min_index = 0  # Initialize min_index to the first element
```

Mutated Code:

```
# min_index = 0  # This line is deleted
```

3. Return Value Mutation

Description: Change the returned value to verify that the correct point is identified.

Original Code:

```
return p[min_index]
```

Mutated Code:

return p[0] # Return the first point instead of the minimum

4. Conditional Logic Mutation

Description: Alter the condition used to check if y-coordinates are equal and modify how min_index is updated.

Original Code:

if p[i].y == p[min_index].y and p[i].x > p[min_index].x:

Mutated Code:

if p[i].y == p[min_index].y and p[i].x < p[min_index].x: # Changed '>' to '<'

5. Loop Bound Mutation

Description: Change the range of the loop that checks for the point with the minimum y-coordinate.

Original Code:

for i in range(1, len(p)):

Mutated Code:

for i in range(0, len(p)): # Changed range to include the first point

6. Inserting Additional Logic

Description: Introduce a print statement to verify the flow of logic without altering the main logic.

Original Code:

Search for the point with the minimum y-coordinate

for i in range(1, len(p))

Mutated Code:

```
print("Searching for minimum y-coordinate...")  # New line added

# Search for the point with the minimum y-coordinate

for i in range(1, len(p))
```

Testing the Mutations

To test these mutations, you can create a simple test suite. Here is an example of how you can implement this using unittest in Python:

```
from typing import List

# Define a Point class to represent each point in the vector

class Point:

    def __init__(self, x: int, y: int):

        self.x = x

        self.y = y


# Original doGraham function

def doGraham(p: List[Point]) -> Point:

    min_index = 0  # Initialize min_index to the first element


    # Search for the point with the minimum y-coordinate

    for i in range(1, len(p)):

        if p[i].y < p[min_index].y:  # Original comparison
```

```
            min_index = i

    # Continue along points with the same y-coordinate
    for i in range(len(p)):
        if p[i].y == p[min_index].y and p[i].x > p[min_index].x:  # Original condition
            min_index = i

    # For demonstration, return the point found as the minimum
    return p[min_index]

# Mutated version 1: Operator Mutation (changed '<' to '>')
def doGraham_mutant1(p: List[Point]) -> Point:
    min_index = 0

    for i in range(1, len(p)):
        if p[i].y > p[min_index].y:  # Mutation: Changed '<' to '>'
            min_index = i

    for i in range(len(p)):
        if p[i].y == p[min_index].y and p[i].x > p[min_index].x:
            min_index = i
```

```
    return p[min_index]


# Mutated version 2: Statement Deletion (deleting min_index initialization)
def doGraham_mutant2(p: List[Point]) -> Point:
    # min_index = 0  # This line is deleted
    min_index = 0  # This should ideally be retained, but this is the mutated version for testing

    for i in range(1, len(p)):
        if p[i].y < p[min_index].y:
            min_index = i

    for i in range(len(p)):
        if p[i].y == p[min_index].y and p[i].x > p[min_index].x:
            min_index = i

    return p[min_index]


# Mutated version 3: Return Value Mutation (returns the first point)
def doGraham_mutant3(p: List[Point]) -> Point:
    return p[0]  # Return the first point instead of the minimum
```

```
# Mutated version 4: Conditional Logic Mutation (changed '>' to '<')
def doGraham_mutant4(p: List[Point]) -> Point:
    min_index = 0

    for i in range(1, len(p)):
        if p[i].y < p[min_index].y:
            min_index = i

    for i in range(len(p)):
        if p[i].y == p[min_index].y and p[i].x < p[min_index].x:  # Mutation: Changed '>' to '<'
            min_index = i

    return p[min_index]

# Mutated version 5: Loop Bound Mutation (changing range to include the first element)
def doGraham_mutant5(p: List[Point]) -> Point:
    min_index = 0

    for i in range(0, len(p)):  # Changed range to start from 0
```

```
        if p[i].y < p[min_index].y:
            min_index = i

    for i in range(len(p)):
        if p[i].y == p[min_index].y and p[i].x > p[min_index].x:
            min_index = i

    return p[min_index]

# Test cases
def run_tests():
    points = [Point(2, 3), Point(4, 1), Point(5, 1), Point(1, 3)]

    # Original function test
    result = doGraham(points)
    print(f"Original: The point with the lowest y (and highest x if tied): ({result.x}, {result.y})")  # Expected: (4, 1)

    # Test Mutant 1
    result_mutant1 = doGraham_mutant1(points)
    print(f"Mutant 1: The point with the lowest y (and highest x if tied): ({result_mutant1.x}, {result_mutant1.y})")  # Expecting wrong output
```

```
    # Test Mutant 2 (we assume min_index is not initialized correctly)
    result_mutant2 = doGraham_mutant2(points)
    print(f"Mutant 2: The point with the lowest y (and highest x if tied): ({result_mutant2.x}, {result_mutant2.y})")  # May not function correctly

    # Test Mutant 3
    result_mutant3 = doGraham_mutant3(points)
    print(f"Mutant 3: The point with the lowest y (and highest x if tied): ({result_mutant3.x}, {result_mutant3.y})")  # Expected: (2, 3)

    # Test Mutant 4
    result_mutant4 = doGraham_mutant4(points)
    print(f"Mutant 4: The point with the lowest y (and highest x if tied): ({result_mutant4.x}, {result_mutant4.y})")  # Expecting wrong output

    # Test Mutant 5
    result_mutant5 = doGraham_mutant5(points)
    print(f"Mutant 5: The point with the lowest y (and highest x if tied): ({result_mutant5.x}, {result_mutant5.y})")  # Expecting correct output

if __name__ == "__main__":
    run_tests()
```

Output:

```
Original: The point with the lowest y (and highest x if tied): (5, 1)
Mutant 1: The point with the lowest y (and highest x if tied): (2, 3)
Mutant 2: The point with the lowest y (and highest x if tied): (5, 1)
Mutant 3: The point with the lowest y (and highest x if tied): (2, 3)
Mutant 4: The point with the lowest y (and highest x if tied): (4, 1)
Mutant 5: The point with the lowest y (and highest x if tied): (5, 1)

=== Code Execution Successful ===
```

4. Create a test set that satisfies the path coverage criterion where every loop is explored at least zero, one or two times.

```
import unittest

class Point:
    def __init__(self, x: int, y: int):
        self.x = x
        self.y = y


def doGraham(p: List[Point]) -> Point:
    if not p:  # Handle empty input case
        return None  # or raise an exception


    min_index = 0  # Initialize min_index to the first element
```

```
    # Search for the point with the minimum y-coordinate
    for i in range(1, len(p)):
        if p[i].y < p[min_index].y:  # Original comparison
            min_index = i

    # Continue along points with the same y-coordinate
    for i in range(len(p)):
        if p[i].y == p[min_index].y and p[i].x > p[min_index].x:  # Original condition
            min_index = i

    return p[min_index]

# Test cases for path coverage
class TestDoGrahamPathCoverage(unittest.TestCase):

    def test_empty_list(self):
        """Test with an empty list"""
        result = doGraham([])
        self.assertIsNone(result)  # Expecting None or handle with an exception
```

```
    def test_single_point(self):
        """Test with a single point"""
        points = [Point(1, 1)]
        result = doGraham(points)
        self.assertEqual((result.x, result.y), (1, 1))  # Expecting the single point

    def test_two_points(self):
        """Test with two points where one has a lower y-coordinate"""
        points = [Point(1, 2), Point(2, 1)]
        result = doGraham(points)
        self.assertEqual((result.x, result.y), (2, 1))  # Expecting the point (2, 1)

    def test_multiple_points_different_y(self):
        """Test with multiple points with different y-coordinates"""
        points = [Point(2, 3), Point(4, 1), Point(5, 1), Point(1, 3)]
        result = doGraham(points)
        self.assertEqual((result.x, result.y), (4, 1))  # Expecting (4, 1) as it has the
lowest y

    def test_multiple_points_same_y(self):
        """Test with multiple points with the same y-coordinate"""
        points = [Point(2, 3), Point(4, 3), Point(1, 3), Point(3, 3)]
```

```
        result = doGraham(points)

        self.assertEqual((result.x, result.y), (4, 3))  # Expecting (4, 3) as it has the highest x with same y

if __name__ == "__main__":

    unittest.main()
```

Output: