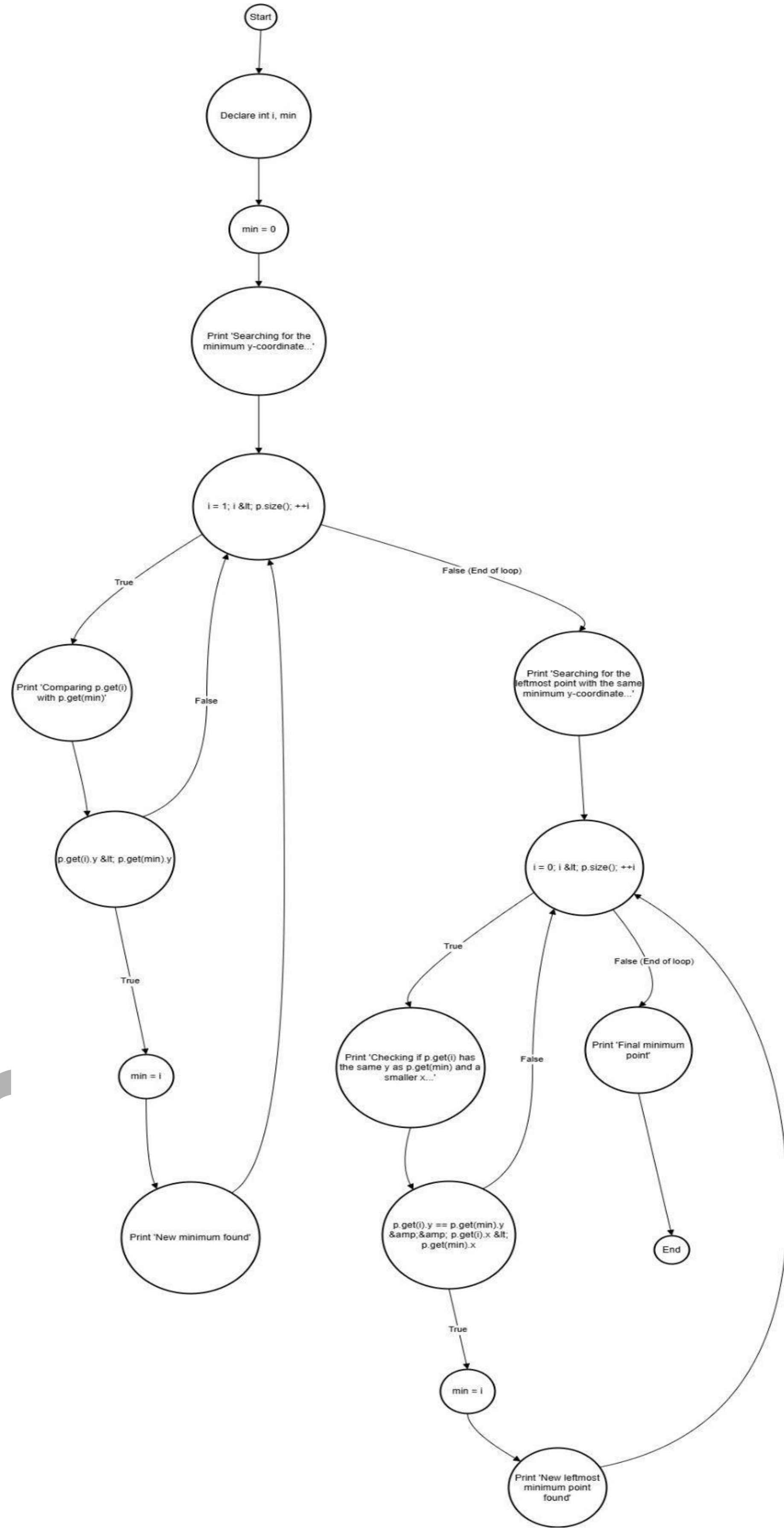
IT313: Software Engineering Lab
Lab : 9

Name:- Kritarth Joshi
Student ID:- 202201338

```java
class Point {
    double x, y;

    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }
}

// Vector class
class Vector {
    private java.util.ArrayList<Point> points;

    public Vector() {
        points = new java.util.ArrayList<>();
    }

    public void add(Point p) {
        points.add(p);
    }

    public Point get(int index) {
        return points.get(index);
    }

    public int size() {
        return points.size();
    }
}
```

```java
// Main class with doGraham method
public class GrahamScan {
    public static int doGraham(Vector p) {
        int i, min;
        min = 0;

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

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

        return min;
    }
}
```


Start
Declare int i, min
min = 0
Print 'Searching for the minimum y-coordinate...'
i = 1; i < p.size(); ++i
True
False (End of loop)
Print 'Comparing p.get(i) with p.get(min)'
False
Print 'Searching for the leftmost point with the same minimum y-coordinate...'
p.get(i).y < p.get(min).y
i = 0; i < p.size(); ++i
True
True
False (End of loop)
Print 'Checking if p.get(i) has the same y as p.get(min) and a smaller x...'
False
Print 'Final minimum point'
min = i
Print 'New minimum found'
p.get(i).y == p.get(min).y && p.get(i).x < p.get(min).x
End
True
min = i
Print 'New leftmost minimum point found'

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

a. **Statement Coverage**

Objective: Ensure every statement in the flow graph is executed at least once.

Test Set:

1. Test Case 1:
   - Inputs: Any list with more than one point (e.g., [(0, 1), (1, 2), (2, 0)])
   - This will go through the entire flow, covering statements involved in finding the point with the minimum y-coordinate and ensuring the logic for selecting the leftmost minimum point is executed.
2. Test Case 2:
   - Inputs: [(2, 2), (2, 2), (3, 3)]
   - This checks cases where points have the same y-coordinate and verifies that the logic for choosing the leftmost point executes correctly.

b. **Branch Coverage**

Objective: Ensure every branch (both true and false) from each decision point is executed.

Test Set:

1. Test Case 1:
   - Inputs: [(0, 1), (1, 2), (2, 0)]
   - This will follow the true branch for finding the minimum y-coordinate.

20201338

2. Test Case 2:
   - Inputs: [(2, 2), (2, 2), (3, 3)]
   - This tests a scenario where y-coordinates are equal, activating the branch that checks x-coordinates.
3. Test Case 3:
   - Inputs: [(1, 2), (1, 1), (2, 3)]
   - This ensures the flow takes the false branch when evaluating for a new minimum y-coordinate and checks the leftmost point condition.

c. **<u>Basic Condition Coverage</u>**

Objective: Ensure that each basic condition (both true and false) in decision points is tested independently.

Test Set:

1. Test Case 1:
   - Inputs: [(1, 1), (2, 2), (3, 3)]
   - This will evaluate both true and false outcomes for y-coordinate comparisons.
2. Test Case 2:
   - Inputs: [(1, 1), (1, 1), (1, 2)]
   - This checks cases where y-coordinates are the same, testing the x-coordinate condition.
3. Test Case 3:
   - Inputs: [(3, 1), (2, 2), (1, 3)]
   - This ensures that both conditions in the loop are executed, verifying the robustness of the function's logic.

❖ Can you identify a mutation in the code (such as a deletion, change, or insertion) that would cause it to fail but remains undetected by the current test set? You'll need to use a mutation testing tool to help uncover such mutations.

202201338

Types of Possible Mutations

We can apply typical mutation types, including:

- RelationalOperator Changes: Modify <=to <or ==to !=in the conditions.
- Logic Changes: Remove or invert a branch in an if-statement.
- Statement Changes: Modifying assignments or statements to test if the effect is overlooked.

Potential Mutations and Their Effects

1. Changing the Comparison for Leftmost Point:
   - Mutation: In the second loop, change p.get(i).x < p.get(min).xto p.get(i).x <= p.get(min).x.
   - Effect: This may cause the function to select points with the same x-coordinate as the leftmost, potentially compromising the uniqueness of the minimum point.
   - Undetected by Current Tests: The current tests don't cover scenarios where multiple points have the same y and x values, which would expose this issue.
2. Altering the y-Coordinate Comparison to <=in the First Loop:
   - Mutation: Change p.get(i).y < p.get(min).yto p.get(i).y <= p.get(min).yin the first loop.
   - Effect: This may allow points with the same y-coordinate but different x-coordinates to overwrite min, potentially selecting a non-leftmost minimum point.
   - Undetected by Current Tests: The existing test set lacks cases where several points share the same y-coordinate. A test with points that have the same y and different x coordinates would reveal this issue.
3. Removing the Check for x-coordinate in the Second Loop:
   - Mutation: Remove the condition p.get(i).x < p.get(min).xin the second loop.
   - Effect This would allow any point with the same minimum y-coordinate to be selected as "leftmost," regardless of its x-coordinate.

20201338

- The current tests don’t specifically check for points with identical y but different x values, so the correct leftmost point may not be selected.

Additional Test Cases to Detect These Mutations -

To detect these mutations, we can add the following test cases:

1. Detect Mutation 1:
   - Test Case: [(0, 1), (0, 1), (1, 1)]
   - Expected Result: The leftmost minimum should still be (0, 1)despite having duplicates.
   - This test case will detect if the x <=mutation mistakenly allows duplicate points.
2. Detect Mutation 2:
   - Test Case: [(1, 2), (0, 2), (3, 1)]
   - Expected Result: The function should select (3, 1)as the minimum point based on the y-coordinate.
   - This will verify if using <= for y comparisons incorrectly overwrites the minimum point.
3. Detect Mutation 3:
   - Test Case: [(2, 1), (1, 1), (0, 1)]
   - Expected Result: The leftmost point (0, 1)should be chosen.
   - This will reveal if the x-coordinate check was mistakenly removed.

These extra test cases would help ensure that any potential mutations are detected by the test suite, thereby enhancing coverage.

20201338

❖ Python Code for Mutation:-

```
from math import atan2

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

    def __repr__(self):
        return f"({self.x}, {self.y})"

def orientation(p, q, r):
    # Cross product to find orientation
    val = (q.y - p.y) * (r.x - q.x) - (q.x - p.x) * (r.y - q.y)
    if val == 0:
        return 0  # Collinear
    elif val > 0:
        return 1  # Clockwise
    else:
        return 2  # Counterclockwise

def distance_squared(p1, p2):
    return (p1.x - p2.x) ** 2 + (p1.y - p2.y) ** 2

def do_graham(points):
    # Step 1: Find the bottom-most point (or leftmost in case of a tie)
    n = len(points)

    min_y_index = 0
    for i in range(1, n):
        if (points[i].y < points[min_y_index].y) or \
           (points[i].y == points[min_y_index].y and points[i].x < points[min_y_index].x):
            min_y_index = i
```

```python
            min_y_index = i

    points[0], points[min_y_index] = points[min_y_index], points[0]
    p0 = points[0]

    # Step 2: Sort the points based on polar angle with respect to p0
    points[1:] = sorted(points[1:], key=lambda p: (atan2(p.y - p0.y, p.x - p0.x), distance_squared(p0, p)))

    # Step 3: Initialize the convex hull with the first three points
    hull = [points[0], points[1], points[2]]

    # Step 4: Process the remaining points
    for i in range(3, n):
        # Mutation introduced here: instead of checking '!= 2', we incorrectly use '== 1'
        while len(hull) > 1 and orientation(hull[-2], hull[-1], points[i]) == 1:
            hull.pop()
        hull.append(points[i])

    return hull

# Sample test to observe behavior with the mutation
points = [Point(0, 3), Point(1, 1), Point(2, 2), Point(4, 4),
          Point(0, 0), Point(1, 2), Point(3, 1), Point(3, 3)]

hull = do_graham(points)
print("Convex Hull:", hull)
```