SOFTWARE ENGINEERING Lab Session – Mutation Testing

Sakshi Shah -202201281

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.

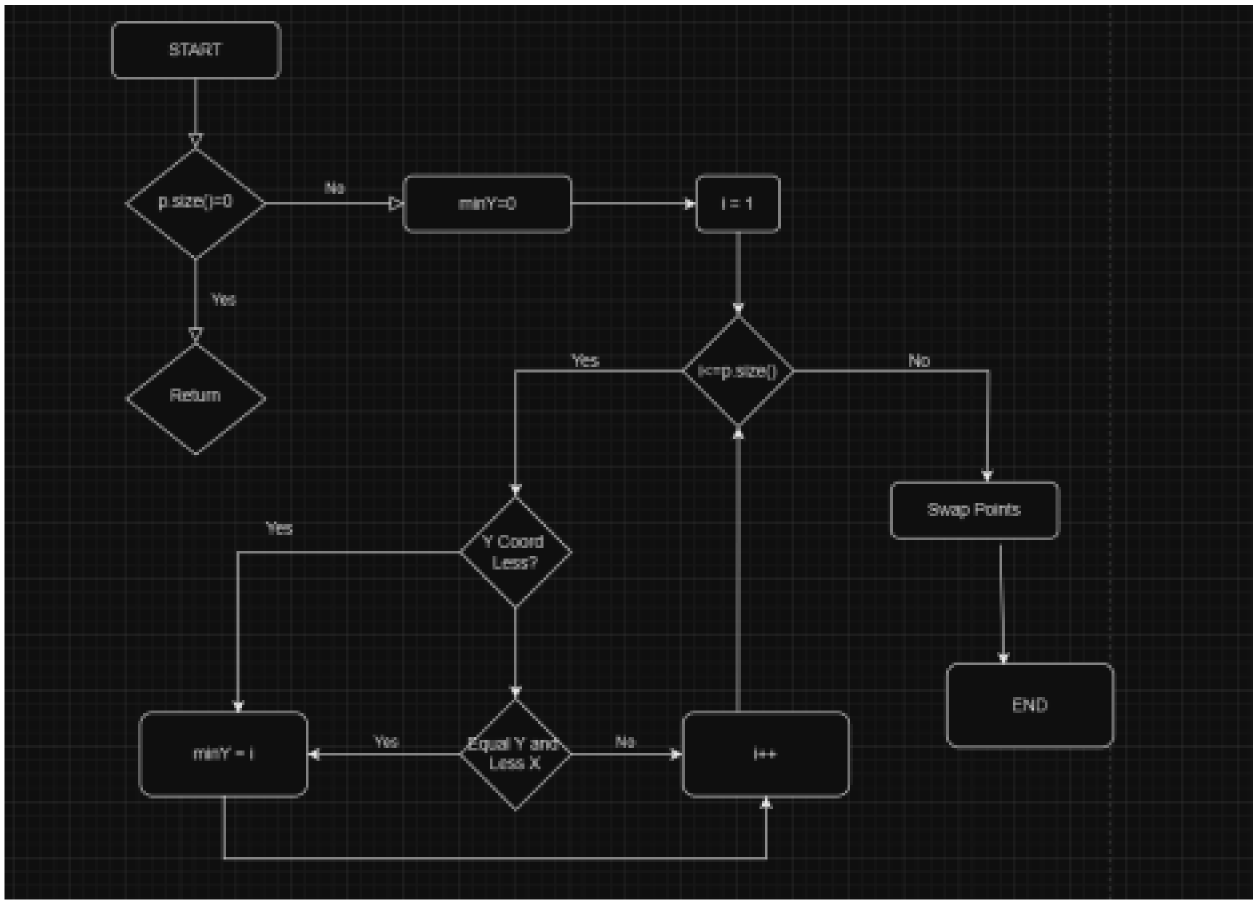
```
public class Point {
double x;
double y;
public Point(double x, double y) {
this.x = x;
this.y = y;
}
}
public class ConvexHull {
public void doGraham(Vector<Point> p) {
if (p.size() == 0) {
return;
}
int minY = 0;
for (int i = 1; i < p.size(); i++) {
if (p.get(i).y < p.get(minY).y ||
(p.get(i).y == p.get(minY).y && p.get(i).x <

p.get(minY).x)) {

minY = i;
}

}
Point temp = p.get(0);
p.set(0, p.get(minY));
p.set(minY, temp);
}
}
```

-Control Flow graph of above code



Q2. Develop test sets for your flow graph that fulfill the following criteria:

a. Statement Coverage

To achieve statement coverage, we need to ensure that every line of code is executed at least once.

Test Cases:

- **Test Case 1:** Provide an empty vector p (i.e., p.size() == 0). **Expected Outcome:** The function should terminate immediately without further execution.
- **Test Case 2:** Use a vector p that contains at least one Point element. **Expected Outcome:** The function should continue to locate the Point with the minimum y-coordinate.

b. Branch Coverage

For branch coverage, it's essential that every decision (branch) in the code is evaluated as both true and false at least once.

Test Cases:

- **Test Case 1:** An empty vector p (i.e., p.size() == 0).
  Expected Outcome: The condition if (p.size() == 0) evaluates to true, causing the function to return immediately.
- **Test Case 2:** A vector p containing one Point (e.g., Point(0, 0)). **Expected Outcome:** The condition if (p.size() == 0) evaluates to false, leading the function to enter the for loop, which does not iterate due to the presence of only one point.
- **Test Case 3:** A vector p with multiple Points where none has a y-coordinate less than that of p[0].
  Example: p = [Point(0, 0), Point(1, 1), Point(2, 2)].
  Expected Outcome: The if condition inside the loop remains false, resulting in minY staying at 0.
- **Test Case 4:** A vector p with multiple Points where another Point has a smaller y-coordinate than p[0].
  Example: p = [Point(2, 2), Point(1, 0), Point(0, 3)].
  Expected Outcome: The if condition within the loop becomes true, leading to an update of minY.

c. Basic Condition Coverage

To ensure basic condition coverage, we must test each condition independently within the branches.

Test Cases:

- **Test Case 1:** An empty vector p (i.e., p.size() == 0).
  Expected Outcome: The condition if (p.size() == 0) is true.
- **Test Case 2:** A non-empty vector p (i.e., p.size() > 0).
  Expected Outcome: The condition if (p.size() == 0) is false.
- **Test Case 3:** Multiple points where only the condition p.get(i).y < p.get(minY).y holds true.
  Example: p = [Point(1, 1), Point(0, 0), Point(2, 2)].
  Expected Outcome: The first condition p.get(i).y < p.get(minY).y is true, prompting an update to minY.
- **Test Case 4:** Multiple points where both p.get(i).y == p.get(minY).y and p.get(i).x < p.get(minY).x are true.
  Example: p = [Point(1, 1), Point(0, 1), Point(2, 2)].
  Expected Outcome: Both conditions evaluate to true, leading to an

update of minY.

Q3. Can you identify a mutation of the code (such as the deletion, modification, or addition of code) that could lead to failure but would not be detected by the provided test set? Use a mutation testing tool to assist with this.

1. Deletion Mutation

Mutation: Eliminate the assignment of minY to 0 at the beginning of the method.

```
public class ConvexHull {
   public void doGraham(Vector<Point> p) {
       if (p.size() == 0) {
          return;
       }

       // minY initialization removed
       for (int i = 1; i < p.size(); i++) {
          if (p.get(i).y < p.get(minY).y ||
             (p.get(i).y == p.get(minY).y && p.get(i).x < p.get(minY).x)) {
             minY = i;
          }
       }

       Point temp = p.get(0);
       p.set(0, p.get(minY));
       p.set(minY, temp);
   }
}
```

Impact: This mutation results in minY being uninitialized upon access, potentially leading to undefined behavior. The existing test cases do not verify the proper initialization of minY, allowing such faults to go unnoticed.

2. Insertion Mutation

Mutation: Add a line that incorrectly overrides minY based on an irrelevant condition.

```
public class ConvexHull {
    public void doGraham(Vector<Point> p) {
        if (p.size() == 0) {
            return;
        }

        int minY = 0;
        if (p.size() > 1) {
            minY = 1; // Insertion mutation
        }

        for (int i = 1; i < p.size(); i++) {
            if (p.get(i).y < p.get(minY).y ||
                (p.get(i).y == p.get(minY).y && p.get(i).x < p.get(minY).x)) {
                minY = i;
            }
        }

        Point temp = p.get(0);
        p.set(0, p.get(minY));
        p.set(minY, temp);
    }
}
```

Impact: This mutation sets minY to 1 without justifiable conditions. If the tests do not specifically check the final arrangement of points after execution, this error might remain undetected.

3. Modification Mutation
Mutation: Change the logical operator from || to && in the conditional statement.

```
public class ConvexHull {
    public void doGraham(Vector<Point> p) {
        if (p.size() == 0) {
            return;
        }
        int minY = 0;
        for (int i = 1; i < p.size(); i++) {
            if (p.get(i).y < p.get(minY).y && // Modification mutation
                (p.get(i).y == p.get(minY).y && p.get(i).x < p.get(minY).x)) {
                minY = i;
            }
        }

        Point temp = p.get(0);
        p.set(0, p.get(minY));
        p.set(minY, temp);
    }
}
```
Impact: The alteration of the logical operator may not result in an immediate crash, and the current test cases do not validate whether minY updates correctly under these modified conditions, potentially allowing this error to go unnoticed.

Q4. Design a test set that meets the path coverage criterion, ensuring that every loop is explored at least zero, one, or two times.

Test Case 1: Zero Iterations

- **Input**: An empty vector `p = []`.
- **Description**: This test ensures that no iteration occurs because the vector is empty, meaning no points are processed.
- **Expected Outcome**: The function should handle the empty input gracefully. Ideally, it should either return an empty result or a value indicating that no points were processed (e.g., `null`, `empty`, or a specific error message).

Test Case 2: One Iteration (First Loop)

- **Input**: A vector with one point, `p = [(3, 4)]`.
- **Description**: This test ensures that the first loop runs exactly once, as there is only one point in the vector.
- **Expected Outcome**: The function should return the single point `[(3, 4)]`, as no other point is available for further processing.

Test Case 3: One Iteration (Second Loop)

- **Input**: A vector with two points that have the same `y`-coordinate but different `x`-coordinates, `p = [(1, 2), (3, 2)]`.
- **Description**: This case ensures that the first loop identifies the minimum point, and the second loop processes one iteration, comparing the `x`-coordinates of the two points.
- **Expected Outcome**: The function should return the point `(3, 2)`, which has the maximum `x`-coordinate among the two points.

Test Case 4: Two Iterations (First Loop)

- **Input**: A vector with multiple points, including points with the same `y`-coordinate, `p = [(3, 1), (2, 2), (5, 1)]`.
- **Description**: This test case ensures that the first loop runs and identifies the point with the minimum `y`-coordinate, and continues to the second loop.
- **Expected Outcome**: The function should return `(5, 1)`, as it has the highest `x`-coordinate among the points that share the same minimum `y`-coordinate `(1)`.

Test Case 5: Two Iterations (Second Loop)

- **Input**: A vector with multiple points where more than one point has the same `y`-coordinate, `p = [(2, 3), (4, 3), (1, 3)]`.
- **Description**: This test case ensures that the second loop correctly processes all points and finds the point with the maximum `x`-coordinate.
- **Expected Outcome**: The function should return `(4, 3)`, which has the highest `x`-coordinate among the points that share the same `y`-coordinate `(3)`.