IT-314 LAB-9

Software Engineering

Prof. Saurabh Tiwari

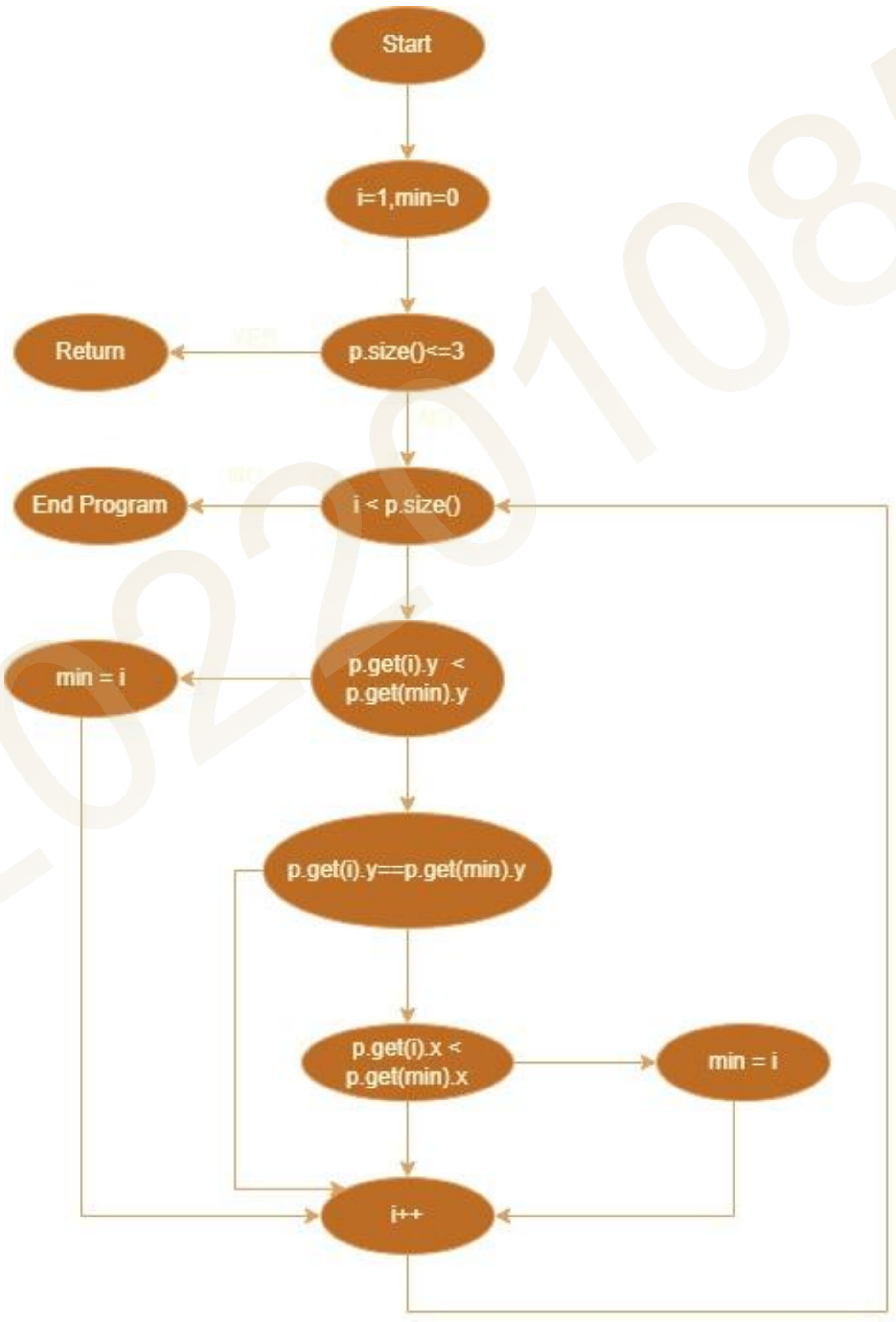
Student ID : 202201085

Name : Dholariya Parth Narendrabhai

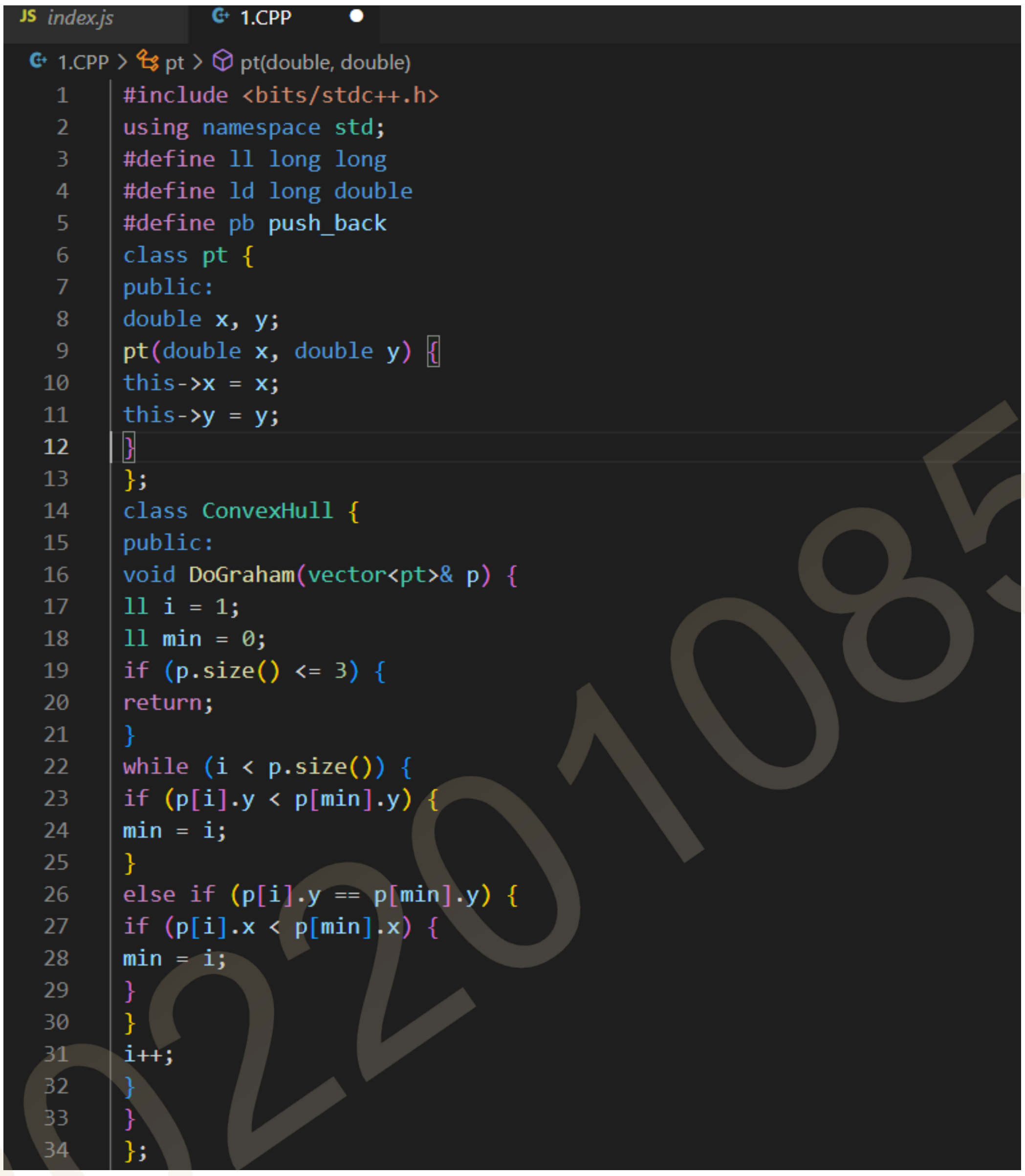
Question-1

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.

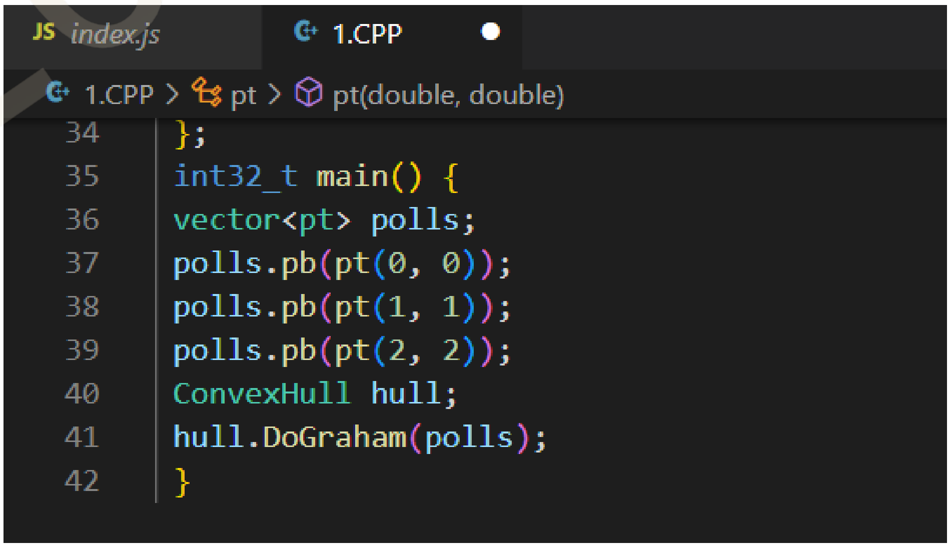
Control Flow Graph



- **C++ CODE**



```cpp
#include <bits/stdc++.h>
using namespace std;
#define ll long long
#define ld long double
#define pb push_back
class pt {
public:
double x, y;
pt(double x, double y) {
this->x = x;
this->y = y;
}
};
class ConvexHull {
public:
void DoGraham(vector<pt>& p) {
ll i = 1;
ll 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++;
}
}
};
```



```cpp
};
int32_t main() {
vector<pt> polls;
polls.pb(pt(0, 0));
polls.pb(pt(1, 1));
polls.pb(pt(2, 2));
ConvexHull hull;
hull.DoGraham(polls);
}
```

Task-2

Construct test sets for your flow graph that are adequate for the following criteria

a) **Statement Coverage:**

Objective: Ensure that every line in the "**DoGraham**" method is executed at least once.

✓ **Test Case 1:** p.size() <= 3

- Input: A vector 'p' containing three or fewer points, such as (3, 3), (4, 4), (5, 5).
- Expected Outcome: The method exits immediately, covering the initial return statement.

✓ **Test Case 2:** p.size() > 3, with points having unique y and x values.

- Input: A vector 'p' with distinct points like (3, 1), (6, 3), (8, 4), (9, 5).
- Expected Outcome: The loop iterates through each point to identify the one with the smallest y-coordinate, covering the loop body and if-else conditions.

✓ **Test Case 3:** p.size() > 3, where points have identical y values but different x values.

- Input: Points such as (3, 2), (5, 2), (1, 2), (7, 3).
- Expected Outcome: The loop selects the point with the smallest x-coordinate among those with the same y value, covering both y and x comparisons.

✓ **Test Case 4:** p.size() > 3, with all points having the same y and x values.

- Input: Points like (2, 2), (2, 2), (2, 2), (2, 2).
- Expected Outcome: The loop completes without changing 'min' since all points are identical, ensuring the loop finishes without updating 'min'.

b) **Branch Coverage**: **Objective**: Ensure that each branch outcome (true/false) for all decision points is tested.

✓ **Test Case 1**: p.size() <= 3

- **Input**: A vector p with 3 or fewer points, such as (4, 4), (5, 5), (6, 6).
- **Expected Outcome**: The method returns immediately, covering the false branch for the loop entry.

✓ **Test Case 2**: p.size() > 3, with all y values distinct.

- **Input**: Points like (2, 0), (4, 1), (6, 2), (8, 3).
- **Expected Outcome**: True branch of the main if condition (p[i].y < p[min].y) is covered since each y value is unique.

- ✓ **Test Case 3**: p.size() > 3, with some points having identical y values but different x values.
  - **Input**: Points such as (3, 2), (5, 2), (1, 2), (7, 3).
  - **Expected Outcome**: Covers true for the if statement (when y is lower) and true/false for the else-if condition (same y, different x).
- ✓ **Test Case 4**: p.size() > 3, with multiple points having the same y and x values.
  - **Input**: Points like (2, 2), (2, 2), (2, 2), (2, 2).
  - **Expected Outcome**: The loop iterates, covering the false branch for all comparisons since no updates to min occur.

c) **Basic Condition Coverage**:

Objective: Independently test each atomic condition within the method to cover all possible outcomes.

Conditions:

1. p.size() <= 3
2. p[i].y < p[min].y
3. p[i].y == p[min].y
4. p[i].x < p[min].x

- ✓ **Test Case 1**: p.size() <= 3
  - **Input**: A vector p containing 3 or fewer points, such as (5, 5), (6, 6), (7, 7).
  - **Expected Outcome**: Evaluates the true condition for p.size() <= 3.
- ✓ **Test Case 2**: p.size() > 3, with points having distinct y values.
  - **Input**: Points like (2, 1), (3, 2), (4, 3), (5, 4).
  - **Expected Outcome**: True outcome for p[i].y < p[min].y, as each point has a progressively higher y value than the initial minimum.
- ✓ **Test Case 3**: p.size() > 3, with points sharing the same y value but differing in x values.
  - **Input**: Points such as (2, 3), (4, 3), (5, 3), (3, 2).
  - **Expected Outcome**: True outcome for p[i].y == p[min].y and both true and false outcomes for p[i].x < p[min].x.
- ✓ **Test Case 4**: p.size() > 3, with identical y and x values.
  - **Input**: Points like (3, 3), (3, 3), (3, 3), (3, 3).
  - **Expected Outcome**: False outcomes for p[i].y < p[min].y, p[i].y == p[min].y, and p[i].x < p[min].x as no updates to min occur.

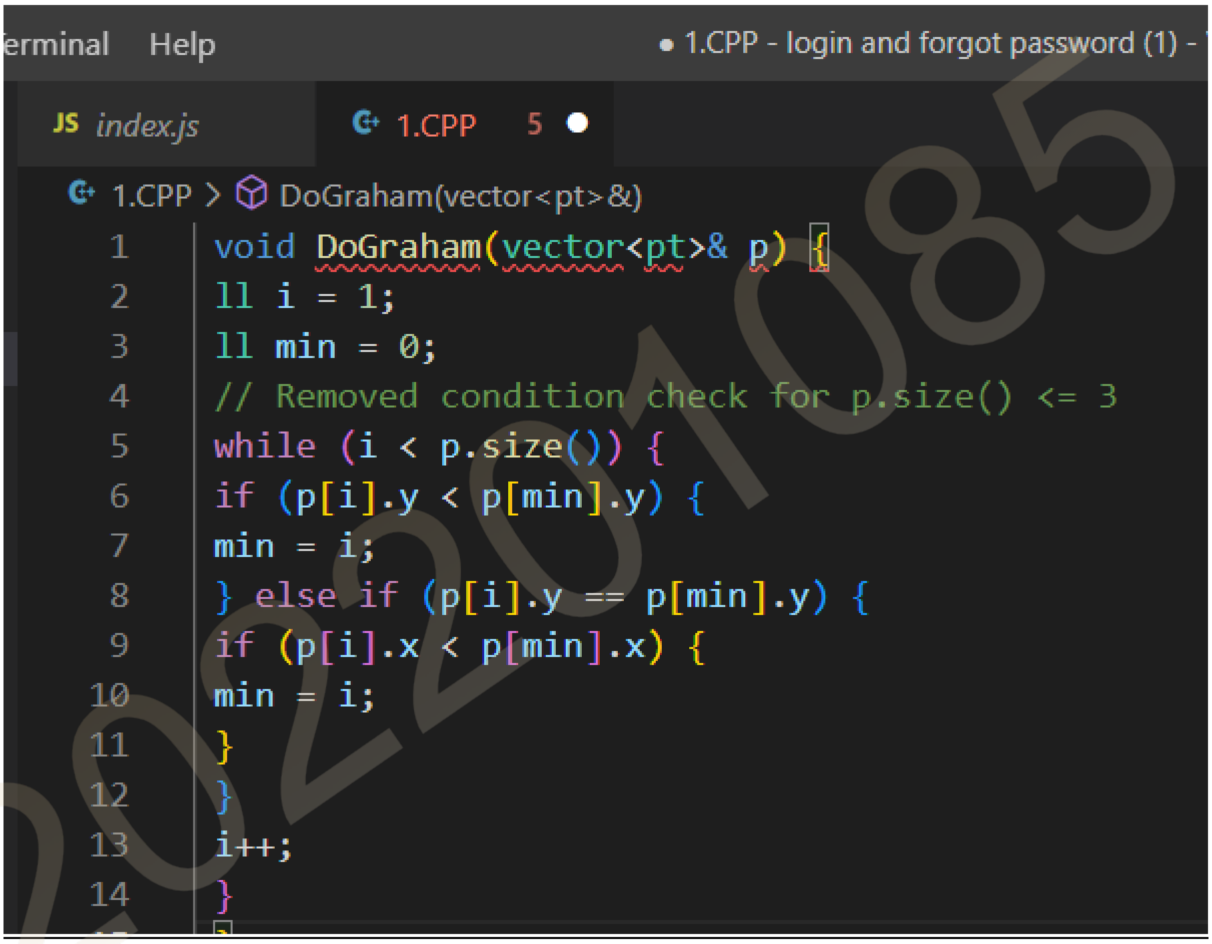
Task- 3

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.

1) Deletion Mutation: Remove specific conditions or lines in the code.

Mutation Example: Remove the condition if (p.size() <= 3) at the start of the method.

Code


erminal
Help
● 1.CPP - login and forgot password (1) -
index.js
1.CPP 5
1.CPP > DoGraham(vector<pt>&)

```
void DoGraham(vector<pt>& p) {
ll i = 1;
ll min = 0;
// Removed condition check for p.size() <= 3
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++;
}
```

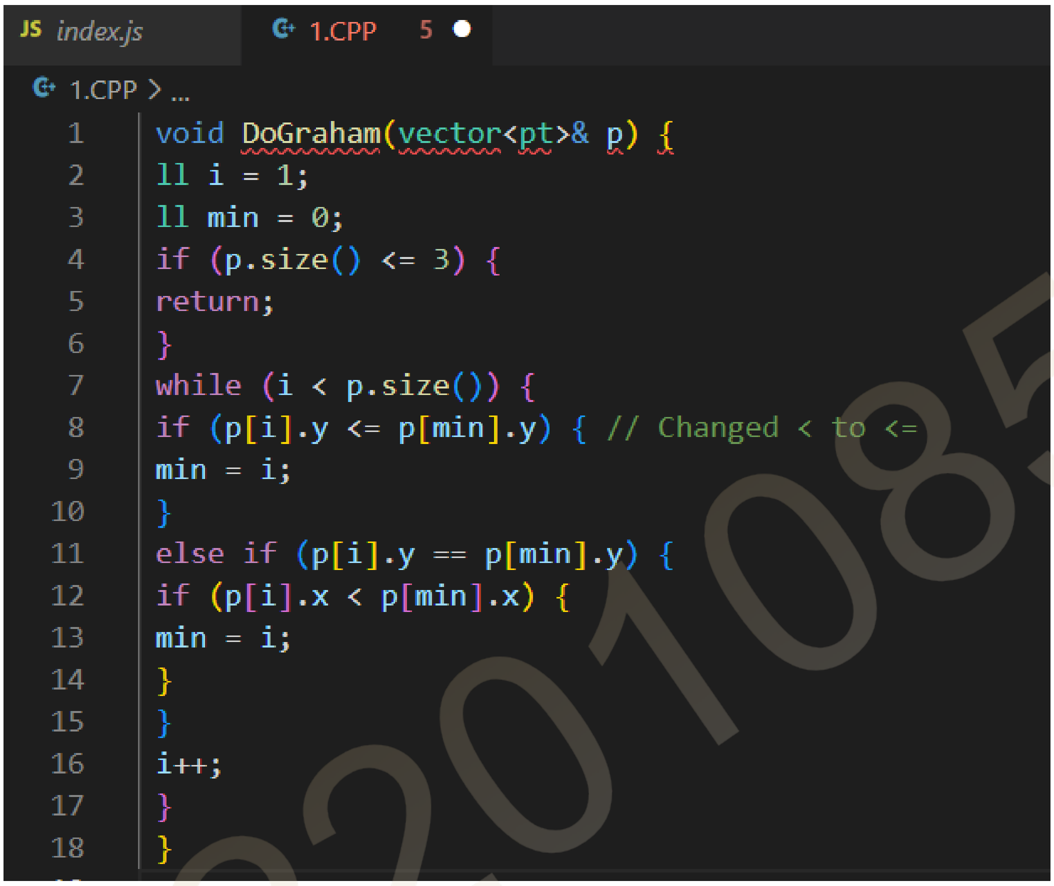
Expected Outcome: Without this condition, the method would attempt to execute even when p.size() <= 3, potentially leading to unnecessary or incorrect calculations when the input size is insufficient. Our existing tests, which only verify cases where p.size() > 3, might overlook this, allowing the mutation to go undetected.

Test Case Requirement: A test case where p.size() is exactly 3 (for example, points (0, 0), (1, 1), and (2, 2)) would help identify this issue, as it would prompt the method to perform unnecessary calculations.

2) **Change Mutation:** Alter a condition, variable, or operator within the code.

Mutation Example: Change the <operator to <= in the condition if(p[i].y< p[min].y).

Code



```
void DoGraham(vector<pt>& p) {
ll i = 1;
ll min = 0;
if (p.size() <= 3) {
return;
}
while (i < p.size()) {
if (p[i].y <= p[min].y) { // Changed < to <=
min = i;
}
else if (p[i].y == p[min].y) {
if (p[i].x < p[min].x) {
min = i;
}
}
i++;
}
}
```

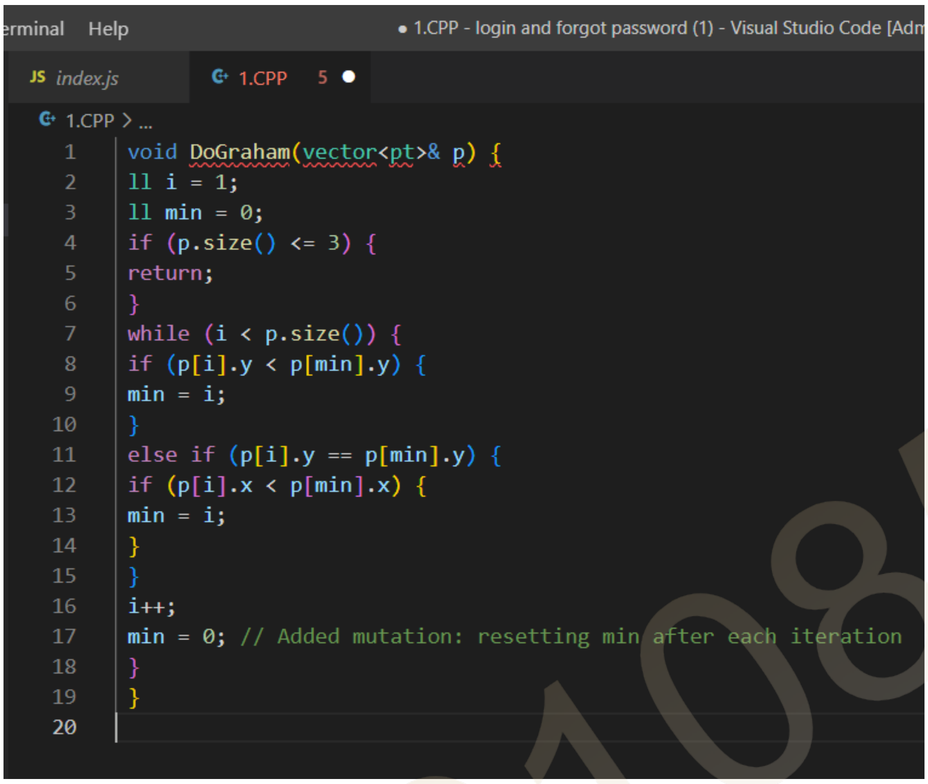
Expected Outcome: With the <= condition, points with equal y values could mistakenly replace min, potentially leading to the incorrect selection of the minimal point. As current test cases may not fully assess scenarios with multiple points having identical y values, this issue might go unnoticed.

Test Case Requirement: Include a test case with multiple points sharing the same y value (e.g., (2, 1), (1, 1), (3, 1), (0, 1)) to ensure that the smallest x is selected correctly. This would reveal any incorrect behaviour of selecting the last occurrence rather than the actual minimum.

3. **Insertion Mutation**: Add extra statements or conditions to the code.

Mutation Example: Insert a line that resets the min index at the end of each loop iteration.

Code


erminal Help
● 1.CPP - login and forgot password (1) - Visual Studio Code [Adm
index.js
1.CPP 5
1.CPP > ...

```
void DoGraham(vector<pt>& p) {
ll i = 1;
ll 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++;
min = 0; // Added mutation: resetting min after each iteration
}
}
```

Expected Outcome: Resetting min after each iteration prevents the code from accurately tracking the actual minimum point found, as min is continuously reset to 0. This can result in incorrect outputs, especially with sequences where the smallest y or x value is not at the start.

Test Case Requirement: A vector p with minimum values in varied positions, such as [(5, 5), (1, 0), (2, 3), (4, 2)], would help reveal this issue. The correct min should persist across iterations, which this mutation would disrupt, causing the test to fail.

Task- 4

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**: p.size() = 0 (No iterations)

- **Input**: p = [] (an empty vector).
- **Expected Outcome**: Since p.size() == 0, the method exits immediately without entering the loop.
- **Path Covered**: This covers the scenario where the loop condition fails at the outset, so the loop does not execute.

✓ **Test Case 2**: p.size() = 1 (No iterations)

- **Input**: p = [(2, 3)] (a single point).
- **Expected Outcome**: As p.size() <= 3, the method returns without processing further, covering the case where the loop is bypassed.
- **Path Covered**: Confirms that the method exits early due to p.size() <= 3.

✓ **Test Case 3**: p.size() = 4 with one loop iteration

- **Input**: p = [(1, 1), (3, 4), (5, 6), (7, 8)].
- **Expected Outcome**: The method checks p.size() > 3, then enters the loop. During the first iteration, it evaluates (3, 4), updates min to the index with the lowest y or x value, and then exits.
- **Path Covered**: Covers the path where the loop executes a single iteration before completion.

✓ **Test Case 4**: p.size() = 4 with two loop iterations

- **Input**: p = [(2, 2), (4, 3), (1, 1), (5, 5)].
- **Expected Outcome**: The method evaluates the first two points in the loop to identify the minimum. After two iterations, it updates min and prepares to continue or exit.
- **Path Covered**: Covers the path where the loop executes exactly twice.

Lab Execution

Q1: After generating the control flow graph, verify whether your CFG aligns with the CFG produced by the Control Flow Graph Factory Tool and the Eclipse flow graph generator.

→ **YES**

Q2: Determine the minimum number of test cases required to cover the code based on the criteria mentioned above.

❖ <u>**Answer**</u>

- Statement Coverage: 3
- Branch Coverage: 3
- Basic Condition Coverage: 2
- Path Coverage: 3

Summary of Minimum Test Cases:

➢ **Total**: 3 (Statement) + 3 (Branch) + 2 (Basic Condition) + 3 (Path) = 11 test cases