Software Engineering (IT314)

Lab 7

Name : Param Mistry

ID : 202001439

Section A

Question :

Consider a program for determining the previous date. Its input is a 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 dates or invalid dates. Design the equivalence class test cases.

Write a set of test cases (i.e., test suite) – a 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.

Answer.

There are three parameters, i.e., day, month and year, in this problem, so there will be separate equivalence classes for each parameter.

Equivalence Classes:

1. **Date**

    Valid: 1 ≤ day ≤ 31

    Invalid: day < 1, day > 31

2. **Month**

    Valid: 1 ≤ month ≤ 12

    Invalid: month < 1, month > 12

3. **Year**

    Valid: 1900 ≤ year ≤ 2015

    Invalid: year < 1900, 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 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: 01/01/2009, 15/03/1990, 31/12/2004
Partition 2: 00/01/2004, -10/03/2001, 32/12/2000
Partition 3: 01/00/2001, 15/13/2011, 31/15/2010
Partition 4: 01/01/0000, 15/03/10000, 31/12/99999
Partition 5: 30/02/2022, 31/04/2023, 28/02/2100
Partition 6: 29/02/2021, 29/02/1900, 29/02/2100

2. 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. 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 sample 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: 2010

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: 1990

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: 1930

Test case 7: Invalid date (boundary value) - Day: 31, Month: 4, Year: 1930

Test case 8: Valid date (within valid range) - Day: 15, Month: 6, Year: 2015

Test case 9: Invalid date (day is outside valid range) - Day: 32, Month: 6, Year: 2010

Test case 10: Invalid date (month is outside valid range) - Day: 15, Month: 13, Year: 2010

Test case 11: Invalid date (year is outside valid range) - Day: 15, Month: 6, Year: 2030

Programs

P1. 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.

```
package tests;
public class Programs{
    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 array a[] of length 1, where v does not exist	-1
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:

```java
public class UnitTesting1 {
    @Test
    public void test1() {
        int arr[] = { 1, 2, 3, 4, 5 };

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

        System.out.println(output);
        assertEquals(0, output);
    }

    @Test
    public void test2() {
        int arr[] = { };

        Programs program = new Programs();
        int output = program.linearSearch(5, arr);

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

    @Test
    public void test3() {
        int arr[] = { 5 };

        Programs program = new Programs();
        int output = program.linearSearch(5, arr);
```

```java
        System.out.println(output);
        assertEquals(0, output);
    }

    @Test
    public void test4() {
        int arr[] = { 10 };

        Programs program = new Programs();
        int output = program.linearSearch(5, arr);

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

    @Test
    public void test5() {
        int arr[] = { 1, 2, 3, 4, 5 };

        Programs program = new Programs();
        int output = program.linearSearch(6, arr);

        System.out.println(output);
        assertEquals(-1, output);
    }
}
```
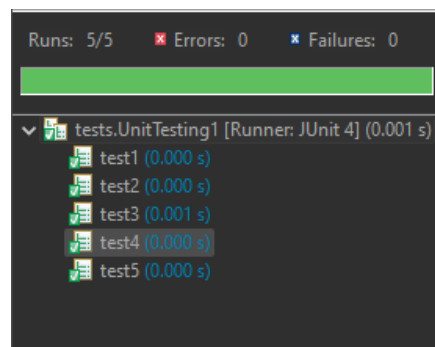
Output of Test Cases:

P2. 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 Outcome
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 Outcome
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 array a[] of length greater than 1, where v exists at the beginning 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 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:

```java
public class UnitTesting2 {
    @Test
    public void test1() {
        int input[] = {   };
        Programs program = new Programs();
        int output = program.countItem(0, input);

        assertEquals(output, 0);
    }

    @Test
    public void test2() {
        int input[] = { 1, 1, 2, 3, 1 };
        Programs program = new Programs();
        int output = program.countItem(10, input);

        assertEquals(output, 0);
    }

    @Test
    public void test3() {
        int input[] = { 1, 1, 2, 3, 1 };
        Programs program = new Programs();
        int output = program.countItem(1, input);

        assertEquals(output, 3);
    }

    @Test
    public void test4() {
        int input[] = { 1, 1, 2, 3, 1 };
        Programs program = new Programs();
        int output = program.countItem(2, input);

        assertEquals(output, 1);
```

```java
        }

        @Test
        public void test5() {
                int input[] = { 1 };
                Programs program = new Programs();
                int output = program.countItem(1, input);

                assertEquals(output, 1);
        }

        @Test
        public void test6() {
                int input[] = { 1 };
                Programs program = new Programs();
                int output = program.countItem(2, input);

                assertEquals(output, 0);
        }

        @Test
        public void test7() {
                int input[] = { 1, 1, 2, 3, 1, 1, 1 };
                Programs program = new Programs();
                int output = program.countItem(1, input);

                assertEquals(output, 5);
        }
}
}
```
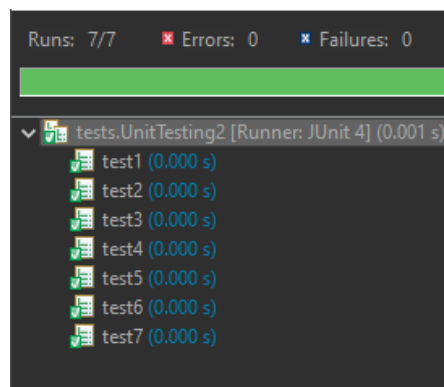
Output:

P3. 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=6, a=[2, 4, 6, 8, 10]	2
v=2, a=[2, 4, 6, 8, 10]	0
v=10, a=[2, 4, 6, 8, 10]	4
v=1, a=[2, 4, 6, 8, 10]	-1
v=12, a=[2, 4, 6, 8, 10]	-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:

```java
public class UnitTesting3 {
    @Test
    public void test1() {
        int input[] = { 2, 4, 6, 8, 10 };
        Programs program = new Programs();
        int output = program.binarySearch(6, input);

        assertEquals(2, output);
    }

    @Test
    public void test2() {
        int input[] = { 2, 4, 6, 8, 10 };
        Programs program = new Programs();
        int output = program.binarySearch(2, input);

        assertEquals(0, output);
    }

    @Test
    public void test3() {
        int input[] = { 2, 4, 6, 8, 10 };
        Programs program = new Programs();
        int output = program.binarySearch(10, input);

        assertEquals(4, output);
    }

    @Test
    public void test4() {
        int input[] = { 2, 4, 6, 8, 10 };
        Programs program = new Programs();
        int output = program.binarySearch(1, input);

        assertEquals(-1, output);
    }

    @Test
    public void test5() {
        int input[] = { 2, 4, 6, 8, 10 };
        Programs program = new Programs();
```

```java
        int output = program.binarySearch(12, input);

        assertEquals(-1, output);
    }

    @Test
    public void test6() {
        int input[] = { 10 };
        Programs program = new Programs();
        int output = program.binarySearch(10, input);

        assertEquals(0, output);
    }

    @Test
    public void test7() {
        int input[] = {   };
        Programs program = new Programs();
        int output = program.binarySearch(5, input);

        assertEquals(-1, output);
    }

    @Test
    public void test8() {
        int input[] = { 5, 7, 9 };
        Programs program = new Programs();
        int output = program.binarySearch(5, input);

        assertEquals(0, output);
    }

    @Test
    public void test9() {
        int input[] = { 1, 3, 5 };
        Programs program = new Programs();
        int output = program.binarySearch(5, input);

        assertEquals(2, output);
    }
}
```
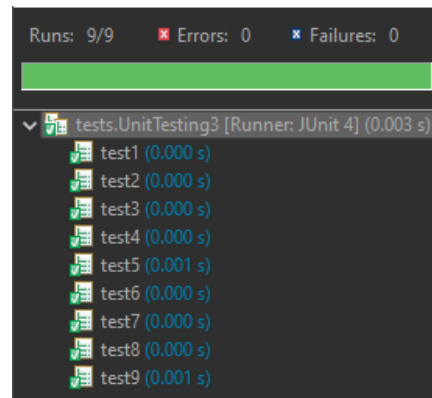
Output:

Here, we can see that all of our test cases are getting passed successfully.

P4. 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:

Tester Actions and Input Data Input Format: (a, b, c)	Expected Output
(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:

Tester Actions and Input Data	Expected Output
(0, 0, 0)	3
a + b = c or b + c = a or c + a = b (eg., (1, 2, 3))	3
(5, 5, 5)	0
a = b != c = 3	1
a != b = c = 3	1
a = c != 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

Test Cases in Eclipse:

```java
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);
    public class UnitTesting4 {
    @Test
    public void test1() {
        int a = 2, b = 2, c = 2;
        Programs program = new Programs();
        int output = program.triangle(a, b, c);

        assertEquals(output, 0);
    }
```

```java
    @Test
    public void test2() {
        int a = 3, b = 3, c = 4;
        Programs program = new Programs();
        int output = program.triangle(a, b, c);

        assertEquals(output, 1);
    }
    @Test
    public void test3() {
        int a = 6, b = 5, c = 4;
        Programs program = new Programs();
        int output = program.triangle(a, b, c);

        assertEquals(output, 2);
    }
    @Test
    public void test4() {
        int a = 0, b = 0, c = 0;
        Programs program = new Programs();
        int output = program.triangle(a, b, c);

        assertEquals(output, 3);
    }
    @Test
    public void test5() {
        int a =-1, b =-1, c = 5;
        Programs program = new Programs();
        int output = program.triangle(a, b, c);

        assertEquals(output, 3);
    }
    @Test
    public void test6() {
        int a = 2, b = 2, c = 1;
        Programs program = new Programs();
        int output = program.triangle(a, b, c);

        assertEquals(output, 1);
    }
    @Test
    public void test7() {
        int a = 0, b = 1, c = 1;
        Programs program = new Programs();
        int output = program.triangle(a, b, c);

        assertEquals(output, 3);
```

```java
    }
    @Test
    public void test8() {
        int a = 1, b = 0, c = 1;
        Programs program = new Programs();
        int output = program.triangle(a, b, c);

        assertEquals(output, 3);
    }
    @Test
    public void test9() {
        int a = 1, b = 1, c = 0;
        Programs program = new Programs();
        int output = program.triangle(a, b, c);

        assertEquals(output, 3);
    }
    @Test
    public void test10() {
        int a = 1, b = 2, c = 3;
        Programs program = new Programs();
        int output = program.triangle(a, b, c);

        assertEquals(output, 3);
    }
    @Test
    public void test11() {
        int a = 3, b = 1, c = 3;
        Programs program = new Programs();
        int output = program.triangle(a, b, c);

        assertEquals(output, 1);
    }
    @Test
    public void test12() {
        int a = 5, b = 4, c = 2;
        Programs program = new Programs();
        int output = program.triangle(a, b, c);

        assertEquals(output, 2);
    }
    @Test
    public void test13() {
        int a = Integer.MAX_VALUE, b = Integer.MAX_VALUE, c =
Integer.MAX_VALUE;
        Programs program = new Programs();
        int output = program.triangle(a, b, c);
```
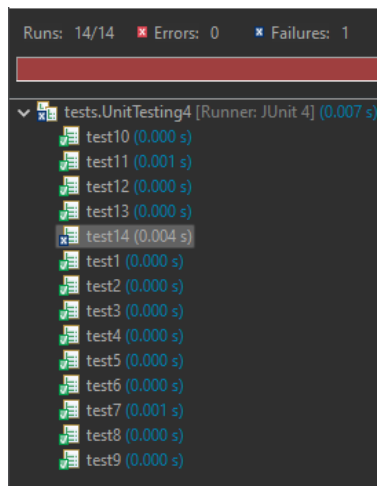
```
            assertEquals(output, 3);
      }
      @Test
      public void test14() {
            int a = Integer.MIN_VALUE, b = Integer.MIN_VALUE, c =
Integer.MIN_VALUE;
            Programs program = new Programs();
            int output = program.triangle(a, b, c);

            assertEquals(output, 3);
      }
}


      return(SCALENE);
}
```

Output:

Here, we can see that the test case (Integer.MIN_VALUE, Integer.MIN_VALUE, Integer.MIN_VALUE) fails. This is because Integer.MIN_VALUE = -2147483648, which when added to itself overflows and becomes 0, so our check of (a >= b + c || b >= a+c || c >= b+c) becomes false and we jump to (a=b=c) condition and the function returns 0 which means that the triangle is EQUILATERAL.

P5. 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).

```java
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: *(str1, str2)*	Expected Output
("good", "good morning")	true
("a", "abc")	true
("", "good morning")	true
("morning", "good morning")	false

Boundary Value Analysis:

Tester Actions and Input Data Input Format: *(str1, str2)*	Expected Output
("", "software")	true
("soft", "software")	true
("software", "soft")	false

("a", "ab")	true
("software", "softwareeee")	true
("abc", "abc")	true
("a", "b")	false
("a", "a")	true
("", "")	true

Test cases in Eclipse:

```java
public class UnitTesting6 {
    @Test
    public void test1() {
        String str1 = "good", str2 = "good morning";

        Programs program = new Programs();
        boolean output = program.prefix(str1, str2);

        assertEquals(output, true);
    }
    @Test
    public void test2() {
        String str1 = "a", str2 = "abc";

        Programs program = new Programs();
        boolean output = program.prefix(str1, str2);

        assertEquals(output, true);
    }
    @Test
    public void test3() {
        String str1 = "", str2 = "good morning";

        Programs program = new Programs();
        boolean output = program.prefix(str1, str2);

        assertEquals(output, true);
    }
    @Test
    public void test4() {
        String str1 = "morning", str2 = "good morning";

        Programs program = new Programs();
```

```java
        boolean output = program.prefix(str1, str2);

        assertEquals(output, false);
    }
    @Test
    public void test5() {
        String str1 = "soft", str2 = "software";

        Programs program = new Programs();
        boolean output = program.prefix(str1, str2);

        assertEquals(output, true);
    }
    @Test
    public void test6() {
        String str1 = "software", str2 = "soft";

        Programs program = new Programs();
        boolean output = program.prefix(str1, str2);

        assertEquals(output, false);
    }
    @Test
    public void test7() {
        String str1 = "a", str2 = "ab";

        Programs program = new Programs();
        boolean output = program.prefix(str1, str2);

        assertEquals(output, true);
    }
    @Test
    public void test8() {
        String str1 = "software", str2 = "software";

        Programs program = new Programs();
        boolean output = program.prefix(str1, str2);

        assertEquals(output, true);
    }
    @Test
    public void test9() {
        String str1 = "abc", str2 = "abc";

        Programs program = new Programs();
        boolean output = program.prefix(str1, str2);
```

```java
            assertEquals(output, true);
    }
    @Test
    public void test10() {
            String str1 = "a", str2 = "b";

            Programs program = new Programs();
            boolean output = program.prefix(str1, str2);

            assertEquals(output, false);
    }
    @Test
    public void test11() {
            String str1 = "a", str2 = "a";

            Programs program = new Programs();
            boolean output = program.prefix(str1, str2);

            assertEquals(output, true);
    }
    @Test
    public void test12() {
            String str1 = "", str2 = "";

            Programs program = new Programs();
            boolean output = program.prefix(str1, str2);

            assertEquals(output, true);
    }
}
```
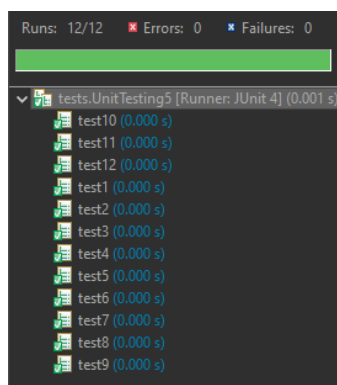
Output:

Here, we can see that all the tests are getting passed.

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 that states whether the triangle, if it can be formed, is scalene, isosceles, equilateral, or right angled.
Determine the following for the above program:

a. Identify the equivalence classes for the system

Following are the equivalence classes for different types of triangles:

Equivalent Classes	Expected Output
E1: $a + b \leq c$	Invalid
E2: $a + c \leq b$	Invalid
E3: $b + c \leq a$	Invalid
E4: $a = b, b = c, c = a$	Equilateral
E5: $a = b, a \,!= c$	Isosceles
E6: $a = c, a \,!= b$	Isosceles
E7: $b = c, b \,!= a$	Isosceles
E8: $a \,!= b, b \,!= c, c \,!= a$	Scalene
E9: $a^2 + b^2 = c^2$	Right angled triangle
E10: $b^2 + c^2 = a^2$	Right angled triangle
E11: $a^2 + c^2 = b^2$	Right angled triangle

b. Identify test cases to cover the identified equivalence classes. Also, explicitly mention which test case would cover which equivalence class.

Test cases Input format: (a, b, c)	Output	Equivalence Class Covered
(2.5, 4.6, 6.1)	Invalid	E1
(-2.6, 5, 6)	Invalid	E2
(7.1, 6.1, 1)	Invalid	E3
(3.1, 3.1, 3.1)	Equilateral	E4
(3.5, 3.5, 5)	Isosceles	E5
(6, 4, 6)	Isosceles	E6
(8, 5, 5)	Isosceles	E7
(6, 7, 8)	Scalene	E8
(3, 4, 5)	Right angled triangle	E9
(0.13, 0.12, 0.05)	Right angled triangle	E10
(7, 25, 23)	Right angled triangle	E11

c. For the boundary condition A + B > C case (scalene triangle), identify test cases to verify the boundary.

Test cases to verify the boundary condition:
1. (5, 5, 9) (a + b = c)
2. (5.2, 5.2, 10.8) (a + b is greater than c)
3. (5.5, 5, 9.6) (a + b is just less than c)

d. For the boundary condition A = C case (isosceles triangle), identify test cases to verify the boundary.

Test cases to verify the boundary condition:
1. (2, 2, 2) (a = c)
2. (4.4, 4.4, 4.6) (a is just less than c)
3. (5.5, 5, 5.3) (a is just greater than c)

e. For the boundary condition A = B = C case (equilateral triangle), identify test cases to verify the boundary.

Test cases to verify the boundary condition:

1. (3, 3, 3) (a = b = c)
2. (10, 10, 9) (a = b but a != c)
3. (10, 11, 10) (a = c but a != b)

f. For the boundary condition $a^2 + b^2 = c^2$ case (right-angle triangle), identify test cases to verify the boundary.

Test cases to verify the boundary condition:

1. (3, 4, 5) ($a^2 + b^2 = c^2$)
2. (0.12, 0.5, 0.14) ($a^2 + b^2$ is less than c^2)
3. (7, 23, 24) ($a^2 + b^2$ is just greater than c^2)

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

Test cases to verify the boundary condition:

1. (1, 2, 3)
2. (5, 5, 10)
3. (0, 0, 0)

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

Test points for non-positive input:

1. (-4.0, 4.2, 4.5)
2. (5, -4.2, -3.2)
3. (4, 5, -10)

Section B

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 and so the focus is on creating test sets that satisfy some particular coverage criterion.

```
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.
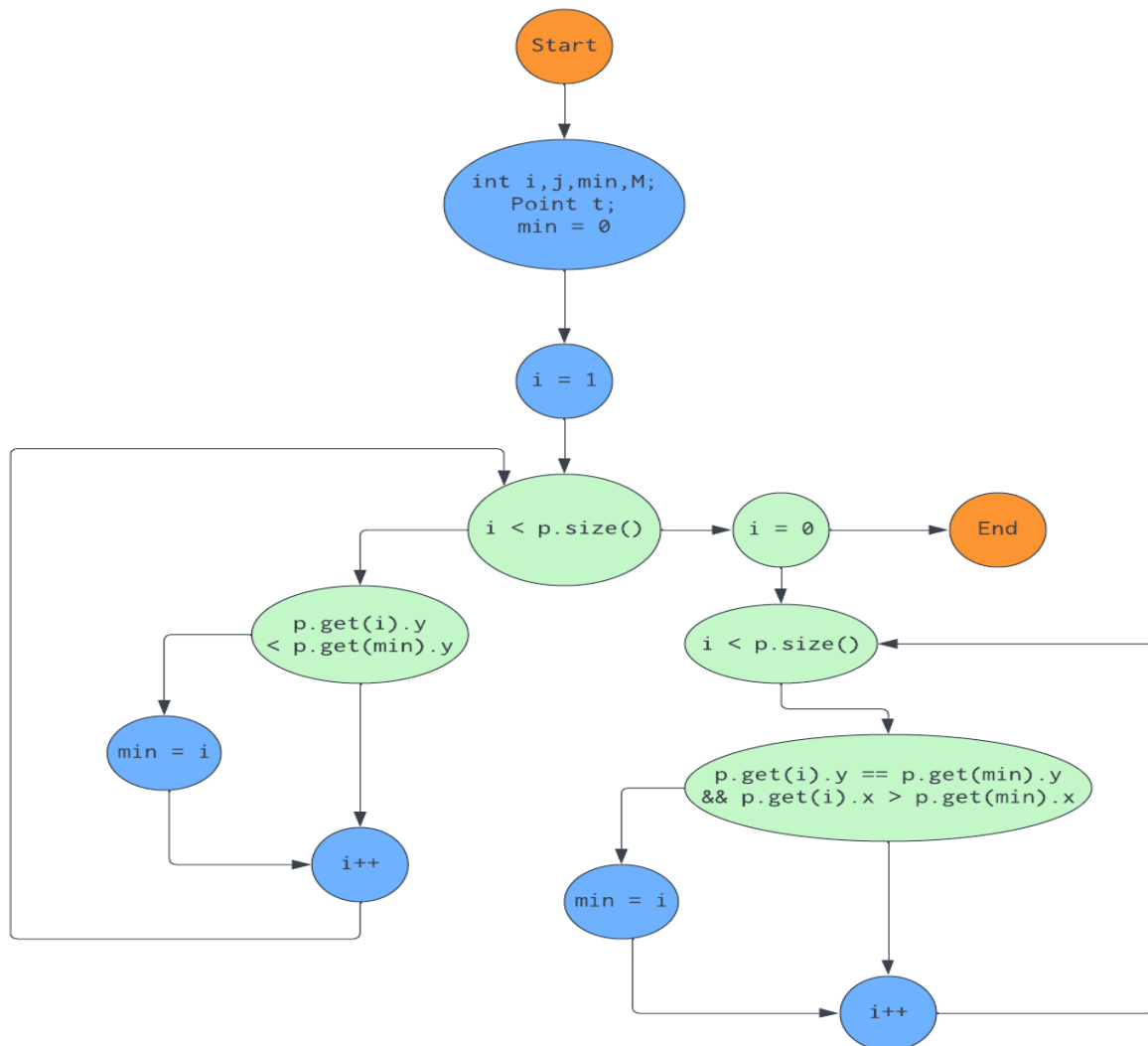1. Convert the Java code comprising the beginning of the doGraham method into a control
flow graph (CFG).
2. Construct test sets for your flow graph that are adequate for the following criteria:
        a. Statement Coverage.
        b. Branch Coverage.
        c. Basic Condition Coverage.

Answer :
Control Flow Graph (CFG):

Consider the following lines for denoting coverage:

```
int i,j,min,M;
Point t;
min=0;
for(i = 1;i < p.size();++i){
    if(((Point)P.get(i)).y < ((Point)P.get(min)).y)
        min=i;
}
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;
}
```

The following are the test cases and their corresponding coverages of statements:

Test cases:

1. p = [(x = 2, y = 2), (x = 2, y = 3), (x = 1, y = 3), (x = 1, y = 4)]
    Statement Covered: { 1, 2, 3, 4, 5, 7, 8 }
    Branches Covered: { 5, 8 }
    Basic Conditions Covered: { 5 - false, 8 - false }

2. p = [(x = 2, y = 3), (x = 3, y = 4), (x = 1, y = 2), (x = 5, y = 6)]
    Statements covered = { 1, 2, 3, 4, 5, 6, 7}
    Branches covered = { 5, 8 }
    Basic conditions covered = {5-false,true, 8-false}

3. p = [(x = 1, y = 5), (x = 2, y = 7), (x = 3 , y = 5), (x = 4, y = 5), (x = 5, y = 6)]
    Statements covered = { 1, 2, 3, 4, 5, 6, 7, 8, 9}
    Branches covered = { 5, 8 }
    Basic conditions covered = { 5 - false, true, 8 - false, true }

4. p = [(x = 1, y = 2)]
    Statements covered = { 1, 2, 3, 7, 8 }
    Branches covered = { 8 }

Basic conditions covered = { }

5. p=[ ]
   Statements covered = { 1, 2, 3 }
   Branches covered = {}
   Basic conditions covered = {}

Thus, the above 5 test cases are covering all statements, branches and conditions.