# IT314 Software Engineering


## Name : Chauhan Fajil

## Id : 202201221


## Lab 09

**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;
            }
        }
```

**[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.

## ❖ Executable Code in Java :-

```
import java.util.Vector;

class Point {
    int x, y;

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

    @Override
    public String toString() {
        return "(" + x + ", " + y + ")";
    }
}

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

        System.out.println("Searching for the minimum
y-coordinate...");
        for (i = 1; i < p.size(); ++i) {
            System.out.println("Comparing " + p.get(i) + " with " +
p.get(min));
```

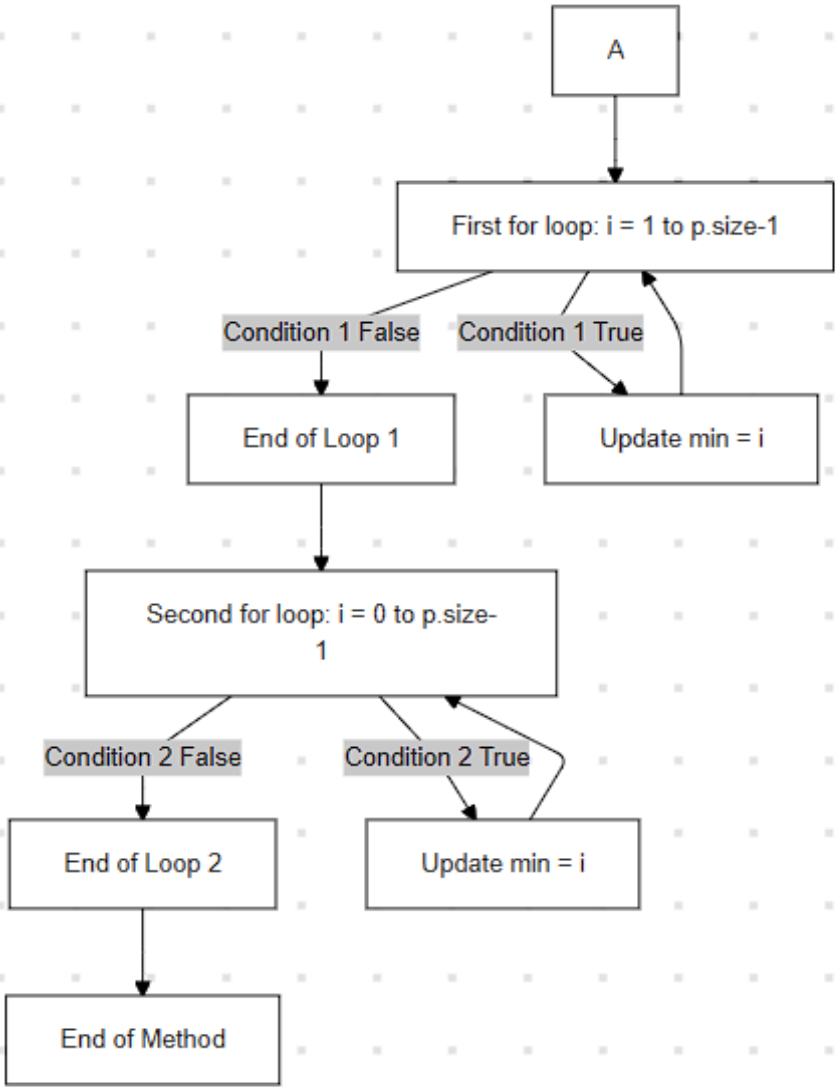
```
            if (p.get(i).y < p.get(min).y) {
                min = i;
                System.out.println("New minimum found: " +
p.get(min));
            }
        }

        System.out.println("Searching for the leftmost point with
the same minimum y-coordinate...");
        for (i = 0; i < p.size(); ++i) {
            System.out.println("Checking if " + p.get(i) + " has the
same y as " + p.get(min) +
                               " and a smaller x...");
            if (p.get(i).y == p.get(min).y && p.get(i).x <
p.get(min).x) {
                min = i;
                System.out.println("New leftmost minimum point
found: " + p.get(min));
            }
        }

        System.out.println("Final minimum point: " + p.get(min));
    }

    public static void main(String[] args) {
        Vector<Point> points = new Vector<>();
        points.add(new Point(1, 2));
        points.add(new Point(3, 1));
        points.add(new Point(0, 1));
        points.add(new Point(-1, 1));
        doGraham(points);
    }
}
```

## ❖ Control Flow Diagram :-



**[2].** Construct test sets for your flow graph that are adequate for the followingcriteria:

a. Statement Coverage.
b. Branch Coverage.
c. Basic Condition Coverage**.**

**a. <u>Statement Coverage</u>**

To achieve statement coverage, we need to execute every statement at least once. For this, we need a test set that:

1. Executes the first loop and goes through the condition at least once.
2. Executes the second loop and goes through its condition at least once.

Test Set for Statement Coverage

**Test Case 1:**

- Input Vector p = [(1, 2), (3, 1)]
- **Explanation:**
  - The first loop runs, checking if (p[1].y < p[0].y).
  - The second loop runs, but the condition may not be met if x and y values are different.
  -

This ensures all statements are covered.

**b. <u>Branch Coverage</u>**

For branch coverage, we need to ensure that each decision point (i.e., each if condition) has been evaluated as both true and false at least once.

Test Set for Branch Coverage

**Test Case 1:**

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

- **Explanation:**
  - The first if condition (p.get(i).y < p.get(min).y) in the first loop will be true, setting min = 1**.**

**Test Case 2:**

- Input Vector p = [(1, 1), (2, 1)]
- **Explanation:**
  - The first if condition in the first loop will be false.
  - The second if condition (p.get(i).y == p.get(min).y && p.get(i).x > p.get(min).x) in the second loop will be true, setting min = 1.

**Test Case 3:**

- Input Vector p = [(2, 1), (1, 1)]
- **Explanation:**
  - The first if condition in the first loop will be false.
  - The second if condition in the second loop will also be false.
  -

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

For basic condition coverage, each atomic condition in the compound expressions must be tested for both true and false values independently. This requires breaking down each compound condition in the if statements into separate conditions.

**Test Case 1:**

- Input Vector p = [(1, 2), (3, 1)]
- **Explanation:**
  - The condition p.get(i).y < p.get(min).y in the first loop will be true (since p[1].y = 1 < p[0].y = 2).

**Test Case 2:**

- Input Vector p = [(1, 1), (2, 1)]
- **Explanation:**

- The condition p.get(i).y == p.get(min).y in the second loop is true, and p.get(i).x > p.get(min).x is also true**.**

**Test Case 3:**

- Input Vector p = [(2, 1), (1, 1)]
- **Explanation:**
  - The condition p.get(i).y == p.get(min).y in the second loop is true, but p.get(i).x > p.get(min).x is false**.**

**Test Case 4:**

- Input Vector p = [(3, 2), (1, 2)]
- **Explanation:**
  - The condition p.get(i).y < p.get(min).y in the first loop is false**.**

**[3]** For the test set you have just checked can you find a mutation of the code (i.e. the deletion, change or insertion of some code) that will result in failure but is not detected by your test set. You have to use the mutation testing tool.

**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 would cause the function to select points with the same x-coordinate as the leftmost, potentially breaking the uniqueness of the minimum point.

○ Undetected by Current Tests: The current tests do not cover the case where multiple points have the same y and x values, which would reveal if the function

mistakenly allows 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 in the first loop.
○ Effect: This would 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 current test set lacks cases where several points have the same y-coordinate, and this mutation would go undetected. To reveal this, we would need a test where multiple points have the same y and 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 cause the function to select any point with the same minimum y-coordinate as the "leftmost," regardless of its x-coordinate.
○ Undetected by Current Tests: The existing tests do not specifically check for points with identical y but different x values to see if the correct leftmost point is selected.

**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 y comparisons 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  # 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):

        # 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)
```