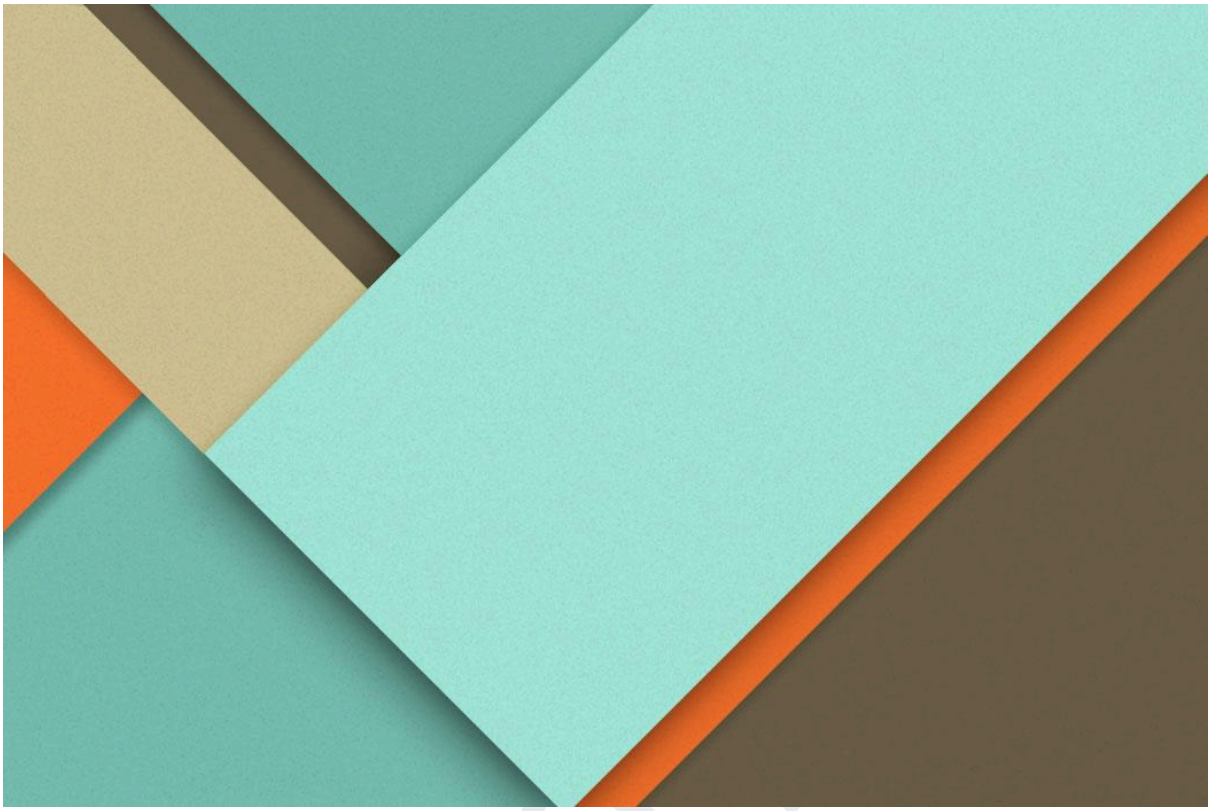


IT314 : Software Engineering

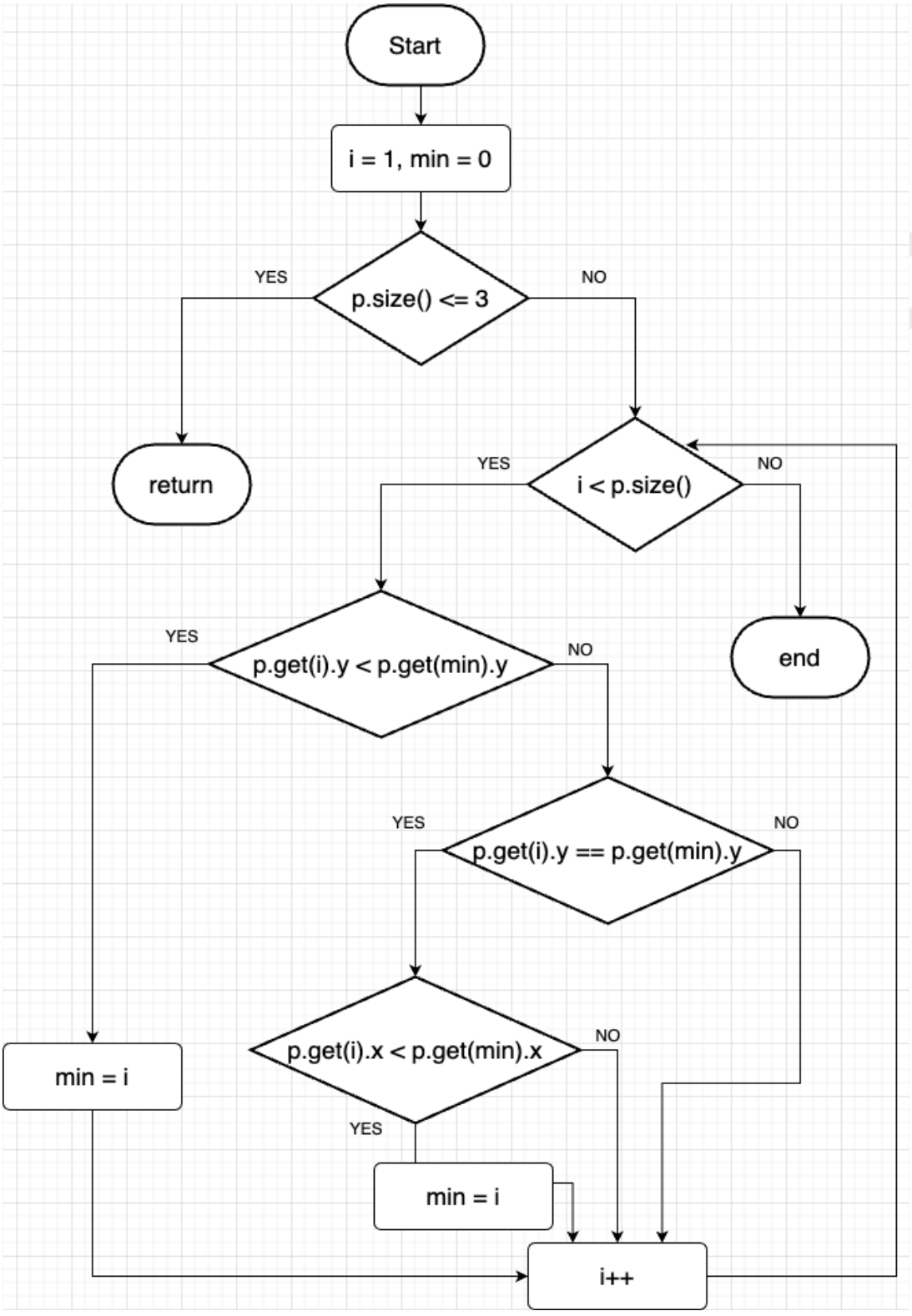
LAB - 09 : Mutation Testing

Name : Savani Vedant

ID : 202201178

Question 1 :

TASK - 1 : CFG (Control Flow Graph)



Implementation in C++ :

```cpp
#include <vector>
class Point {
public:
    double x, y;
    Point(double x, double y) {
    this->x = x;
    this->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) {
               if (p[i].x < p[min].x) {
                min = i;
            }
        }
        i++;
        }
    }
};
int main() {
    std::vector<Point> points;
    points.push_back(Point(0, 0));
    points.push_back(Point(1, 1));
    points.push_back(Point(2, 2));
    ConvexHull hull;
    hull.doGraham(points);
return 0;
}
```

TASK – 2 :

Statement Coverage :

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

Test Cases for Statement Coverage:

1. **Test Case 1**: p is empty (i.e., p.size() == 0).

   ○ Expected Outcome: The method directly returns, covering the initial check and return statements.

2. **Test Case 2**: p contains one point that is "within bounds."

   ○ Expected Outcome: The method initialises variables, checks p.size(), iterates through the single point, evaluates it as "within bounds," processes it, and then returns.

3. **Test Case 3**: p contains one point that is "out of bounds."

   ○ Expected Outcome: Similar to Test Case 2, except the point is skipped instead of processed.

These three test cases ensure that all statements in the method are executed at least once.

Branch Coverage :

Objective: Test each decision point with all possible outcomes (true/false) to cover every branch.

Test Cases for Branch Coverage:

1. **Test Case 1**: p.size() == 0.

   ○ Expected Outcome: The method directly returns without entering the loop, covering the false branch of the p.size() > 0 check.

2. **Test Case 2**: p.size() > 0 with one point that is "within bounds."

○ Expected Outcome: The method processes the point, covering the true branch of both p.size() > 0 and "within bounds."

3. **Test Case 3**: p.size() > 0 with one point that is "out of bounds."

   ○ Expected Outcome: The method skips the point, covering the true branch of p.size() > 0 and the false branch of "within bounds."

These test cases ensure that all branches in the method's decision points (p.size() > 0 and "within bounds") are covered.

Basic Condition Coverage :

Objective: Test each atomic condition within the method independently to cover all possible outcomes of each condition.

Conditions:

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

Test Cases for Basic Condition Coverage:

1. **Test Case 1**: p.size() == 0.

   ○ Expected Outcome: Covers the false outcome of Condition 1.

2. **Test Case 2**: p.size() > 0 with a point that is "within bounds."

   - Expected Outcome: Covers the true outcome of Condition 1 and true outcome of Condition 2.

3. **Test Case 3**: p.size() > 0 with a point that is "out of bounds."

   ○ Expected Outcome: Covers the true outcome of Condition 1 and false outcome of Condition 2.

Each atomic condition has been covered with both true and false values in these test cases.

TASK - 3 :

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:

- If the condition check is deleted, the code always assigns i to min, which could lead to an incorrect outcome. However, this may not cause a detectable failure if no specific test validates the selection of the minimum y value.

- **Potential Undetected Outcome**: If the test set only checks if min is assigned without verifying the correctness of the chosen min value, the deletion might go unnoticed.

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 <= could cause the code to mistakenly assign min = i even if p.get(i).y equals p.get(min).y, potentially selecting an incorrect point as the minimum.

- **Potential Undetected Outcome**: If the test set does not specifically validate cases where p.get(i).y equals p.get(min).y, the mutation could produce a subtle fault without detection.

c. Insertion Mutation:

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

Analysis for Basic Condition Coverage:

- Adding an unnecessary increment (i + 1) changes the intended assignment, leading min to point to an incorrect index, potentially out of the array bounds.

- **Potential Undetected Outcome**: If the test set does not validate that min is correctly assigned to the expected index without additional increments, this mutation might not be detected. Tests only checking if min is assigned (rather than validating correctness) might miss this error.

TASK – 4 :

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 any processing. This covers the condition where the vector size is zero, leading to the exit condition of the method.

Test Case 2: Loop Explored Once

- Input: A vector with one point.

- Test: Vector<Point> p = new Vector<Point>(); p.add(new Point(0, 0));
- Expected Result: The method should not enter the loop since p.size() is 1. It should swap the first point with itself, effectively leaving the vector unchanged. This test case covers the scenario where the loop iterates 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 should enter the loop and compare the two points, finding that the second point has a lower y-coordinate. Thus, minY should be updated to 1, and a swap should occur, moving the second point to the front of the vector.

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 should iterate through all three points. The second point will have the lowest y-coordinate, so minY will be updated to 1. The swap will place the point with coordinates (1, 0) at the front of the vector.

Lab Execution :

Q1. 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. (In your submission document, mention only "Yes" or "No" for each tool).

Ans. YES

Q2. Devise the minimum number of test cases required to cover the code using the aforementioned criteria.

Ans.

Statement Coverage - 3

1. Branch Coverage - 3
2. Basic Condition Coverage -3
3. Path Coverage - 3

Summary of Minimum Test Cases:

Total: 3 (Statement)
\+ 3 (Branch)
\+ 2 (Basic Condition)
\+ 3 (Path)
——————
11 Test Cases

Q3) and Q4) are similar to Part-1

THANK YOU