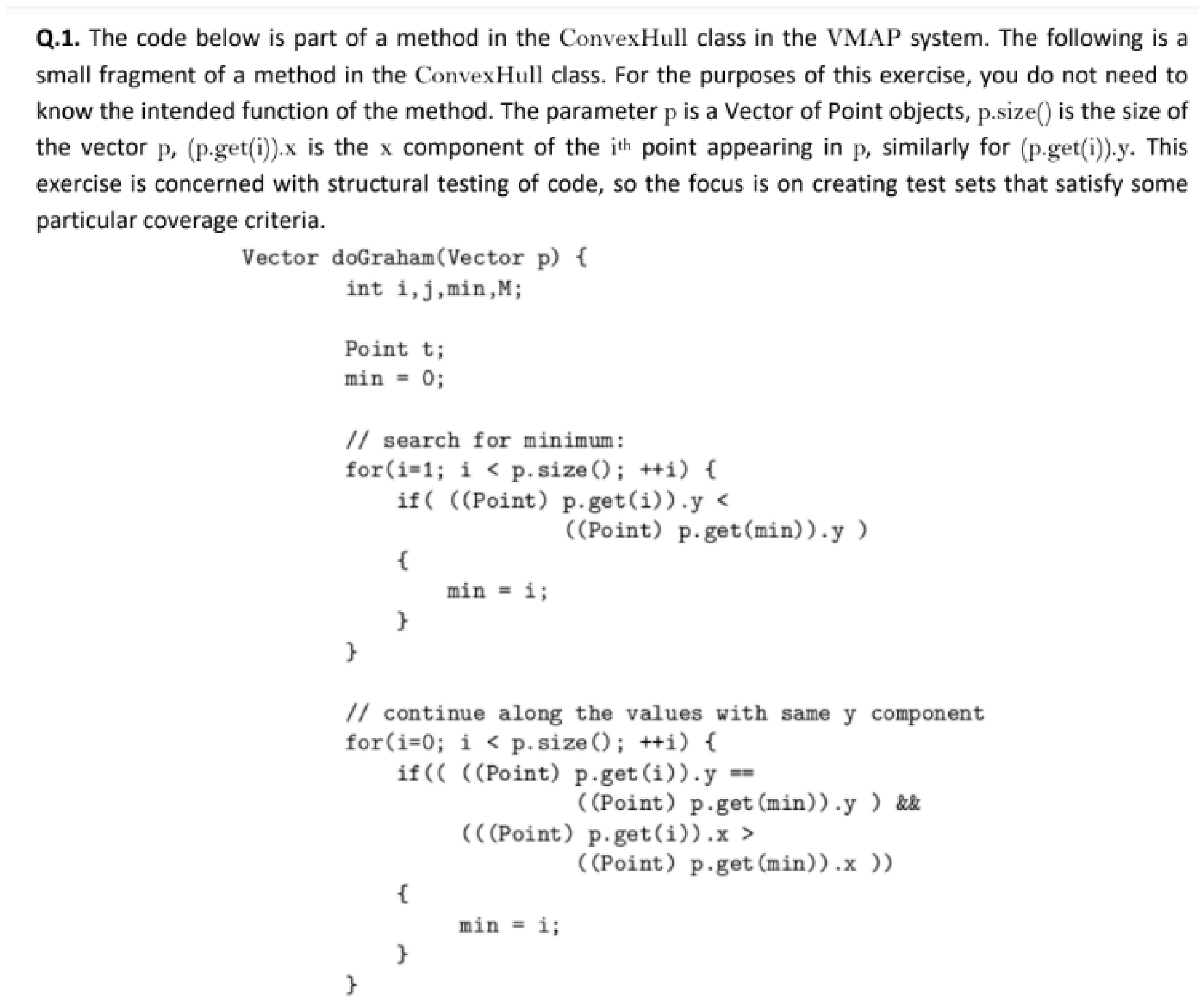
Software Engineering(IT-314)
LAB 9 MUTATION TESTING

Vraj Dobariya- 202201106



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. This exercise is concerned with structural testing of code, so the focus is on creating test sets that satisfy some particular coverage criteria.

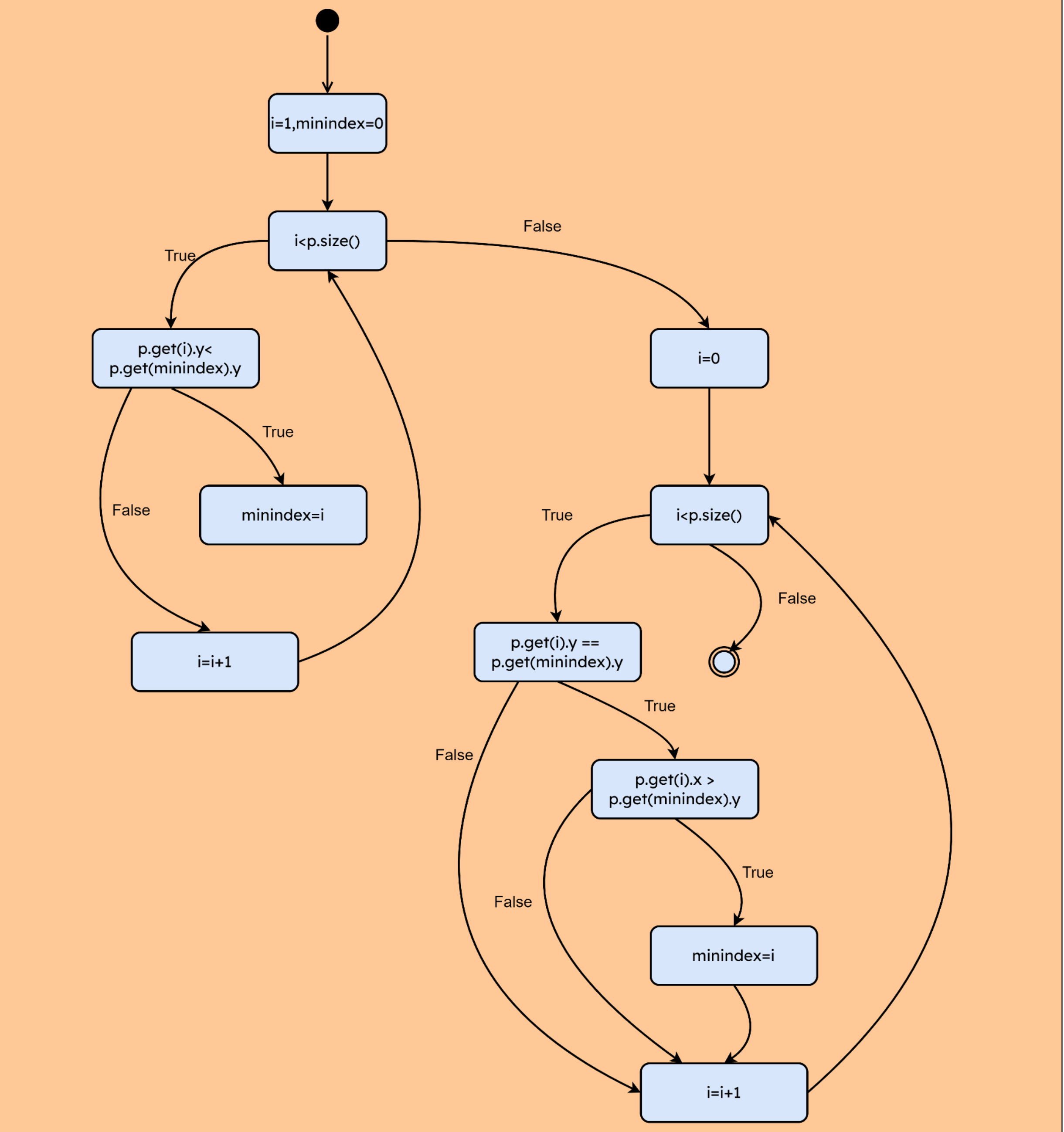
```
Vector doGraham(Vector p) {
        int i,j,min,M;

        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;
            }
        }
```

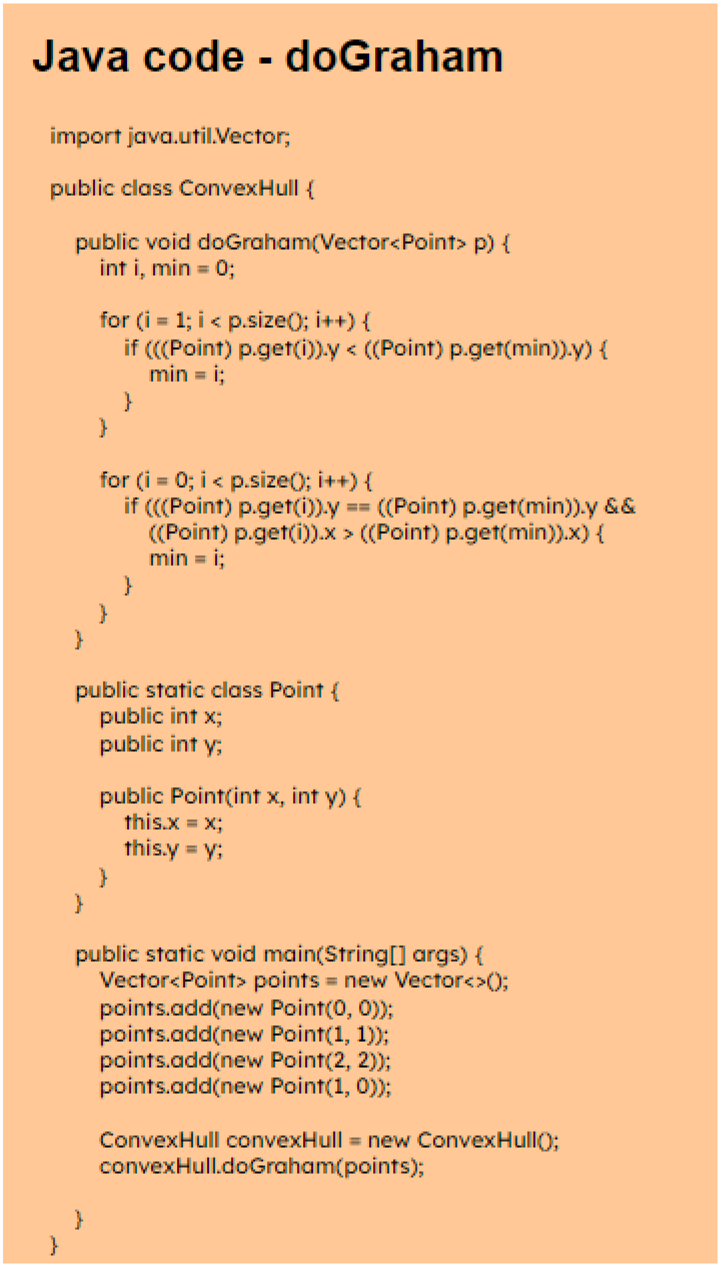
For the given code fragment, you should carry out the following activities.

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.

Control Flow Graph - doGraham





Java code - doGraham

```
import java.util.Vector;

public class ConvexHull {

    public void doGraham(Vector<Point> p) {
        int i, min = 0;

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

        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;
            }
        }
    }

    public static class Point {
        public int x;
        public int y;

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

    public static void main(String[] args) {
        Vector<Point> points = new Vector<>();
        points.add(new Point(0, 0));
        points.add(new Point(1, 1));
        points.add(new Point(2, 2));
        points.add(new Point(1, 0));

        ConvexHull convexHull = new ConvexHull();
        convexHull.doGraham(points);

    }
}
```

Control Flow Graph Factory Tool YES
Eclipse flow graph generator YES

a) Statement Coverage:

Test Case 1:

- Input: p = [(0, 1), (1, 2), (2, 3)]

- Explanation: This input ensures we go through both loops and perform minimum checks in both y and x comparisons.
- Expected Outcome: index 2

b) Branch Coverage:

Test Case 2:

- Input: p = [(1, 3), (2, 1), (3, 3)]
- Explanation: This input allows the code to take both paths in p.get(i).y < p.get(min).y and p.get(i).y == p.get(min).y. The x-comparison will also be tested when y values are equal.
- Expected Outcome: index 2

Test Case 3:

- Input: p = [(0,3),(1,3),(2,3)]
- Explanation: Ensures the code covers cases where multiple points have the same y value and tests the branch where x values are compared.
- Expected Outcome: Index 2

c) Basic Condition Coverage:

Test Case 4:

- Input: p = [(2, 2), (1, 1), (: This set allows for basic condition testing where each part of p.get(i).y < p.get(min).y, p.get(i).y == p.get(min).y, and p.get(i).x > p.get(min).x evaluates as both true and false.
- Expected Outcome: index 2

Test Case 5:

- Input: p = [(1, 1), (1, 1), (2, 2)]
- Explanation: This input tests both true and false branches of each condition in isolation.
- Expected Outcome: Since the first two points are identical, the second loop tests the y equality and x comparison in a controlled manner. Min should be updated to reflect the highest x among points with the smallest y.

Identifying Undetected Code Mutations:

For the test suite you have recently analyzed, can you pinpoint a mutation in the code (such as a deletion, alteration, or addition) that would result in a failure but is not captured by your current tests? This task should be performed using a mutation testing tool.

Types of Possible Mutations

Several common mutation types can be applied, including:

- **Changes to Relational Operators:** Modify `<=` to `<` or switch `==` to `!=` in conditional statements.

- **Logic Modifications:** Remove or invert branches in if-statements.
- **Statement Adjustments:** Alter assignments or statements to see if the outcome goes unnoticed.

Potential Mutations and Their Consequences:

Modifying the Comparison for the 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.
- **Consequence:** This change could lead to the selection of points sharing the same x-coordinate as the leftmost point, undermining the uniqueness of the minimum point.

Undetected by Current Tests: The existing test cases do not address situations where multiple points have identical x and y values, which would highlight if the function mistakenly includes such points as the leftmost.

Changing the y-Coordinate Comparison to <= in the First Loop:

- **Mutation:** Alter p.get(i).y < p.get(min).y to p.get(i).y <= p.get(min).y in the first loop.
- **Consequence:** This could allow points with the same y-coordinate but different x-coordinates to overwrite the minimum, potentially selecting a non-leftmost minimum point.

Undetected by Current Tests: The current test set lacks scenarios with multiple points sharing the same y-coordinate, which could cause this mutation to remain undetected. To expose this issue, a test with points having the same y but different x values is necessary.

Eliminating the x-coordinate Check in the Second Loop:

- **Mutation:** Remove the condition p.get(i).x < p.get(min).x from the second loop.
- **Consequence:** This would permit the selection of any point with the minimum y-coordinate as the "leftmost," irrespective of its x-coordinate.

Undetected by Current Tests: The existing tests do not verify whether the correct leftmost point is selected when multiple points share the same y-coordinate but have different x values.

Additional Test Cases to Identify These Mutations:

To effectively detect these mutations, consider implementing the following test cases:

Test Case for Mutation 1:

- **Input:** [(0, 1), (0, 1), (1, 1)]
- **Expected Outcome:** The leftmost minimum should remain (0, 1) despite duplicates. This case will check if the x <= mutation incorrectly includes duplicate points.

Test Case for Mutation 2:

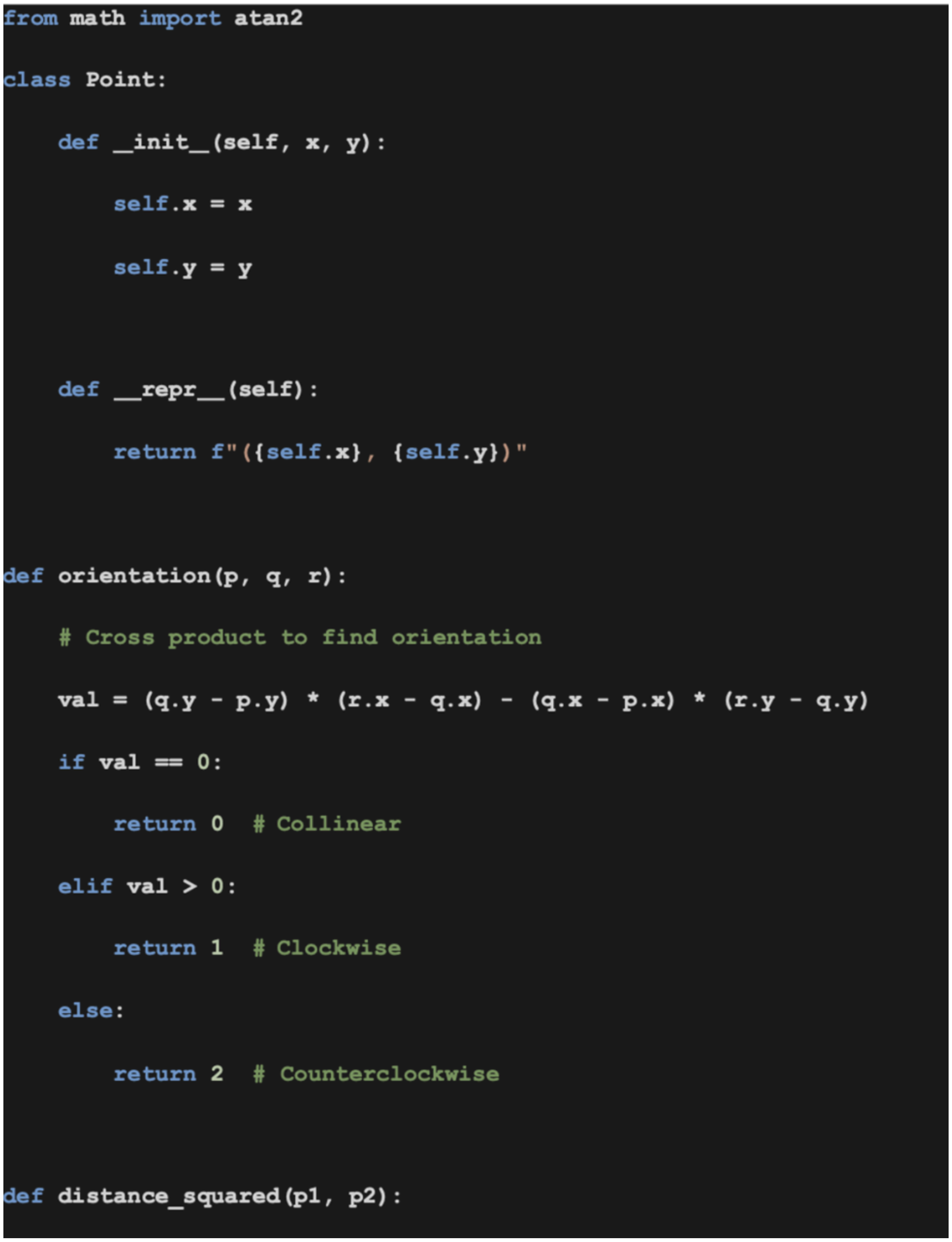
- **Input:** [(1, 2), (0, 2), (3, 1)]
- **Expected Outcome:** The function should identify (3, 1) as the minimum point based on the y-coordinate. This test will confirm whether using <= for y comparisons erroneously overwrites the minimum point.

Test Case for Mutation 3:

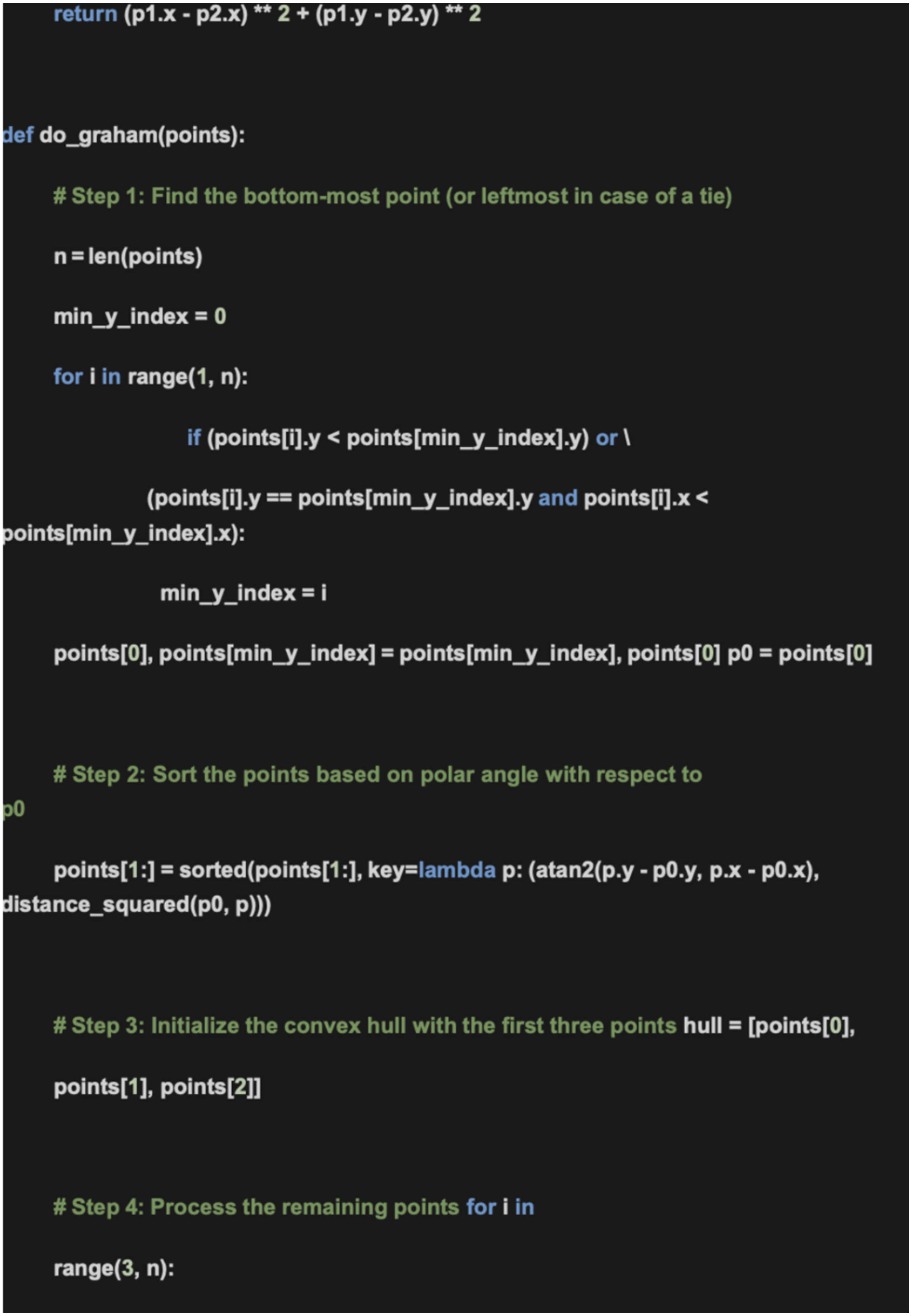
- **Input:** [(2, 1), (1, 1), (0, 1)]
- **Expected Outcome:** The leftmost point should be (0, 1). This case will help determine if the x-coordinate check was incorrectly removed.

By adding these specific test cases, you can strengthen the test suite to ensure that these mutations are effectively caught.

Python Mutation CODE



```python
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

    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):
```

```python
        # 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)
```