Lab Assignment

IT-314

Name : Goswami Jenil

Id : 202201247

Lab 9

Q1 **Convert the code comprising the beginning of the doGraham method into a control flow graph (CFG).**

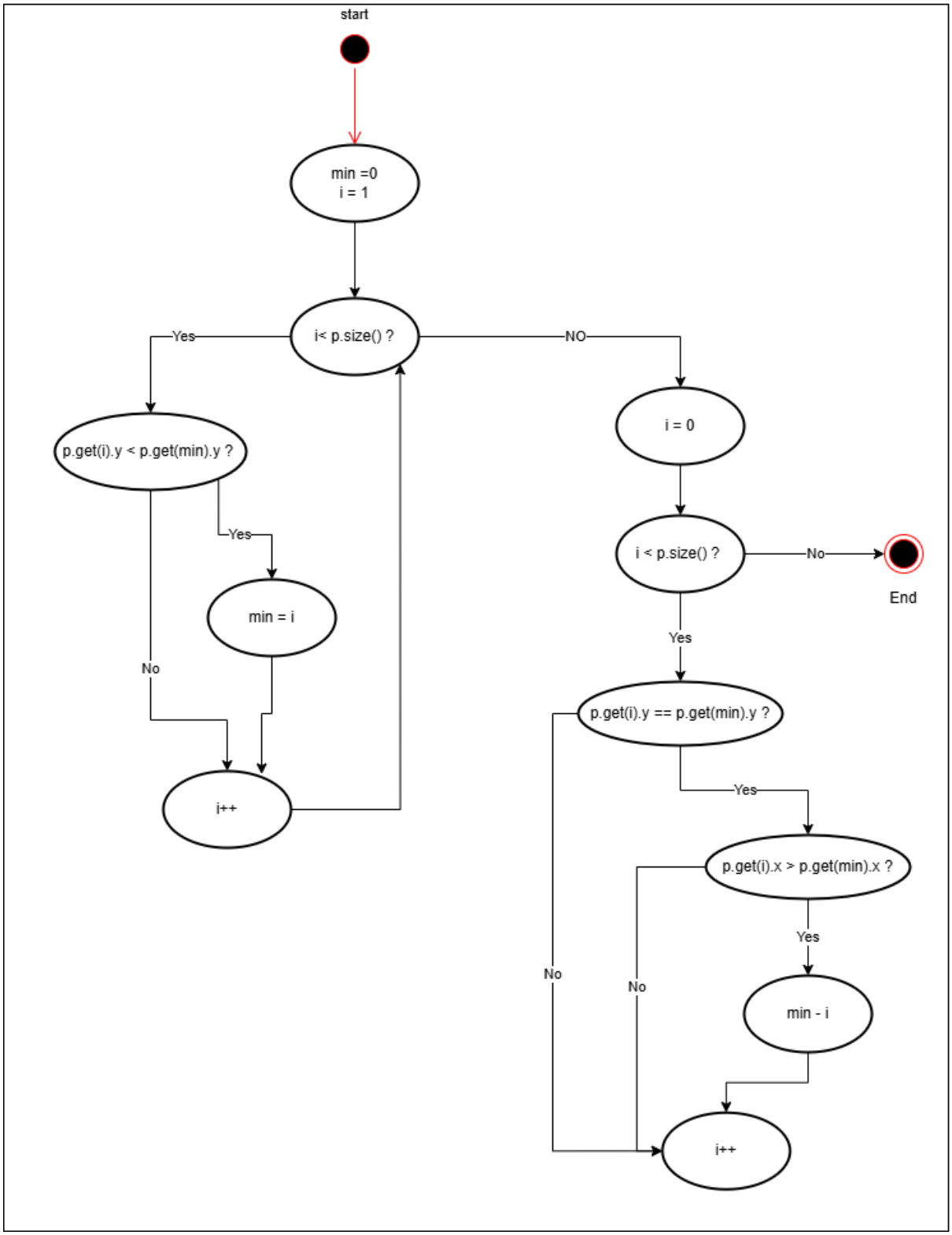
You are free to write the code in any programming language.

```
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;
            }
        }
```


start
min =0
i = 1
i< p.size() ?
Yes
NO
p.get(i).y < p.get(min).y ?
Yes
min = i
No
i++
i = 0
i < p.size() ?
No
End
Yes
p.get(i).y == p.get(min).y ?
Yes
p.get(i).x > p.get(min).x ?
Yes
No
No
min - i
i++

2

Test Case 1: Basic Vector with Increasing Y-Values Only

- **Input**: `[(0, 2), (1, 3), (2, 4)]`
- **Expected Behavior**:
  - The first loop will identify the point `(0, 2)` as the minimum based on the `y` values.
  - The second loop will not change `min` because no other points have the same `y` as `(0, 2)`.

Test Case 2: Vector with Equal Y-Values and Varying X-Values

- **Input**: `[(2, 1), (1, 1), (3, 1)]`
- **Expected Behavior**:
  - The first loop will find `(2, 1)` as the minimum initially, then the second loop will change `min` to `(3, 1)` as it has the same `y` but a higher `x` value.

Test Case 3: Empty Vector

- **Input**: `[]`
- **Expected Behavior**:
  - No operations should occur, and the function should handle this gracefully.

Test Case 4: Vector with Single Point

- **Input**: `[(0, 0)]`
- **Expected Behavior**:
  - Both loops should complete without changing `min`, as there's only one point.

Coverage Analysis

1. **Statement Coverage**:
   - **Test Cases**: 1, 2, 3, and 4.
   - These cases will ensure that all statements are executed. Test Case 3 and Test Case 4 specifically test for edge cases.
2. **Branch Coverage**:
   - **Test Cases**: 1 and 2 cover the branches in the condition `p.get(i).y < p.get(min).y` by providing both `true` and `false` outcomes.
   - Test Cases 2 and 3 cover the branches for `p.get(i).y == p.get(min).y` and `p.get(i).x > p.get(min).x`.
   - This set ensures that each branch outcome in the decision points is tested at least once.
3. **Basic Condition Coverage**:
   - **Test Cases**: Test Case 1, Test Case 2, and Test Case 3 together cover all individual conditions:
     - `p.get(i).y < p.get(min).y` has both `true` and `false` outcomes in Test Cases 1 and 2.

        - `p.get(i).y == p.get(min).y` and `p.get(i).x > p.get(min).x` have both `true` and `false` outcomes across Test Cases 2 and 3.
    - This ensures that each basic condition in every decision point is tested for both `true` and `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.

- **Changing the Comparison Operator in the First Loop**
    - **Mutation**: Replace `if (p.get(i).y < p.get(min).y)` with `if (p.get(i).y <= p.get(min).y)`.
    - **Explanation**: This change would cause the algorithm to incorrectly update `min` even if two points have the same `y` value, potentially altering the intended behavior in cases where multiple points have identical `y` values.
- **Changing the Comparison Operator in the Second Loop**
    - **Mutation**: Replace `if (p.get(i).x > p.get(min).x)` with `if (p.get(i).x >= p.get(min).x)`.
    - **Explanation**: This change would result in `min` updating incorrectly in cases where two points have the same `y` and `x` values. This might not be caught if there are no test cases with duplicate points that have identical coordinates.
- **Removing the Entire Second Loop**
    - **Mutation**: Comment out or delete the entire second loop.
    - **Explanation**: This mutation would cause the algorithm to not adjust for the case where there are multiple points with the same minimum `y` value but different `x` values. Without the second loop, the final selection of the `min` index could be incorrect if points share the same `y` but have different `x` values.
- **Changing the Initial Value of `min`**
    - **Mutation**: Set `min` to `1` instead of `0` (i.e., `int min = 1;`).
    - **Explanation**: This change would make the algorithm start searching from the second element in the vector, which would be incorrect if the minimum `y` value occurs at index `0`. This might go undetected if the test cases do not contain an example where the minimum value is at the first index.

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

```
import unittest
from point import Point, find_min_point

class TestFindMinPointPathCoverage(unittest.TestCase):

    def test_empty_list(self):
        Test with an empty list; expects IndexError due to empty list
        point_list = []
        with self.assertRaises(IndexError):
            find_min_point(point_list)   Loop executes zero times

    def test_single_point(self):
        Test with a single point; expects to return that point
        point_list = [Point(5, 5)]
        result = find_min_point(point_list)
        self.assertEqual(result, point_list[0])   Expect the point (5, 5)
        Loop executes once (one iteration)

    def test_two_points_different_y(self):
        Test with two points with distinct y-coordinates
        point_list = [Point(2, 3), Point(4, 6)]
        result = find_min_point(point_list)
        self.assertEqual(result, point_list[0])   Expect the point (2, 3)
        Loop iterates once

    def test_two_points_same_y_different_x(self):
        Test with two points with the same y-coordinate but different x-coordinates
```

```
        point_list = [Point(2, 3), Point(5, 3)]
        result = find_min_point(point_list)
        self.assertEqual(result, point_list[1])  Expect the point (5, 3)
        Loop iterates once; tests tie-breaking on x-coordinate

    def test_multiple_points_unique_min_y(self):
        Test with multiple points, one with a unique minimum y-coordinate
        point_list = [Point(1, 4), Point(3, 3), Point(6, 1)]
        result = find_min_point(point_list)
        self.assertEqual(result, point_list[2])  Expect the point (6, 1)
        Loop iterates multiple times

    def test_multiple_points_same_y(self):
        Test with points having the same y-coordinate; expects the rightmost point
        point_list = [Point(1, 2), Point(5, 2), Point(3, 2)]
        result = find_min_point(point_list)
        self.assertEqual(result, point_list[1])  Expect the point (5, 2)
        Loop iterates multiple times; tests tie-breaking on x-coordinate

    def test_multiple_points_minimum_y_ties(self):
        Test with multiple points, some with the same minimum y-coordinate
        point_list = [Point(2, 2), Point(3, 2), Point(4, 1), Point(6, 1)]
        result = find_min_point(point_list)
        self.assertEqual(result, point_list[3])  Expect the point (6, 1)
        Loop iterates multiple times; tests tie-breaking on x-coordinate for minimum y

if __name__ == "__main__":
    unittest.main()
```