

IT 314 Software Engineering

Lab.7

Khush Patel-202001127



Date: Apr 13, 2023

Section A

Q1: 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 the previous date or invalid date. Design the equivalence class test cases?

Ans:

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: $\text{year} < 1900, \text{year} > 2015$

Some Test Cases:

1. Valid test cases:

- a. (1, 1, 1900) – the minimum valid date
- b. (8, 12, 2014) – a valid date
- c. (31, 12, 2015) – the maximum valid date

2. Invalid test cases:

- a. (0, 7, 1999) – day is less than 1
- b. (32, 1, 2008) – day is greater than 31
- c. (29, 2, 1900) – the year is not a leap year, but the day is 29
- d. (31, 4, 2016) – the year is greater than 2015
- e. (15, 13, 1998) – month is greater than 12
- f. (-10, -5, 2022) – all parameters are invalid

These test cases represent the equivalence classes and should cover all possible scenarios.

Programs:

```
package UnitTesting;
```

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

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

```
public static 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;
}

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

public static 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;
}

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;
}

}

```

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.

```

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

```

Code:

```
package Tests;

import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.Test;

class TestForProg_1{

    @Test
    public void test1() {
        int[] arr1 = { 1, 2, 3, 4, 5 };

        Procedures obj1 = new procedures();
        int output_1 = obj1.linearSearch(1, arr1);
        int output_2 = obj1.linearSearch(3, arr1);

        assertEquals(0, output_1);
        assertEquals(2, output_2);
    }

    @Test
    public void test2() {
        int a[] = {1, 2, 3, 4};
        procedures obj1 = new procedures();
        int output_f = obj1.linearSearch(3, a);

        assertEquals(2, output_f);
    }

    @Test
    public void test3() {

        int[] arr2 = { 5, 4, 3, 2, 1 };

        procedures obj1 = new procedures();
        int output_1 = obj1.linearSearch(5, arr2);
        int output_2 = obj1.linearSearch(6, arr2);

        assertEquals(0, output_1);
        assertEquals(-1, output_2);
    }
}
```

```

@Test
public void test4() {

    int[] arr3 = { -1, 0, 1 };

    Functions obj1 = new Functions();
    int output_1 = obj1.linearSearch(1, arr3);
    int output_2 = obj1.linearSearch(-2, arr3);

    assertEquals(2, output_1);
    assertEquals(-1, output_2);
}

@Test
public void test5() {

    int[] arr = null;

    Functions obj1 = new Functions();
    int output_2 = obj1.linearSearch(1, arr);

    assertEquals(-1, output_2);
}
}

```

Equivalence Partitioning:

Tester Action and Input Data	Expected Outcome
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 Outcome
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 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 index where v is found

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);  
}
```

Code:

```
package Tests;

import static org.junit.Assert.*;

import org.junit.Test;

public class TestForProg 2 {

    @Test
    public void test1() {
        int[] arr1 = {1, 2, 3, 4, 5};

        procedures obj1 = new procedures();
        int output_f = obj1.countItem(3, arr1);

        assertEquals(0, output_f);
    }

    @Test
    public void test2() {
        int[] arr2 = {5, 5, 5, 5, 5};

        procedures obj1 = new procedures();
        int output_f = obj1.countItem(5, arr2);

        assertEquals(5, output_f);
    }

    @Test
    public void test3() {
        int[] arr3 = {0, 0, 0, 0, 0};

        procedures obj1 = new procedures();
        int output_f = obj1.countItem(0, arr3);
    }
}
```



```
        assertEquals(5, output_f);
    }

    @Test
    public void test4() {

        int[] arr4 = {-1, -2, -3, -4, -5};

        procedures obj1 = new procedures();
        int output_f = obj1.countItem(7, arr4);

        assertEquals(0, output_f);
    }
```

```
    @Test
    public void test5() {

        int[] arr5 = {2, 4, 6, 8, 10};

        procedures obj1 = new procedures();
        int output_f = obj1.countItem(3, arr5);

        assertEquals(0, output_f);
    }
```

```
}
```

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 an 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 0	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[]

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. Assumption: the elements in the array `a` are sorted in non-decreasing order.

```
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);
}
```

Code:

```
package Tests;

import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.Test;

class TestForProg 3 {

    @Test
    public void test1() {
        int[] array = {1, 2, 3, 4, 5, 6, 7, 8, 9};
        int element = 7;
        int expectedIndex = 5;

        procedures obj1 = new procedures();
        int output_f = obj1.binarySearch(element, array);

        assertEquals(expectedIndex, output_f);
    }

    @Test
    public void test2() {
        int[] array = {1, 2, 3, 4, 5, 6, 7, 8, 9};
        int element = 10;
        int expectedIndex = -1;

        procedures obj1 = new procedures();
        int output_f = obj1.binarySearch(element, array);

        assertEquals(expectedIndex, output_f);
    }

    @Test
    public void test3()
    {int[] array = {};
        int element = 1;
        int expectedIndex = -1;

        procedures obj1 = new procedures();
        int output_f = obj1.binarySearch(element, array);
```

```

        assertEquals(expectedIndex, output_f);
    }

    @Test
    public void test4()
    { int[] array =
      {5};int element =
      5;
      int expectedIndex = 0;

      Functions obj1 = new Functions();
      int output_f = obj1.binarySearch(element, array);

      assertEquals(expectedIndex, output_f);
    }

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

        Functions obj1 = new Functions();
        int output_f = obj1.binarySearch(element, array);

        assertEquals(expectedIndex, output_f);
    }
}

```

Equivalence Partitioning:

Tester Action and Input Data	Expected Outcome
v=5, a=[1, 3, 5, 7, 9]	2
v=1, a=[1, 3, 5, 7, 9]	0
v=9, a=[1, 3, 5, 7, 9]	4
v=4, a=[1, 3, 5, 7, 9]	-1
v=11, a=[1, 3, 5, 7, 9]	-1

Boundary Value Analysis:

Tester Action and Input Data	Expected Outcome
v=1, a=[1]	0
v=9, a=[9]	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)

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);
}
```

Code:

```
package Tests;
import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.Test;

class TestForProg 4 {

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

    @Test
    public void Test1() {
        int a = 5;
        int b = 5;
        int c = 5;
        int expectedType = EQUILATERAL;

        procedures obj1 = new procedures();
        int output_f = obj1.triangle(a, b, c);

        assertEquals(expectedType, output_f);
    }

    @Test
    public void Test2() {
        int a = 7;
        int b = 7;
        int c = 10;
        int expectedType = ISOSCELES;

        procedures obj1 = new procedures();
        int output_f = obj1.triangle(a, b, c);

        assertEquals(expectedType, output_f);
    }

    @Test
    public void Test3() {
        int a = 3;
```

```

    int b = 4;
    int c = 5;
    int expectedType = SCALENE;

    procedures obj1 = new procedures();
    int output_f = obj1.triangle(a, b, c);

    assertEquals(expectedType, output_f);
}

@Test
public void Test4() {
    int a = 2;
    int b = 2;
    int c = 5;
    int expectedType = INVALID;

    procedures obj1 = new procedures();
    int output_f = obj1.triangle(a, b, c);

    assertEquals(expectedType, output_f);
}

@Test
public void Test5() {
    int a = 0;
    int b = 0;
    int c = 0;
    int expectedType = INVALID;

    procedures obj1 = new procedures();
    int output_f = obj1.triangle(a, b, c);

    assertEquals(expectedType, output_f);
}

@Test
public void Test6() {
    int a = -1;
    int b = 2;
    int c = 5;
    int expectedType = INVALID;

```



```

    Functions obj1 = new Functions();
    int output_f = obj1.triangle(a, b, c);

    assertEquals(expectedType, output_f);
}

@Test
public void Test7()
{
    int a = 1000000;
    int b = 1000000;
    int c = 1000000;
    int expectedType = EQUILATERAL;

    Functions obj1 = new Functions();
    int output_f = obj1.triangle(a, b, c);

    assertEquals(expectedType, output_f);
}
}

```

Equivalence Partitioning:

Tester Action and Input Data	Expected Outcome
Valid input: a=3, b=3, c=3	EQUILATERAL
Valid input: a=4, b=4, c=5	ISOSCELES
Valid input: a=5, b=4, c=3	SCALENE
Invalid input: a=0, b=0, c=0	INVALID
Invalid input: a=-1, b=2, c=3	INVALID
Valid input: a=1, b=1, c=1	EQUILATERAL
Valid input: a=2, b=2, c=1	ISOSCELES
Valid input: a=3, b=4, c=5	SCALENE
Invalid input: a=0, b=1, c=1	INVALID
Invalid input: a=1, b=0, c=1	INVALID
Invalid input: a=1, b=1, c=0	INVALID

Boundary Value Analysis:

Tester Action and Input Data	Expected Outcome
Invalid inputs: $a = 0, b = 0, c = 0$	INVALID
Invalid inputs: $a + b = c$ or $b + c = a$ or $c + a = b$ ($a=3, b=4, c=8$)	INVALID
Equilateral triangles: $a = b = c = 1$	EQUILATERAL
Equilateral triangles: $a = b = c = 100$	EQUILATERAL
Isosceles triangles: $a = b \neq c = 10$	ISOSCELES
Isosceles triangles: $a \neq b = c = 10$	ISOSCELES
Isosceles triangles: $a = c \neq b = 10$	ISOSCELES
Scalene triangles: $a = b + c - 1$	SCALENE
Scalene triangles: $b = a + c - 1$	SCALENE
Scalene triangles: $c = a + b - 1$	SCALENE
Maximum values: $a, b, c = \text{Integer.MAX_VALUE}$	INVALID
Minimum values: $a, b, c = \text{Integer.MIN_VALUE}$	INVALID

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).

```
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;
}
```

Code:

```
package Tests;

import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.Test;

class TestForProg 5 {

    @Test
    public void test1()
    {
        String s1 =
            "hello";

        String s2 = "hello world";
        boolean expectedResult = true;

        procedures obj1 = new procedures();
        boolean output_f = obj1.prefix(s1, s2);

        assertEquals(expectedResult, output_f);
    }

    @Test
    public void test2()
    {
        String s1 = "";
        String s2 = "hello";
        boolean expectedResult = true;

        procedures obj1 = new procedures();
        boolean output_f = obj1.prefix(s1, s2);

        assertEquals(expectedResult, output_f);
    }

    @Test
    public void test3()
    {
        String s1="hello";

        String s2 = "hello";
        boolean expectedResult = true;

        procedures obj1 = new procedures();
        boolean output_f = obj1.prefix(s1, s2);
    }
}
```

```

    assertEquals(expectedResult, output_f);
}

@Test
public void test4()
{
    String s1 =
    "hello";String s2 =
    "world";
    boolean expectedResult = false;

    Functions obj1 = new Functions();
    boolean output_f = obj1.prefix(s1, s2);

    assertEquals(expectedResult, output_f);
}

@Test
public void test5()
{
    String s1 = " ";
    String s2 = "hello";
    boolean expectedResult = false;

    Functions obj1 = new Functions();
    boolean output_f = obj1.prefix(s1, s2);

    assertEquals(expectedResult, output_f);
}
}

```

Equivalence Partitioning:

Tester Action and Input Data	Expected Outcome
Valid Inputs: s1 = "hello", s2 = "hello world"	true
Valid Inputs: s1 = "a", s2 = "abc"	true
Invalid Inputs: s1 = "", s2 = "hello world"	false
Invalid Inputs: s1 = "world", s2 = "hello world"	false

Boundary Value Analysis:

Tester Action and Input Data	Expected Outcome
s1 = "", s2 = "abc"	False
s1 = "ab", s2 = "abc"	True
s1 = "abc", s2 = "ab"	False
s1 = "a", s2 = "ab"	True
s1 = "hello", s2 = "hellooo"	True
s1 = "abc", s2 = "abc"	True
s1 = "a", s2 = "b"	False
s1 = "a", s2 = "a"	True

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:

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

Ans:

Equivalence Class:

Tester Action and Input Data	Expected Outcome
$a = -1, b = 2, c = 3$	Invalid input
$a = 1, b = 1, c = 1$	Equilateral triangle
$a = 2, b = 2, c = 3$	Isosceles triangle
$a = 3, b = 4, c = 5$	Scalene right-angled triangle
$a = 3, b = 5, c = 4$	Scalene right-angled triangle
$a = 5, b = 3, c = 4$	Scalene right-angled triangle
$a = 3, b = 4, c = 6$	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 = Integer.MAX_VALUE, b = Integer.MAX_VALUE, c = 1

a = Double.MAX_VALUE, b = Double.MAX_VALUE, c = 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

h) Non-positive input:

a = -1, b = -2, c = -3 a = 0, b = 1, c = 2

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 `i`th 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;
        }
    }

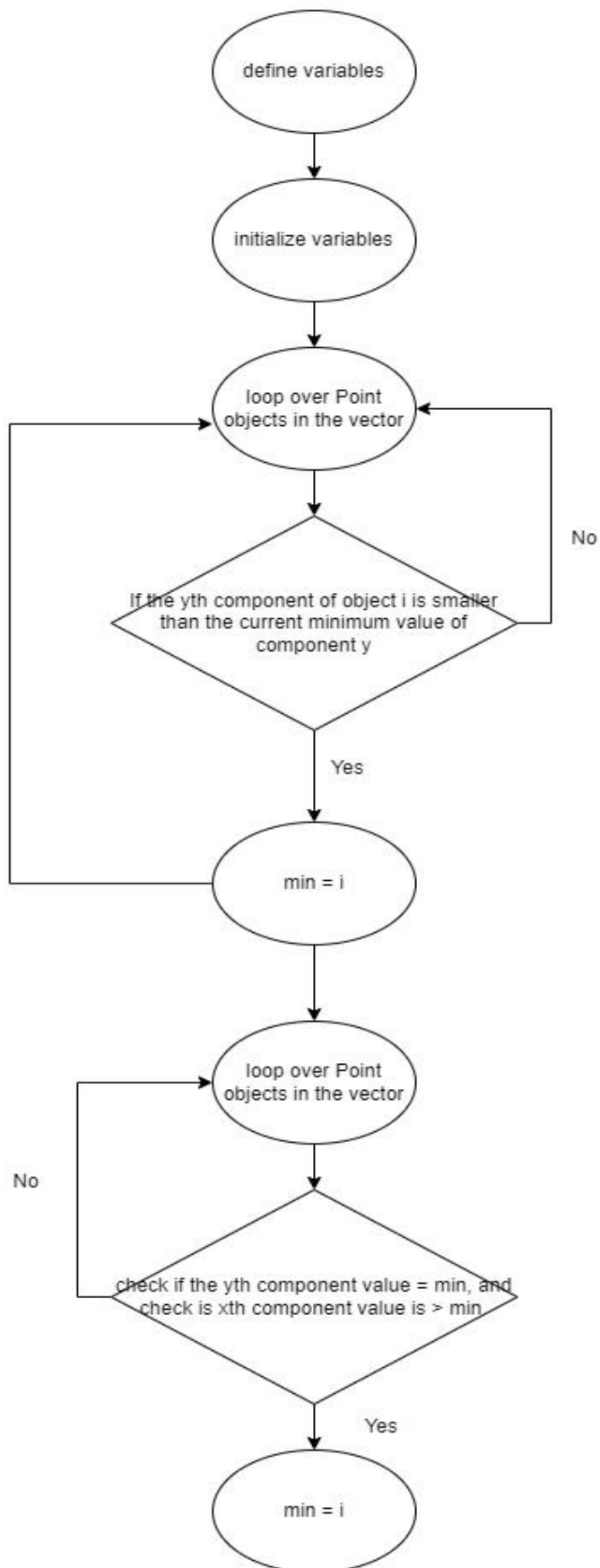
    // continue along the values with same y component
    for(i=0; i < p.size(); ++i) {
        if( ((Point) p.get(i)).y ==
            ((Point) p.get(min)).y ) &&
            (((Point) p.get(i)).x >
              ((Point) p.get(min)).x ))
        {
            min = i;
        }
    }
}
```

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.

Ans:

Control Flow Graph (CFG):



Test sets for each coverage criterion:

a. Statement Coverage:

- Test 1: p = empty vector
- Test 2: p = vector with one point
- Test 3: p = vector with two points with the same y component
- Test 4: p = vector with two points with different y components
- Test 5: p = vector with three or more points with different y components
- Test 6: p = vector with three or more points with the same y component

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

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

b. Branch Coverage:

- Test 1: p = empty vector
- Test 2: p = vector with one point
- Test 3: p = vector with two points with the same y component
- Test 4: p = vector with two points with different y components
- Test 5: p = vector with three or more points with different y components, and none of them have the same x component
- Test 6: p = vector with three or more points with the same y component, and some of them have the same x component
- Test 7: p = vector with three or more points with the same y component, and all of them have the same x component

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

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

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

c. Basic Condition Coverage:

- Test 1: p = empty vector
- Test 2: p = vector with one point
- Test 3: p = vector with two points with the same y component, and the first point has a smaller x component
- Test 4: p = vector with two points with the same y component, and the second point has a smaller x component
- Test 5: p = vector with two points with different y components
- Test 6: p = vector with three or more points with different y components, and none of them have the same x component
- Test 7: p = vector with three or more points with the same y component, and some of them have the same x component
- Test 8: p = vector with three or more points with the same y component, and all of them have the same x component.

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