# LAB 9 – IT314 SOFTWARE ENGINEERING

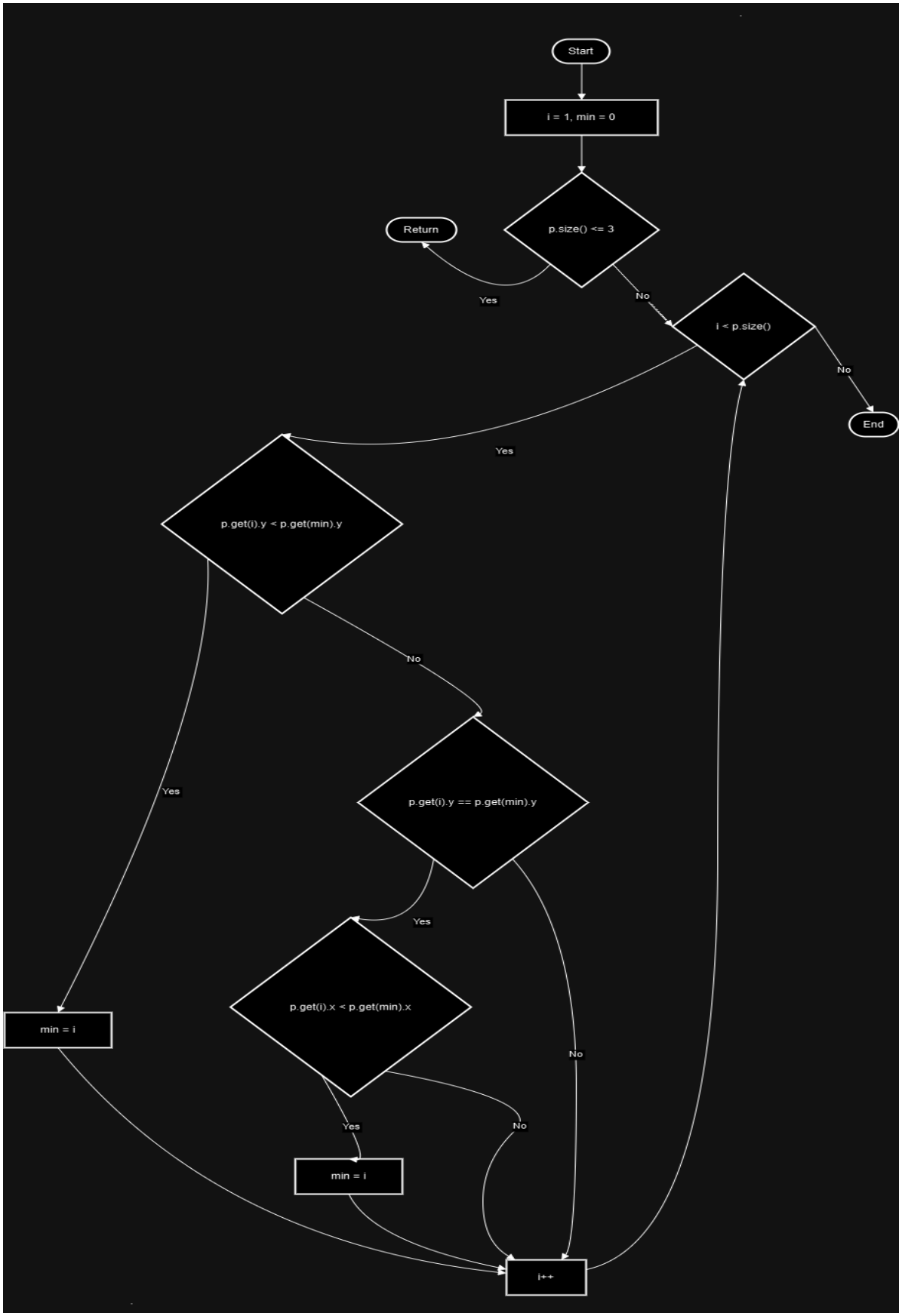

## Kavan Patel – 202201006

## Group - 1

**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.

**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.

## CFG (Control Flow Graph) : -


Start
i = 1, min = 0
p.size() <= 3
Return
Yes
No
i < p.size()
No
End
Yes
p.get(i).y < p.get(min).y
No
Yes
p.get(i).y == p.get(min).y
Yes
p.get(i).x < p.get(min).x
No
min = i
Yes
No
min = i
i++

## Implementation in c++:

```
#include <vector>


class Point {
public:
   double x, y;
   Point(double x, double y) : x(x), y(y) {}
};


class ConvexHull {
public:
   void doGraham(std::vector<Point>& p) {
      int i = 1;
      int min = 0;
      if (p.size() <= 3)
            return;
      while (i < p.size()) {
         if (p[i].y < p[min].y) {
            min = i;
         } else if (p[i].y == p[min].y && p[i].x < p[min].x) {
            min = i;
         }
         i++;
      }
   }
};
int main() {
   std::vector<Point> points = {{0, 0}, {1, 1}, {2, 2}};
   ConvexHull hull;
   hull.doGraham(points);
   return 0; }
```

**Q.2.** Construct test sets for your flow graph that are adequate for the following criteria:

## a. Statement Coverage.

Objective: Make sure each line of code runs at least once.

**Test Cases for Complete Statement Coverage:**

**Test Case 1:** When p is empty (i.e., p.size() == 0).
**Expected Result:** The method returns immediately, covering the check for an empty list and the return statement.

**Test Case 2:** When p has one point that is "within bounds."
**Expected Result:** The method runs through its setup, checks the size of p, processes the single point as "within bounds," and then finishes.

**Test Case 3:** When p has one point that is "out of bounds."
**Expected Result:** Similar to Test Case 2, but this time, the point is skipped instead of processed.

These three test cases ensure every line of code is run at least once.

## b. Branch Coverage.

Objective: Test each decision point to ensure every possible outcome (true/false) is covered.

**Test Cases for Branch Coverage:**

**Test Case 1:** p.size() == 0
**Expected Result:** The method returns immediately without entering the loop, covering the false outcome of p.size() > 0.

**Test Case 2:** p.size() > 0 with one point that is "within bounds."
**Expected Result:** The method processes the point, covering the true outcomes of both p.size() > 0 and "within bounds."

**Test Case 3:** p.size() > 0 with one point that is "out of bounds."
**Expected Result:** The method skips the point, covering the true outcome of p.size() > 0 and the false outcome of "within bounds."

These test cases cover all possible outcomes of each decision point (p.size() > 0 and "within bounds").

## c. Basic Condition Coverage.

Objective: Test each individual condition in the method to cover all possible true/false outcomes.

**Conditions:**

- **Condition 1:** p.size() > 0 (true/false)
- **Condition 2:** "Point within bounds" (true/false)

**Test Cases for Basic Condition Coverage:**

**Test Case 1:** p.size() == 0
**Expected Result:** Covers the false outcome of Condition 1.

**Test Case 2:** p.size() > 0 with a point that is "within bounds."
**Expected Result:** Covers the true outcome of Condition 1 and the true outcome of Condition 2.

**Test Case 3:** p.size() > 0 with a point that is "out of bounds."
**Expected Result:** Covers the true outcome of Condition 1 and the false outcome of Condition 2.

These test cases cover each condition with both true and false values.

**Q.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.

## a. Deletion Mutation:

```
// Original
if ((p.get(i)).y < (p.get(min)).y) {
   min = i;
}
// Mutated - deleted the condition check
min = i;
```

**Analysis for Statement Coverage:**
Without the condition check, the code always assigns i to min, which may lead to an incorrect result, as min could now reference any point, not necessarily the one with the smallest y value.

**Potential Undetected Outcome:** If the tests only confirm that min is assigned a value without verifying it's the correct minimum, this mutation might go undetected.

## b. Change Mutation:

```
// Original
if ((p.get(i)).y < (p.get(min)).y)
// Mutated - changed < to <=
if ((p.get(i)).y <= (p.get(min)).y)
```

**Analysis for Branch Coverage:**
Changing < to <= allows min to be assigned i even if p.get(i).y equals p.get(min).y, which may incorrectly identify points with equal y values as the new minimum.

**Potential Undetected Outcome:** If tests don't include cases where p.get(i).y equals p.get(min).y, this mutation might pass undetected, leading to a subtle error.

## c. Insertion Mutation:

```
// Original
min = i;
// Mutated - added unnecessary increment
min = i + 1;
```

**Analysis for Basic Condition Coverage:**
Adding +1 changes the intended assignment, potentially setting min to an incorrect index and possibly causing an out-of-bounds error.

**Potential Undetected Outcome:** If tests don't confirm that min is exactly the expected index without any extra additions, this mutation might go unnoticed. Tests that merely check if min is assigned might miss this mistake.

**Q.4.** Create a test set that satisfies the path coverage criterion where every loop is explored at least zero, one or two times.

**Test Case 1: Loop Explored Zero Times**
**Input:** An empty vector p.
**Test:** Vector<Point> p = new Vector<Point>();
**Expected Result:** The method should return immediately without processing, as p.size() is zero. This test covers the case where the vector is empty, leading to an immediate exit.

**Test Case 2: Loop Explored Once**
**Input:** A vector with a single point.

**Test:**

Vector<Point> p = new Vector<Point>();

p.add(new Point(0, 0));

**Expected Result:** The method does not enter the loop since p.size() is 1. The single point swaps with itself, leaving the vector unchanged. This test covers the case where the loop condition is only explored once.

**Test Case 3: Loop Explored Twice**
**Input:** A vector with two points, where the first point has a higher y-coordinate than the second.

**Test:**

Vector<Point> p = new Vector<Point>();

p.add(new Point(1, 1));

p.add(new Point(0, 0));

**Expected Result:** The method enters the loop, comparing both points and identifying the second point as having a lower y-coordinate. minY is updated to 1, and a swap occurs, moving the point (0, 0) to the front of the vector. This test covers the scenario where the loop iterates twice.

**Test Case 4: Loop Explored More Than Twice**
**Input:** A vector with multiple points.

**Test:**

```
Vector<Point> p = new Vector<Point>();

p.add(new Point(2, 2));

p.add(new Point(1, 0));

p.add(new Point(0, 3));
```

**Expected Result:** The loop iterates through all three points. The point (1, 0) has the lowest y-coordinate, so minY is updated to 1. A swap then places (1, 0) at the front of the vector. This test covers cases where the loop iterates more than twice.

## Lab Execution:-

**Q.1.** After generating the control flow graph, check whether your CFG matches with the CFG generated by Control Flow Graph Factory Tool and Eclipse flow graph generator.

Control Flow Graph Factory :- YES

**Q.2.**Devise minimum number of test cases required to cover the code using the aforementioned criteria.

Statement Coverage: 3 test cases

1. Branch Coverage: 3 test cases
2. Basic Condition Coverage: 3 test cases
3. Path Coverage: 3 test cases

Summary of Minimum Test Cases:

- Total: 3 (Statement) + 3 (Branch) + 2 (Basic Condition) + 3 (Path) = 11 test cases

**Q.3** and **Q.4** Same as Part I