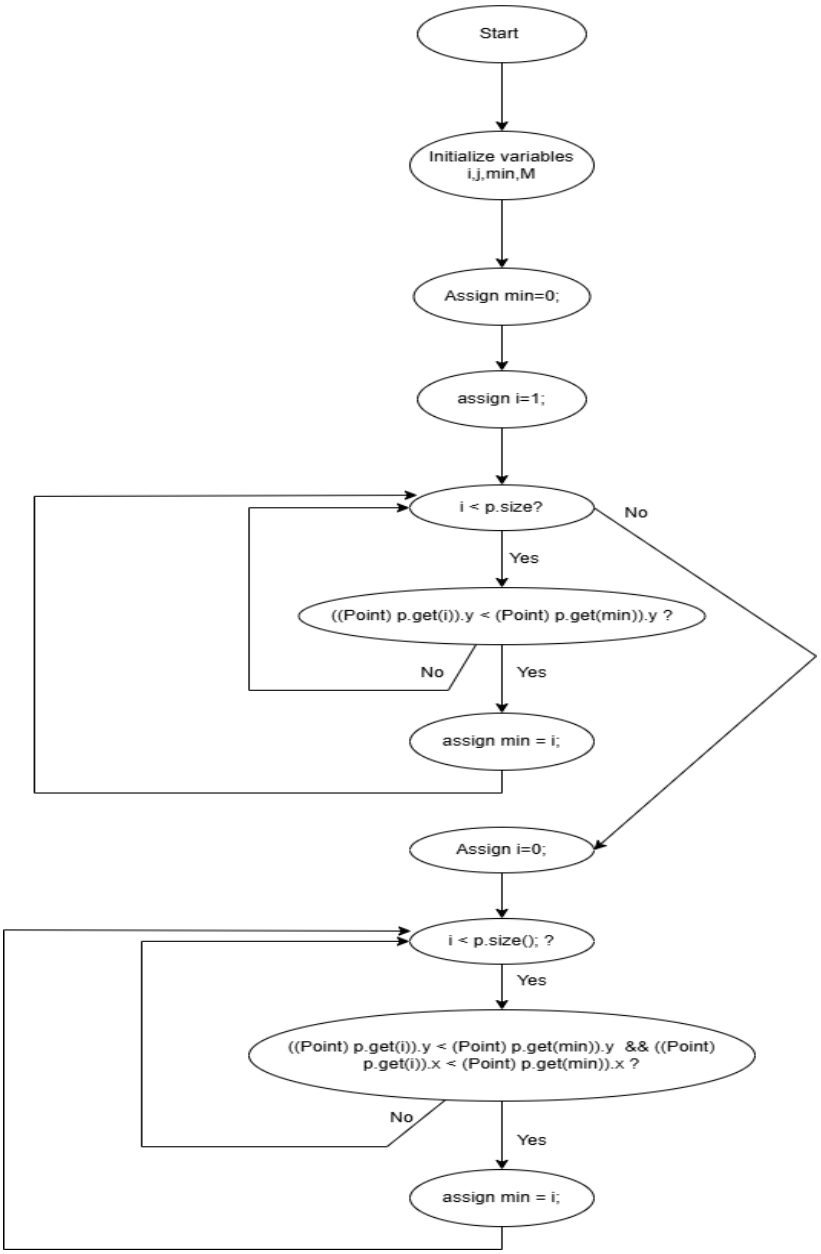
Software Engineering

Lab - 09 (Mutation testing)

Name- Alpesh Yadav
ID – 202201264

Q1.



- Objective: Develop comprehensive test sets that ensure adequate coverage of the following criteria in the flow graph:

- Statement Coverage: Design test cases to execute every executable statement in the flow graph at least once.

- Branch Coverage: Develop test cases that execute every possible branch (decision outcome) in the flow graph to ensure that all possible paths are taken.

- Basic Condition Coverage: Create test cases that ensure each basic condition in the decision nodes is evaluated to both true and false at least once, regardless of other conditions in the same decision.

1. Statement Coverage

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

Test Set:

- **Test Case 1:**
  - **Inputs:** A list with more than one point, e.g., [(0, 1), (1, 2), (2, 0)]
  - **Description:** This test will traverse the entire flow, covering statements related to finding the minimum y-coordinate and the leftmost minimum point.
- **Test Case 2:**
  - **Inputs:** A list with points having equal y-coordinates, e.g., [(2, 2), (2,2),(3, 3)]
  - **Description:** This test checks for points with the same y-coordinate, ensuring the logic for selecting the leftmost point is executed correctly.

2. Branch Coverage

Objective: Ensure that every decision point in the flow graph has both its true and false branches executed.

Test Set:

- **Test Case 1:**
  - **Inputs:** [(0, 1), (1, 2), (2, 0)]
  - **Description:** This test ensures that the true branch for finding the minimumy-coordinate is executed.
- **Test Case 2:**
  - **Inputs:** [(2, 2),(2,2) ,(3, 3)]

- **Description:** This test case ensures that the branch checking x-coordinates istriggered when y-coordinates are equal.

- **Test Case 3:**
  - **Inputs:** [(1, 2), (1, 1), (2, 3)]
  - **Description:** This test ensures the false branch is taken when checking for new minimum y-coordinates, as well as ensuring the leftmost check is executed.

3. Basic Condition Coverage

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

Test Set:

- **Test Case 1:**
  - **Inputs:** `[(1, 1), (2, 2), (3, 3)]`
  - **Description:** This test will evaluate both the true and false conditions for comparing the y-coordinates.
- **Test Case 2:**
  - **Inputs:** `[(1, 1), (1, 1), (1, 2)]`
  - **Description:** This test ensures the x-coordinate comparison is tested when multiple points have the same y-coordinate.
- **Test Case 3:**
  - **Inputs:** `[(3, 1), (2, 2), (1, 3)]`
  - **Description:** This test ensures that both conditions in the loop (y-coordinate and x-coordinate) are tested, confirming the correctness of the logic.

Mutation Testing for Undetected Failures

Objective: Identify mutations in the code that might lead to test failures but are not currently detected by the existing test set.

Mutation testing involves making small changes to the code, such as altering conditions or removing statements, to see if the tests can detect the issues.

Types of Mutations:

- **Relational Operator Changes:** For example, changing `<=` to `<` or `==` to `!=` in conditions.
- **Logic Changes:** Modifying or removing branches in if-statements.
- **Statement Changes:** Altering assignments or statements to observe if the changes go 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 mutation would result in the function selecting points with the same x-coordinate as the leftmost, which could break the uniqueness of the minimum point.
- **Undetected by Current Tests:** The current test set does not include cases where multiple points have the same y- and x-values. This would fail if the function allows multiple points with identical coordinates to be selected as the leftmost.

2. **Altering the y-Coordinate Comparison to `<=`:**
   - **Mutation:** Change `p.get(i).y < p.get(min).y` to `p.get(i).y <= p.get(min).y` in the first loop.
   - **Effect:** This change would allow points with the same y-coordinate but different x-coordinates to overwrite the minimum y-coordinate, potentially selecting a non-leftmost point.
   - **Undetected by Current Tests:** The current test cases do not cover scenarios where multiple points have the same y-coordinate. This mutation would not be detected unless we add tests with multiple points sharing 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:** Without this condition, the function would select any point with the same minimum y-coordinate as the "leftmost," disregarding its x-coordinate.
   - **Undetected by Current Tests:** The current test set does not specifically test for points with identical y but different x values, meaning this mutation would go undetected unless a case involving such points is added.

Additional Test Cases to Detect These Mutations

To detect the mutations described above, the following additional test cases should be added:

1. **Detecting Mutation 1 (Leftmost Point Comparison):**
   - **Test Case:** `[(0, 1), (0, 1), (1, 1)]`
   - **Expected Result:** The leftmost minimum should be `(0, 1)` even if there are duplicate points with the same coordinates.
   - **Purpose:** This test case will verify that the `<=` mutation does not mistakenly allow duplicate points with the same coordinates.
2. **Detecting Mutation 2 (y-Coordinate Comparison to `<=`):**
   - **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.

- **Purpose:** This test case ensures that changing the `y < min.y` condition to `y <= min.y` does not incorrectly overwrite the minimum point.

3. **Detecting Mutation 3 (Removing x-Coordinate Check):**
   - **Test Case:** `[(2, 1), (1, 1), (0, 1)]`
   - **Expected Result:** The leftmost point `(0, 1)` should be chosen, as it has the smallest x-coordinate among those with the same y-coordinate.
   - **Purpose:** This test case will reveal if the condition to check the x-coordinate was mistakenly removed.

Conclusion

The additional test cases described above will help detect the mutations not covered by the current test set, ensuring that the function behaves as expected and maintains its correctness even with small changes in the code.

By applying mutation testing, we can enhance the robustness of the test suite and improve code quality by ensuring that edge cases and potential logic errors are properly handled.

Mutated Code :

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <cmath>

using namespace std;

class Point
{
public:
    double x, y;


    Point(double x, double y) : x(x), y(y) {}


    friend ostream &operator<<(ostream &os, const Point &p)
    {
```

```
        os << "(" << p.x << ", " << p.y << ")";
        return os;
    }
};

int orientation(const Point &p, const Point &q, const Point &r)
{
    double val = (q.y - p.y) * (r.x - q.x) - (q.x - p.x) * (r.y - q.y);
    if (val == 0)
        return 0;
    return (val > 0) ? 1 : 2;
}


double distanceSquared(const Point &p1, const Point &p2)
{
    return (p1.x - p2.x) * (p1.x - p2.x) + (p1.y - p2.y) * (p1.y - p2.y);
}


vector<Point> doGraham(vector<Point> &points)
{
    int n = points.size();

    // Step 1: Find the bottom-most (or leftmost in case of a tie) point
    int minYIndex = 0;
    for (int i = 1; i < n; i++)
    {
        if (points[i].y < points[minYIndex].y ||
            (points[i].y == points[minYIndex].y && points[i].x <
points[minYIndex].x))
        {
            minYIndex = i;
        }
    }

    // Swap the point at minYIndex with the first point
    swap(points[0], points[minYIndex]);
    Point p0 = points[0];
```

```
    // Step 2: Sort the points based on the polar angle with respect to p0
    sort(points.begin() + 1, points.end(), [&p0](const Point &p1, const
Point &p2)
         {
        double angle1 = atan2(p1.y - p0.y, p1.x - p0.x);
        double angle2 = atan2(p2.y - p0.y, p2.x - p0.x);
        if (angle1 != angle2) {
            return angle1 < angle2;
        } else {
            return distanceSquared(p0, p1) < distanceSquared(p0, p2);
        } });

    // Step 3: Initialize the convex hull with the first three points
    vector<Point> hull;
    hull.push_back(points[0]);
    hull.push_back(points[1]);
    hull.push_back(points[2]);

    // Step 4: Process the remaining points
    for (int i = 3; i < n; i++)
    {
        // Mutation introduced here: instead of checking != 2, we
incorrectly use == 1
        while (hull.size() > 1 && orientation(hull[hull.size() - 2],
hull[hull.size() - 1], points[i]) == 1)
        {
            hull.pop_back();
        }
        hull.push_back(points[i]);
    }

    return hull;
}

int main()
{

    vector<Point> 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)};
```

```
    vector<Point> hull = doGraham(points);


    cout << "Convex Hull:\n";
    for (const Point &p : hull)
    {
        cout << p << " ";
    }

    return 0;
}
```