Software Engineering

Lab 7

Name : Chirag Chavda
ID : 202001164
Date: 13/04/2023

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?

Equivalent classes :

E1 - {1<=day<=31} , E2 - {day<1} , E3 - {day>31} ,
E4 - {1<=month<=12} , E5 - {month<1} , E6 - {month>12} ,
E7 - {1900<=year<=2015} , E8 - {year<1900} , E9 - {year>2015}

Here are the possible dates that comes under each class giving some output :

Class	day	month	year	output
E1	30	4	2003	29/4/2003
E2	0	4	2003	INVALID
E3	32	4	2003	INVALID
E4	4	11	1976	3/11/1976
E5	4	-11	1976	INVALID
E6	4	13	1976	INVALID
E7	23	10	1975	22/10/1975
E8	23	10	1899	INVALID
E9	23	10	2023	INVALID

PROGRAMS

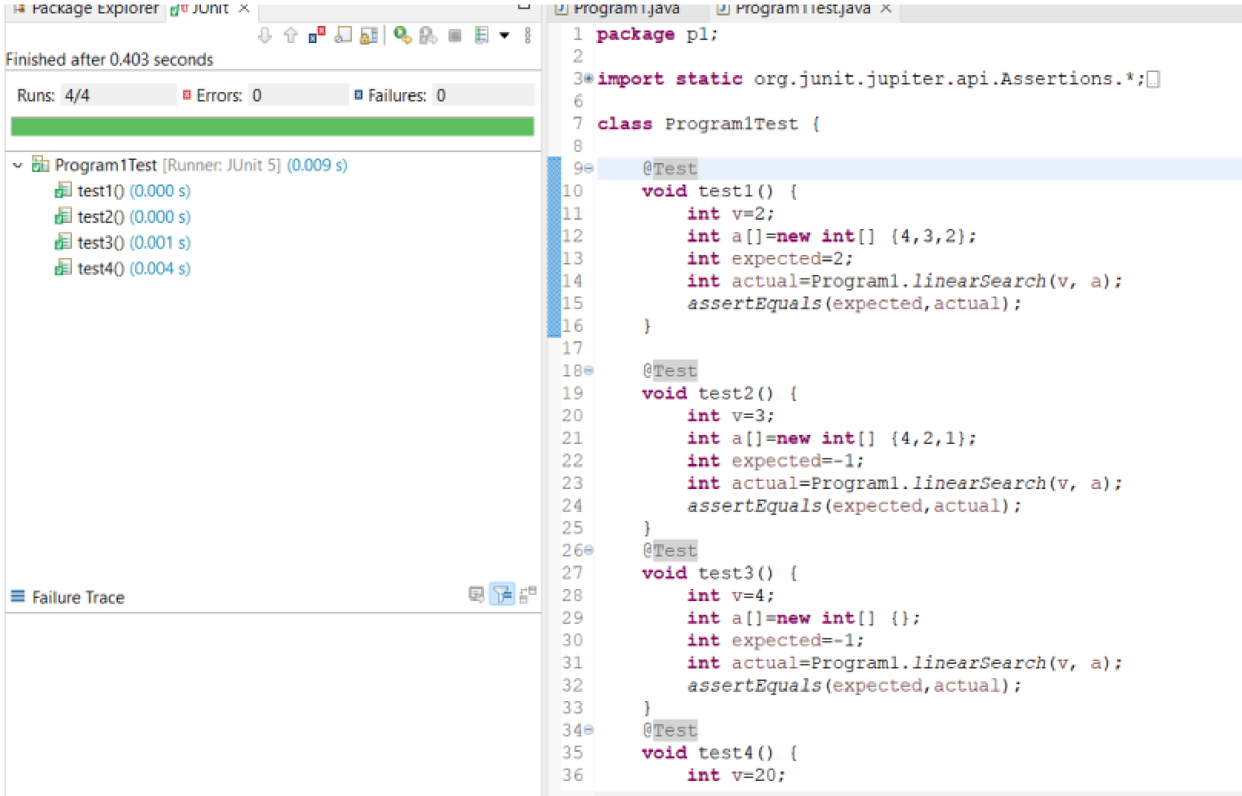
P1 : The function linearSearch searches for a value v in an array of integers a. If v appears in the array a, then the function returns the first index i, such that a[i] == v; otherwise, -1 is returned.

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

Test cases :

Input	**output**
v = 7 , a = {7,8,6}	**0**
v = 0 , a = {2,8,4}	**-1**
v = 0 , a = {12,34,2,5,2,8}	**-1**
v = 11 , a = {2,57,4,11,23}	**3**
v = 100 , a = {}	**-1**

Tester action and input data equivalence partitioning	**expected output**
v = 7 , a = {7,8,6}	0
v = 0 , a = {2,8,4}	-1
v = 0 , a = {12,34,2,5,2,8}	-1
v = 11 , a = {2,57,4,11,23}	3
Boundary values analysis	
v = 100 , a = {}	-1



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

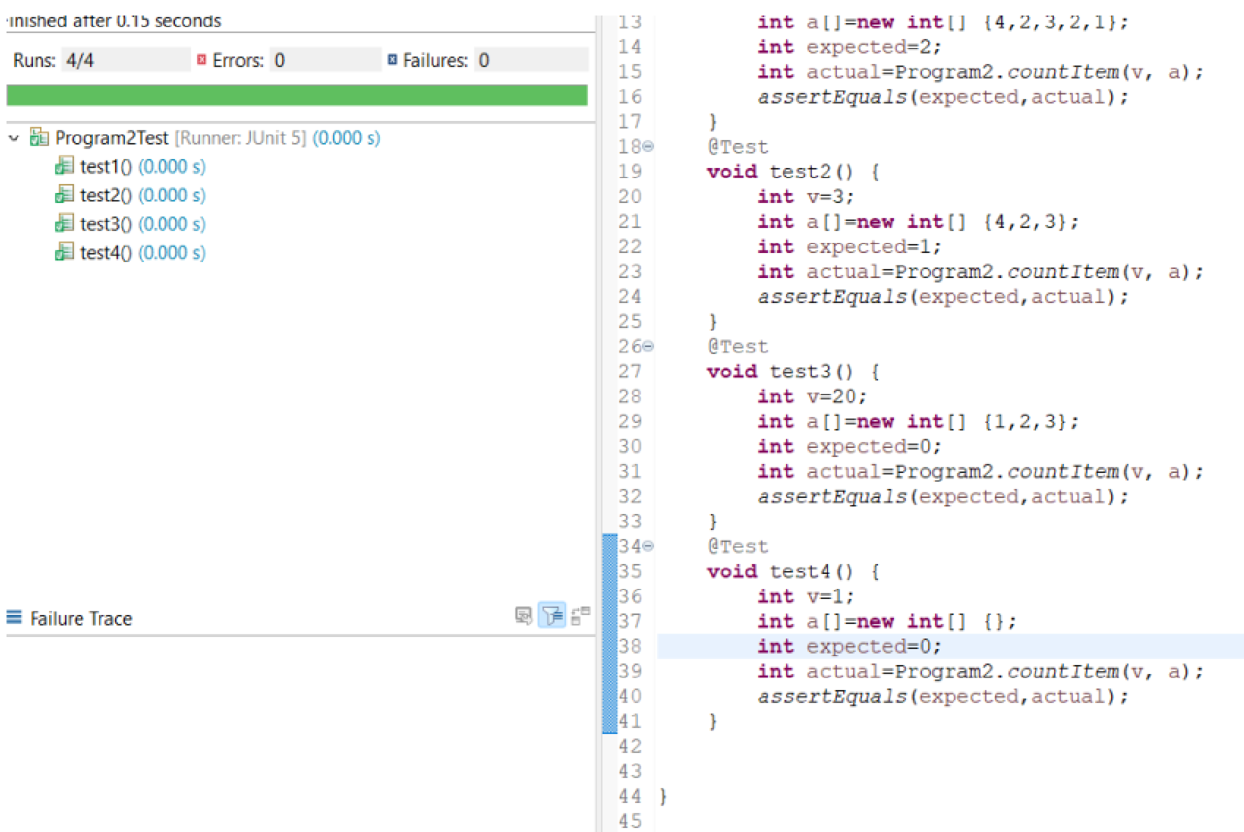
```
int countItem(int v, int a[])
{
        int count = 0;
        for (int i = 0; i < a.length; i++)
        {
        if (a[i] == v) count++;
        }
        return (count);
}
```

Test cases :

Input	output
v = 7 , a = {7,90,38,29,7}	2
v = 0 , a = {2,8,0,47,2,2,2,2}	1
v = 0 , a = {12,34,2,5,2,8}	0

v = 11 , a = {2,57,4,1,1,1,1,1,1,23}	**-1**
v = 99 , a = {}	**-1**

Tester action and input data equivalence partitioning	**expected output**
v = 7 , a = {7,90,38,29,7}	2
v = 0 , a = {2,8,0,47,2,2,2,2}	1
v = 0 , a = {12,34,2,5,2,8}	0
v = 11 , a = {2,57,4,1,1,1,1,1,1,23}	0
Boundary values analysis	
v = 99 , a = {}	0



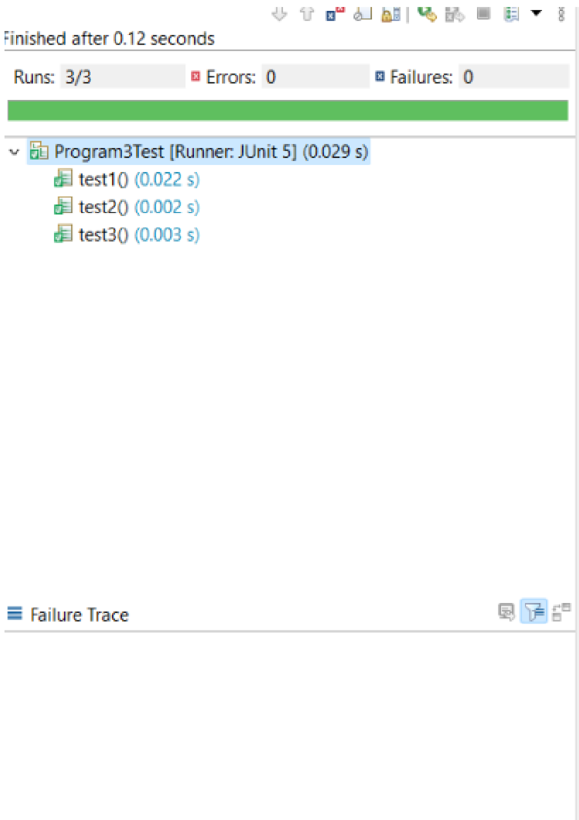
P3 : The function binarySearch searches for a value v in an ordered array of integers a. If v appears in the array a, then the function returns an index i, such that a[i] == v; otherwise, -1 is returned.

```
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);
 }
```

Test cases :

Input	**output**
v = 7 , a = {6,7,8,9,10}	**1**
v = 0 , a = {-2,-2,-1,0,1,1,2}	**3**
v = 53 , a = {12,34,45,55,72,88}	**-1**
v = 23 , a = {2,3,4,11,23}	**4**
v = 53 , a = {12,34,53,53,72,88}	**2 or 3**

Tester action and input data equivalence partitioning	**expected output**
v = 7 , a = {6,7,8,9,10}	1
v = 0 , a = {-2,-2,-1,0,1,1,2}	3
v = 23 , a = {2,3,4,11,23}	4
v = 53 , a = {12,34,53,53,72,88}	2 or 3
Boundary values analysis	
v = 53 , a = {12,34,45,55,72,88}	-1




```
@Test
void test1() {
    int v=2;
    int a[]=new int[] {0,1,2,3,4};
    int expected=2;
    int actual=Program3.binarySearch(v, a);
    assertEquals(expected,actual);
}

@Test
void test2() {
    int v=-4;
    int a[]=new int[] {1,2,3,4,5};
    int expected=-1;
    int actual=Program3.binarySearch(v, a);
    assertEquals(expected,actual);
}

@Test
void test3() {
    int v=5;
    int a[]=new int[] {2,3,4,5,5,6};
    int expected=3;
    int actual=Program3.binarySearch(v, a);

    // two possible correct outputs
    try {
    assertEquals(expected,actual);
    }
    catch(AssertionError e)
    {
        assertEquals(4,actual);
    }
}
```

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).

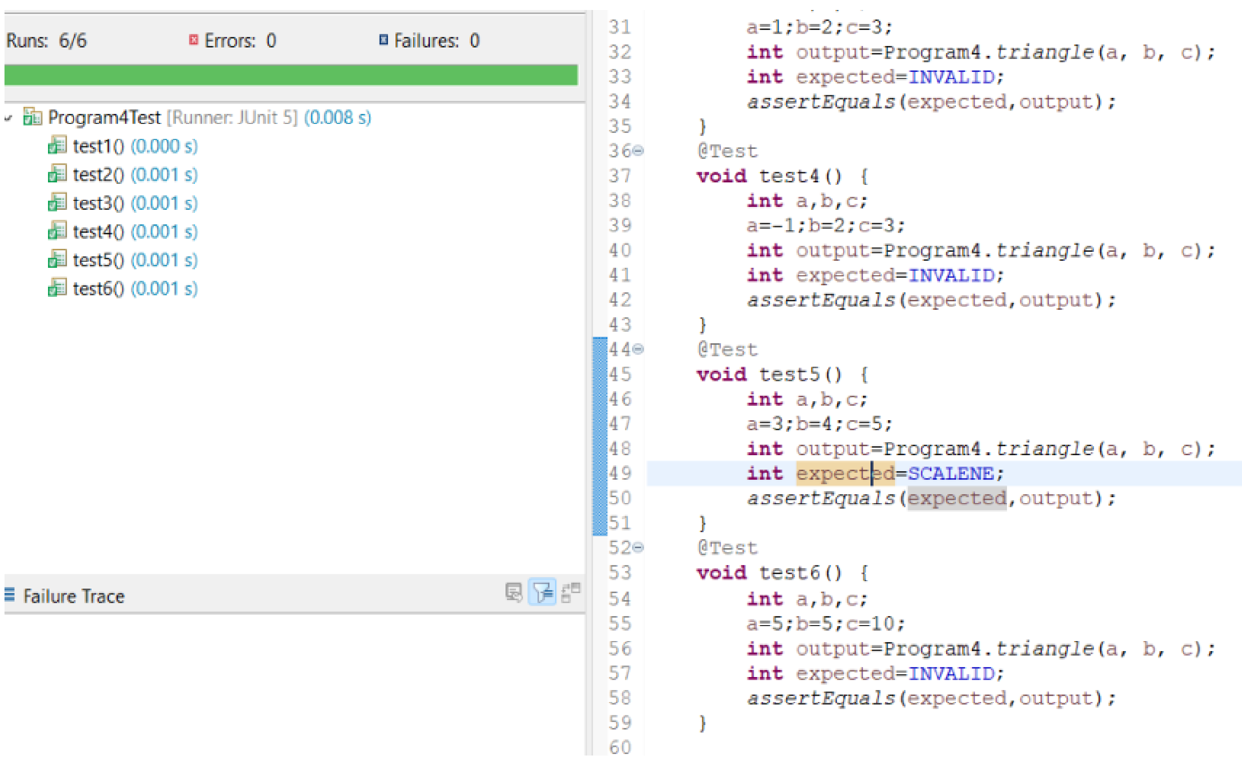
```
final int EQUILATERAL = 0;
final int ISOSCELES = 1;
final int SCALENE = 2;
final int INVALID = 3;

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);
 }
```

Test cases :

Input	output

a=3, b=4, c=5	SCALENE
a=7, b=7, c=7	EQUILATERAL
a=5, b=5, c=6	ISOSCELES
a=2, b=3, c=5	INVALID
a=0, b=5, c=10	INVALID

Tester action and input data equivalence partitioning	**expected output**
a=3, b=4, c=5	SCALENE
a=7, b=7, c=7	EQUILATERAL
a=5, b=5, c=6	ISOSCELES
Boundary values analysis	
a=2, b=3, c=5	INVALID
a=0, b=5, c=10	INVALID



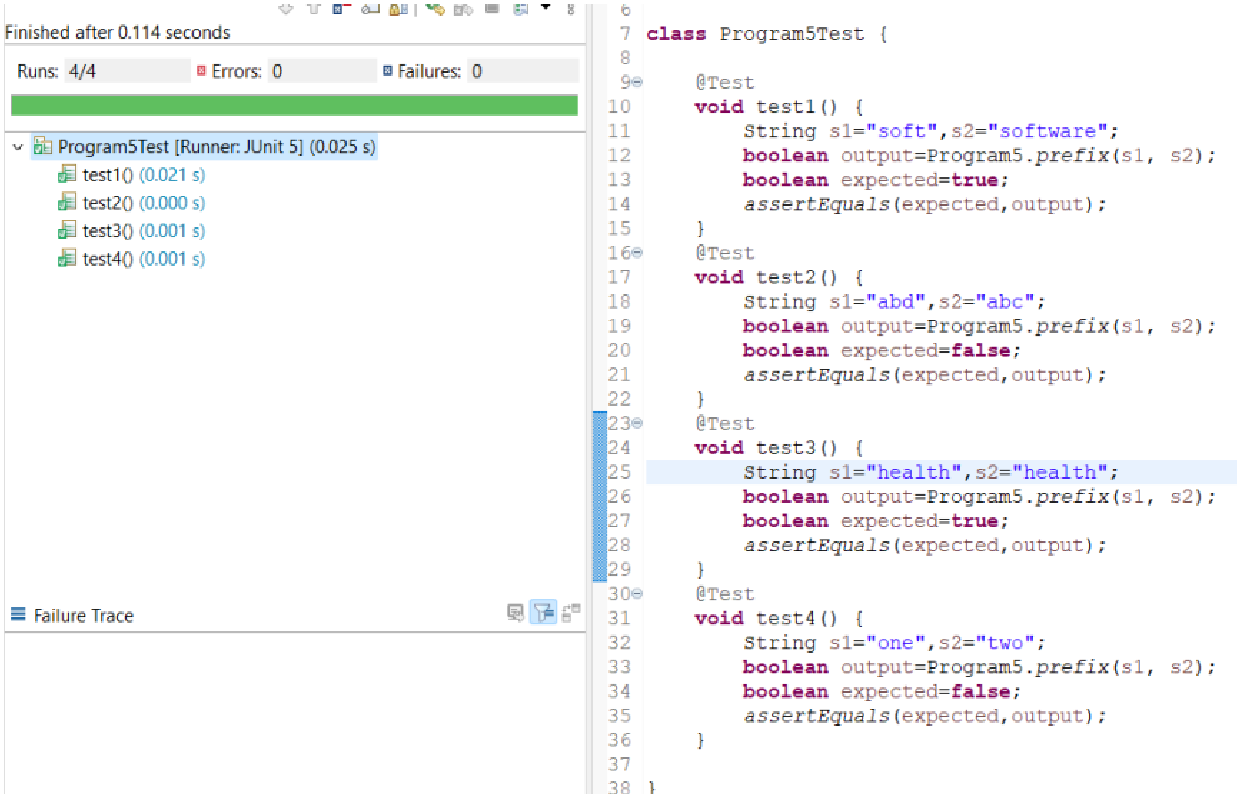
P5 : The function prefix (String s1, String s2) returns whether or not the string s1 is a prefix of string s2 (you may assume that neither s1 nor s2 is null).

```
        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;
  }
```

Test cases :

Input	**output**
s1=”chirag”, s2=”chiragchavda”	true
s1=”daiict”, s2=”gandhinagar”	false
s1=”infocity” , s2=”information”	false
s1=”abc” , s2=”kbc”	false
s1=”” , s2=”ict”	true

Tester action and input data equivalence partitioning	**expected output**
s1=”chirag”, s2=”chiragchavda”	true
s1=”daiict”, s2=”gandhinagar”	false
s1=”infocity” , s2=”information”	false
s1=”abc” , s2=”kbc”	false
Boundary values analysis	
s1=”” , s2=”ict”	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:

a) Identify the equivalence classes for the system

The equivalence classes for different types of triangles are as follows:

Invalid case:

E1 : a+b<=c

E2 : a+c<=b

E3: b+c <=a

Equilateral case:

E4 : a=b,b=c,c=a

Isosceles case:

E5 : a=b , a!=c

E6: a= c, a!=b

E7: b=c, b!=a

Scalene case:

E8 : a!=b ,b!=c, c!=a

Right-angled triangle case:

E9 : a^2 + b^2 = c^2
E10: b^2+c^2 =a^2
E11: a^2 +c^2=b^2

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)

Test case	**Output**	**Equivalent class covered**
a=1.5,b=2.6,c=4.1	IS NOT TRIANGLE	E1
a=-1,6,b=5,c=6	IS NOT TRIANGLE	E2
a=7.1,b=6.1,c=1	IS NOT TRIANGLE	E3
a=7.8,b=7.8,c=7.8	EQUILATERAL	E4
a=9.8,b=9.8,c=11	ISOSCELES	E5
a=9,b=2,c=9	ISOSCELES	E6
a=11,b=11,c=7	ISOSCELES	E7
a=6,b=7,c=8	SCALENE	E8
a=3,b=4,c=5	RIGHT TRIANGLE	E9
a=5,b=12,c=13	RIGHT TRIANGLE	E10
a=7,b=25,c=23	RIGHT TRIANGLE	E11

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

Input (a+b>c)
a=9,b=10,c=17
a=10,b=3,c=11
a=7,b=20,c=21

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

Input (a=c)
a=17,b=9,c=17
a=10,b=3,c=10
a=7,b=10,c=7

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

Input (a=b=c)
a=9,b=9,c=9
a=10,b=10,c=10
a=20,b=20,c=20

f) For the boundary condition A^2 + B^2 = C^2 case (right-angle triangle), identify test cases to verify the boundary.

Input
a=3,b=4,c=5
a=5,b=12,c=13
a=7,b=25,c=23

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

Input
a=9,b=9,c=0
a=10,b=3,c=13
a=0,b=0,c=0

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

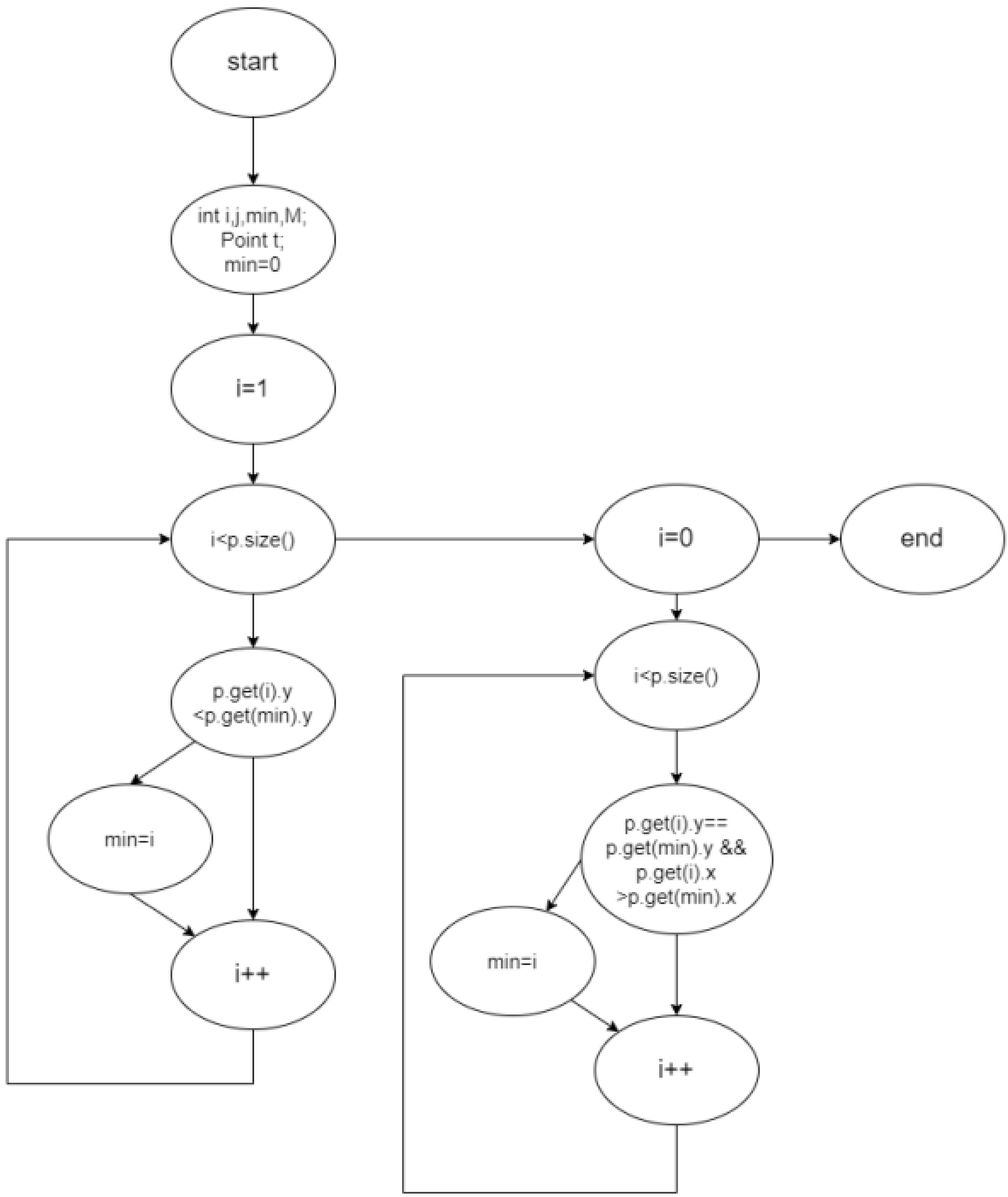
Input (a<0 or b<0 or c<0)
a=9,b=-1,c=17
a=10,b=3,c=-10
a=-20,b=20,c=20

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.

Test Case

1	Empty Vector
2	Vector with one point
3	vector with two points with the same y component
4	vector with two points with different y components
5	vector with three or more points with different y components
6	vector with three or more points with same y components

b. Branch Coverage.

1	Empty Vector
2	vector with one point
3	vector with two points with the same y component
4	vector with two points with different y components
5	vector with three or more points with different y components, and none of them have the same x component

6 vector with three or more points with the same y component, and some of them have the same x component

7 vector with three or more points with the same y component, and all of them have the same x component

c. Basic Condition Coverage

1 Empty vector

2 vector with one point

3 vector with two points with the same y component, and the first point has a smaller x component

4 vector with two points with the same y component, and the second point has a smaller x component

5 vector with two points with different y components

6 vector with three or more points with different y components, and none of them have the same x component

7 vector with three or more points with the same y component, and some of them have the same x component

8 vector with three or more points with the same y component, and all of them have the same x component.