IT314 Software Engineering

Name : Kosiya Kenil Hasmukhbhai
ID : 202001235
Lab : 7

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?

Solution :

Day	Month	Year	Expected output
1	5	2000	30-04-2000
2	2	2005	01-02-2005
29	2	2007	Invalid
1	3	2001	28-02-2001
1	3	2012	29-02-2012
31	04	2012	Invalid
1	1	2000	31-12-1999
31	12	2015	30-12-2015
30	4	2016	Invalid
1	1	1994	Invalid

Equivalent Class	Range	Validity
E1	1<=Day<=31	Valid
E2	Day<1	Invalid
E3	Day>31	Invalid
E4	1<=Month<=31	Valid
E5	Month<1	Invalid
E6	Month>12	Invalid
E7	1900<=Year<=2015	Valid
E8	Year<1900	Invalid
E9	Year>2015	Invalid

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.

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.

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

Equivalence Partitioning:

Tester Action and Input Data	Expected Outcome
v is present in the array a=[5,6,7,8], v=7	1
v is not present in the array a=[5,6,7,8], v=50	-1

Boundary Value Analysis:

Tester Action and Input Data	Expected Outcome
Empty array a=[], v=7	-1
Value Present at particular index a=[5,6,7,8], v=8	3
a=[5,6,5,7,8,7,7], v=7	3

```
1  package test1;
2
3  import static org.junit.Assert.*;
4
5  import org.junit.Test;
6
7  public class linearsearch {
8
9      @Test
10     public void test1() {
11         c1 obj = new c1();
12
13         int[] arr2 = {-3, 0, 3, 7, 11};
14         int[] arr3 = {1, 3, 5, 7, 9};
15
16         assertEquals(2, obj.linearSearch(3, arr2));
17         assertEquals(4, obj.linearSearch(9, arr3));
18     }
19
```

P2. The function countItem returns the number of times a value v appears in an array of integers a.

```
public 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
v not present in the array a=[5,6,7,8], v=10	0
v appear a single time	1
v appear multiple times a=[5,6,5,7,8,7,7], v=5	2

Boundary Value Analysis:

Tester Action and Input Data	Expected Outcome
Empty array a=[], v=7	0
Value Present at single index a=[5,6,7,8], v=8	1
Value Present at multiple index a=[5,6,5,7,8,7,7], v=7	3

```
19
20    @Test
21    public void test2() {
22        c1 counter = new c1();
23
24        int[] arr1 = {1, 2, 3, 4, 5};
25        int[] arr2 = {1, 2, 3, 4, 5, 6, 7, 8, 9};
26        int[] arr3 = {1, 2, 3, 4, 4, 4, 5, 6, 7, 8, 9};
27
28        int v1 = 3;
29        int v2 = 10;
30
31        assertEquals(1, counter.countItem(v1, arr1));
32        assertEquals(0, counter.countItem(v2, arr1));
33        assertEquals(0, counter.countItem(v2, arr2));
34        assertEquals(1, counter.countItem(v1, arr3));
35        assertEquals(0, counter.countItem(v2, arr3));
36    }
```

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.

```
public 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 Action and Input Data	Expected Outcome
v is present in the array a=[5,6,7,8], v=7	1
v is not present in the array a=[5,6,7,8], v=50	-1

Boundary Value Analysis:

Tester Action and Input Data	Expected Outcome
Empty array a=[], v=7	-1
Value Present at particular index a=[5,6,7,8], v=7	2
v is not present in the array a=[5,6,7,8], v=50	-1

```java
@Test
public void test3() {
    c1 bs = new c1();

    int[] arr1 = {1, 3, 5, 7, 9};
    assertEquals(0, bs.binarySearch(1, arr1)); // search for 1 in {1, 3, 5, 7, 9}
    assertEquals(2, bs.binarySearch(5, arr1)); // search for 5 in {1, 3, 5, 7, 9}
    assertEquals(4, bs.binarySearch(9, arr1)); // search for 9 in {1, 3, 5, 7, 9}
    assertEquals(-1, bs.binarySearch(4, arr1)); // search for 4 in {1, 3, 5, 7, 9}

    int[] arr2 = {2, 4, 6, 8, 10, 12};
    assertEquals(-1, bs.binarySearch(1, arr2)); // search for 1 in {2, 4, 6, 8, 10, 12}
    assertEquals(2, bs.binarySearch(6, arr2)); // search for 6 in {2, 4, 6, 8, 10, 12}
    assertEquals(5, bs.binarySearch(12, arr2)); // search for 12 in {2, 4, 6, 8, 10, 12}
    assertEquals(-1, bs.binarySearch(7, arr2)); // search for 7 in {2, 4, 6, 8, 10, 12}
}
```

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).

```java
final static int EQUILATERAL = 0;
final static int ISOSCELES = 1;
final static int SCALENE = 2;
final static 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);
}
```

Equivalence Partitioning:

Tester Action and Input Data	Expected Outcome
If any value is negative	Invalid
Invalid triangle: a+b<=c, a+c<=b, b+c<=a	Invalid
Valid equilateral triangle: a=b=c	equilateral

Valid isosceles triangle: a=b<c, a=c<b, b=c<a	isosceles
Valid scalene triangle: a<b<c (here all combinations are valid)	scalene

Boundary Value Analysis:

Tester Action and Input Data	Expected Outcome
a = b = c = 0	invalid
a = b = c = 1	equilateral
a=10 b=10 c=9	isosceles
a=3 b=4 c=5	scalene

```
@Test
public void testEquilateral() {
    assertEquals(0, c1.triangle(3, 3, 3));
  }

@Test
public void testIsosceles() {
    assertEquals(1, c1.triangle(5, 5, 6));

}

@Test
public void testScalene() {
    assertEquals(2, c1.triangle(3, 4, 5));
}

@Test
public void testIncorrectInput() {
    assertEquals(3, c1.triangle(1, 2, 3));

}
```

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;
}
```

Equivalence Partitioning:

Tester Action and Input Data	Expected Outcome
Both Empty strings	True
s1 empty , s2 non empty	True
s1 is prefix of s2	True
s1 is not a prefix of s2	False
s1 length id higher than s2 length	False

Boundary Value Analysis:

Tester Action and Input Data	Expected Outcome
s1=”kenil”, s2=”kenilkoshiya”	True
s1=”kenil”, s2=”Koshiya”	False
s1=””, s2=”vrvbejnd”	True
s1=”nefjv”, s2=””	False
s1=””,s2=””	True

```
@Test
public void testPrefix() {
        String s1 = "hello";
        String s2 = "hello world";
        assertTrue(c1.prefix(s1, s2));

        s1 = "abc";
        s2 = "abcd";
        assertTrue(c1.prefix(s1, s2));

        s1 = "";
        s2 = "hello";
        assertTrue(c1.prefix(s1, s2));

        s1 = "hello";
        s2 = "hi";
        assertFalse(c1.prefix(s1, s2));

        s1 = "abc";
        s2 = "def";
        assertFalse(c1.prefix(s1, s2));
    }
```

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

E1 : Negative values or zero
E2 : Sum of two sides is less than third side
E3: Scalene triangle a != b & b != c &c != a
E4: Isosceles triangle a=b, b=, a=c
E5: Equilateral trianglea=b=c
E6: Right-angled triangle (Pythagorean theorem should be valid for sides)

b) 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)

E1: -5,5,7; 0,84,1
E2: 5,10,50
E3: 3, 4, 5
E4: 5, 5, 7
E5: 10,10,10
E6: 3,4,5

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

Case 1 : 3,4,5

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

Case 1 : 10,10,15
Case 2 : 4,5,5

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

Case 1 : 5,5,5
Case 2 : 8,8,8

f) For the boundary condition A2 + B2 = C2 case (right-angle triangle), identify test cases to verify the boundary.

Case 1 : 3,4,5
Case 2 : 5,12,13

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

Case 1 : 1,2,4
Case 2 : 2,4,8

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

Case 1 : 5,0,10
Case 2 : 6,-8,9

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.

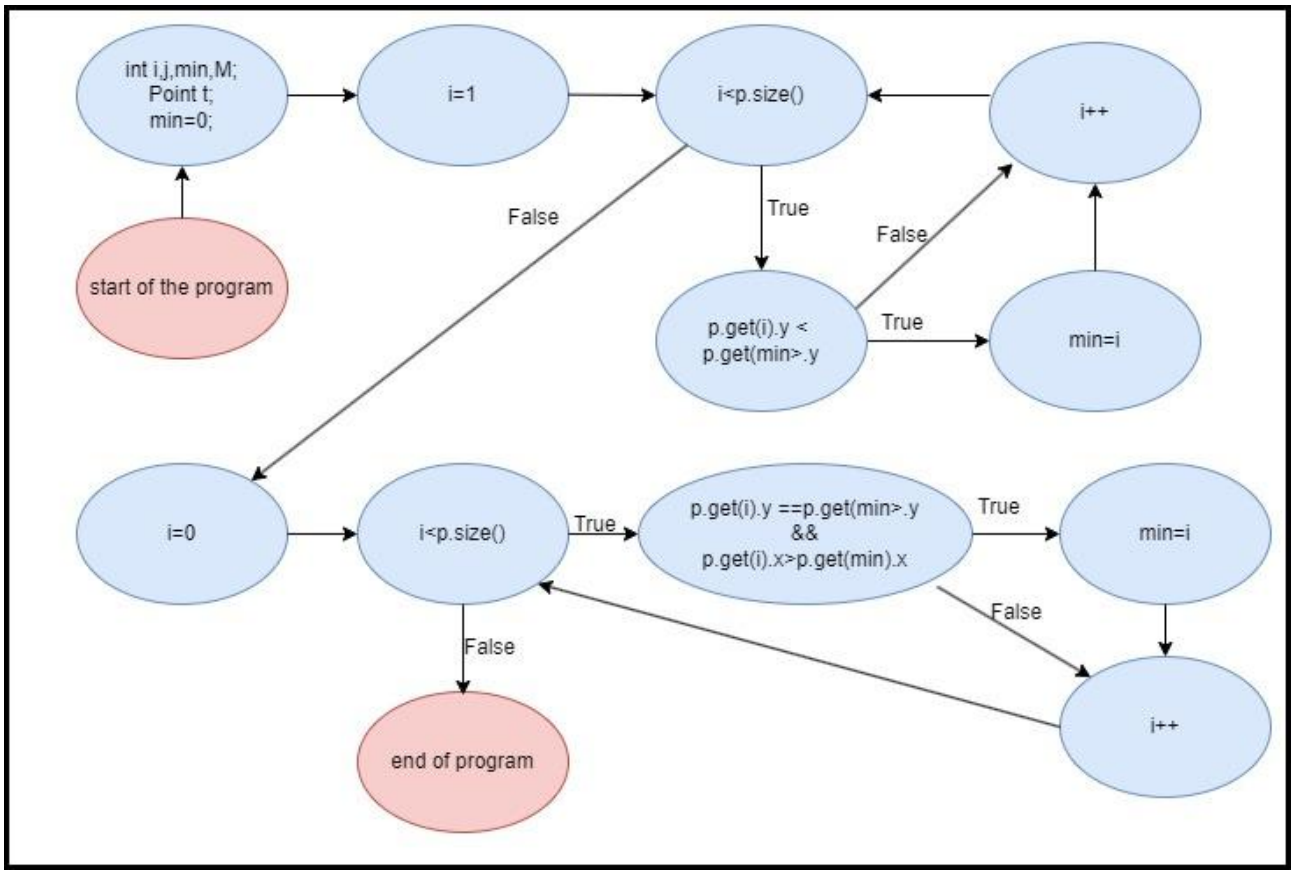
```
Vector doGraham(Vector p) {
        int i,j,min,M;

        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;
            }
        }
```

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.

To satisfy statement coverage, we need to ensure that each statement in the CFG is executed at least once. We can achieve this by providing a test case with a single point in the vector. In this case, both loops will not execute, and the return statement will be executed. A test set that satisfies statement coverage would be: p = [Point (0,0)]

b. Branch Coverage.

To satisfy branch coverage, we need to ensure that each branch in the CFG is executed at least once. We can achieve this by providing a test case with two points such that one of the points has the minimum y-coordinate, and the other has a greater x-coordinate than the minimum. In this case, both loops will execute, and the second branch in the second loop will be taken. A test set that satisfies branch coverage would be:

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

c. Basic Condition Coverage.

To satisfy basic condition coverage, we need to ensure that each condition in the CFG is evaluated to both true and false at least once. We can achieve this by providing a test case with three points such that two of the points have the same y-coordinate, and the other has a greater x-coordinate than the minimum. In this case, both loops will execute, and the second condition in the second loop will be evaluated to true and false. A test set that satisfies basic condition coverage would be:
p = [Point (0,0), Point (1,1), Point (2,0)]