IT314

Software Engineering

(Lab - 7)

Name : Deep kanani
ID : 202001454
Date : 18/04/23

Section A

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

Sol: From the given constraints 1 <= month <= 12, 1 <= day <= 31, 1900 <= year <= 2015, The following are the equivalence classes obtained.

Equivalent Classes:

There are a total of 9 equivalence classes.

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

The following are the weak normal equivalence class test cases:

Equivalent Class	Day	Month	Year	Output
E1	2	3	2011	1/3/2011
E2	0	4	2022	Invalid Date
E3	34	5	2000	Invalid Date
E4	1	1	1980	31/12/1989
E5	21	-4	1970	Invalid
E6	20	15	1943	Invalid
E7	4	5	1980	3/5/1980
E8	5	6	1899	Invalid
E9	4	3	2016	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.

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.**

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);
}
```

Equivalence Partitioning:

Tester Action and Input Data	Expected Outcome
'v' is not present in array 'a'	-1
'v' is present in array 'a'	Index of 'v'

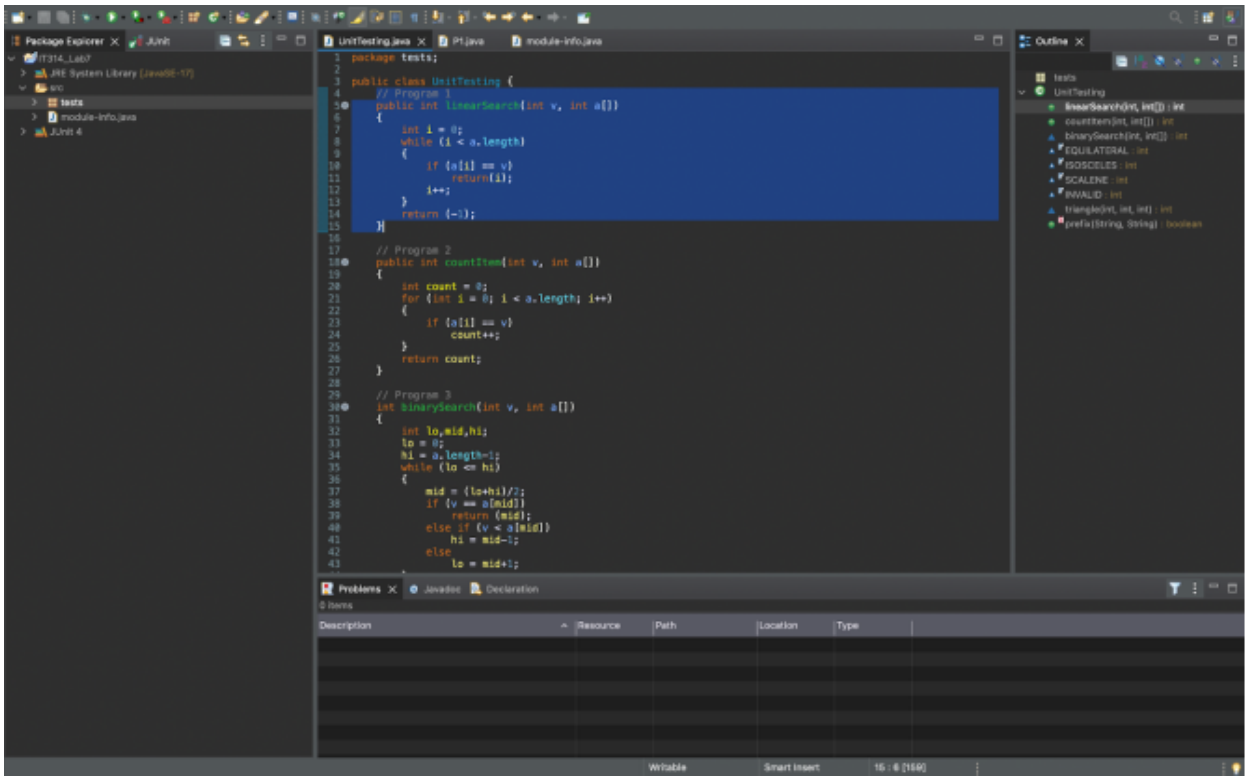
Boundary Value Analysis:

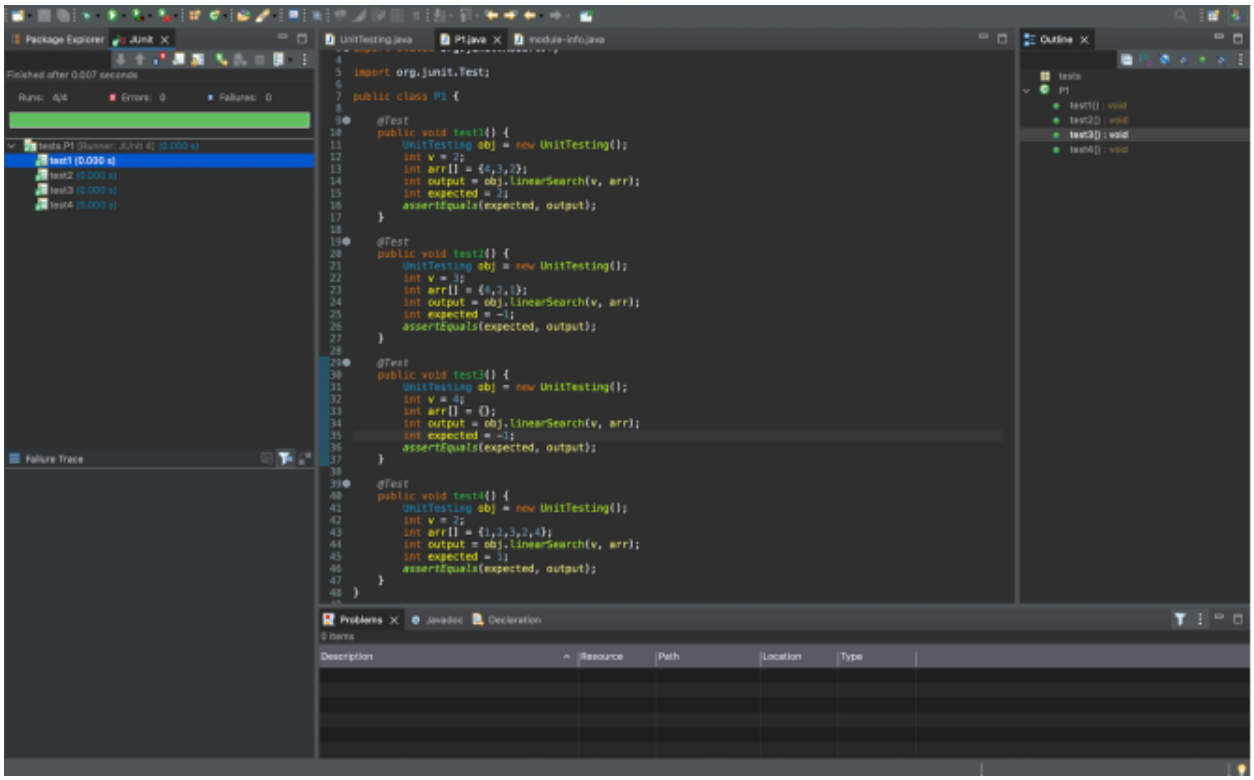
Tester Action and Input Data	Expected Outcome
Empty array 'a'	-1
'v' is present at 2nd index in array 'a'	2
'v' is not present in array 'a'	-1

Test Cases:

1. v = 2, a = {4,3,2}, expected output: 2
2. v = 3, a = {4,2,1}, expected output: -1
3. v = 4, a = {}, expected output: -1
4. v = 2, a = {1,2,3,2,4}, expected output: 1

JUnit Testing:





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
'v' is not present in array 'a'	0
'v' is present in array 'a'	Number of times 'v' appears in array 'a'

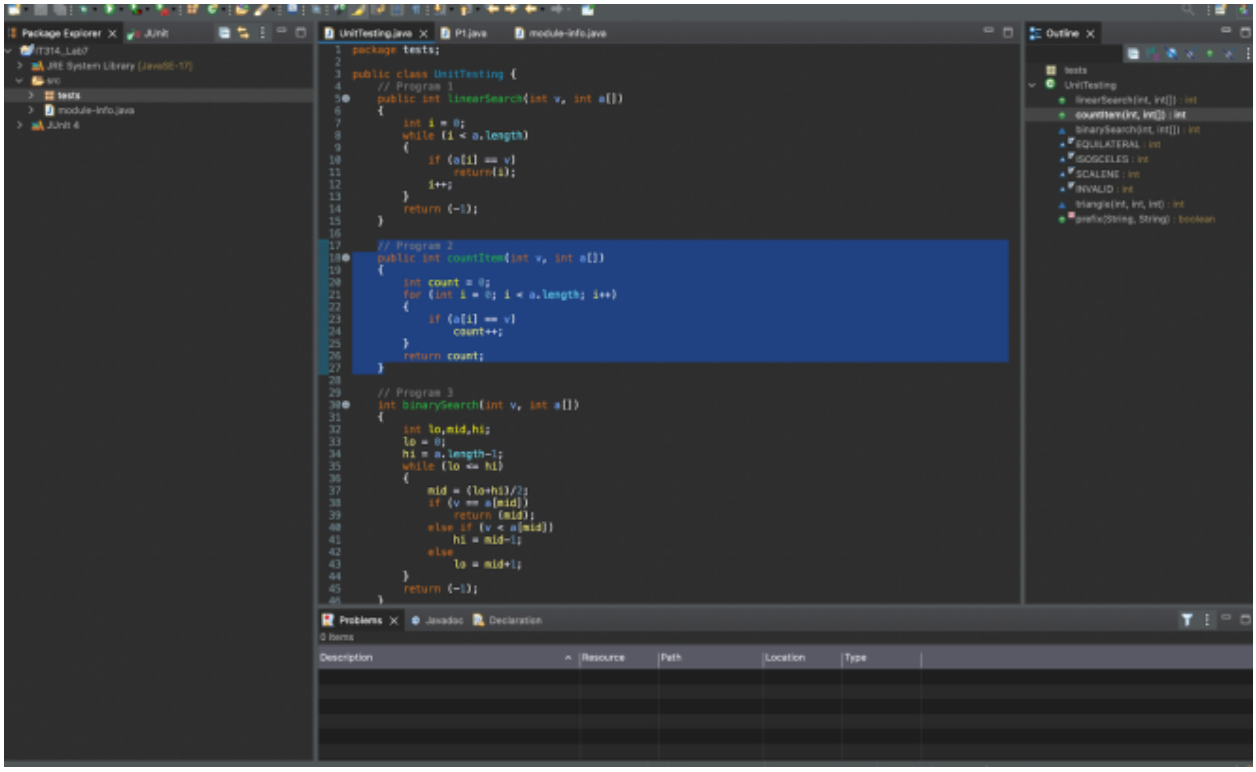
Boundary Value Analysis:

Tester Action and Input Data	Expected Outcome
Empty array 'a'	-1
'v' is present once in array 'a'	1
'v' is present multiple times in array 'a'	Number of times 'v' is present
'v' is not present in array 'a'	0

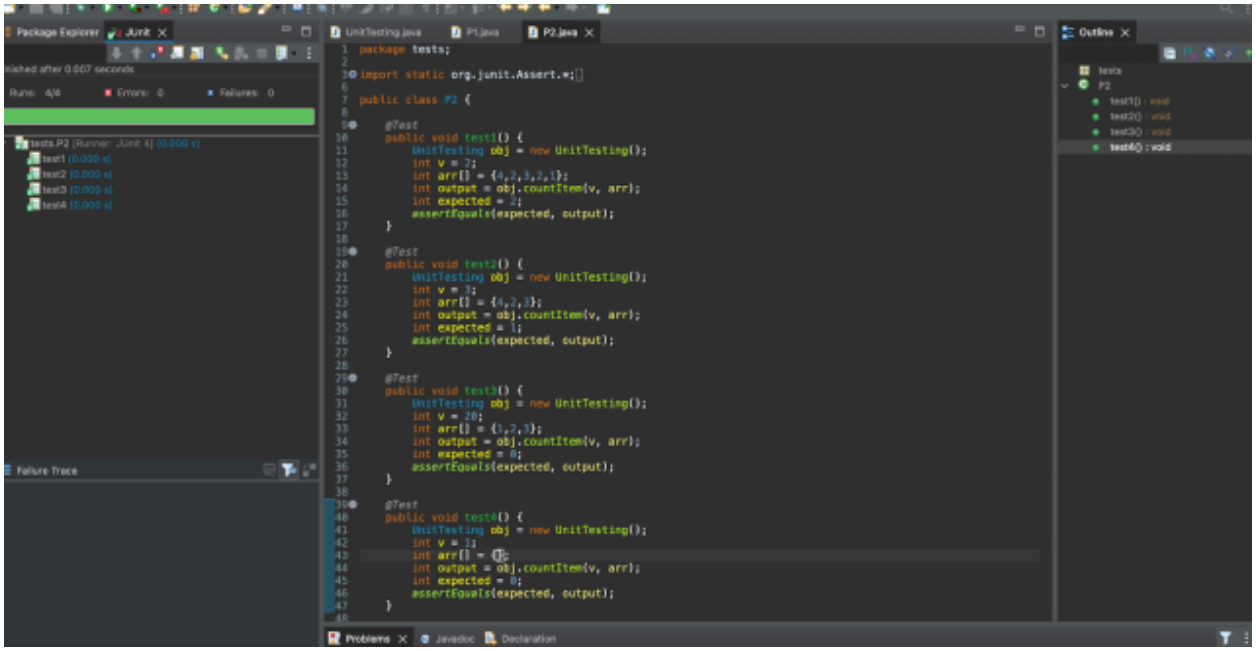
Test Cases:

1. v = 2, a = {4,2,3,2,1}, expected output: 2
2. v = 3, a = {4,2,3}, expected output: 1
3. v = 20, a = {1,2,3}, expected output: 0
4. v = 1, a = {}, expected output: 0

JUnit Testing:



```
package tests;

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

    // Program 2
    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;
    }

    // Program 3
    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);
    }
```



```
package tests;

import static org.junit.Assert.*;

public class P2 {

    @Test
    public void test1() {
        UnitTesting obj = new UnitTesting();
        int v = 2;
        int arr[] = {4,2,3,2,1};
        int output = obj.countItem(v, arr);
        int expected = 2;
        assertEquals(expected, output);
    }

    @Test
    public void test2() {
        UnitTesting obj = new UnitTesting();
        int v = 3;
        int arr[] = {4,2,3};
        int output = obj.countItem(v, arr);
        int expected = 1;
        assertEquals(expected, output);
    }

    @Test
    public void test3() {
        UnitTesting obj = new UnitTesting();
        int v = 20;
        int arr[] = {1,2,3};
        int output = obj.countItem(v, arr);
        int expected = 0;
        assertEquals(expected, output);
    }

    @Test
    public void test4() {
        UnitTesting obj = new UnitTesting();
        int v = 1;
        int arr[] = {};
        int output = obj.countItem(v, arr);
        int expected = 0;
        assertEquals(expected, 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. 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);
}
```

Equivalence Partitioning:

Tester Action and Input Data	Expected Outcome
'v' is not present in array 'a'	-1
'v' is present in array 'a'	Index of 'v' in array 'a'

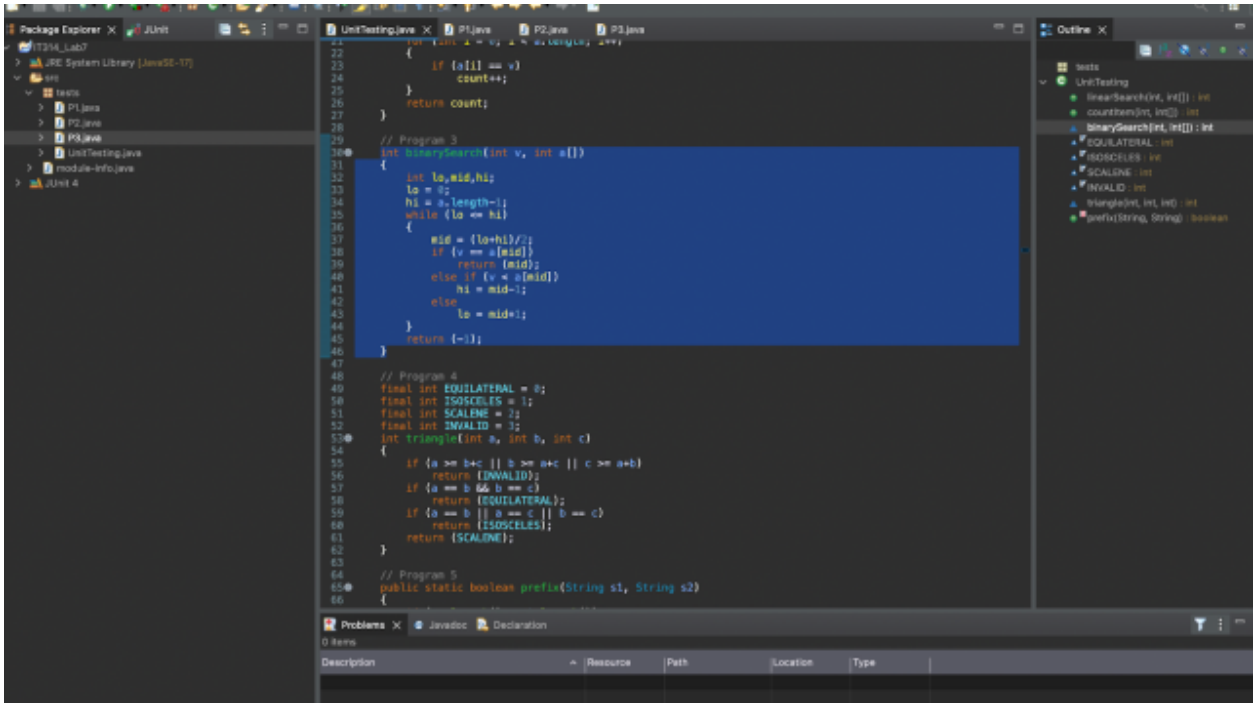
Boundary Value Analysis:

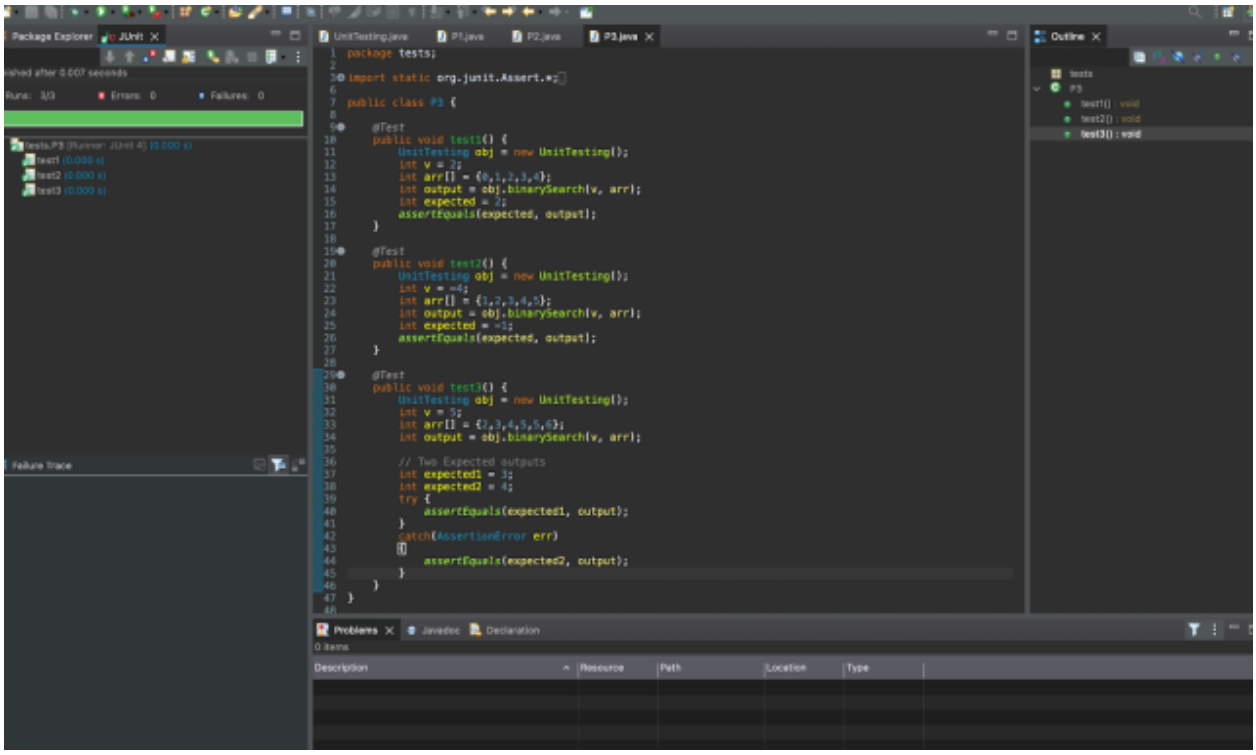
Tester Action and Input Data	Expected Outcome
Empty array ‘a’	-1
‘v’ is present at first index in array ‘a’	0
‘v’ is not present in array ‘a’	-1

Test Cases:

1. v = 2, a = {0,1,2,3,4}, expected output: 2
2. v = -4, a = {1,2,3,4,5}, expected output: -1
3. v = 5, a = {2,3,4,5,5,6}, expected output: 3 or 4

JUnit Testing:





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 Action and Input Data	Expected Outcome
Invalid triangle (a+b<=c)	INVALID
Valid equilateral triangle (a=b=c)	EQUILATERAL
Valid isosceles triangle (a=b<c)	ISOSCELES
Valid scalene triangle (a<b<c)	SCALENE

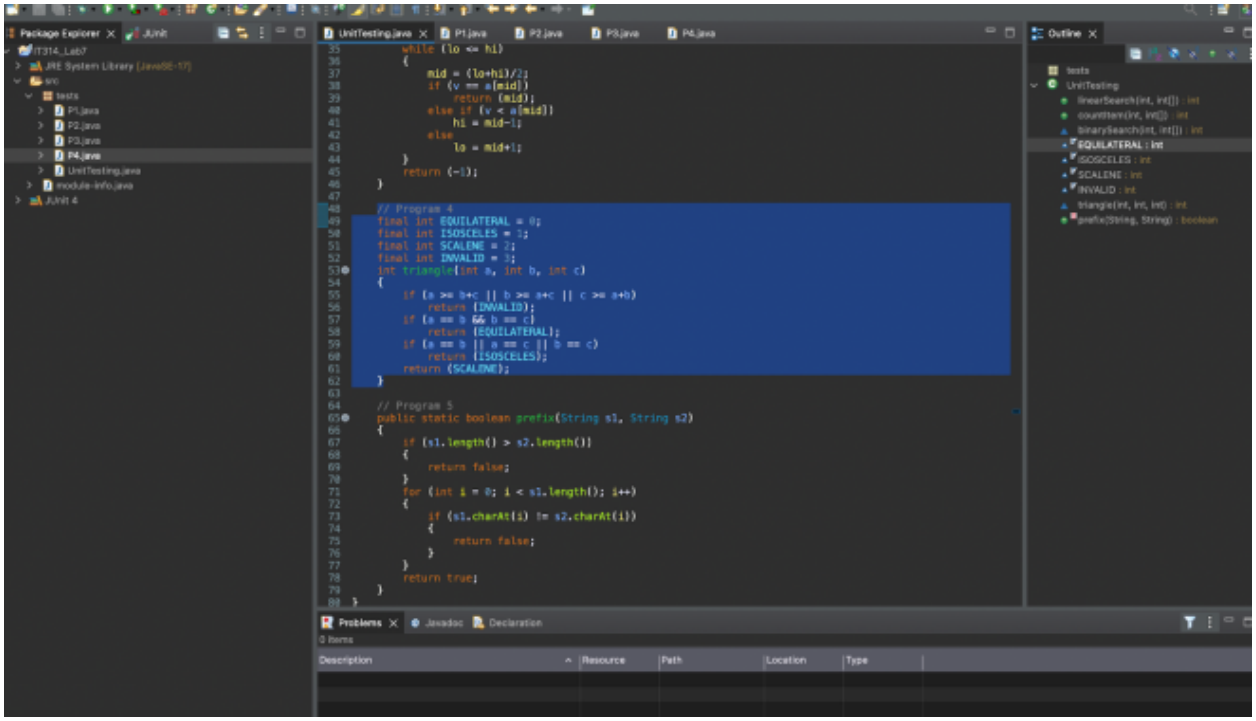
Boundary Value Analysis:

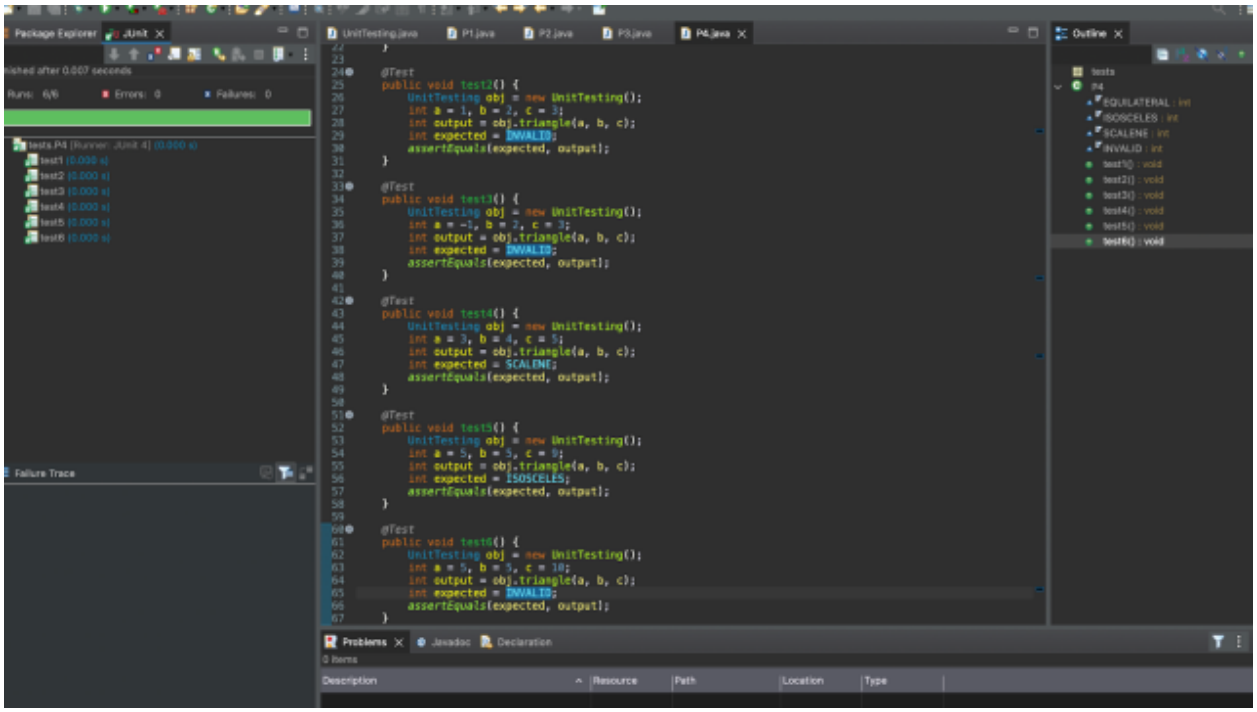
Tester Action and Input Data	Expected Outcome
Invalid triangle (a+b<=c)	INVALID
Invalid triangle (a+c<=b)	INVALID
Invalid triangle (b+c<=a)	INVALID
Valid equilateral triangle (a=b=c)	EQUILATERAL
Valid isosceles triangle (a=b<c)	ISOSCELES
Valid isosceles triangle (a=c<b)	ISOSCELES
Valid isosceles triangle (b=c<a)	ISOSCELES
Valid scalene triangle (a<b<c)	SCALENE

Test Cases:

1. a = 4, b = 4, c = 4, expected output: EQUILATERAL
2. a = 1, b = 2, c = 3, expected output: INVALID
3. a = -1, b = 2, c = 3, expected output: INVALID
4. a = 3, b = 4, c = 5, expected output: SCALENE
5. a = 5, b = 5, c = 9, expected output: ISOSCELES
6. a = 5, b = 5, c = 10, expected output: INVALID

JUnit Testing:





P5. The function prefix(Strings1,Strings2) 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
Empty string s1 and s2	True
Empty string s1 and non-empty s2	True
Non-empty s1 is a prefix of non-empty s2	True
Non-empty s1 is not a prefix of s2	False
Non-empty s1 is longer than s2	False

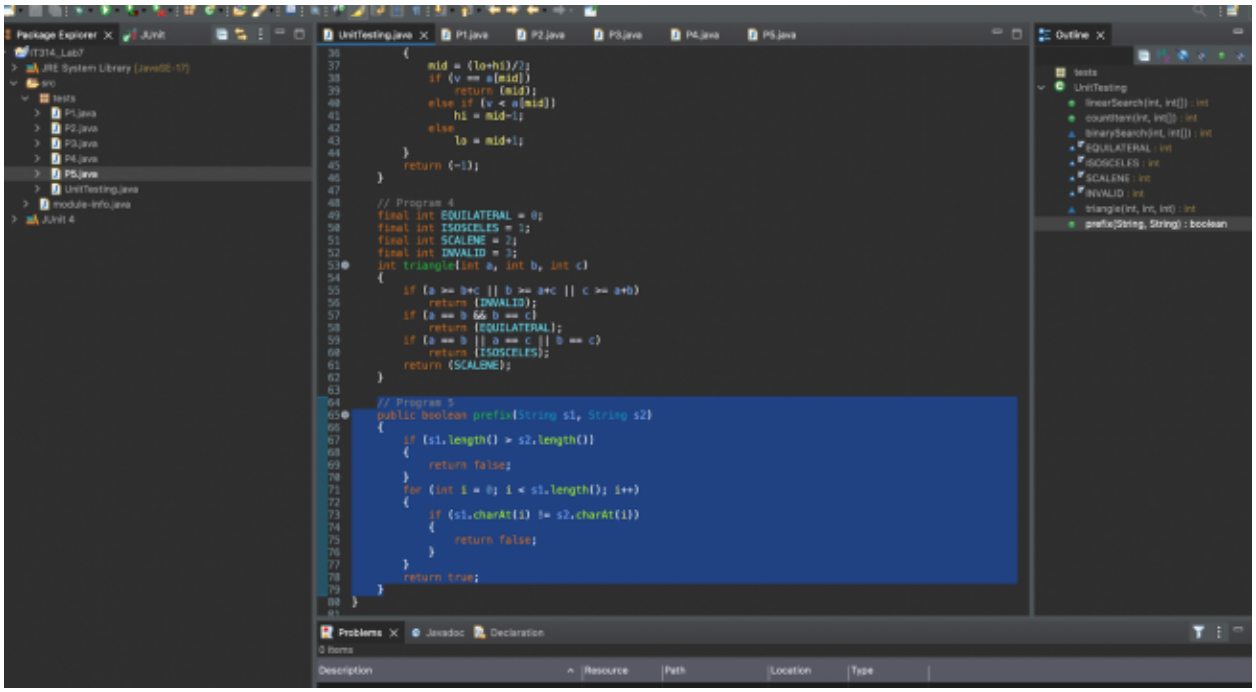
Boundary Value Analysis:

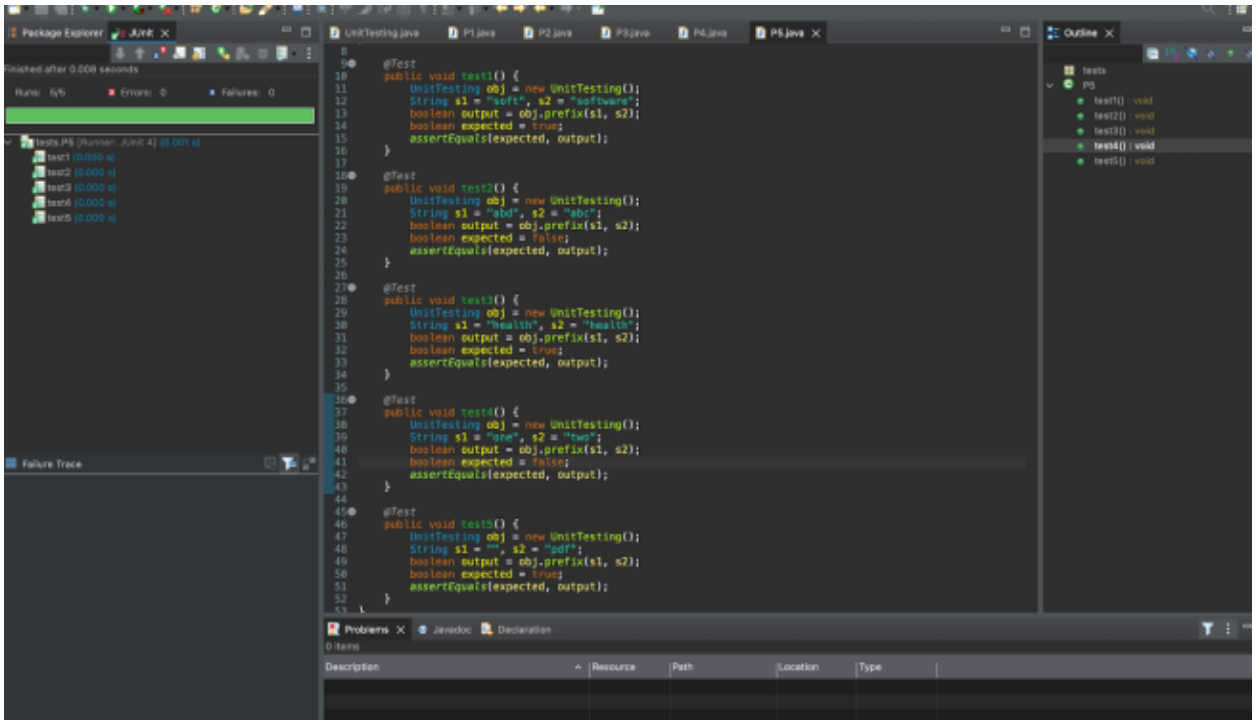
Tester Action and Input Data	Expected Outcome
Empty string s1 and s2	True
Empty string s1 and non-empty s2	True
Non-empty s1 is not a prefix of s2	False
Non-empty s1 is longer than s2	False

Test Cases:

1. s1 = “soft”, s2 = “software”, expected output: true
2. s1 = “abd”, s2 = “abc”, expected output: False
3. s1 = “health”, s2 = “health”, expected output: True
4. s1 = “one”, s2 = “two”, expected output: False
5. s1 = “”, s2 = “pdf”, expected output: True

JUnit Testing:





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 following are the equivalence classes for different types of triangles.
1. INVALID case:
→ E1: a + b <= c
→ E1: a + c <= b
→ E1: b + c <= a

2. EQUILATERAL case:
→ E1: a = b, b = c, c = a

3. ISOSCELES case:
→ E1: a = b, a != c
→ E1: a = c, a != b
→ E1: b = c, b != a

4. SCALENE case:
→ E1: a != b, b != c, c != a

5. RIGHT-ANGLED TRIANGLE case:
→ E1: a2 + b2 = c2
→ E1: b2 + c2 = a2
→ E1: a2 + c2 = b2

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	Equivalence Class
a = 1.5, b = 2.6, c = 4.1	INVALID	E1
a = -1.6, b = 5, c = 6	INVALID	E2
a = 7.1, b = 6.1, c = 1	INVALID	E3
a = 5.5, b = 5.5, c = 5.5	EQUILATERAL	E4
a = 4.5, b = 4.5, c = 5	ISOSCELES	E5
a = 6, b = 4, c = 6	ISOSCELES	E6
a = 8, b = 5, c = 5	ISOSCELES	E7
a = 6, b = 7, c = 8	SCALENE	E8
a = 3, b = 4, c = 5	RIGHT-ANGLED TRIANGLE	E9
a = 0.13, b = 0.12, c = 0.05	RIGHT-ANGLED TRIANGLE	E10
a = 7, b = 25, c = 23	RIGHT-ANGLED TRIANGLE	E11

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

The test cases to verify boundary condition:

1. a = 5, b = 4, c = 5
2. a = 5, b = 5, c = 9
3. a = 5, b = 6, c = 12

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

1. a = 5, b = 4, c = 5
2. a = 5, b = 4, c = 5.1
3.. a = 5, b = 4, c = 4.9

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

1. a = 5, b = 5, c = 5 (a =b=c)
2. a = 10, b = 10, c = 9 (a=b but a!=c)
3. a = 10, b = 11, c = 10 (a=c but a!=b)

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

1. a = 3, b = 4, c = 5
2. a = 5, b = 12, c = 13

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

1. a = 1, b = 2, c = 3
2. a = 4.5, b = 5.5, c = 10

h) For non-positive input, identify test points.
The test points for non-positive inputs:

1. a = -4.0, b = 3.2, c = 4.5
2. a = 5, b = -4.2, c = -3.2

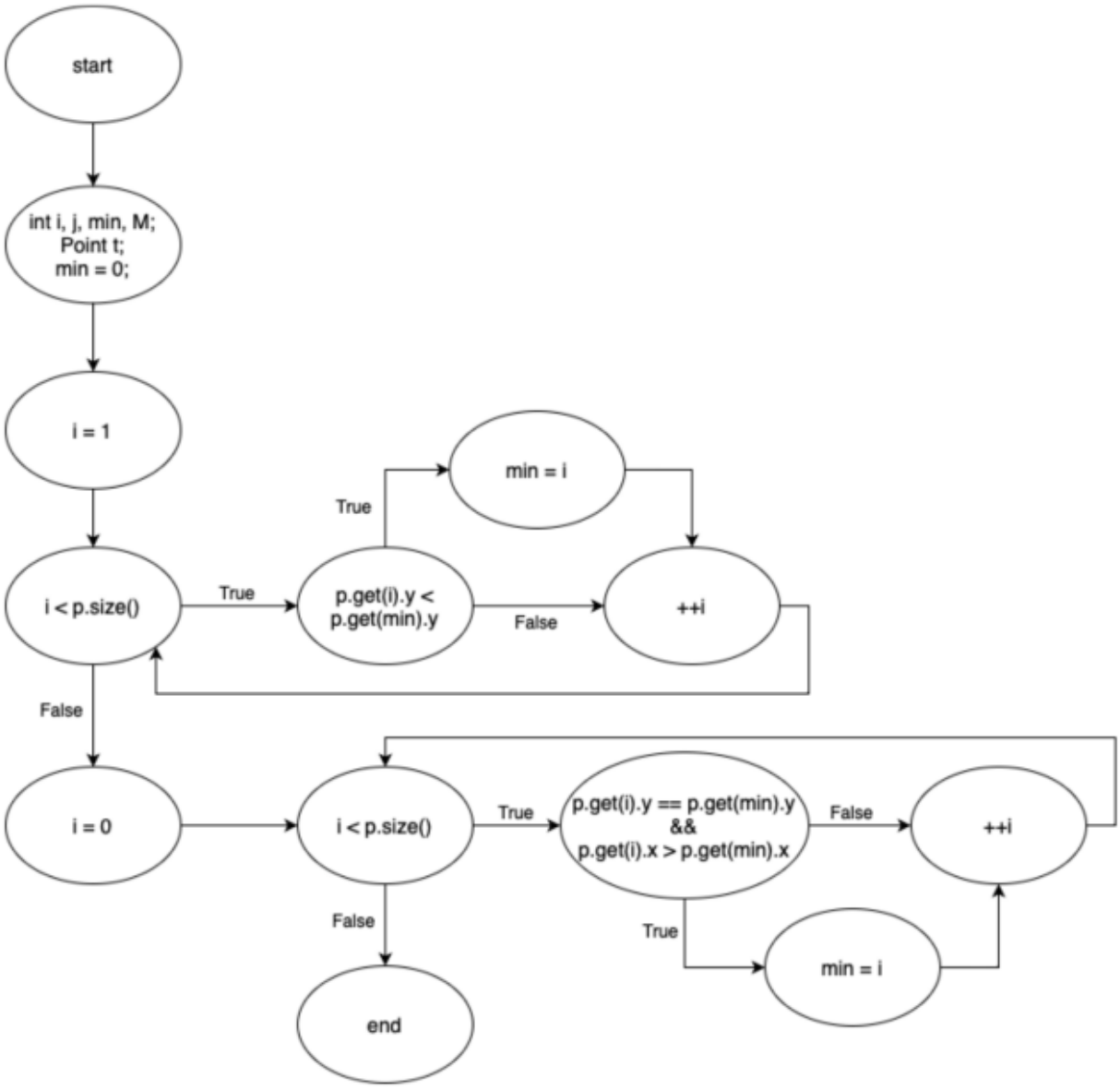
Section B

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 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).



Control Flow Graph (CFG) of doGraham Method

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.

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

Test cases:

1. p=[(x = 2, y = 2), (x = 2, y = 3), (x = 1, y = 3), (x = 1, y = 4)]
   Statements 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. These 5 test cases are adequate for statement coverage, branch coverage and basic condition coverage.