Software Engineering

Lab No: 9

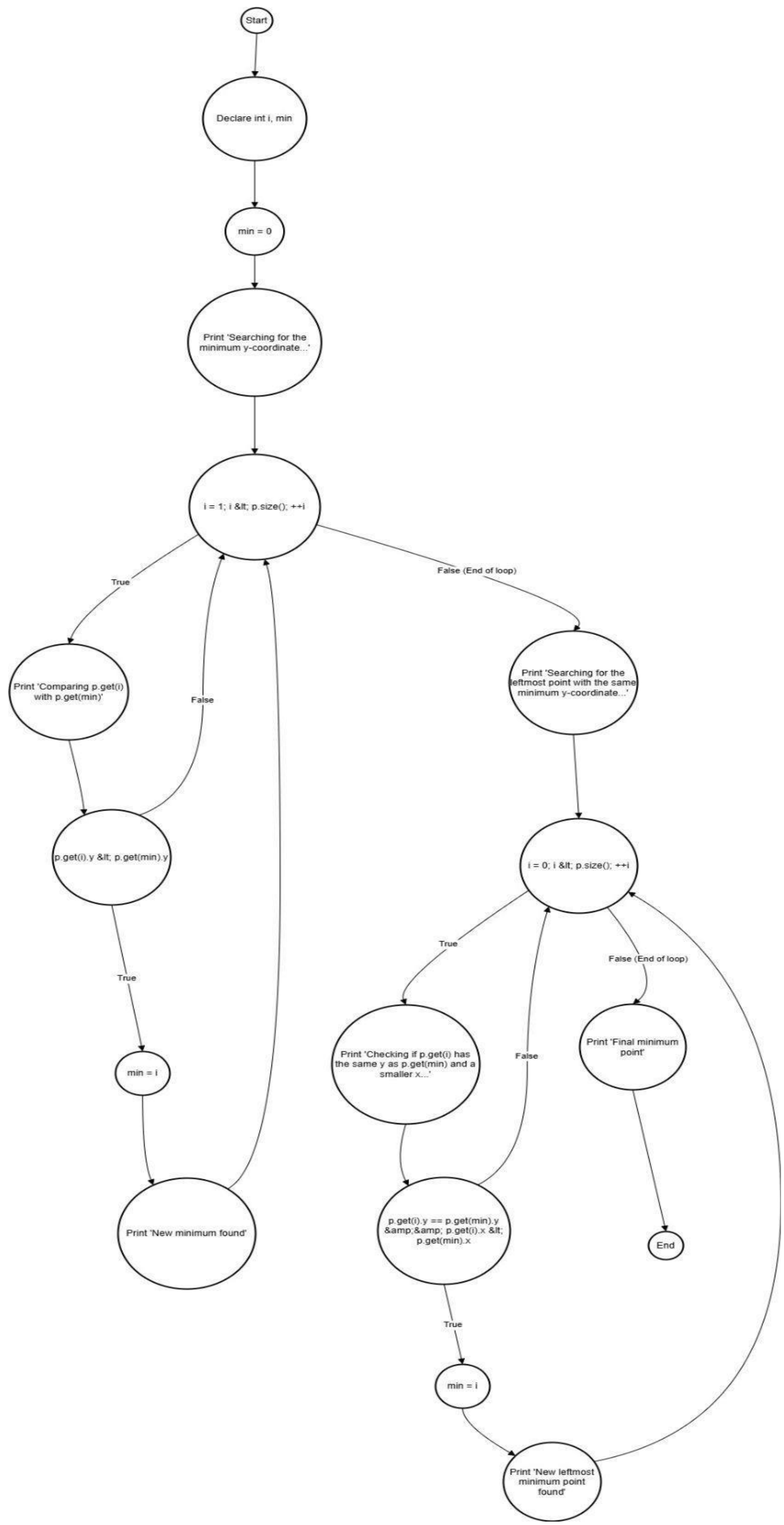
Student ID:- 202201175

Name:- Kathan Kadiya

❖ **Code :**

```
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();
}
```

```
// 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).yp.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
p.get(i).y < p.get(min).y
True
min = i
Print 'New minimum found'
Print 'Searching for the leftmost point with the same minimum y-coordinate...'
i = 0; i < p.size(); ++i
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'
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 that each 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 traverse through the entire flow, covering statements related to finding the minimum y-coordinate and leftmost minimum point.
2. **Test Case 2:**
   - Inputs: [(2, 2), (2, 2), (3, 3)]
   - This checks for points with the same y-coordinate and ensures the leftmost point logic executes.

b. Branch Coverage

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

Test Set:

1. **Test Case 1:**
   - Inputs: [(0, 1), (1, 2), (2, 0)]
   - This will take the true branch for finding the minimum y-coordinate.
2. **Test Case 2:**
   - Inputs: [(2, 2), (2, 2), (3, 3)]
   - This will test the scenario where y-coordinates are equal, triggering the branch for checking x-coordinates.
3. **Test Case 3:**
   - Inputs: [(1, 2), (1, 1), (2, 3)]
   - This ensures the flow takes the false branch when checking for new minimum y-coordinates and the leftmost check.

c. Basic Condition Coverage

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 conditions for the y-coordinate comparisons.
2. **Test Case 2:**
   - Inputs: [(1, 1), (1, 1), (1, 2)]
   - This checks the scenario where the y-coordinates are the same but evaluates 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, confirming the function's logic is robust.

- ❖ **Using a mutation testing tool, identify any mutations of the code (such as deletions, modifications, or insertions) that would cause a failure but are not detected by your current test set.**

❖ Types of Possible Mutations :

We can apply typical mutation types, including:

- **Relational Operator Changes**: Modify <=to <or ==to !=in the conditions.
- **Logic Changes**: Remove or invert a branch in an if-statement.
- **Statement Changes**: Modify assignments or statements to see if the effect goes undetected.

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).x** to**p.get(i).x <= p.get(min).x**.
   - **Effect** : This could cause the function to select points that share the same x-coordinate as the leftmost point, potentially disrupting the uniqueness of the minimum point.

- **Not Detected by Current Tests** : The existing tests do not address the scenario where multiple points have the same x and y values, which would indicate whether the function incorrectly accepts such points as the leftmost.

2. **Altering the y-Coordinate Comparison to <=in the First Loop**:
   - **Mutation**: Change **p.get(i).y < p.get(min).y to p.get(i).y <= p.get(min).y** inthe first loop.
   - **Effect**: This would permit points with the same y-coordinate but different x-coordinates to overwrite the minimum, potentially resulting in the selection of a non-leftmost minimum point.
   - **Undetected by Current Tests**: The current test set does not include cases where multiple points share the same y-coordinate, allowing this mutation to remain undetected. To uncover this issue, we would need a test case where several points have the same y-coordinate but different x-coordinates.
3. **Removing the Check for x-coordinate in the Second Loop**:
   - **Mutation** : Remove the condition **p.get(i).x < p.get(min).x** in the second loop.
   - **Effect** : This would lead the function to choose any point that has the same minimum y-coordinate as the "leftmost," irrespective of its x-coordinate.
   - **Undetected by Current Tests**: The current tests do not explicitly verify whether the correct leftmost point is selected when multiple points have the same y value but different x values.

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 test case will confirm if using <=for ycomparisons mistakenly 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 additional test cases would help ensure that any such mutations do not survive undetected by the test suite, strengthening the coverage.

❖ 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 # Collinearelif val > 0:

        return 1 # Clockwiseelse:

        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

        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 inrange(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)
```