# IT-314 SoftWare Engineering
# Lab : 9

## MUTATION TESTING

Name : vaghela vivek

Student ID: 202201197S

Lab Group: 3

**IT313: Software Engineering**

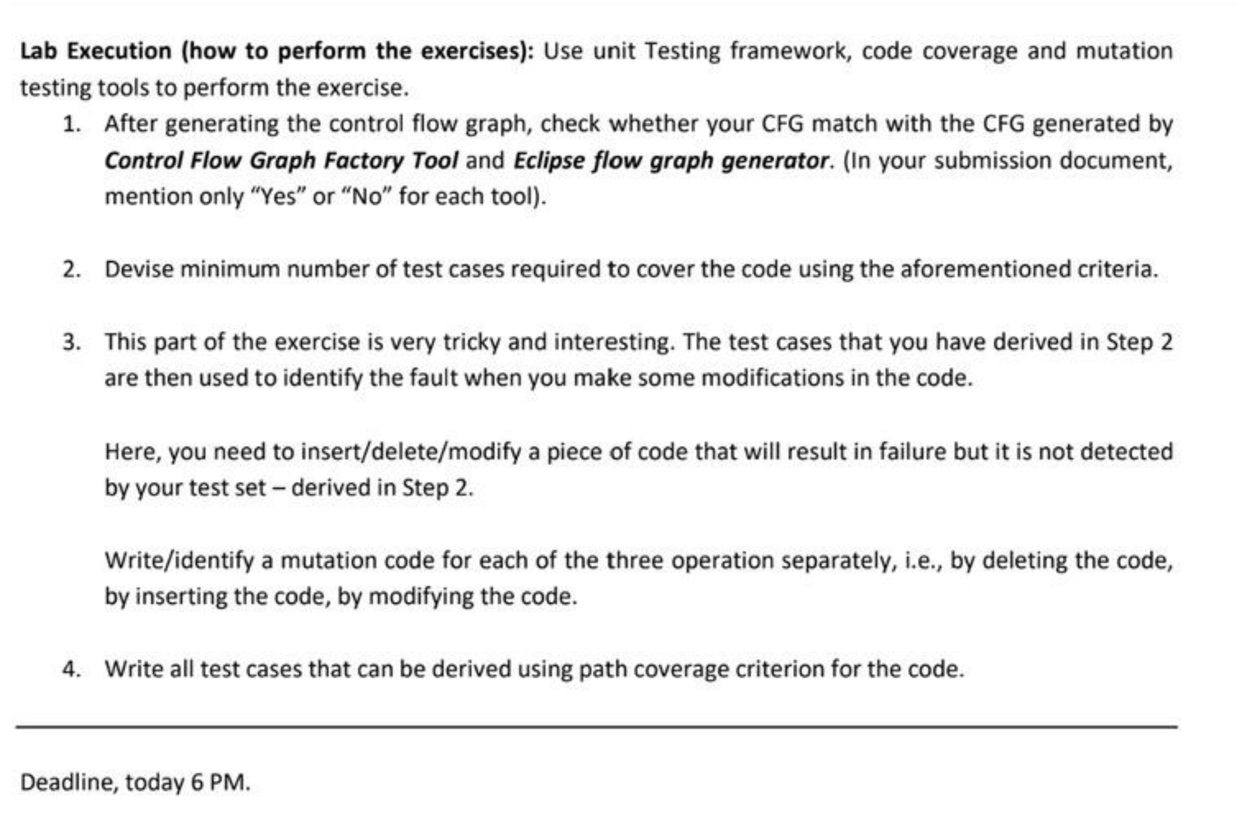
**Lab Session – Mutation Testing**

**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.

```
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;
            }
        }
```

For the given code fragment, you should carry out the following activities.

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



**Lab Execution (how to perform the exercises):** Use unit Testing framework, code coverage and mutation testing tools to perform the exercise.

1. After generating the control flow graph, check whether your CFG match with the CFG generated by ***Control Flow Graph Factory Tool*** and ***Eclipse flow graph generator***. (In your submission document, mention only "Yes" or "No" for each tool).

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

3. This part of the exercise is very tricky and interesting. The test cases that you have derived in Step 2 are then used to identify the fault when you make some modifications in the code.

   Here, you need to insert/delete/modify a piece of code that will result in failure but it is not detected by your test set – derived in Step 2.

   Write/identify a mutation code for each of the three operation separately, i.e., by deleting the code, by inserting the code, by modifying the code.

4. Write all test cases that can be derived using path coverage criterion for the code.

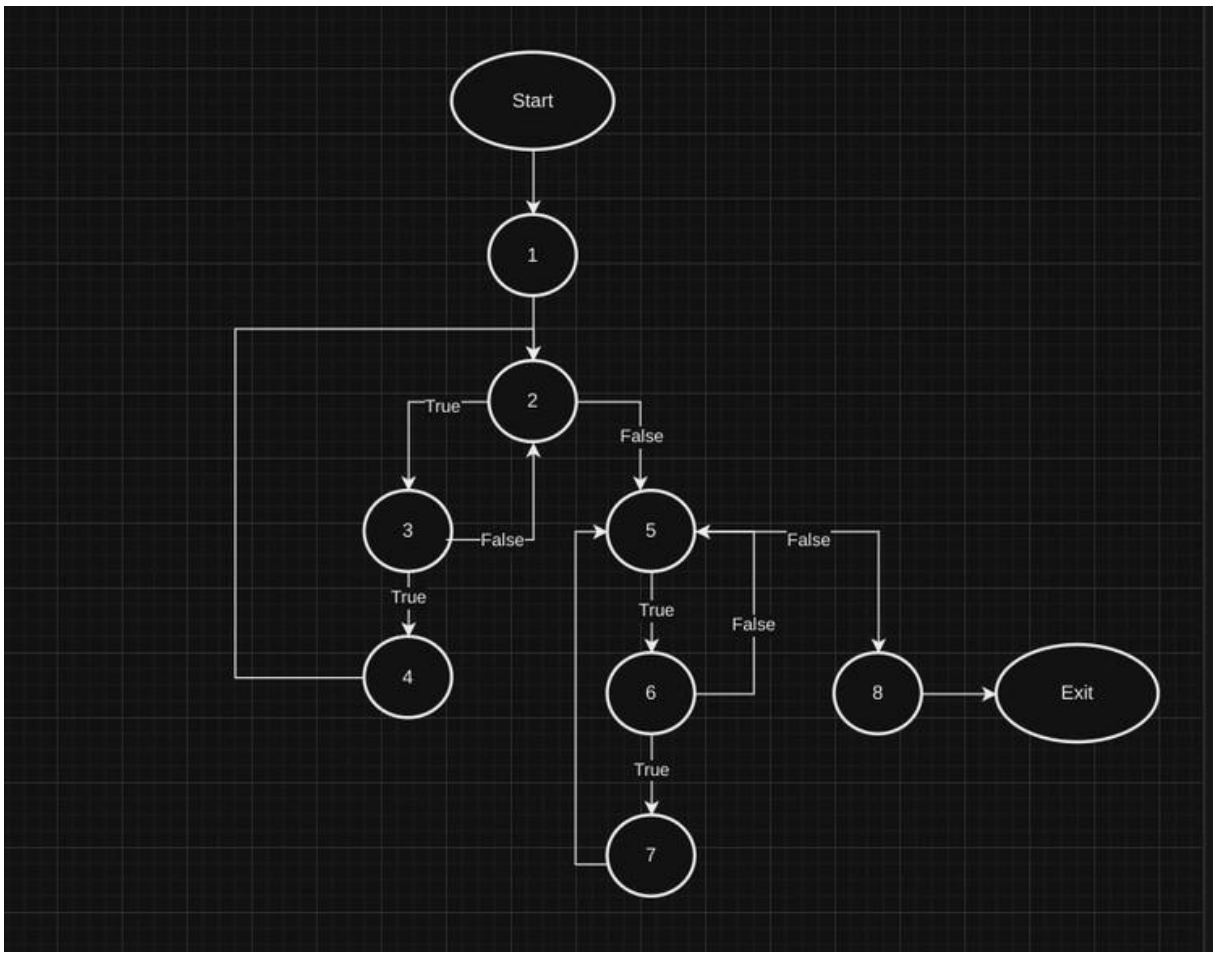
---

Deadline, today 6 PM.

```
class Point:
  def __init__(self, x, y):
    self.x = x
    self.y = y
class Vector:
     def __init__(self, points):
          self.points = points
     def size(self):
          return len(self.points)
     def get(self, i):
               return self.points[i]
```

```
def doGraham(p):
  (1   min_idx = 0
  )    for i in range(1, p.size()):
  (2  (3)if p.get(i).y < p.get(min_idx).y:
  )          (4) min_idx  =  i
    (5)  for i in range(p.size()):
        (6)  if p.get(i).y == p.get(min_index).y and p.get(i).x >
p.get(min_idx).x:
            (7)  min_idx = i
 (8) return min_idx
```

**Control Flow Graph:**



## 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.

In doGraham, there are two main sections:

1. FirstLoop:Thisloopiteratesoverthepointstofindtheonewiththe smallest y value. If there are multiple points with the same y value, it keeps the first one encountered.
   - Condition 1: i < p.size() – controls the termination of the loop.
   - Condition2:((Point) p.get(i)).y < ((Point) p.get(min)).y – updates min if the y value of the current point is less than the current minimum.
2. SecondLoop:Thisloopiteratesoverthepointsagaintofindthepoint with the same y value as min but the largest x value.
   - Condition3:((Point) p.get(i)).y == ((Point) p.get(min)).y – checks if the current point has the same y as the minimum.
   - Condition 4: ((Point) p.get(i)).x > ((Point) p.get(min)).x –updates min ifthecurrentpoint's x islarger thanthecurrentmaximum wixththesame . y

Coverage Criteria

- StatementCoverage:Ensureseverylineofcodeisexecutedatleast once.
- Branch Coverage: Ensures each branch (i.e., each possible true/false path) is taken at least once.
- Basic Condition Coverage: Ensures each basic condition (part of a compound condition) is tested for both true and false outcomes independently.

## 1. Statement Coverage

To achieve statement coverage, each line of code must be executed at least once. This can be done with minimal cases to exercise each part of the code:

- TestCase1:Asinglepoint(0, 0)
  - This will initialize min = 0 but won't update it, as there's only one point.
  - Coverstheinitializationandbothloopswithoutmakingany updates.
- TestCase2:Twopointswithdifferentyvalues,e.g.,[(0, 0), (1, 1)]
  - This will cover the update to min in the first loop.
  - The second loop will execute without updating `min`, as the `y` values are different.

## 2. Branch Coverage

To achieve branch coverage, each branch in the code must be tested, ensuring both the true and false outcomes of each condition are covered. This requires more diverse points:

- TestCase3:Multiplepointswithincreasingyvalues,e.g.,[(0, 1), (1, 2), (2, 3)]
  - Ensures `Condition 2` evaluatestobothtrueandfalse.Here,the first loop will find the minimum y value, but no updates will occur in the second loop due to differing y values.
- Test Case 4: Points where one point has a larger x value but the same y astheminimum,e.g.,[(0, 0), (2, 0), (1, 1)]
  - ThiswillcoverbothbranchesofCondition 4inthesecond loop.
  - Inthiscase,thesecondloopwillfindthatminshouldbeupdated to the point (2, 0) as it has the same y as (0, 0) but a larger x.

## 3. Basic Condition Coverage

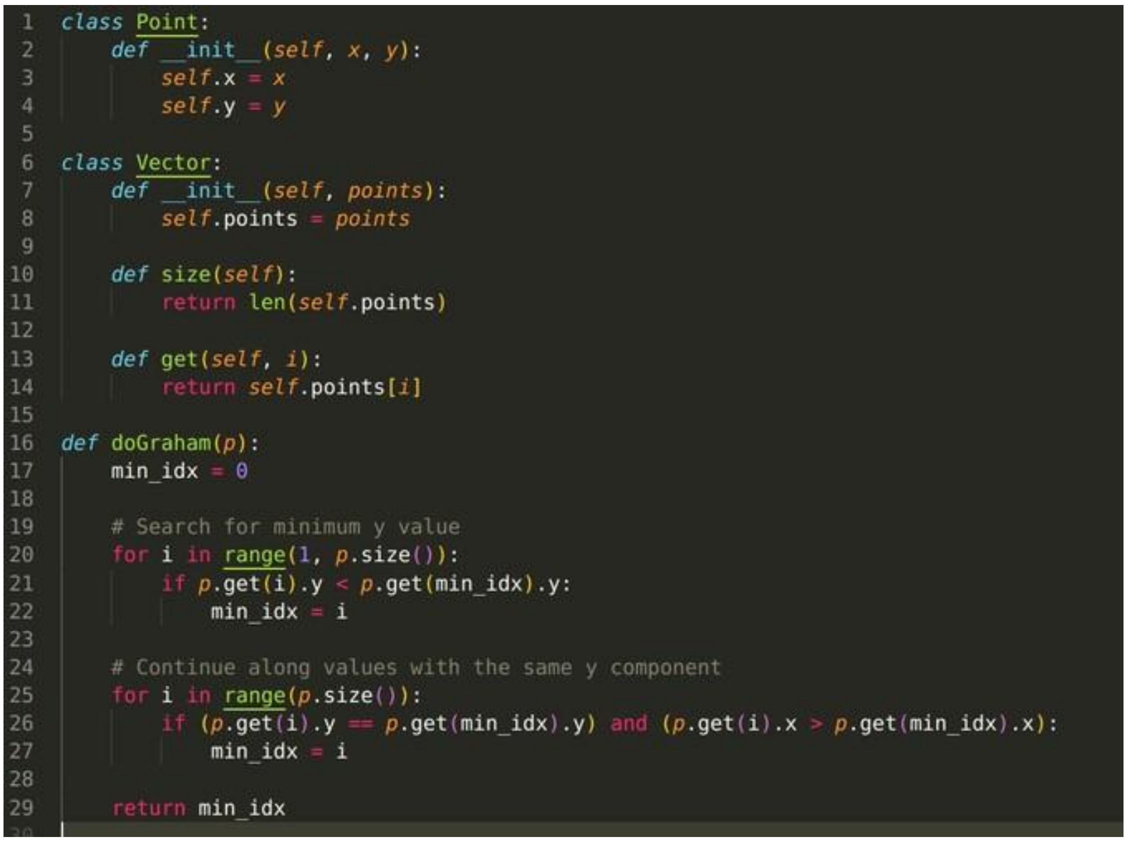
To achieve basic condition coverage, each basic condition (each part of a compound condition) must independently evaluate to true and false. This requires a more nuanced set of points to trigger all possible combinations of true and false for each condition:

- Test Case 5: Points with equal y values but different x values, e.g., `[(0, 0), (2, 0), (1, 0)]`
  - ThiswilltestCondition 4toensurethatminisupdatedtothe point with the largest x, which would be (2, 0).
- Test Case 6: Points with different y values and some points sharing the `samey,e.g.,[(1, 1), (2, 1), (0, 0)]`
  - EnsuresbothCondition 2(ycomparisoninfirstloop)and Condition 3(same yconditioninsecondloop)areevaluated to 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**

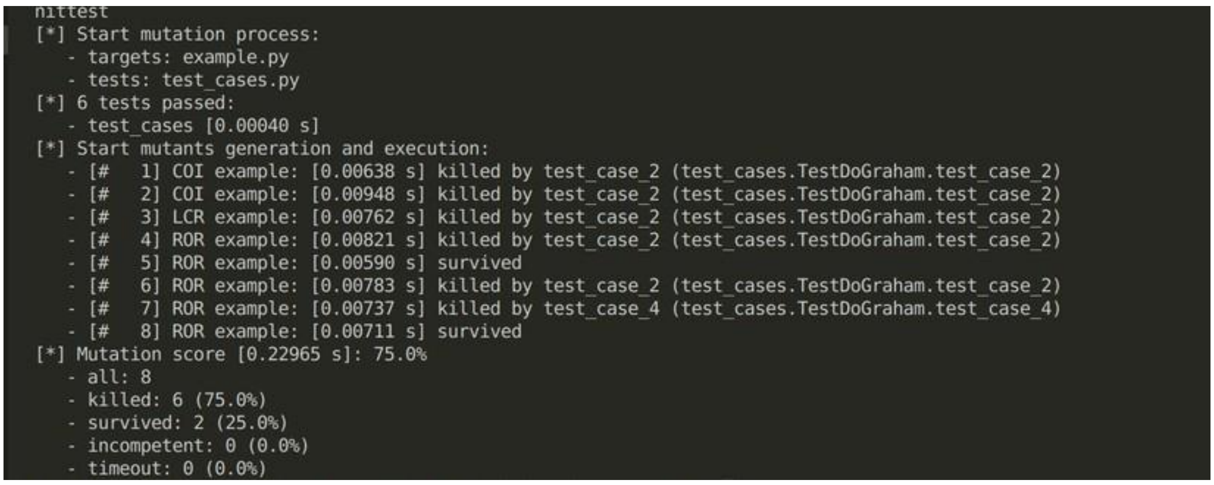
**or insertion of some code) that will result in failure but is not detected by your test set. You have to use the mutation testing tool.**

**Python Code for mutation testing:**



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

class Vector:
    def __init__(self, points):
        self.points = points

    def size(self):
        return len(self.points)

    def get(self, i):
        return self.points[i]

def doGraham(p):
    min_idx = 0

    # Search for minimum y value
    for i in range(1, p.size()):
        if p.get(i).y < p.get(min_idx).y:
            min_idx = i

    # Continue along values with the same y component
    for i in range(p.size()):
        if (p.get(i).y == p.get(min_idx).y) and (p.get(i).x > p.get(min_idx).x):
            min_idx = i

    return min_idx
```

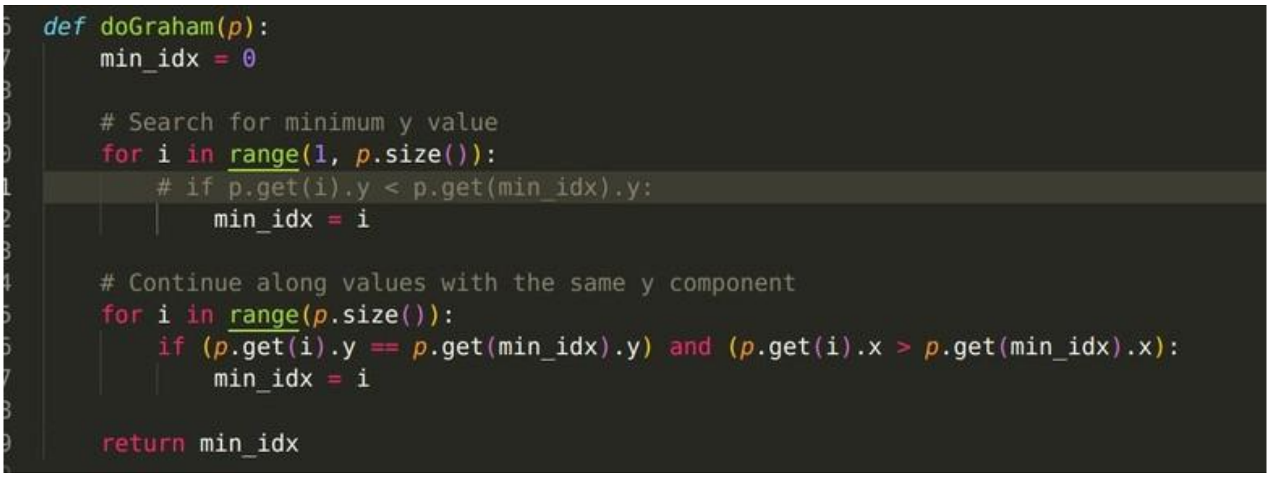
**Using the mut.py as mutation testing tool for python**



```
nittest
[*] Start mutation process:
   - targets: example.py
   - tests: test_cases.py
[*] 6 tests passed:
   - test_cases [0.00040 s]
[*] Start mutants generation and execution:
   - [#   1] COI example: [0.00638 s] killed by test_case_2 (test_cases.TestDoGraham.test_case_2)
   - [#   2] COI example: [0.00948 s] killed by test_case_2 (test_cases.TestDoGraham.test_case_2)
   - [#   3] LCR example: [0.00762 s] killed by test_case_2 (test_cases.TestDoGraham.test_case_2)
   - [#   4] ROR example: [0.00821 s] killed by test_case_2 (test_cases.TestDoGraham.test_case_2)
   - [#   5] ROR example: [0.00590 s] survived
   - [#   6] ROR example: [0.00783 s] killed by test_case_2 (test_cases.TestDoGraham.test_case_2)
   - [#   7] ROR example: [0.00737 s] killed by test_case_4 (test_cases.TestDoGraham.test_case_4)
   - [#   8] ROR example: [0.00711 s] survived
[*] Mutation score [0.22965 s]: 75.0%
   - all: 8
   - killed: 6 (75.0%)
   - survived: 2 (25.0%)
   - incompetent: 0 (0.0%)
   - timeout: 0 (0.0%)
```

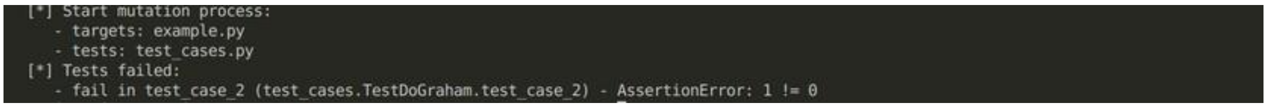
## 1. Mutation by Deleting Code:

**Commented the if condition:**



```python
def doGraham(p):
    min_idx = 0

    # Search for minimum y value
    for i in range(1, p.size()):
        # if p.get(i).y < p.get(min_idx).y:
            min_idx = i

    # Continue along values with the same y component
    for i in range(p.size()):
        if (p.get(i).y == p.get(min_idx).y) and (p.get(i).x > p.get(min_idx).x):
            min_idx = i

    return min_idx
```

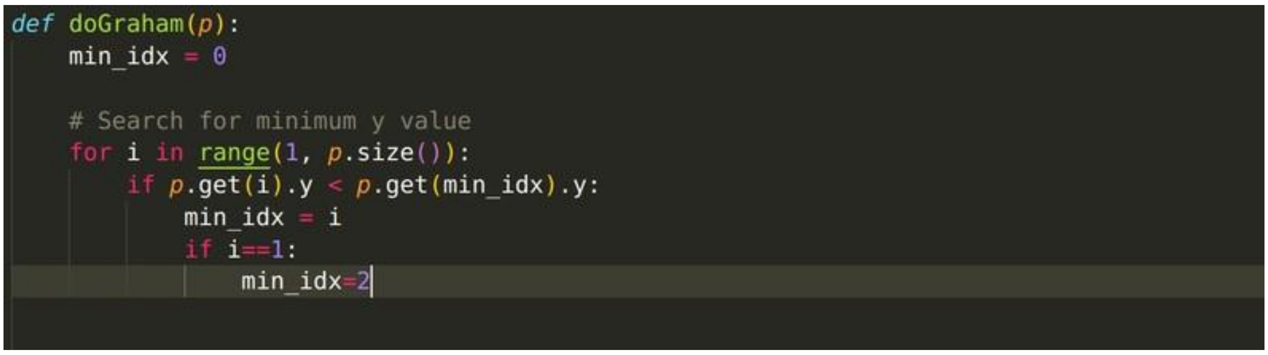
**Output of the Mutation testing:**



```
[*] Start mutation process:
   - targets: example.py
   - tests: test_cases.py
[*] Tests failed:
   - fail in test_case_2 (test_cases.TestDoGraham.test_case_2) - AssertionError: 1 != 0
```

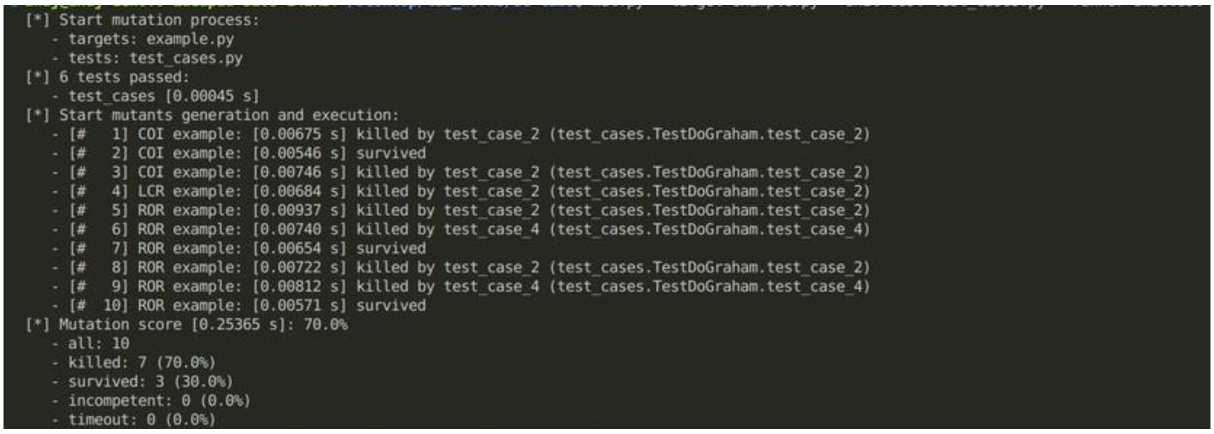
## 2. Mutation by Inserting Code:

**Added an if Condition:**



```python
def doGraham(p):
    min_idx = 0

    # Search for minimum y value
    for i in range(1, p.size()):
        if p.get(i).y < p.get(min_idx).y:
            min_idx = i
            if i==1:
                min_idx=2
```

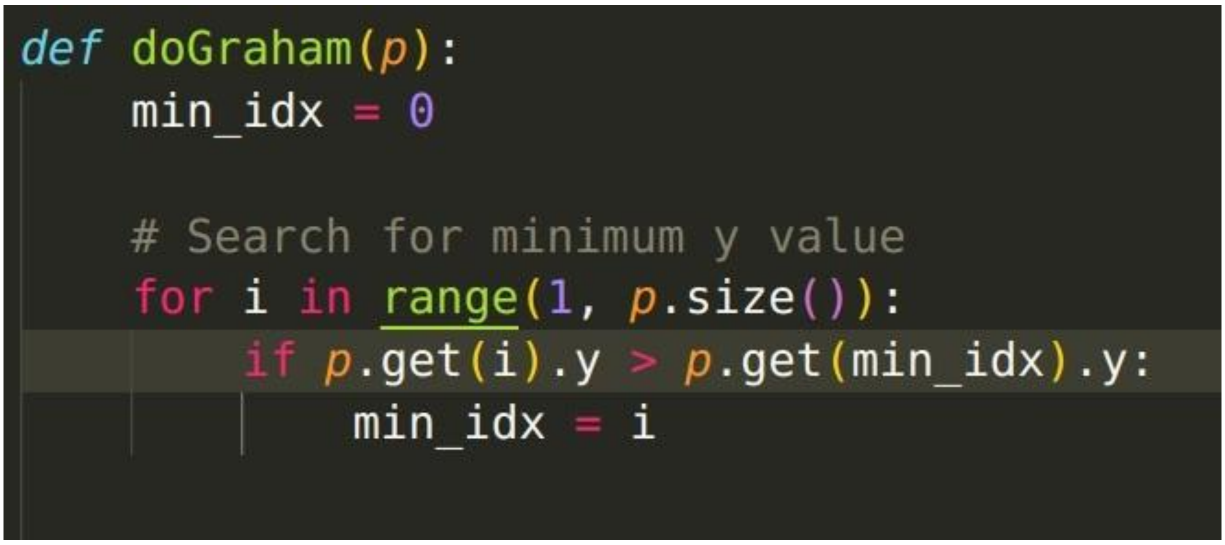
**Output of Mutation testing:**



```
[*] Start mutation process:
   - targets: example.py
   - tests: test_cases.py
[*] 6 tests passed:
   - test_cases [0.00045 s]
[*] Start mutants generation and execution:
   - [#   1] COI example: [0.00675 s] killed by test_case_2 (test_cases.TestDoGraham.test_case_2)
   - [#   2] COI example: [0.00546 s] survived
   - [#   3] COI example: [0.00746 s] killed by test_case_2 (test_cases.TestDoGraham.test_case_2)
   - [#   4] LCR example: [0.00684 s] killed by test_case_2 (test_cases.TestDoGraham.test_case_2)
   - [#   5] ROR example: [0.00937 s] killed by test_case_2 (test_cases.TestDoGraham.test_case_2)
   - [#   6] ROR example: [0.00740 s] killed by test_case_4 (test_cases.TestDoGraham.test_case_4)
   - [#   7] ROR example: [0.00654 s] survived
   - [#   8] ROR example: [0.00722 s] killed by test_case_2 (test_cases.TestDoGraham.test_case_2)
   - [#   9] ROR example: [0.00812 s] killed by test_case_4 (test_cases.TestDoGraham.test_case_4)
   - [#  10] ROR example: [0.00571 s] survived
[*] Mutation score [0.25365 s]: 70.0%
   - all: 10
   - killed: 7 (70.0%)
   - survived: 3 (30.0%)
   - incompetent: 0 (0.0%)
   - timeout: 0 (0.0%)
```

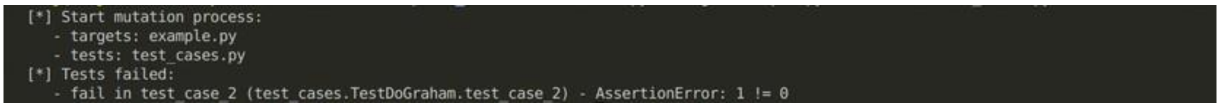
## 3. Mutation by Modifying Code:

### Changing the inequality < to >:



```python
def doGraham(p):
    min_idx = 0

    # Search for minimum y value
    for i in range(1, p.size()):
        if p.get(i).y > p.get(min_idx).y:
            min_idx = i
```

### Output of Mutation testing:



```
[*] Start mutation process:
   - targets: example.py
   - tests: test_cases.py
[*] Tests failed:
   - fail in test_case_2 (test_cases.TestDoGraham.test_case_2) - AssertionError: 1 != 0
```

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

**one or two times.**

**1. Test Case: Zero Iterations of the First Loop**

- Description:Thistestverifiesthebehaviorofthefunctionwhen only one point is provided, ensuring that the first loop does not execute.
- Input:
    - Points: (0,0)(0, 0)(0,0)
- Expected Output: 0 (The index of the only point)

**2. Test Case: One Iteration of the First Loop**

- Description:Thistestchecksthecasewithtwopoints,wherethe first point has a lower y value than the second, causing the first loop to run once.

- Input:
    - Points:(0,0),(1,1)(0,0),(1,1)(0,0),(1,1)
- Expected Output: 0 (The index of the point with the minimum y value)

**3. Test Case: Two Iterations of the First Loop**

- Description:Thistestassessesthefunction’shandlingofthree points with increasing y values, ensuring the first loop iterates twice.
- Input:
  - Points:(0,1),(1,2),(2,3)(0,1),(1,2),(2,3)(0,1),(1,2),(2,3)
- ExpectedOutput:0(Theindexofthepointwiththeminimum y value)

**4. Test Case: Zero Iterations of the Second Loop**

- Description:Thistestverifiesthatthesecondloopdoesnot execute when only one point is provided.
- Input:
  - Points: (0,0)(0, 0)(0,0)
- Expected Output: 0 (The index of the only point)

**5. Test Case: One Iteration of the Second Loop**

- Description:Thistestchecksthebehaviorofthefunctionwith two points that have the same y value, ensuring that the second loop runs once.
- Input:
  - Points:(0,0),(1,0)(0,0),(1,0)(0,0),(1,0)
- ExpectedOutput:1(Theindexofthepointwiththelargerx value)

**6. Test Case: Two Iterations of the Second Loop**

- Description:Thistestassessesthefunctionwiththreepoints that all have the same y value, ensuring the second loop runs twice and correctly selects the last point.
- Input:
  - Points:(0,0),(1,0),(2,0)(0,0),(1,0),(2,0)(0,0),(1,0),(2,0)
- Expected Output: 2 (The index of the point with the largest x value)