IT 314 - Software Engineering

Date - Apr 13, 2023

Name - Chhagani Krunal Ajaybhai

Id - 202001158

Section A:

Consider a program for determining the previous date. Its input is triple of day, month and year with the following ranges 1 <= month <= 12, 1 <= day <= 31, 1900 <= year <= 2015.The possible output dates would be previous date or invalid date. Design the equivalence class test cases?

Consider a program for determining the previous date. Its input is triple of day, month and year with the following ranges 1 <= month <= 12, 1 <= day <= 31, 1900 <= year <= 2015.The possible output dates would be the previous date or invalid date. Design the equivalence class test cases?

Class 1: Valid Dates (Dates from (1, 1, 1900) to (31, 12, 2015) such that day, month and year are in the given ranges. ex: (13, 2, 2002), (3, 7, 2010), (21, 12, 1995) etc.)
Class 2: Invalid Dates (Dates from (1, 1, 1900) to (31, 12, 2015) such that day, month and year are in the given ranges but the input is invalid. ex: (31, 6, 2002), (29, 2, 2010), (30, 2, 2000) etc.)
Class 3: Invalid Range (Dates from (1, 1, 1900) to (31, 12, 2015) such that day, month and year are not in the given ranges. ex: (33, 6, 2020), (1,16,2033), (15,0,2022) etc.)
Class 4: Invalid Input (Dates from (1, 1, 1900) to (31, 12, 2015) such that day, month and year are in the given ranges but the input is invalid. ex: (2, 1.2, 2022), (12, a, 2013), (-2, 5, 2010) etc.)

Tester Action and Input Data Expected Outcome:

Valid Dates:
Test Case 1:
Input: (13, 2, 2002)
Output: (12, 2, 2002)
Test Case 2:
Input: (3,7,2010)
Output: (2,7,2010)
Test Case 2:
Input: (21, 12, 1995)
Output: (20, 12, 1995)

Invalid Dates:
Test Case 1:

Input: (31, 6, 2002)

Output: (30, 6, 2002)

Test Case 2:

Input: (29, 2, 2010)

Output: (28, 2, 2010)

Test Case 3:

Input: (3,7,2010)

Output: (2,7,2010)

Invalid Range:

Test Case 1:

Input: (33, 6, 2020)

Output: Invalid Date

Test Case 2:

Input: (1,16,2033)

Output: Invalid Date

Test Case 3:

Input: (15,0,2022)

Output: Invalid Date

Invalid Input:

Test Case 1:

Input: (2, 1.2, 2022)

Output: Invalid Date

Test Case 2:

Input: (12, a, 2013)

Output: Invalid Date

Test Case 3:

Input: (-2, 5, 2010)

Output: Invalid Date

Boundary Value Analysis:

Test Case 1: Valid First Possible Date

Input: (1,1,1900)

Output: (31,12,1899) Which is not in range.

Test Case 2: Valid Last Possible Date

Input: (31,12,2015)

Output: (30,12,2015)

Test Case 3: One Day Before First Possible Date

Input: (31,1,1899)

Output: Invalid Input

Test Case 4: One Day After Last Possible Date

Input: (1,1,2016)

Output: Invalid Input

Test Case 5: Valid Leap Year Date

Input: (29,2,2000)

Output: (28,2,2000)

Test Case 6: Invalid Leap Year Date

Input: (29,2,1900)

Output: Invalid input

Test Case 7: Valid Date After Leap Year Date

Input: (1,3,2000)

Output: (29,2,2000)

Test Case 8: Valid Date After Non Leap Year Date

Input: (1,3,2019)

Output: (28,2,2019)

Test Case 9: Valid First Day of Month

Input: (1,3,2000)

Output: (31,12,1999)

Test Case 10: Valid First Day of Year

Input: (1,1,2000)

Output: (31,12,1999)

Write a set of test cases (i.e., test suite) – specific set of data – to properly test the programs. Your test suite
should include both correct and incorrect inputs.

1. Enlist which set of test cases have been identified using Equivalence Partitioning and Boundary Value
Analysis separately.
2. Modify your programs such that it runs on eclipse IDE, and then execute your test suites on the
program. While executing your input data in a program, check whether the identified expected
outcome (mentioned by you) is correct or not.

Programs :

1. The function linearSearch searches for a value v in an array of integers a. If v appears in the array a, then the function returns the first index i, such that a[i] == v; otherwise, -1 is returned.

Equivalence partitioning and Boundary Value Analysis :

Test Actions and Input Data	Expected Output
Equivalence Partitioning	
a = [2,3,4,5,6] , v = 5	3
a = [1,2,4,5] , v = 10	-1
a = [2,3,3,3,4] , v = 3	1
a = null , v = 4	Error message
Boundary Value Analysis	
Minimum Array length a = [], v = 5	-1
Maximum array length a = [1,2,3,4,5,6,7,8,9,0,20,19,18,17,16,15,14,13,12,11], v = 3	2
Minimum value of v : a = [3,4,5], v = 3	0
Maximum value of v : a = [3,4,5], v = 5	2

2. The function countItem returns the number of times a value v appears in an array of integers a.

Equivalence Partitioning and Boundary Value Analysis :

Test Actions and Input Data	Expected Output
Equivalence Partitioning	
Invalid input: v is not an integer	Error Message
Empty array: a = []	0
Single item array: a = [v], v = a[0]	1
Multiple item array with v appearing:	
v appears once	1
v appears multiple times	Count > 1
Multiple item array with v not appearing	0
Boundary Value Analysis	
Minimum input values: v = a[0] = 1	Count > 0
Maximum input values: v = a[9999]	Count > 0

= 10000	
One occurrence of v: a = [1, 2, 3, ..., 9999, v, 10000]	1
All occurrences of v: a = [v, v, v, ..., v, v]	10000
No occurrences of v: a = [1, 2, 3, ..., 9999]	0

3. The function binarySearch searches for a value v in an ordered array of integers a. If v appears in the array a, then the function returns an index i, such that a[i] == v; otherwise, -1 is returned.\

Equivalence Partitioning :

Test cases for correct inputs :

Test Actions and Input Data	Expected Output
a = [2,3,4,5,6,7,8,9] , v = 5	3
a = [1,2,4,5,5,8,8] , v = 5	3
a = [2,3,3,3,4] , v = 3	1

Test Cases for incorrect inputs :

Test Actions and Input Data	Expected Output
Equivalence Partitioning	
a= [], v = 1	-1
a = [1,2,4,5] , v = 10	-1
a = [2,3,3,3,4] , v = 5	-1

Boundary value analysis:

Testcase for correct inputs:

Tester Action and Input data	Expected Outcome
Equivalence Partitioning	
A = [3,5,6,10,11] , v = 5	1
A = [2,2,3,3,4,5,5], v = 5	5
A = [1,2,3] , v = 1	0
A = null, v = 1	An error message
Boundary value analysis	
Minimum length of array: A = [] , v = 5	-1
Maximum length of array: a =	2

[2,3,4,5,5,6,10,11,12,13], v = 4	
Minimum value of v: a = [5,7,9] , v = 5	0
Maximum value of v: a = [1,2,3], v = 3	2

Test Cases for Incorrect Inputs:

Tester Action and Input Data	Expected Outcome
v = 4, a = [1, 3, 5, 7, 9]	-1
v = 11, a = [1, 3, 5, 7, 9]	-1
v = 2, a = [1, 3, 5, 7, 9]	-1
v = 9, a = [2, 3, 4, 5, 6]	-1
v = 1, a = [2, 3, 4, 5, 6]	-1
v = 10, a = [5, 6, 7, 8, 9]	-1

4. The following problem has been adapted from The Art of Software Testing, by G. Myers (1979). The function triangle takes three integer parameters that are interpreted as the lengths of the sides of a triangle. It returns whether the triangle is equilateral (three lengths equal), isosceles (two lengths equal),scalene (no lengths equal), or invalid (impossible lengths).

Tester Action and Input Data	Expected Outcome
Equivalence Partitioning:	
a=b=c, where a, b, c are positive integers	EQUILATERAL
a=b<c, where a, b, and c are positive integers	ISOSCELES
a=b=c=0	INVALID
Tester Action and Input Data	Expected Outcome
Equivalence Partitioning:	
a<b+c, b<a+c, c<a+b, where a, b, c are positive integers	SCALENE
a=b>0, c=0	INVALID
a>b+c	INVALID
Boundary Value Analysis:	
a=1, b=1, c=1	EQUILATERAL
a=1, b=2, c=2	ISOSCELES
a=0, b=0, c=0	INVALID
a=2147483647, b=2147483647, c=2147483647	EQUILATERAL
a=2147483646, b=2147483647, c=2147483647	ISOSCELES
a=1, b=1, c=2^31-1	SCALENE
a=0, b=1, c=1	INVALID

5. The function prefix (String s1, String s2) returns whether or not the string s1 is a prefix of string s2 (you may assume that neither s1 nor s2 is null).

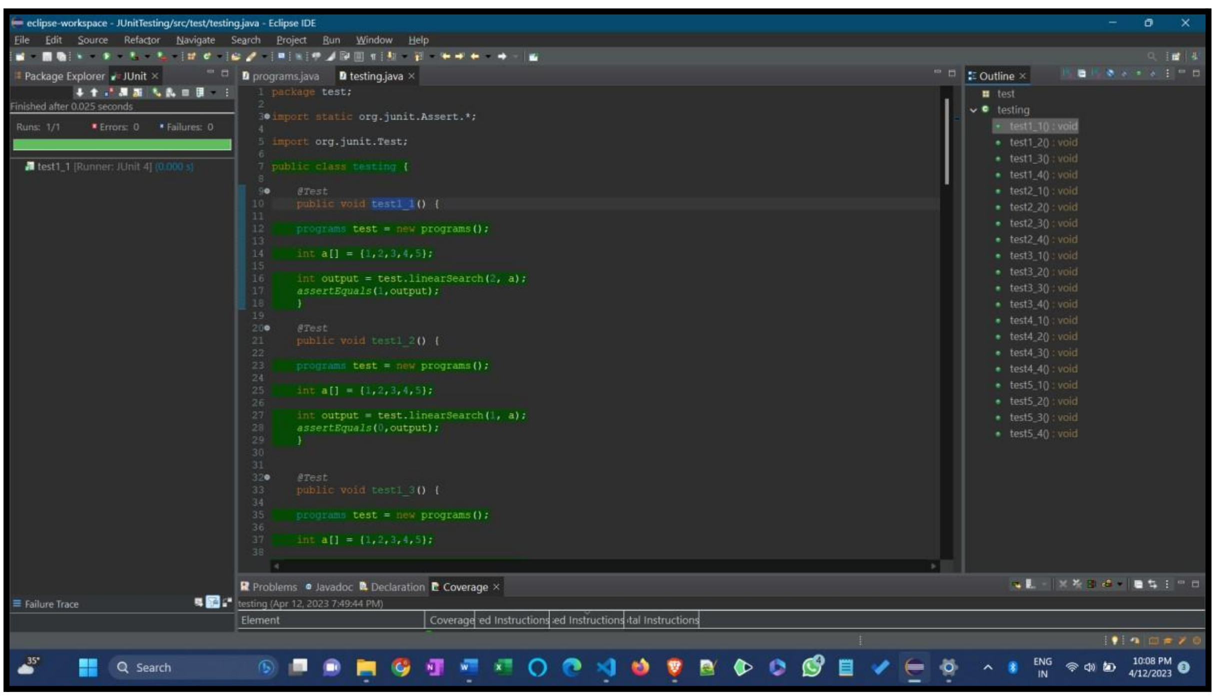
Equivalence Partitioning and Boundary Value Analysis

Tester Action and Input Data	Expected Outcome
Equivalence Partitioning:	
s1 is empty, s2 is non-empty string	true
s1 is non-empty string, s2 is empty	false
s1 is a prefix of s2	true
s1 is not a prefix of s2	false
s1 has same characters as s2, but not a prefix	false
Boundary Value Analysis:	
s1 = "a", s2 = "ab"	true
s1 = "ab", s2 = "a"	false
s1 = "a", s2 = "a"	true
s1 = "a", s2 = "A"	false

Modify your programs such that it runs on eclipse IDE, and then execute your test suites on the program. While executing your input data in a program, check whether the identified expected outcome (mentioned by you) is correct or not.

I've collected 20 test cases, four for each programme. wherein 12 are true while eight are false or invalid.
Code coverage is shown in screenshots of code fragments.

Tester Action and Input Data	Expected Outcome
s1 = "abcdefghijklmnopqrstuvwxyz", s2 = "abcdefghijklmnopqrstuvwxyz"	true
s1 = "abcdefghijklmnopqrstuvwxyz", s2 = "abcdefghijklmno"	true
s1 = "", s2 = ""	true



Modified Java codes of given programs (P1 – P5)

```
package test;

public class programs {

      public int linearSearch(int v, int a[]) // p1
      {

            int i = 0;
            while (i < a.length)
            {
            if (a[i] == v)
            return(i);
            i++;
            }
            return (-1);
            }


      public int countItem(int v, int a[]) //p2
      {

      int count = 0;

      for (int i = 0; i < a.length; i++)

      {

      if (a[i] == v)

      count++;
      }
      return (count);
```

```
	{
		int lo,mid,hi;

		lo = 0;

		hi = a.length-1;
		while (lo <= hi)

		{

		mid = (lo+hi)/2;

		if (v == a[mid])

		return (mid);

		else if (v < a[mid])

		hi = mid-1;

		else

		lo = mid+1;

		}

		return(-1);

		}

	final int EQUILATERAL = 0;
	final int ISOSCELES = 1;
	final int SCALENE = 2;
	final int INVALID = 3;
	public int triangle(int a, int b, int c) //p4
	{
	if (a >= b+c || b >= a+c || c >= a+b)
	return(INVALID);
	if (a == b && b == c)
	return(EQUILATERAL);
	if (a == b || a == c || b == c)
	return(ISOSCELES);
	return(SCALENE);

	}

	public boolean prefix(String s1, String s2) //p5
	{
	if (s1.length() > s2.length())
	{
	return false;
	}
	for (int i = 0; i < s1.length(); i++)
	{
	if (s1.charAt(i) != s2.charAt(i))
	{
	return false;
```

```java
public void test2_1() { // no of element p2

    programs test = new programs();

    int a[] = {1,2,3,4,5};

    int output = test.countItem(2, a);
    assertEquals(2,output);
}

@Test
public void test2_2() { //no of element p2

    programs test = new programs();

    int a[] = {1,2,3,4,5};

    int output = test.countItem(4, a);
    assertEquals(2,output);
}
@Test
public void test2_3() { //no of element p2

    programs test = new programs();

    int a[] = {1,2,3,4,5};

    int output = test.countItem(6, a);
    assertEquals(0,output);
}

@Test
public void test2_4() { //no of element p2

    programs test = new programs();

    int a[] = {1,2,3,4,5};

    int output = test.countItem(6, a);
    assertEquals(-1,output);
}

@Test
public void test3_1() { //binary search p3

    programs test = new programs();

    int a[] = {1,2,3,4,5};

    int output = test.binarySearch(2, a);
    assertEquals(1,output);
}

@Test
public void test3_2() { //binary search p3
```

```
programs test = new programs();

int a[] = {1,2,3,4,5};
```

```java
        int output = test.binarySearch(3, a);
        assertEquals(3,output);
        }


        @Test
        public void test3_3() { //binary search p3

        programs test = new programs();

        int a[] = {1,2,3,4,5};

        int output = test.binarySearch(8, a);
        assertEquals(-1,output);
        }

        @Test
        public void test3_4() { //binary search p3

        programs test = new programs();

        int a[] = {1,2,3,4,5};

        int output = test.binarySearch(8, a);
        assertEquals(-1,output);
        }

        @Test
        public void test4_1() {
        programs test = new programs();
        int output = test.triangle(8,8,8);
        assertEquals(0,output);
        }

        @Test
        public void test4_2() {
        programs test = new programs();
        int output = test.triangle(8,8,10);
        assertEquals(2,output);
        }

        @Test
        public void test4_3() {
        programs test = new programs();
        int output = test.triangle(0,0,0);
        assertEquals(1,output);
        }

        @Test
        public void test4_4() {
        programs test = new programs();
        int output = test.triangle(0,0,0);
        assertEquals(3,output);
        }

        @Test
```

```
    public void test5_1() {
        programs test = new programs();
        boolean output = test.prefix("","nonEmpty");
        assertEquals(true,output);
    }

    @Test
    public void test5_2() {  // example of s1 is prefix of s2
        programs test = new programs();
        boolean output = test.prefix("hello","hello world");
        assertEquals(true,output);
    }

    @Test
    public void test5_3() {  // example of s1 is not prefix of s2
        programs test = new programs();
        boolean output = test.prefix("hello","world hello");
        assertEquals(false,output);
    }

    @Test
    public void test5_4() {  // example of s1 is not prefix of s2
        programs test = new programs();
        boolean output = test.prefix("hello","world hello");
        assertEquals(true,output);
    }

}
```

P6: Consider again the triangle classification program (P4) with a slightly different specification: The program reads floating values from the standard input. The three values A, B, and C are interpreted as representing the lengths of the sides of a triangle. The program then prints a message to the standard output stating whether the triangle can be formed, is scalene, isosceles, equilateral, or right-angled. Determine the following for the above program:

a) Equivalence classes for the system are

Class 1: Invalid inputs (negative or zero values)

Class 2: Non-triangle (sum of the two shorter sides is not greater than the longest side)

Class 3: Scalene triangle (no sides are equal)

Class 4: Isosceles triangle (two sides are equal)

Class 5: Equilateral triangle (all sides are equal)

Class 6: Right-angled triangle (satisfies the Pythagorean theorem

b) Test cases to cover the identified equivalence classes:

Class 1: -1, 0

Class 2: 1, 2, 5

Class 3: 3, 4, 5

Class 4: 5, 5, 7

Class 5: 6, 6, 6

Class 6: 3, 4, 5

Test case 1 covers class 1, test case 2 covers class 2, test case 3 covers class 3, test case 4 covers class 4, test case 5 covers class 5, and test case 6 covers class 6.

c) Test cases to verify the boundary condition A + B > C for the scalene triangle:

(1) 2, 3, 6

(2) 3, 4, 8

Both test cases have two sides shorter than the third side and should not form a triangle.

d) Test cases to verify the boundary condition A = C for the isosceles triangle:

(1) 2, 3, 3

(2) 5, 6, 5

Both test cases have two equal sides and should form an isosceles triangle.

e) Test cases to verify the boundary condition A = B = C for the equilateral triangle:

(1) 5, 5, 5

(2) 9, 9, 9

Both test cases have all sides equal and should form an equilateral triangle.

f) Test cases to verify the boundary condition A^2 + B^2 = C^2 for the right-angled triangle:

(1) 3, 4, 5

(2) 5, 12, 13

Both test cases satisfy the Pythagorean theorem and should form a right-angled triangle.

g) For the non-triangle case, identify test cases to explore the boundary.

(1) 2, 2, 4

(2) 3, 6, 9

Both test cases have two sides that add up to the third side and should not form a triangle.

h) For non-positive input, identify test points.

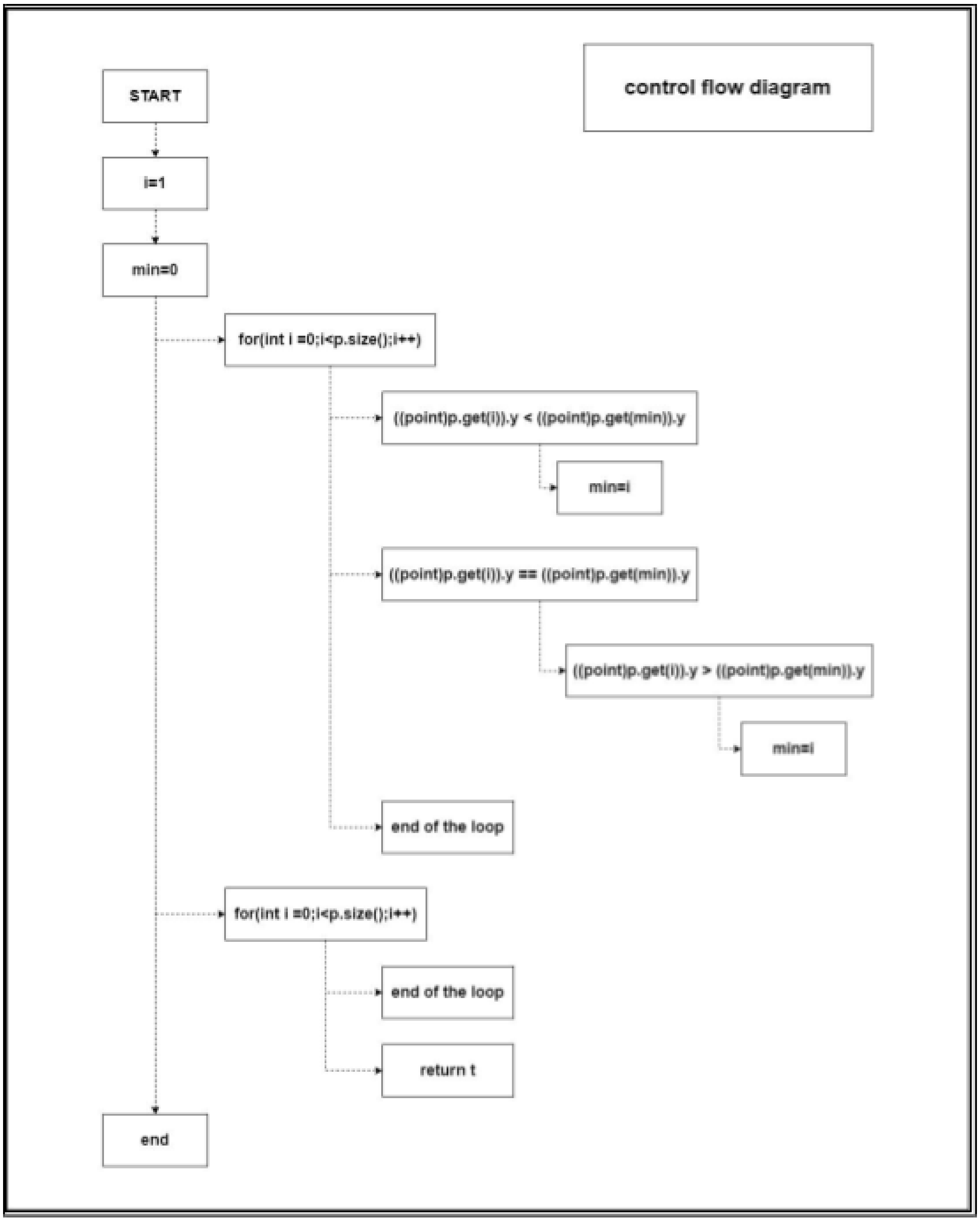
(1) 0, 1, 2

(2) -1, -2, -3

Both test cases have at least one non-positive value, which is an invalid input.

Section A :

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



1. **Construct test sets for your flow graph that are adequate for the following criteria:**

a. Statement Coverage:

We must make sure that every statement in the CFG is run at least once in order to satisfy statement coverage. By offering a test case with just one vector point, we can accomplish this. In this scenario, the return statement will be carried out instead of either of the loops. Statement coverage would be satisfied by the test set p = [Point (0,0)].

b. Branch Coverage:

We must make sure that every branch in the CFG is executed at least once in order to satisfy branch coverage. To do this, we need to create a test case with two points where one of them has the minimum y-coordinate and the other has an x-coordinate that is greater than the minimum. Both loops will run in this situation, and the second branch of the second loop will be taken. An example of a test set that satisfies branch coverage

p = [Point (0,0), Point (1,1)]

c. Basic Condition Coverage:

Every statement in the CFG must be run at least once in order to satisfy the requirement of statement coverage. This can be done by giving a test case that contains only one vector point. Both loops won't run in this scenario, and the return statement will instead be carried out. The test set p = [Point (0,0)] would satisfy statement coverage.

p = [Point (0,0), Point (1,1), Point (2,0)]