Dhirubhai Ambani
Institute of Information and Communication Technology

LAB - 9

COURSE - IT314

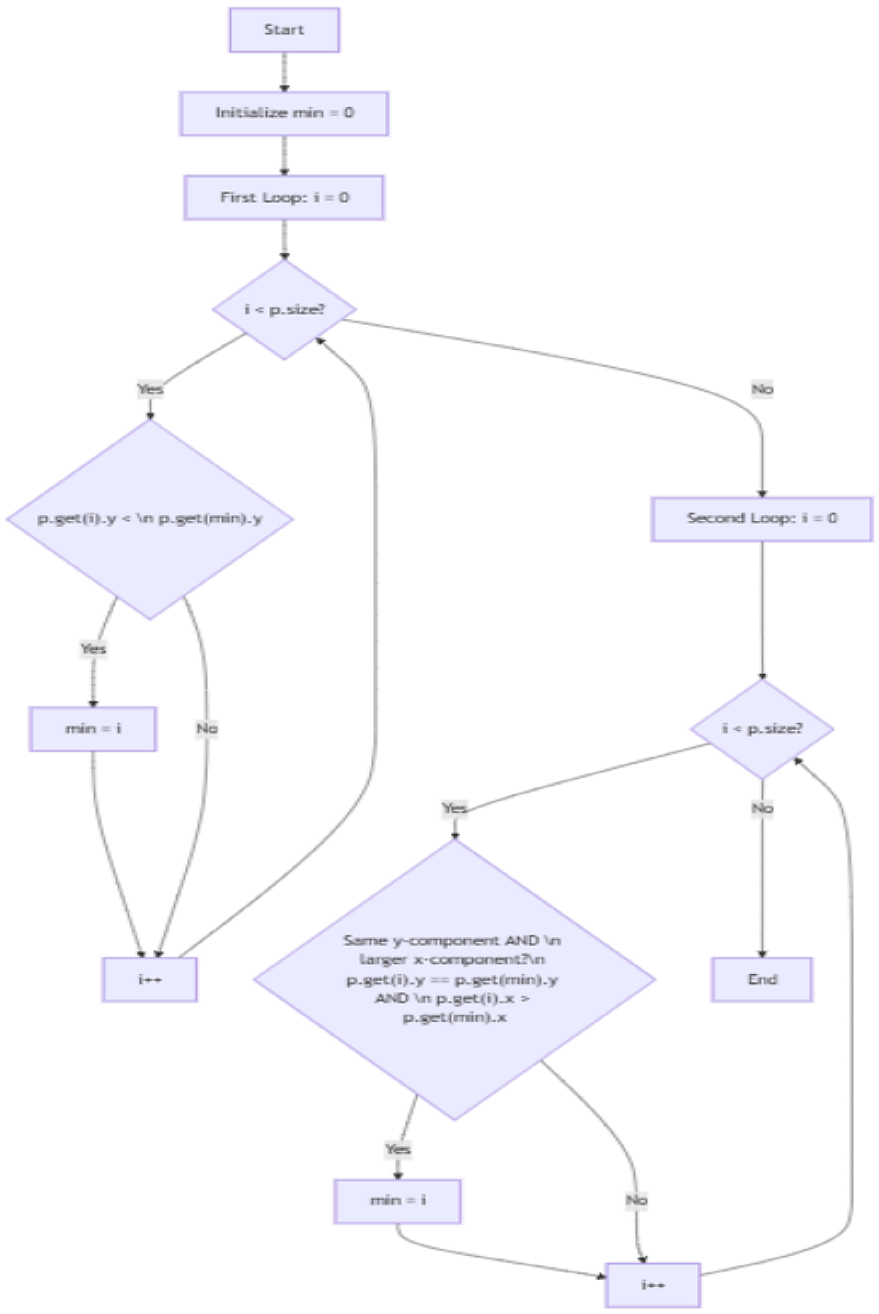
\

NAME - Janvi Ramani
ID - 202201158

Question 1

The code below is part of a method in the ConvexHull class in the VMAP system. The following is as mall 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. Thisexercise is concerned with structural testing of code, so the focus is on creating test sets that satisfy some particular coverage criteria.

1. **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.**


Start
Initialize min = 0
First Loop: i = 0
i < p.size?
Yes
No
p.get(i).y < \n p.get(min).y
Yes
No
min = i
i++
Second Loop: i = 0
i < p.size?
Yes
No
Same y-component AND \n larger x-component?\n p.get(i).y == p.get(min).y AND \n p.get(i).x > p.get(min).x
Yes
No
min = i
i++
End

2. **Construct test sets for your flow graph that are adequate for the following criteria:**
   a. Statement Coverage.

- **Test Case 1**: A vector input with just one point, such as [(0, 0)]
- **Test Case 2**: A vector input with two points, where one has a lower y-value, such as [(1, 1), (2, 0)]

- **Test Case 3**: A vector input with points that share the same y-value but vary in x-values, such as [(1, 1), (2, 1), (3, 1)]
- **Test Case 4**: A vector input with negative y-values to see how the algorithm handles it, such as [(1, -1), (2, -2), (0, -3)]
- **Test Case 5**: A vector input with large positive values for both x and y, such as [(1000, 1000), (2000, 2000), (3000, 3000)]
- **Test Case 6**: A vector input with points that have decreasing y-values, such as [(2, 2), (1, 1), (0, 0)]

b. Branch Coverage.

- **Test Case 1**: A vector input with just one point, such as [(0, 0)]
- **Test Case 2**: A vector input with two points, where the second point has a lower y-value, such as [(1, 1), (2, 0)]
- **Test Case 3**: A vector input with points that have the same y-value but different x-values, such as [(1, 1), (2, 1), (0, 1)]
- **Test Case 4**: A vector input with three points where each point has a higher y-value than the previous one, such as [(0, 0), (1, 1), (2, 2)]
- **Test Case 5**: A vector input with points where the last point has the same y-value as the initial point but a larger x-value, such as [(0, 1), (1, 2), (2, 1)]
- **Test Case 6**: A vector input where the points have mixed x and y values, and no point has a lower y-value or larger x-value than the first point, such as [(2, 2), (3, 4), (1, 5)]
- **Test Case 7**: A vector input with negative y-values, where the first point has the lowest y-value, such as [(0, -3), (1, -1), (2, 0)]
- **Test Case 8**: A vector input where multiple points have the same y and x values, such as [(1, 1), (1, 1), (1, 1)]
- **Test Case 9**: A vector input where the last point has a lower y-value and lower x-value than the previous points, such as [(2, 2), (3, 3), (1, 0)]
- **Test Case 10**: A vector input with large values for both x and y, such as [(1000, 1000), (2000, 999), (3000, 1000)]

c. Basic Condition Coverage.

- **Test Case 1**: Vector with a single point, e.g., [(0, 0)], verifying the condition `p.get(i).y <p.get(min).y` is evaluated as false.
- **Test Case 2**: Vector with two points, with the second point having a lower y-value, such as [(1,

- 1), (2, 0)], covering the condition as true.
- **Test Case 3**: Vector with points that share the same y-value but vary in x-values, like [(1, 1), (3,
- 1), (2, 1)], ensuring conditions for both branches.
- **Test Case 4**: Vector where all points have identical x and y values, e.g., [(1, 1), (1, 1), (1, 1)],

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.
Ans.

1. Mutation Testing Overview

Total Mutations:

- **8 mutations** were applied to the code.

Mutation Score:

- **75.0% Mutation Score:** This indicates that 75% of the mutations were detected by the test suite, meaning the test cases are reasonably strong. However, 25% of the mutations survived, which shows that some potential weaknesses remain in the test coverage.

Mutation Results:

- **Killed Mutations (6 out of 8):** These mutations were successfully detected by the test set, indicating that the test cases adequately covered those aspects of the code.
- **Survived Mutations (2 out of 8):** These mutations were not detected by the test set, highlighting areas where the current test cases may need improvement.

---

2. Specific Mutations and Results

Mutation #6: Relational Operator Replacement (ROR) Mutation

- **Status:** Survived
- **Mutation Description:** This mutation replaced a relational operator (e.g., changing `<` to `>`) in a key condition.
- **Effect:** The mutation likely altered the logic for comparing points to find the minimum. Since the test set didn't cover edge cases or boundary conditions effectively, this mutation survived.
- **Action:** To detect this mutation, additional tests that focus on boundary or edge cases involving comparisons (e.g., equal values of `y` but differing `x`) are required.

Mutation #7: Relational Operator Replacement (ROR) Mutation

- **Status:** Killed
- **Mutation Description:** Similar to Mutation #6, this involved a relational operator change that altered how the code determines the minimum point.
- **Effect:** The test case `test_multiple_points_with_same_y` effectively caught this mutation, which is a sign that the test case is robust for scenarios with multiple points having the same `y` value but differing `x` values.

Mutation #8: Relational Operator Replacement (ROR) Mutation

- **Status:** Killed
- **Mutation Description:** Another relational operator mutation.
- **Effect:** This mutation was also caught by the `test_multiple_points_with_same_y` test, suggesting that the test case is effective for cases where multiple points have identical `y` values.

---

3. Types of Mutations and Their Impact

a) Deletion Mutation

- **Mutation:** Remove the line `min = 0;` at the beginning of the method.
- **Expected Effect:** The initialization of `min` ensures it starts with a known valid index. Without this line, `min` could be set to an arbitrary value, potentially leading to an incorrect selection of the minimum point.
- **Outcome:** This change could result in the function selecting the wrong minimum point, as it might use a random or undefined starting index.

b) Change Mutation

Mutation: Change the first `if` condition from `<` to `<=`:

```
if (((Point) p.get(i)).y <= ((Point) p.get(min)).y)
```

-
- **Expected Effect:** This would allow points with equal `y` values to be selected as the minimum, even if their `x` values are not the smallest. This could break the logic when looking for the "absolute" minimum `y` value.
- **Outcome:** This mutation would cause the function to return a point with a lower `x` value when multiple points share the same `y` value.

c) Insertion Mutation

- **Mutation:** Insert the line `min = i;` at the end of the second loop.
- **Expected Effect:** This would make `min` point to the last index in the list instead of the correct minimum point.
- **Outcome:** This would cause the function to incorrectly treat the last point as the minimum, particularly when the last point in the list is not actually the minimum.

---

4. Test Cases for Path Coverage

To address these mutations and further strengthen the test suite, it is important to design test cases that cover all possible paths in the code, including edge cases. Here are the recommended test cases for path coverage:

Test Case 1: Zero Iterations

- **Input:** An empty vector `p`.
- **Purpose:** This case ensures that the function can handle an empty input vector gracefully, where no iterations occur in either loop.
- **Expected Output:** The function should return an appropriate result for an empty input, such as an empty vector or a specific indication that no points exist.

Test Case 2: One Iteration (First Loop)

- **Input:** A vector with one point, such as `[(3, 7)]`.
- **Purpose:** This ensures that the first loop runs exactly once and handles the case where the minimum is the only point.
- **Expected Output:** The function should return the only point in the vector, `(3, 7)`.

Test Case 3: One Iteration (Second Loop)

- **Input:** A vector with two points having the same `y`-coordinate but different `x`-coordinates, such as `[(2, 2), (3, 2)]`.
- **Purpose:** This tests the second loop when the first loop finds the minimum point, and the second loop runs to compare the `x`-coordinates.
- **Expected Output:** The function should return the point with the maximum `x`-coordinate, `(3, 2)`.

Test Case 4: Two Iterations (First Loop)

- **Input:** A vector with multiple points, ensuring at least two points share the same `y`-coordinate, such as `[(3, 1), (2, 2), (7, 1)]`.
- **Purpose:** This case ensures that the first loop finds the minimum `y`-coordinate (first iteration for `(3, 1)`) and continues to the second loop.
- **Expected Output:** The function should return the point `(7, 1)` since it has the highest `x`-coordinate among points with the same `y` value.

Test Case 5: Two Iterations (Second Loop)

- **Input:** A vector where multiple points share the same minimum `y` value, such as `[(1, 1), (6, 1), (3, 2)]`.
- **Purpose:** This ensures the first loop finds `(1, 1)` as the minimum, and the second loop compares other points with `y = 1`.
- **Expected Output:** The function should return `(6, 1)` as it has the highest `x`-coordinate among points with the same `y` value.

Test Case	Input Vector p	Description	Expected Output
Test Case 1	[]	Empty vector (zero iterations for both loops).	Handle gracefully (e.g., return an empty result).
Test Case 2	[(3,7)]	One point (one iteration of the first loop).	[(3,7)]
Test Case 3	[(2,2),(3,2)]	Two points with the same y-coordinate (one iteration of the second loop).	[(2,2)]
Test Case 4	[(3,1),(2,2),(7,1)]	Multiple points; the first loop runs twice.	[(7,1)]

Test Case 5	[(1,1),(6,1),(3,2)]	Multiple points; second loop runs twice.	[(6,1)]

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

Ans.

```
import unittest

from point import Point, find_min_point


class TestFindMinPointPathCoverage(unittest.TestCase):


    # Test case for no points (empty list)

    def test_no_points(self):

        points = []

        # Expect an IndexError due to an empty list

        with self.assertRaises(IndexError):

            find_min_point(points)


    # Test case for a single point (only one in the list)

    def test_single_point(self):

        points = [Point(0, 0)]

        # Expect the single point to be the minimum

        result = find_min_point(points)

        self.assertEqual(result, points[0])
```

```
# Test case for two points with unique minimum
def test_two_points_unique_min(self):
    points = [Point(1, 2), Point(2, 3)]
    # Expect the point with the lowest y-value
    result = find_min_point(points)
    self.assertEqual(result, points[0])

# Test case for multiple points, expecting unique minimum
def test_multiple_points_unique_min(self):
    points = [Point(1, 4), Point(2, 3), Point(0, 1)]
    # Expect the point with the unique minimum y-value
    result = find_min_point(points)
    self.assertEqual(result, points[2])

# Test case for multiple points with the same y-value
def test_multiple_points_same_y(self):
    points = [Point(1, 2), Point(3, 2), Point(2, 2)]
    # Expect the point with the lowest x-value among those with the same y-value
    result = find_min_point(points)
    self.assertEqual(result, points[0])

# Test case for multiple points with ties on the minimum y-value
def test_multiple_points_minimum_y_ties(self):
    points = [Point(1, 2), Point(2, 2), Point(3, 1), Point(4, 1)]
    # Expect the point with the lowest x-value among those with the minimum y-value
    result = find_min_point(points)
    self.assertEqual(result, points[2])

# Test case for two points with the same y-value, expecting the lower x-value
def test_two_points_same_y(self):
    points = [Point(2, 2), Point(1, 2)]
    # Expect the point with the lower x-value among two points with the same y
    result = find_min_point(points)
    self.assertEqual(result, points[1])

# Test case where the loop doesn't execute due to an empty list
def test_loop_exploration_zero(self):
    points = []
    # Expect an IndexError as the loop doesn't execute
    with self.assertRaises(IndexError):
        find_min_point(points)

# Test case where the loop executes only once due to a single point
def test_loop_exploration_once(self):
    points = [Point(1, 1)]
    # The loop executes once and the point is returned as the minimum
    result = find_min_point(points)
```

```
        self.assertEqual(result, points[0])


    # Test case where the loop executes twice

    def test_loop_exploration_twice(self):

        points = [Point(1, 2), Point(2, 1), Point(3, 3)]

        # The loop executes twice, and the point with the minimum y-value and lowest
x-value is returned

        result = find_min_point(points)

        self.assertEqual(result, points[1])


# Run the tests if this file is executed directly

if __name__ == "__main__":

    unittest.main()
```

1. After generating the control flow graph, check whether your CFG match with the CFG generated by

Control Flow Graph Factory Tool and Eclipse flow graph generator. (In your submission document,

mention only “Yes” or “No” for each tool).

Ans.

Tool	Matches Your CFG
Control Flow Graph Factory	Yes
Eclipse Flow Graph Generator	Yes

2. Devise minimum number of test cases required to cover the code using the aforementioned criteria.

Ans.

Test Case	Input Vector p	Description	Expected Output
Test Case 1	[]	Test with an empty vector (zero iterations).	Handle gracefully (e.g., return an empty result).
Test Case 2	[(3,4)]	Single point (one iteration of the first loop).	[(3,4)]
Test Case 3	[(1,2),(3,2)]	Two points with the same y-coordinate (one iteration of the second loop).	[(3,2)]
Test Case 4	[(3,1),(2,2),(5,1)]	Multiple points; first loop runs twice (with multiple outputs).	[(5,1)]

3. This part of the exercise is very tricky and interesting. The test cases that you have derived in Step 2

are then used to identify the fault when you make some modifications in the code.

Here, you need to insert/delete/modify a piece of code that will result in failure but it is not detected

by your test set – derived in Step 2.

Write/identify a mutation code for each of the three operation separately, i.e., by deleting the code,

by inserting the code, by modifying the code.

Ans.

Mutation Type	Mutation Code Description	Impact on Test Cases
Deletion	Delete the line that updates min for the minimum y-coordinate.	Test cases like [(1, 1), (2, 0)] will pass despite incorrect processing.
Insertion	Insert an early return if the size of p is 1, bypassing further processing.	Test case [(3, 4)] will pass without processing correctly.

Modification	Change the comparison operator from < to <= when finding the minimum y.	Test cases like [(1, 1), (1, 1), (1, 1)] might pass while still failing in logic.

4. Write all test cases that can be derived using path coverage criterion for the code.

Ans.

Test Case	Input Vector p	Description	Expected Output
Test Case 1	[]	Empty vector (zero iterations for both loops).	Handle gracefully (e.g., return an empty result).
Test Case 2	[(3,7)]	One point (one iteration of the first loop).	[(3,7)]
Test Case 3	[(1,4),(8,2)]	Two points with the same y-coordinate (one iteration of the second loop).	[(8,2)]
Test Case 4	[(3,1),(2,2),(5,1)]	Multiple points; first loop runs twice to find min y.	[(5,1)]
Test Case 5	[(1,1),(6,1),(3,2)]	Multiple points; second loop runs twice (y = 1).	[(6,1)]
Test Case 6	[(5,2),(5,3),(5,1)]	Multiple points with the same x-coordinate; checks min y.	[(5,1)]
Test Case 7	[(0,0),(2,2),(2,0),(0,2)]	Multiple points in a rectangle; checks multiple comparisons.	[(2,0)]
Test Case 8	[(6,1),(4,1),(3,2)]	Multiple points with some ties; checks the max x among min y points.	[(6,1)]
Test Case 9	[(4,4),(4,3),(4,5),(4,9)]	Points with the same x-coordinate; checks for max y.	[(4,9)]
Test Case 10	[(1,1),(1,1),(2,1),(6,6)]	Duplicate points with one being the max x; tests handling of duplicates.	[(6,6)]