

**IT-314**  
**Software Engineering**

**Name- Archit Singh**

**Student ID- 202001272**

**Group 5**

**Project 27**

## **Section A**

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

Answer-

1. Equivalence Classes for the Day parameter:
  - a. Valid classes:  $1 \leq \text{day} \leq 31$
  - b. Invalid classes:  $\text{day} < 1$ ,  $\text{day} > 31$
2. Equivalence Classes for the Month parameter:
  - a. Valid classes:  $1 \leq \text{month} \leq 12$
  - b. Invalid classes:  $\text{month} < 1$ ,  $\text{month} > 12$
3. Equivalence Classes for the Year parameter:
  - a. Valid classes:  $1900 \leq \text{year} \leq 2015$
  - b. Invalid classes:  $\text{year} < 1900$ ,  $\text{year} > 2015$

### **Test Cases**

#### **1. Equivalence Partitioning**

In equivalence partitioning, we divide the input data into groups, or partitions, where each group contains a set of equivalent or similar values that are expected to exhibit similar behavior in the system under test.

Here are some partitions based on different values:

**Partition 1:** Valid dates with a day between 1 and 31, a month between 1 and 12, and a year between 1900 and 2015.

**Partition 2:** Invalid dates with a day less than 1 or greater than 31.

**Partition 3:** Invalid dates with a month less than 1 or greater than 12.

**Partition 4:** Invalid dates with a year less than 1900 or greater than 2015.

**Partition 5:** Invalid dates with a day that is out of range for a given month (e.g., February 30).

**Partition 6:** Invalid dates with a day that is out of range for a given year (e.g., February 29 in a non-leap year).

Some **sample test cases** for different partitions:

**Partition 1:** 01/05/2008, 15/04/1992, 31/8/2005

**Partition 2:** 00/01/2004, -11/04/2002, 32/11/2001

**Partition 3:** 05/00/2002, 10/13/2010, 30/16/2011

**Partition 4:** 02/10/0000, 15/03/11000, 03/10/9999

**Partition 5:** 30/02/2023, 31/04/2023, 02/05/2101

**Partition 6:** 29/02/2021, 29/02/1900, 29/02/2102

## **Boundary Value Analysis**

In boundary value analysis, we check for input values near the boundaries of valid and

invalid values that are more likely to cause errors, so testing these boundary values can help identify potential problems in the software.

We first identify the boundary values for day, month, and year

- Day: 1, 28, 29, 30, 31
- Month: 1, 2, 12
- Year: 1, 4, 100, 400 (for checking leap years).

We then find valid and invalid input ranges for day, month, and year

- **Day:** valid input range is from 1 to 31, invalid input range is from 32 to infinity.
- **Month:** valid input range is from 1 to 12, invalid input range is from 13 to infinity.
- **Year:** valid input range is from 1900 to 2015, invalid range is anything outside that range.

Using these to generate test cases:

**Test case 1:** Valid date (boundary value) - Day: 1, Month: 1, Year: 2010

**Test case 2:** Valid date (boundary value) - Day: 31, Month: 12, Year: 1990

**Test case 3:** Valid date (boundary value) - Day: 29, Month: 2, Year: 2000 (leap year)

**Test case 4:** Invalid date (boundary value) - Day: 32, Month: 1, Year: 2002

**Test case 5:** Invalid date (boundary value) - Day: 13, Month: 2, Year: 1910

**Test case 6:** Invalid date (boundary value) - Day: 30, Month: 2, Year: 2003

**Test case 7:** Invalid date (boundary value) - Day: 31,  
Month: 4, Year: 1935  
**Test case 8:** Valid date (within valid range) - Day: 11,  
Month: 8, Year: 2013  
**Test case 9:** Invalid date (year is outside the valid range) -  
Day: 10, Month: 7, Year: 2032  
**Test case 10:** Invalid date (day is outside the valid range) -  
Day: 32, Month: 7, Year: 2012  
**Test case 11:** Invalid date (month is outside the valid range)  
- Day: 15, Month: 13, Year: 2014

## **Programs**

### **Program 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.**

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

## Boundary Partitioning:

Tester Action and Input Data	Expected Output
Test with v as a non-existent value and an empty array a[]	-1
Test with v as a non-existent value and a non-empty array a[]	-1
Test with v as an existent value and an empty array a[]	-1
Test with v as an existent value and a non-empty array a[] where v exists	the index of v in a[]
Test with v as an existent value and a non-empty array a[] where v does not exist	-1

## Boundary Value Analysis:

Tester Action and Input Data	Expected Output
Test with v as a non-existent value and an empty array a[]	-1
Test with v as a non-existent value and a non-empty array a[]	-1
Test with v as an existent value and an array a[] of length 0	-1
Test with v as an existent value and an array a[] of length 1, where v exists	0
Test with v as an existent value and an	-1

array a[] of length 1, where v does not exist	
Test with v as an existent value and an array a[] of length greater than 1, where v exists at the beginning of the array	0
Test with v as an existent value and an array a[] of length greater than 1, where v exists at the end of the array	the last index where v is found

### Test Cases in Eclipse:

```

public class UnitTesting1
{
    @Test
    public void test1() {
        int arr[] = { 10, 20, 30, 40, 50 };

        Programs program = new Programs();
        int output = program.linearSearch(20, arr);

        System.out.println(output);
        assertEquals(1, output);
    }
    @Test
    public void test2() {
        int arr[] = { 50 };

        Programs program = new Programs();
        int output = program.linearSearch(50, arr);
    }
}

```

```

        System.out.println(output);
        assertEquals(0, output);
    }
    @Test
    public void test3() {
        int arr[] = { };

        Programs program = new Programs();
        int output = program.linearSearch(10, arr);

        System.out.println(output);
        assertEquals(-1, output);
    }
    @Test
    public void test4() {
        int arr[] = { 100 };

        Programs program = new Programs();
        int output = program.linearSearch(1, arr);

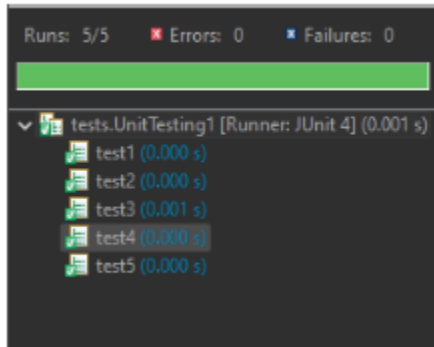
        System.out.println(output);
        assertEquals(-1, output);
    }
    @Test
    public void test5() {int arr[] = { 10, 20, 30 };
        Programs program = new Programs();
        int output = program.linearSearch(60, arr);

        System.out.println(output);
        assertEquals(-1, output);
    }

```



## Output:



## Program 2.

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

```
int countItem(int v, int a[])
{
    int count = 0;
    for (int i = 0; i < a.length; i++)
    {
        if (a[i] == v)
            count++;
    }
    return (count);
}
```

## Equivalence Partitioning:

Tester Action and Input Data	Expected Output
Test with v as a non-existent value and an empty array a[]	0
Test with v as a non-existent value and a non-empty array a[]	0
Test with v as an existent value and a non-empty array a[] where v exists multiple times	The number of occurrences of v in a[]
Test with v as an existent value and a non-empty array a[] where v exists only once	1

### Boundary Value Analysis:

Tester Action and Input Data	Expected Output
Test with v as a non-existent value and an empty array a[]	0
Test with v as a non-existent value and a non-empty array a[]	0
Test with v as an existent value and an array a[] of length 1, where v exists	1
Test with v as an existent value and an array a[] of length 1, where v does not exist	0
Test with v as an existent value and an	The number of

array a[] of length greater than 1, where v exists at the beginning of the array	occurrences of v in a[]
Test with v as an existent value and an array a[] of length greater than 1, where v exists at the end of the array	The number of occurrences of v in a[]
Test with v as an existent value and an array a[] of length greater than 1, where v exists in the middle of the array	The number of occurrences of v in a[]

### Test cases in Eclipse:

```

public class UnitTesting2 {
    @Test
    public void test1() {
        int arr[] = {};
        p2 program = new p2();
        int o = program.countItem(0, arr);

        assertEquals(o, 0);
    }
    @Test
    public void test2() {
        int arr[] = { 1, 1, 1 };
        p2 program = new p2();
        int o = program.countItem(1, arr);

        assertEquals(o, 3);
    }
    @Test

```

```

public void test3() {
    int arr[] = { 2 };
    p2 program = new p2();
    int o = program.countItem(2, arr);

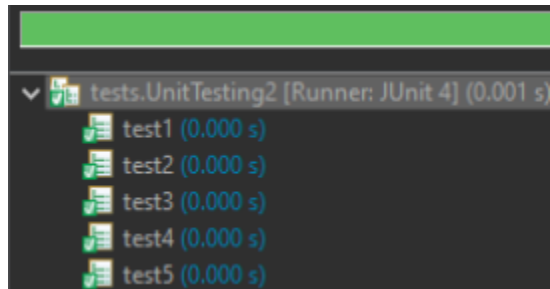
    assertEquals(o, 1);
}
@Test
public void test4() {
    int arr[] = { 3, 1 };
    p2 program = new p2();
    int o = program.countItem(2, arr);

    assertEquals(o, 0);
}
@Test
public void test5() {
    int arr[] = { 11, 2 };
    Programs program = new Programs();
    int o = program.countItem(11, arr);

    assertEquals(o, 1);
}
}
}

```

## Output:



### Program 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.**

```
int binarySearch(int v, int a[])
{
    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);
}
```

### Equivalence Partitioning:

Tester Actions and Input Data	Expected Output
v=20, a=[20, 40, 60, 80, 100 ]	0

v=100, a=[20, 40, 60, 80, 100 ]	4
v=40, a=[20, 40, 60, 80, 100 ]	2
v=1, a=[20, 40, 60, 80, 100 ]	-1
v=12, a=[20, 40, 60, 80, 100 ]	-1

### Boundary Value Analysis:

Tester Actions and Input Data	Expected Output
v=10, a=[10]	0
v=5, a=[]	-1
v=5, a=[5, 7, 9]	0 (smallest element in the array)
v=5, a=[1, 3, 5]	2 (largest element in the array)

### Test cases in Eclipse:

```

public class UnitTesting3 {
    @Test
    public void test1() {
        int input[] = { 20, 40, 60, 80, 100 };
        Programs program = new Programs();
        int output = program.binarySearch(20, input);

        assertEquals(0, output);
    }
    @Test

```

```
public void test2() {  
    int input[] = { 20, 40, 60, 80, 100 };  
    Programs program = new Programs();  
    int output = program.binarySearch(100, input);  
  
    assertEquals(4, output);}
```

@Test

```
public void test3() {  
    int input[] = { 20, 40, 60, 80, 100  
    };  
    Programs program = new Programs();  
    int output = program.binarySearch(40, input);  
  
    assertEquals(1, output);
```

```
}
```

@Test

```
public void test4() {  
    int input[] = { 20, 40, 60, 80, 100  
    };  
    Programs program = new Programs();  
    int output = program.binarySearch(1, input);  
  
    assertEquals(-1, output);
```

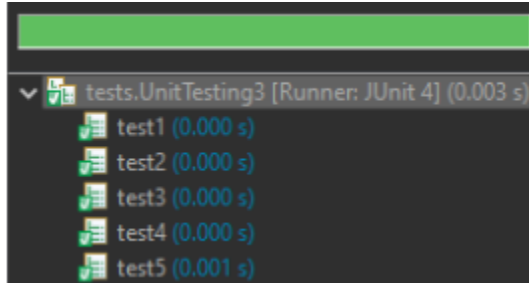
```
}
```

@Test

```
public void test5() {  
    int input[] = { 20, 40, 60, 80, 100  
    };  
    Programs program = new Programs();  
    int output = program.binarySearch(12, input);
```

```
        assertEquals(-1, output);
    }
}
```

## Output:



**Program 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).

```
final int EQUILATERAL = 0;
final int ISOSCELES = 1;
final int SCALENE = 2;
final int INVALID = 3;

int triangle(int a, int b, int c)
{
    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);
}
```

## Equivalence Partitioning:

<b>Tester Actions and Input Data</b> <b>Input Format: (a, b, c)</b>	<b>Expected Output</b>
(2, 2, 2)	0
(3, 3, 4)	1
(6, 5, 4)	2
(0, 0, 0)	3
(-1, -1, 5)	3
(2, 2, 1)	1
(0, 1, 1)	3
(1, 0, 1)	3
(1, 1, 0)	3

## Boundary Value Analysis:

<b>Tester Actions and Input Data</b>	<b>Expected Output</b>
(0, 0, 0)	3
$a + b = c$ or $b + c = a$ or $c + a = b$ (eg., (1, 2, 3))	1=3
(5, 5, 5)	0
$a = b \neq c = 3$	1

$a \neq b = c = 3$	1
$a = c \neq b = 3$	1
$\{a = b + c - 1\}$ or $\{b = a + c - 1\}$ or $\{c = a + b - 1\}$ (eg., (5, 4, 2))	2
$a = b = c =$ Integer.MAX_VALUE or $a = b = c =$ Integer.MIN_VALUE	3

## Program 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).**

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

```

    return true;
}

```

### Equivalence Partitioning:

Tester Actions and Input Data Input Format: ( <i>str1</i> , <i>str2</i> )	Expected Output
("good", "good morning")	true
("a", "abc")	true
("", "good morning")	false
("morning", "good morning")	false

### Boundary Value Analysis:

Tester Actions and Input Data Input Format: ( <i>str1</i> , <i>str2</i> )	Expected Output
("", "software")	false
("soft", "software")	true
("software", "soft")	false
("a", "ab")	true
("software", "softwareeee")	true
("abc", "abc")	true
("a", "b")	false

("a", "a")	true
("", "")	false

## Program 6.

**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 that states whether the triangle, if it can be formed, is scalene, isosceles, equilateral, or right angled.**

**Determine the following for the above program:**

- Identify the equivalence classes for the system
- Identify test cases to cover the identified equivalence classes.  
Also, explicitly mention which test case would cover which equivalence class. (Hint: you must need to be ensure that the identified set of test cases cover all identified equivalence classes)
- For the boundary condition  $A + B > C$  case (scalene triangle), identify test cases to verify the boundary.
- For the boundary condition  $A = C$  case (isosceles triangle), identify test cases to verify the
- boundary.

- f. For the boundary condition  $A = B = C$  case (equilateral triangle), identify test cases to verify the boundary.
- g. For the boundary condition  $A^2 + B^2 = C^2$  case (right-angle triangle), identify test cases to verify the boundary.
- h. For the non-triangle case, identify test cases to explore the boundary.
- i. For non-positive input, identify test points.

**Answer-**

**Equivalence Class:**

<b>Tester Action and Input Data(a,b,c)</b>	<b>Expected Output</b>
(-1,2,3)	Invalid input
(1,1,1)	Equilateral triangle
(2,2,3)	Isosceles triangle
(3,4,5)	Scalene right-angled triangle
(3,5,4)	Scalene right-angled triangle
(5,3,4)	Scalene right-angled triangle
(3,4,6)	Not a triangle

**Test Case:**

**Invalid inputs:**

$$a = 0, b = 0, c = 0, a + b = c, b + c = a, c + a = b$$

**Invalid inputs:**

$$a = -1, b = 1, c = 1, a + b = c$$

**Equilateral triangles:**

$$a = b = c = 1, a = b = c = 100$$

### **Isosceles triangles:**

$$a = b = 10, c = 5;$$

$$a = c = 10, b = 3;$$

$$b = c = 10,$$

$$a = 6$$

### **Scalene triangles:**

$$a = 4, b = 5, c = 6;$$

$$a = 10, b = 11, c = 13$$

### **Right angled triangle:**

$$a = 3, b = 4, c = 5;$$

$$a = 5, b = 12, c = 13$$

### **Non-triangle:**

$$a = 1, b = 2, c = 3$$

### **Non-positive input:**

$$a = -1, b = -2, c = -3$$

### **c) Boundary condition $A + B > C$ :**

$$a = \text{Integer.MAX\_VALUE}, b = \text{Integer.MAX\_VALUE}, c = 1$$

$$a = \text{Double.MAX\_VALUE}, b = \text{Double.MAX\_VALUE}, c = \text{Double.MAX\_VALUE}$$



**d) Boundary condition  $A = C$ :**

a = Integer.MAX\_VALUE,  
b = 2,  
c = Integer.MAX\_VALUE  
a = Double.MAX\_VALUE,  
b = 2.5,  
c = Double.MAX\_VALUE

**e) Boundary condition  $A = B = C$ :**

a = Integer.MAX\_VALUE, b = Integer.MAX\_VALUE, c =  
Integer.MAX\_VALUE a =  
Double.MAX\_VALUE, b = Double.MAX\_VALUE, c =  
Double.MAX\_VALUE

**f) Boundary condition  $A^2 + B^2 = C^2$ :**

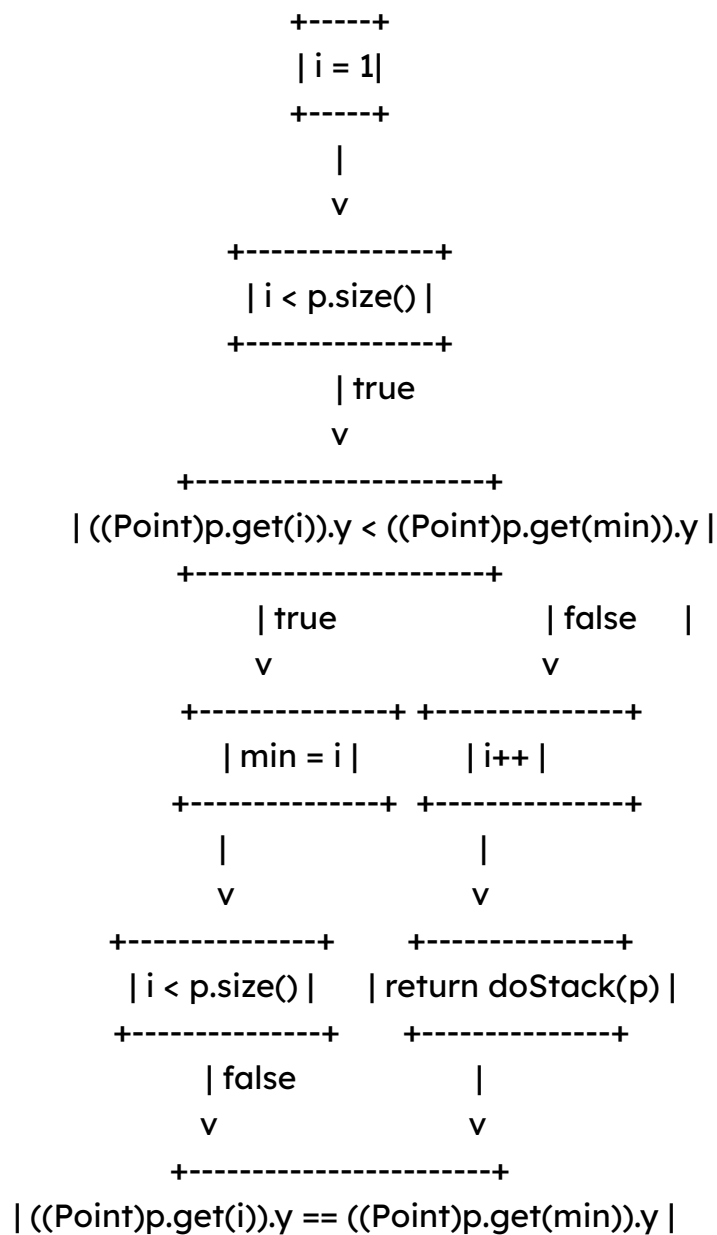
a = Integer.MAX\_VALUE,  
b = Integer.MAX\_VALUE,  
c = Integer.MAX\_VALUE  
a = Double.MAX\_VALUE,  
b = Double.MAX\_VALUE, c =  
Math.sqrt(Math.pow(Double.MAX\_VALUE, 2) +  
Math.pow(Double.MAX\_VALUE, 2))

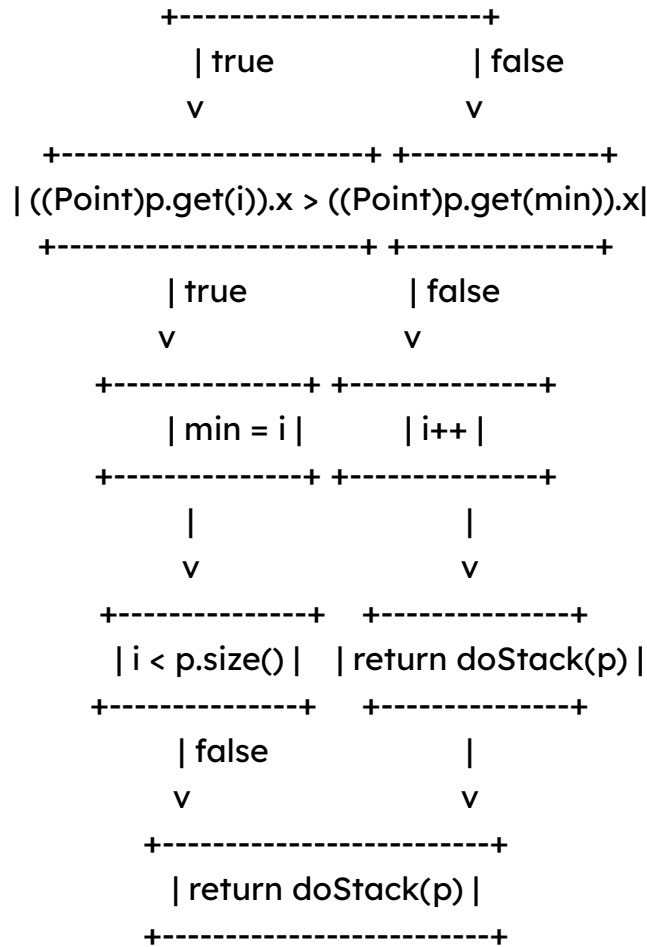
## g) Non-triangle:

a = 1, b = 2, c = 4 a = 2, b = 4, c = 8

## Section B

### 1. Control Flow Graph (CFG)





## Test sets for each coverage criterion:

### a. Statement Coverage:

Test 1:  $p = \{\text{new Point}(0, 0), \text{new Point}(1, 1)\}$

Test 2:  $p = \{\text{new Point}(0, 0), \text{new Point}(1, 0), \text{new Point}(2, 0)\}$

### b. Branch Coverage:

Test 1:  $p = \{\text{new Point}(0, 0), \text{new Point}(1, 1)\}$

Test 2:  $p = \{\text{new Point}(0, 0), \text{new Point}(1, 0), \text{new Point}(2, 0)\}$

Test 3:  $p = \{\text{new Point}(0, 0), \text{new Point}(1, 0), \text{new Point}(1, 1)\}$

c. Basic Condition Coverage:

Test 1:  $p = \{\text{new Point}(0, 0), \text{new Point}(1, 1)\}$

Test 2:  $p = \{\text{new Point}(0, 0), \text{new Point}(1, 0), \text{new Point}(2, 0)\}$

Test 3:  $p = \{\text{new Point}(0, 0), \text{new Point}(1, 0), \text{new Point}(1, 1)\}$

Test 4:  $p = \{\text{new Point}(0, 0), \text{new Point}(1, 0), \text{new Point}(0, 1)\}$

Test 5:  $p = \{\text{new Point}(0, 0), \text{new Point}(0, 1), \text{new Point}(1, 1)\}$