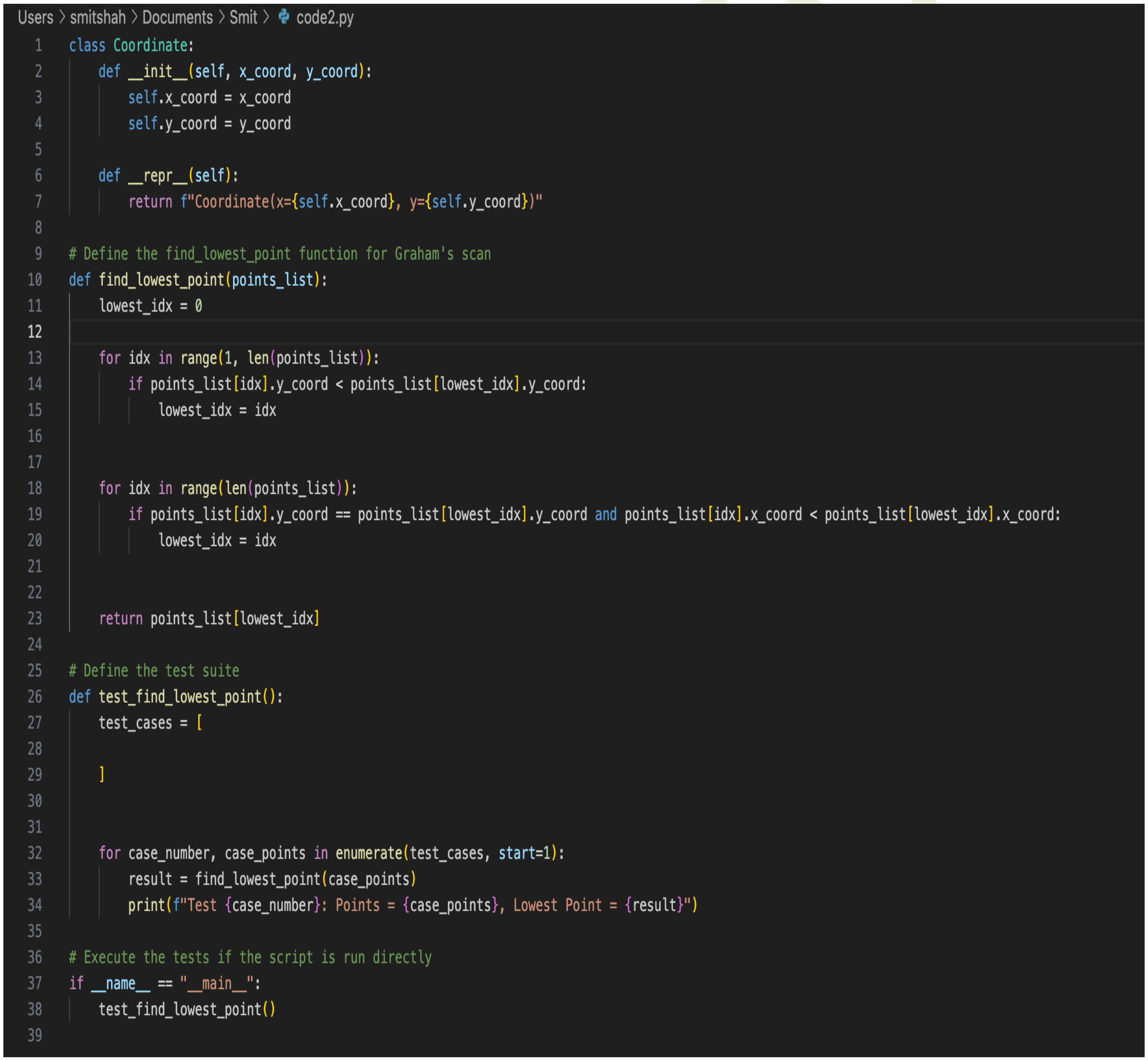
IT314 – SOFTWARE ENGINEERING

LAB - 9

202201192 – SMIT SHAH

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.

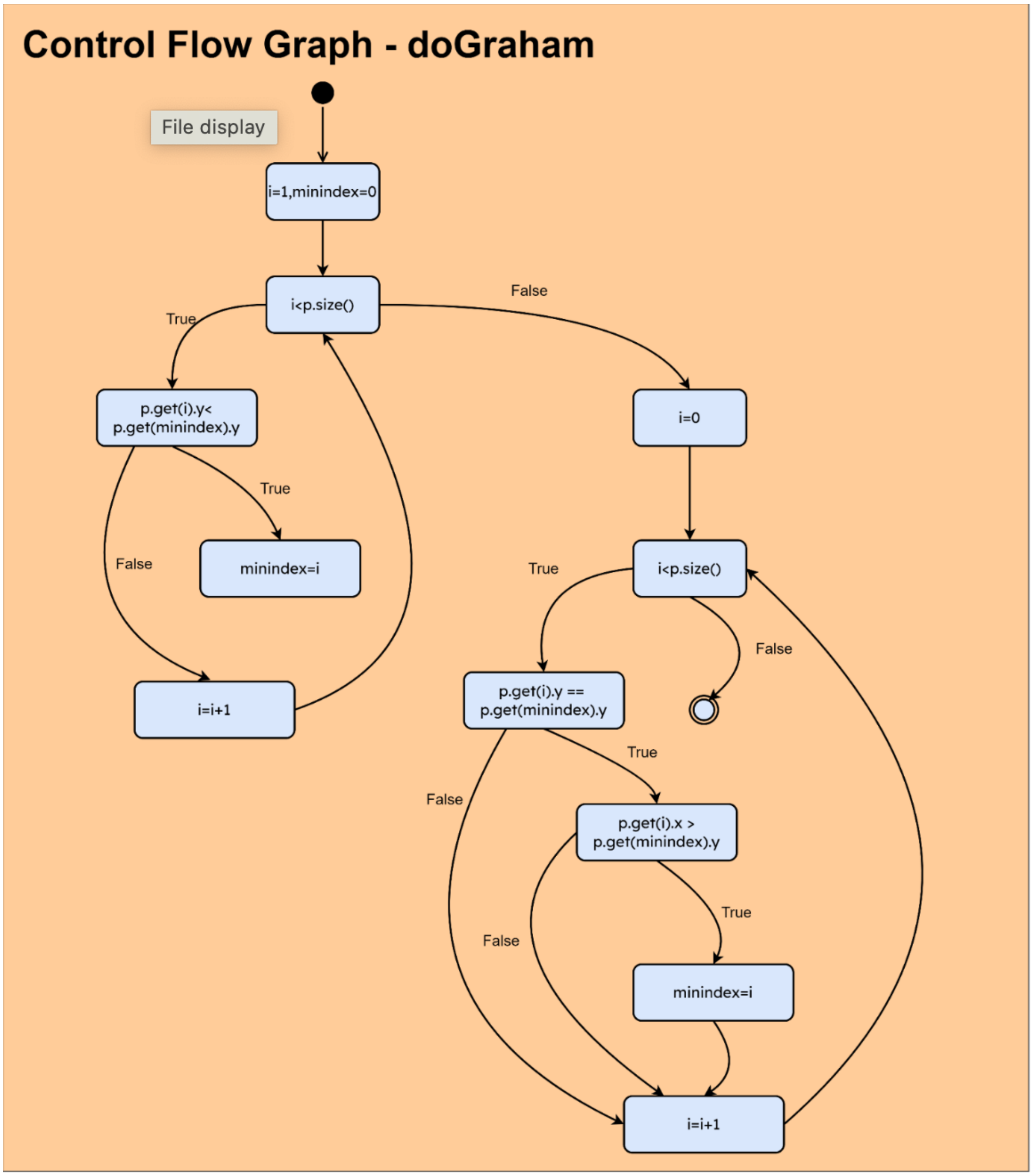
Sol.



Users > smitshah > Documents > Smit > code2.py

```python
class Coordinate:
    def __init__(self, x_coord, y_coord):
        self.x_coord = x_coord
        self.y_coord = y_coord

    def __repr__(self):
        return f"Coordinate(x={self.x_coord}, y={self.y_coord})"

# Define the find_lowest_point function for Graham's scan
def find_lowest_point(points_list):
    lowest_idx = 0

    for idx in range(1, len(points_list)):
        if points_list[idx].y_coord < points_list[lowest_idx].y_coord:
            lowest_idx = idx


    for idx in range(len(points_list)):
        if points_list[idx].y_coord == points_list[lowest_idx].y_coord and points_list[idx].x_coord < points_list[lowest_idx].x_coord:
            lowest_idx = idx


    return points_list[lowest_idx]

# Define the test suite
def test_find_lowest_point():
    test_cases = [

    ]


    for case_number, case_points in enumerate(test_cases, start=1):
        result = find_lowest_point(case_points)
        print(f"Test {case_number}: Points = {case_points}, Lowest Point = {result}")

# Execute the tests if the script is run directly
if __name__ == "__main__":
    test_find_lowest_point()

```

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.**



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.

Sol.
Test Cases for Statement Coverage

Test Case 1:
- Input points are (2,3) , (1,2) and (3,1)
- Purpose of this test case is to ensure that all statements in the flow graph are executed at least once.

Test Cases for Branch Coverage
Test Case 1:
- Input points are (2,3) , (1,2) and (3,1)
- Purpose of this test case is to ensure that both branches (True/False) of the conditional statements are tested at least once.

Test Case 2:
- Input points are (3,3) , (4,3) and (5,3)
- Purpose of this test case is to ensure that both branches (False/False) of the conditional statements are tested.

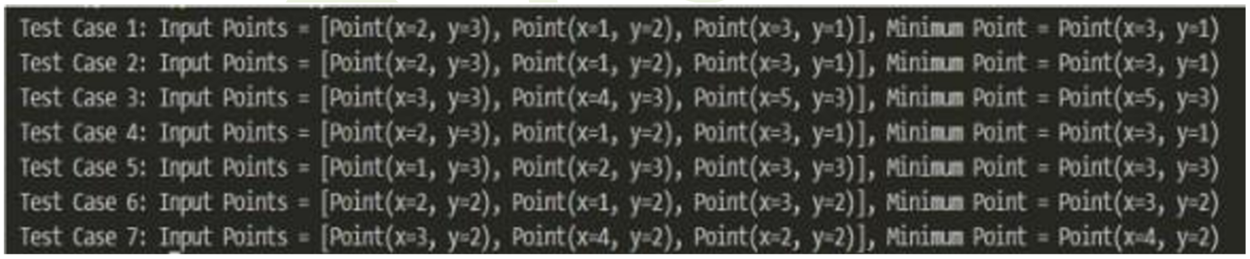
Test Cases for Basic Condition Coverage
Test Case 1:
- Input points are (2,3) , (1,2) and (3,1)
- Purpose of this test case is to ensure that the condition p[i].y < p[min].y is evaluated as True.

Test Case 2:
- Input points are (1,3) , (2,3) and (3,3) .
- Purpose of this test case is to ensure that the condition p[i].y < p[min].y is evaluated as False.

Test Case 3:
- Input points are (2,2) , (1,2) and (3,2) .
- Purpose of this test case is to ensure that both conditions p[i].y == p[min].y and p[i].y < p[min].y are evaluated as True.

Test Case 4:
- Input points are (3,2) , (4,2) and (2,2)
- Purpose of this test case is to ensure that the condition p[i].y == p[min].y is True
  and the condition
  p[i].y < p[min].y is False.

Output :



```
Test Case 1: Input Points = [Point(x=2, y=3), Point(x=1, y=2), Point(x=3, y=1)], Minimum Point = Point(x=3, y=1)
Test Case 2: Input Points = [Point(x=2, y=3), Point(x=1, y=2), Point(x=3, y=1)], Minimum Point = Point(x=3, y=1)
Test Case 3: Input Points = [Point(x=3, y=3), Point(x=4, y=3), Point(x=5, y=3)], Minimum Point = Point(x=5, y=3)
Test Case 4: Input Points = [Point(x=2, y=3), Point(x=1, y=2), Point(x=3, y=1)], Minimum Point = Point(x=3, y=1)
Test Case 5: Input Points = [Point(x=1, y=3), Point(x=2, y=3), Point(x=3, y=3)], Minimum Point = Point(x=3, y=3)
Test Case 6: Input Points = [Point(x=2, y=2), Point(x=1, y=2), Point(x=3, y=2)], Minimum Point = Point(x=3, y=2)
Test Case 7: Input Points = [Point(x=3, y=2), Point(x=4, y=2), Point(x=2, y=2)], Minimum Point = Point(x=4, y=2)
```

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

Sol.
a. Deletion Mutation:

Original Code:
```
if ((p.get(i)).y < (p.get(min)).y) {
    min = i;
}
```

Mutated Code (Deletion Mutation):
```
min = i;
```

Analysis for Statement Coverage

→ The deletion mutation removes the condition check, causing min = i to execute unconditionally. While statement coverage is still satisfied (since min = i is executed), the mutation may go undetected if the test cases do not verify the correctness of the minimum value selection, potentially leading to incorrect results without triggering a failure.

b. Change Mutation:

Original Code:
```
if ((p(i)).y< (p.get(min)).y) {
    min = i;
}
```

Mutated Code (Deletion Mutation):
```
min = i;
```

Statement Coverage Analysis:
→ In the original condition, the comparison checks if the y value of p.get(i)).y is strictly less than the y value of p.get(min)).y. After mutating,

the condition will now check if the y value of p.get(i)).y is less than or equal

to the y value of p.get(min)).y.

This change could impact how the algorithm behaves, particularly if y values are equal for multiple points. The updated condition could now include cases where y values are equal, potentially resulting in a different behavior (such as including equal y values in the logic).

c. Insertion Mutation:

Original Code :
min = i;

Mutated Code:
min = i + 1;

Analysis for Basic Condition Coverage:

Impact of the Mutation:

- **Altered Functionality:** The mutation modifies the assignment of min, changing it to point to the index immediately following i (i.e., min = i + 1), instead of keeping it at i. This results in tracking an incorrect index, leading to the potential for errors in selecting the correct minimum element.

- **Risk of Index Out-of-Bounds:** If the index i refers to the last element of the list, the mutation causes an out-of-bounds reference by attempting to access i + 1, which is beyond the valid range of indices. This could either cause a runtime error or unexpected behavior within the algorithm.

- **Inaccurate Algorithm Behavior:** In algorithms that rely on correctly tracking the smallest (or largest) element, such as sorting or searching, this mutation will disrupt the proper functioning of the algorithm. By improperly updating min to i + 1, the algorithm may fail to find the correct element or misidentify the intended value.

Potential for Undetected Errors:

- **Overlooking the Problem:** If the tests only check whether min is updated (i.e., if the assignment happens) without verifying the correctness of the index itself, this error could go unnoticed. Specifically, tests that do not validate whether min points to the correct element after each update may miss this issue, potentially leading to incorrect behavior without triggering a failure in the test suite.

3. 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 list of points (p).

Test:

```
Vector p = new Vector();
```

● **Expected Result:** Since p has zero elements (len(p) == 0), the loop will not execute. The method should return immediately without any processing. This test case covers the scenario where the loop is not executed at all due to the absence of points.

Test Case 2: Loop Explored Once

● **Input:** A vector with a single point.

Test:

```
Vector p = new Vector();
p.add(new Point(0, 0));
```

● **Expected Result:** The list contains only one point (len(p) == 1), so the loop will be skipped. The method should perform no changes, and the point remains in its original position. This test case ensures that the method handles the case where there is only one element, so no swapping is needed.

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 p = new Vector();
p.add(new Point(1, 1));
p.add(new Point(0, 0));
```

● **Expected Result:** The loop will iterate once, comparing the first and second points: The first point ((1,1)) has a higher y-coordinate than the second point ((0,0)), so the loop will perform a swap. The vector will now be [(0, 0), (1, 1)]. This test case checks the loop's behavior when it iterates once, performing a swap.

Test Case 4: Loop Explored More Than Twice (Loop Iterates Over Multiple Points)

● **Input:** A vector with multiple points.

Test:

```
Vector p = new Vector();
p.add(new Point(2, 2));
p.add(new Point(1, 0));
p.add(new Point(0, 3));
```

Expected Result:

The loop will iterate through all three points:

1. **First iteration:** The first point `(2, 2)` is compared with `(1, 0)`.
   - Since `(1, 0)` has a lower y-coordinate, `miny` will be updated, and a swap will occur, moving `(1, 0)` to the front.
2. **Second iteration:** The newly swapped first point `(1, 0)` will be compared with `(0, 3)`.
   - Since `(1, 0)` has a lower y-coordinate than `(0, 3)`, no further swap will be needed, as `miny` remains at index 1.

The final vector will be `[(1, 0), (2, 2), (0, 3)]`. This case ensures the loop iterates through multiple points and performs at least one swap.

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.

Ans. Control Flow Graph Factory :- YES

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

Ans.
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

Q3) and Q4) Same as Part I