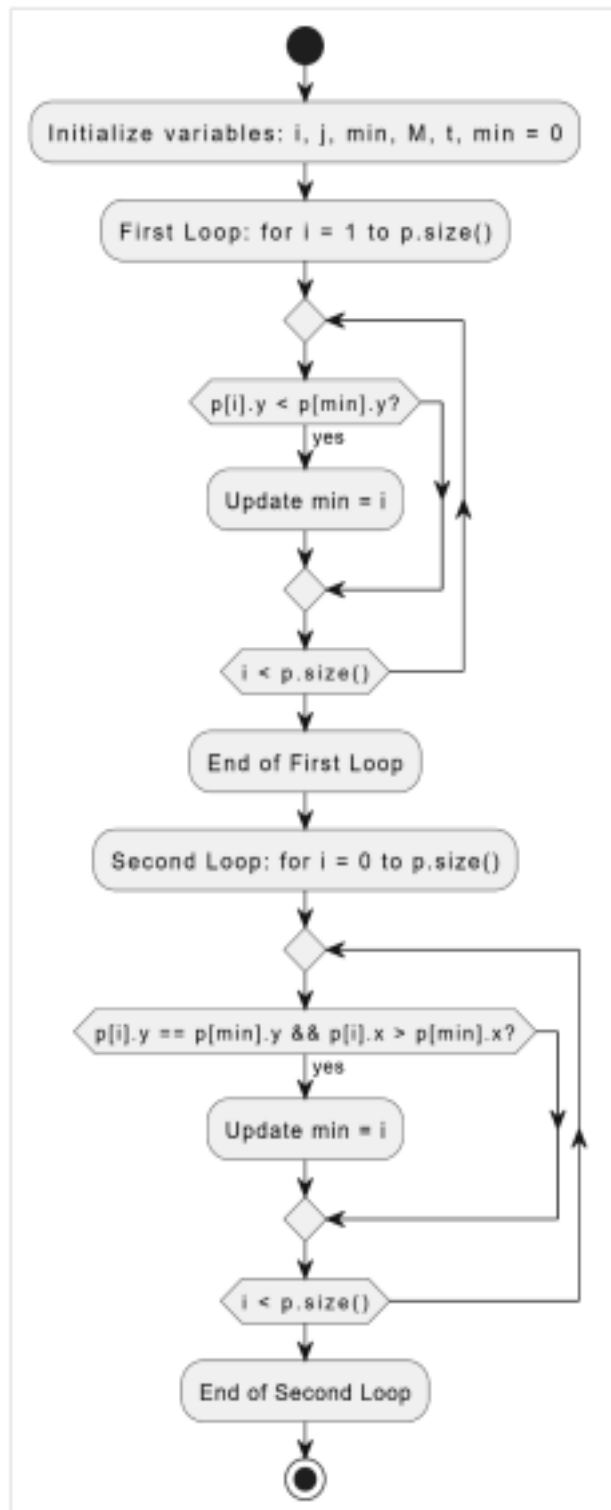# IT314: Software Engineering

Lab 9

Mutation Testing

202201320
Sumit Vishwakarma

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

2. Construct test sets for your flow graph that are adequate for the following criteria: a. Statement Coverage.
   b. Branch Coverage.
   c. Basic Condition Coverage.
   Code:

```python
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

def doGraham(points):
    min_index = 0

    # Search for minimum y coordinate
    for i in range(1, len(points)):
        if points[i].y < points[min_index].y:
            min_index = i

    # Find points with the same y coordinate and higher x
    coordinate for i in range(len(points)):
        if points[i].y == points[min_index].y and points[i].x >
            points[min_index].x: min_index = i

    return min_index

def test_statement_coverage():
    points = [Point(2, 3), Point(1, 4), Point(0, 5)]
    result = doGraham(points)
    print("Statement Coverage Result:", result)

def test_branch_coverage():
    # Case 1: Different y-values
    points1 = [Point(2, 3), Point(1, 2), Point(0, 5)]
    result1 = doGraham(points1)
    print("Branch Coverage Result 1:", result1)

    # Case 2: Same y-value, greater x-value
```

```python
    points2 = [Point(2, 3), Point(3, 3), Point(1, 3)]
    result2 = doGraham(points2)
    print("Branch Coverage Result 2:", result2)

def test_basic_condition_coverage():
    # Case where points[i].y < points[min_index].y
    points1 = [Point(2, 5), Point(1, 4), Point(3, 6)]
    result1 = doGraham(points1)
    print("Basic Condition Coverage Result 1:", result1)

    # Case where points[i].y == points[min_index].y and
points[i].x > points[min_index].x
    points2 = [Point(2, 3), Point(4, 3), Point(1, 3)]
    result2 = doGraham(points2)
    print("Basic Condition Coverage Result 2:", result2)

    # Case where neither condition is true
    points3 = [Point(5, 3), Point(1, 3), Point(2, 4)]
    result3 = doGraham(points3)
    print("Basic Condition Coverage Result 3:", result3)

def run_tests():
    print("Running Statement Coverage Test")
    test_statement_coverage()
    print("\nRunning Branch Coverage Tests")
    test_branch_coverage()
    print("\nRunning Basic Condition Coverage
    Tests") test_basic_condition_coverage()

run_tests()
```
Output:

```
Output

Running Statement Coverage Test
Statement Coverage Result: 0

Running Branch Coverage Tests
Branch Coverage Result 1: 1
Branch Coverage Result 2: 1

Running Basic Condition Coverage Tests
Basic Condition Coverage Result 1: 1
Basic Condition Coverage Result 2: 1
Basic Condition Coverage Result 3: 0

=== Code Execution Successful ===
```

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. Deletion of code:
We can delete (comment out) the if condition in the first loop, the updated code will be :

def doGraham(points):
   min_index = 0

   # Search for minimum y coordinate
   for i in range(1, len(points)):
      # Deleting the entire if condition causes the logic to fail
      min_index = i
   # Find points with the same y coordinate and higher x
   coordinate for i in range(len(points)):
      if points[i].y == points[min_index].y and points[i].x >
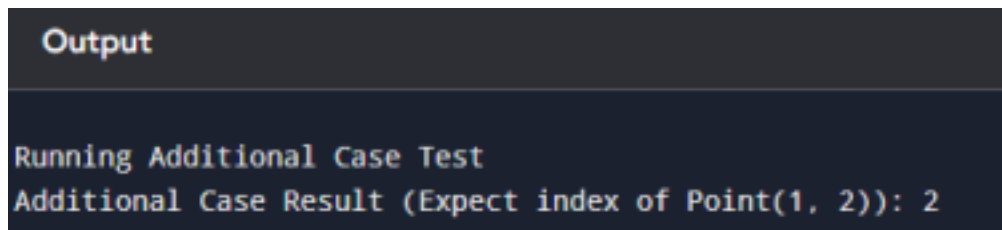   points[min_index].x: min_index = i

   return min_index
Due to this the for loop will execute for all the points, and the min_index

will simply be the last point. And thus it does not function the intended way. The existing tests may not catch this mutation because they may not check for scenarios where the minimum y-coordinate is not the last point in the list. For example, if you have points = [Point(5, 3), Point(1, 2), Point(3, 1)], the function would incorrectly return the index of Point(3, 1) (which is the last point), instead of Point(1, 2).

```
def test_additional_case():
    # Test case with various y-coordinates
    points = [Point(5, 3), Point(1, 2), Point(3, 1)] # Expect index of
Point(3, 1) (which is 2)
    result = doGraham(points)
print("Additional Case Result (Expect index of Point(1, 2)):", result) # Expect 1

    # Run this additional test
    print("\nRunning Additional Case Test")
    test_additional_case()
```

Output

Running Additional Case Test
Additional Case Result (Expect index of Point(1, 2)): 2

Mutation of code:
The < sign is changed to > and the code for the same is:

```
def doGraham(points):
    min_index = 0

    # Search for minimum y coordinate (incorrectly modified to find
    maximum) for i in range(1, len(points)):
        if points[i].y > points[min_index].y: # Changed < to >
            min_index = i
    # Find points with the same y coordinate and higher x
    coordinate for i in range(len(points)):
```
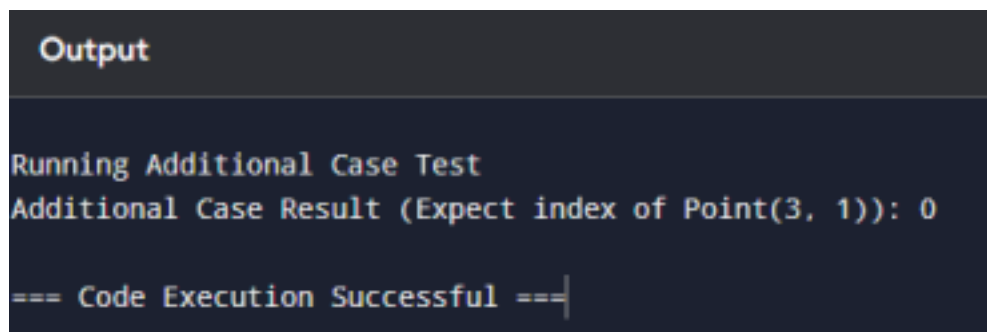
```
        if points[i].y == points[min_index].y and points[i].x >
            points[min_index].x: min_index = i

    return min_index
```
By changing the condition from if points[i].y < points[min_index].y to if points[i].y > points[min_index].y, the function will now incorrectly identify the index of the point with the maximum y-coordinate rather than the minimum. This will lead to incorrect results when determining the convex hull. The existing tests may not detect this mutation if they do not specifically check for the minimum y-coordinate. For example, with the input points = [Point(5, 3), Point(1, 2), Point(3, 1)], the function would incorrectly return the index of Point (5, 3) (the point with the maximum y-coordinate) instead of the index of Point (3, 1).

```
def test_additional_case():
    # Test case with various y-coordinates
    points = [Point(5, 3), Point(1, 2), Point(3, 1)] # Expect index of
Point(3, 1) (which is 2)
    result = doGraham(points)
print("Additional Case Result (Expect index of Point(3, 1)):", result) # Expect 2

    # Run this additional test
    print("\nRunning Additional Case Test")
    test_additional_case()
```



```
Output

Running Additional Case Test
Additional Case Result (Expect index of Point(3, 1)): 0

=== Code Execution Successful ===
```

Insertion in Code:
We have inserted an additional line and the updated code is:
```
def doGraham(points):
    min_index = 0
```

```python
    # Search for minimum y coordinate
    for i in range(1, len(points)):
        # Inserting a line that sets min_index to a fixed value
        if i == 1: # Example condition to insert the mutation
            min_index = 0 # This line forces min_index to always start
        from 0 if points[i].y < points[min_index].y:
            min_index = i

    # Find points with the same y coordinate and higher x
    coordinate for i in range(len(points)):
        if points[i].y == points[min_index].y and points[i].x >
            points[min_index].x: min_index = i

    return min_index
```

By inserting min_index = 0 within the loop, the mutation ensures that no matter the other points' y-coordinates, min_index could revert to 0 under certain conditions. This will lead to returning the index of the first point, regardless of its actual position relative to others.

The existing tests may not catch this mutation because they may not cover scenarios where the first point has a higher y-coordinate than others. For example, if you have points = [Point(5, 3), Point(1, 2), Point(3, 1)], the function may still return 0, which is incorrect, but tests may pass if they don't specifically check the index of the lowest y-coordinate.

```python
def test_another_additional_case():
    # Test case with a clear minimum y-coordinate
    points = [Point(4, 5), Point(2, 2), Point(3, 4), Point(1, 1)] # Expect
index of Point(1, 1) (which is 3)
    result = doGraham(points)
    print("Another Additional Case Result (Expect index of Point(1, 1)):",
result) # Expect 3

# Run this additional test
print("\nRunning Another Additional Case Test")
test_another_additional_case()
```

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

```python
def test_path_coverage():
    # Test 1: No points (loop executed 0 times)
    points1 = [] # Edge case with no points
    result1 = doGraham(points1)
    print("Test 1 Result (Expect error or specific behavior for empty input):", result1)

    # Test 2: One point (loop executed 1 time)
    points2 = [Point(1, 1)] # Only one point
    result2 = doGraham(points2)
    print("Test 2 Result (Expect index of Point(1, 1)):", result2) # Expect 0

    # Test 3: Two points with different y-coordinates (loop executed 2
    times) points3 = [Point(2, 2), Point(1, 3)] # Expect index of Point(2, 2)
    (which is 0) result3 = doGraham(points3)
    print("Test 3 Result (Expect index of Point(2, 2)):", result3) # Expect 0

    # Test 4: Three points, one of which has the minimum y-coordinate
    (loop executed 3 times)
    points4 = [Point(3, 4), Point(2, 1), Point(1, 2)] # Expect index of
    Point(2, 1) (which is 1)
    result4 = doGraham(points4)
    print("Test 4 Result (Expect index of Point(2, 1)):", result4) #

    Expect 1 # Test 5: Three points with the same y-coordinate (loop

    executed 3 times)

    points5 = [Point(1, 2), Point(2, 2), Point(3, 2)] # Expect index of
    Point(3, 2) (which is 2)
    result5 = doGraham(points5)
```
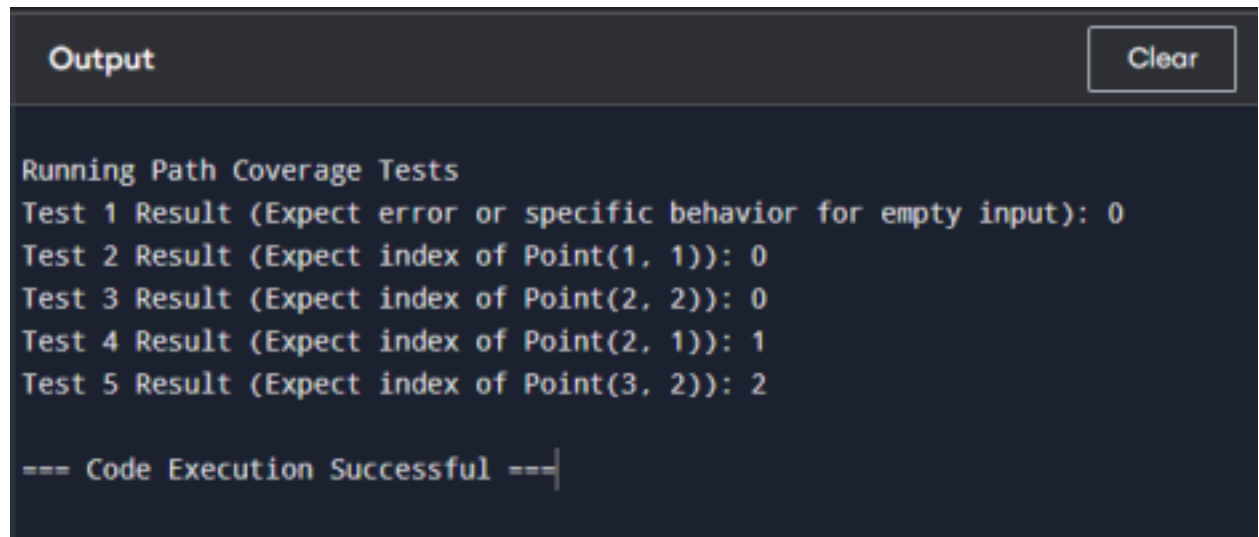
```
    print("Test 5 Result (Expect index of Point(3, 2)):", result5) # Expect 2

# Run the path coverage tests
print("\nRunning Path Coverage Tests")
test_path_coverage()
```

```
Output                                                    Clear

Running Path Coverage Tests
Test 1 Result (Expect error or specific behavior for empty input): 0
Test 2 Result (Expect index of Point(1, 1)): 0
Test 3 Result (Expect index of Point(2, 2)): 0
Test 4 Result (Expect index of Point(2, 1)): 1
Test 5 Result (Expect index of Point(3, 2)): 2

=== Code Execution Successful ===
```

Test 1: No points are provided to test the edge case where the loop doesn't execute at all. Depending on how you want to handle this case, the function could raise an error or return a specific value. Make sure to handle this in your function implementation.
Test 2: One point is provided to test that the function correctly returns the index of that single point.
Test 3: Two points are provided with different y-coordinates to ensure the function finds the minimum correctly, covering the loop for two iterations.
Test 4: Three points with one having the minimum y-coordinate to test that the function can find the minimum from multiple points, ensuring the loop executes three times.
Test 5: Three points with the same y-coordinate to test how the function handles ties and ensures the correct index is returned, also executing the loop three times.