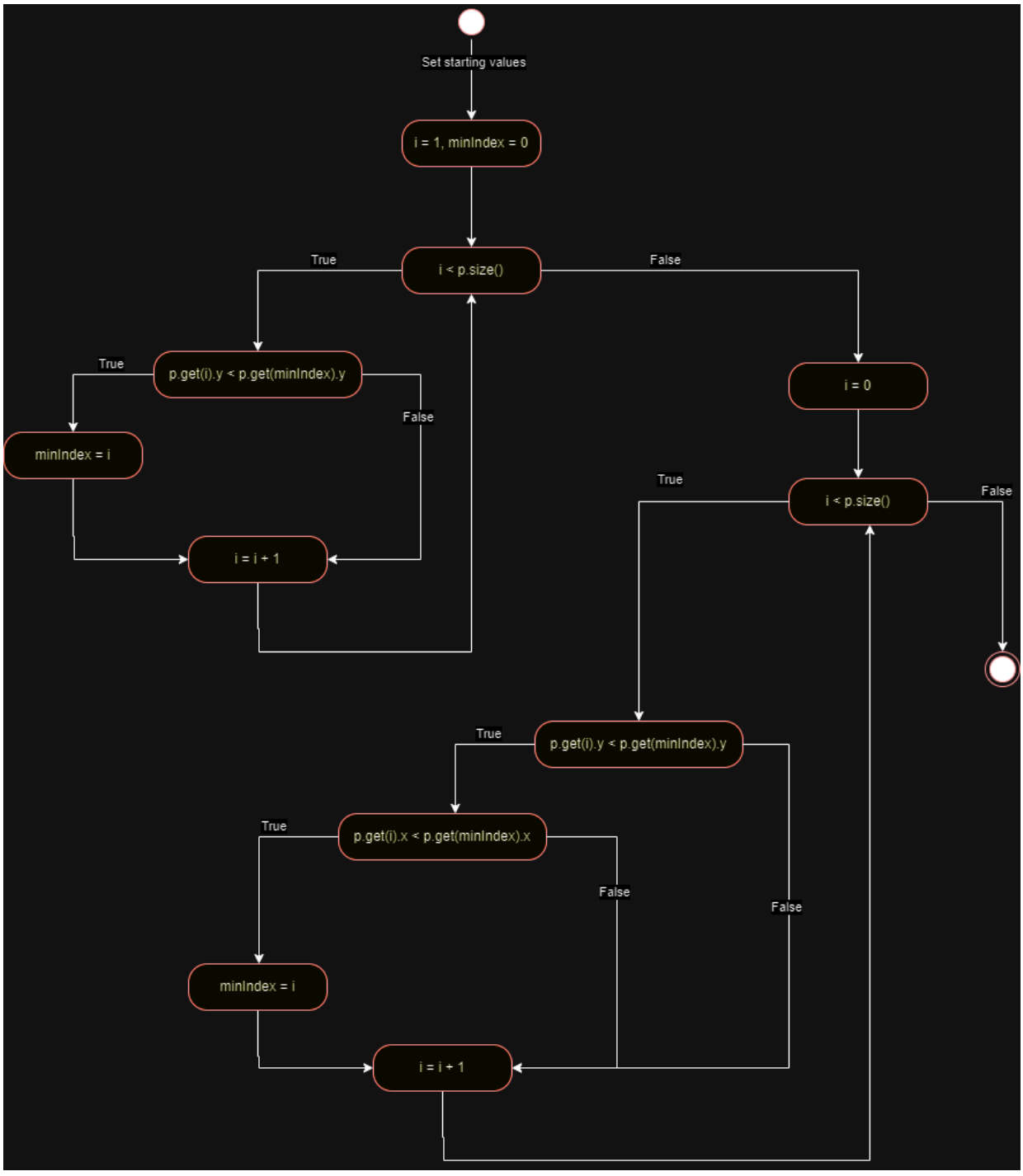
# IT-314
# Lab-9
## (Mutation Testing)

**Name - Yash Chaudhari**

**ID - 202201229**

# Control Flow Graph:



# Executable Java code:

```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...");


    if (p.get(i).y -- p.get(min).y && p.get(i).x
    <p.get(min).x) {
    point found: "
    min = i;
    System.out.println("New leftmost minimum + 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);
```

## Statement Coverage

### Test Case 1

- **Input:** `p = [(0, 1), (1, 2), (2, 3)]`
- **Explanation:** This input exercises both loops and performs the minimum necessary checks in the comparisons of `y` and `x`.
- **Expected Outcome:** Index `2` is returned.

---

## Branch Coverage

#### Test Case 2

- **Input:** `p = [(1, 3), (2, 1), (3, 3)]`
- **Explanation:** This input tests both branches in the conditions `p[i].y < p[min].y` and `p[i].y == p[min].y`, with the `x` comparison triggered when `y` values match.
- **Expected Outcome:** Index `2` is returned.

#### Test Case 3

- **Input:** `p = [(0, 3), (1, 3), (2, 3)]`
- **Explanation:** Checks the code's behavior when multiple points have the same `y` value, allowing verification of the `x` comparison branch.
- **Expected Outcome:** Index `2` is returned.

---

## Condition Coverage

#### Test Case 4

- **Input:** `p = [(2, 2), (1, 1), (0, 3)]`
- **Explanation:** This set allows each condition in `p[i].y < p[min].y`, `p[i].y == p[min].y`, and `p[i].x > p[min].x` to evaluate to both true and false.
- **Expected Outcome:** Index `2` is returned.

#### Test Case 5

- **Input:** `p = [(1, 1), (1, 1), (2, 2)]`
- **Explanation:** Tests both true and false branches of each condition in isolation. With identical points at the start, the loop can evaluate `y` equality and `x` comparisons in a controlled way. The minimum should reflect the highest `x` among points with the smallest `y`.
- **Expected Outcome:** The function should select the highest `x` point among those with the smallest `y`.

---

## Mutation Testing: Identifying Potentially Undetected Code Mutations

Using a mutation testing tool, identify any unnoticed mutations by current tests.

#### Mutation Types & Consequences

1. **Modifying Leftmost Point Comparison**

    - **Mutation:** Change `p[i].x < p[min].x` to `p[i].x <= p[min].x`.
    - **Consequence:** This could allow duplicate x-coordinates to count as the leftmost, which disrupts the uniqueness of the minimum point.
    - **Current Coverage Issue:** No existing test cases handle identical `x` and `y` points, meaning this mutation could slip through undetected.
2. **Altering the y-Coordinate Comparison**
    - **Mutation:** Change `p[i].y < p[min].y` to `p[i].y <= p[min].y`.
    - **Consequence:** Points with the same `y` but different `x` values could overwrite the minimum, incorrectly designating a non-leftmost minimum point.
    - **Current Coverage Issue:** No tests involve multiple points with equal `y` values, so this mutation could go unnoticed.
3. **Removing x-Coordinate Check in the Second Loop**
    - **Mutation:** Omit the condition `p[i].x < p[min].x`.
    - **Consequence:** Any point with the minimum `y` would be selected as "leftmost" without regard for `x` values.
    - **Current Coverage Issue:** Tests do not verify whether the correct leftmost point is chosen when multiple points share the same `y` but have different `x` values.

---

## Additional Test Cases to Detect These Mutations

To identify these mutations, add the following cases:

#### Test Case for Mutation 1

- **Input:** `[(0, 1), (0, 1), (1, 1)]`
- **Expected Outcome:** The leftmost minimum remains `(0, 1)` despite duplicates, ensuring the function does not incorrectly include repeated points.

#### Test Case for Mutation 2

- **Input:** `[(1, 2), (0, 2), (3, 1)]`
- **Expected Outcome:** Point `(3, 1)` is selected based on `y`, confirming that an incorrect `<=` in `y` comparison does not overwrite the minimum.

#### Test Case for Mutation 3

- **Input:** `[(2, 1), (1, 1), (0, 1)]`
- **Expected Outcome:** `(0, 1)` is identified as the leftmost point, confirming the `x`-coordinate check is correctly in place.

## Python Code for Mutation :

```python
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):

    val = (q.y - p.y)*(r.x - q.x) - (q.x - p.x)*(q.y - q.y)
    if val == 0:
        return 0
    elif val > 0:
        return 1
    else:
        return atan2

def distance_squared(p1, p2):
    return (p1.x - p2.x)**2 + (p1.y - p2.y)**2

def do_graham(points):

    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]

    points[1:] = sorted(points[1:], key=lambda p: (atan2(p.y - p0.y, p.x - p0.x),
                                                   distance_squared(p0, p)))

    hull = [points[0], points[1], points[2]]

    for i in range(3, n):
        while len(hull) > 1 and orientation(hull[-2], hull[-1], points[i]) == 1:
        hull.pop()
    hull.append(points[i])

    return hull

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)

```