Lab-7 Testing

Name-Parikh Riki Anilbhai
Id-202001241

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?
Ans:

Variables	**Valid class**	**Invalid class**
Days	1 <= day <= 31	day<1 && day>31
Month	1 <= month <= 12	month<1 && month>12
Year	1900 <= year <= 2015	year<1900 && year>2015

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.

Some test cases:
Valid test cases:

a. 1,3,2000 -valid min date, valid mouth,valid year
b. 31,3,2000 -valid max date, valid mouth,valid year
c. 3,1,2000 -valid date, valid min mouth,valid year
d. 3,12,2000 -valid date, valid max mouth,valid year
e. 2,4,1900 - valid date, valid max mouth,valid min year
f. 2,4,2015 - valid date, valid max mouth,valid max year

Invalid test cases:

a. 0,12,2000 -invalid min date, valid mouth,valid year
b. 34,4,2000- invalid max date, valid mouth,valid year
c. 3,-1,2010 -valid date, invalid min mouth,valid year
d. 3,15,2010 -valid date, invalid max mouth,valid year
e. 3,4,1800 -valid date, valid mouth,invalid min year
f. 3,4,2100 -valid date, valid mouth,invalid max year

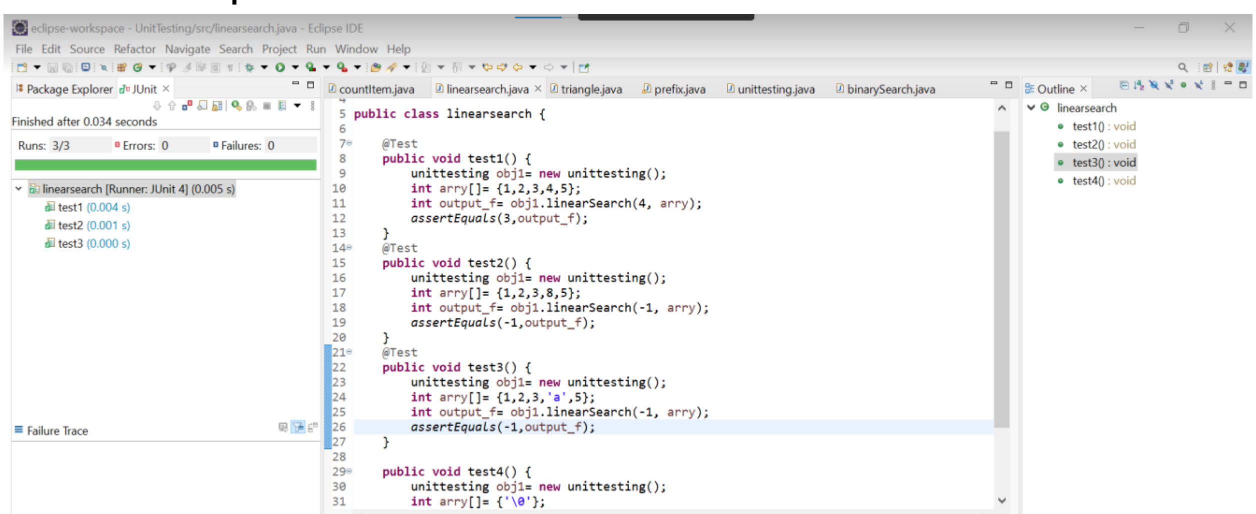
These test cases represent the equivalence classes and should cover all possible scenarios.

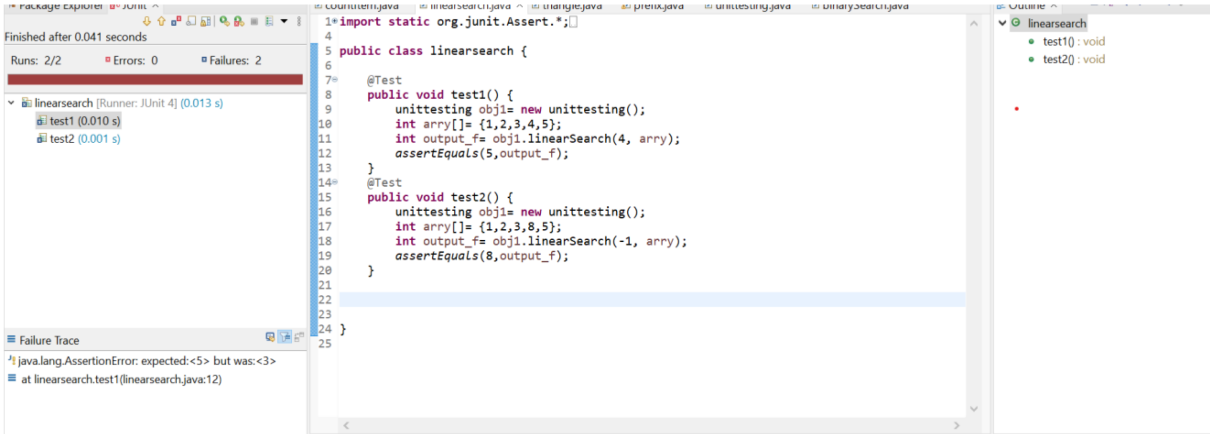
Programs:

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 Case in Eclipse:





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:

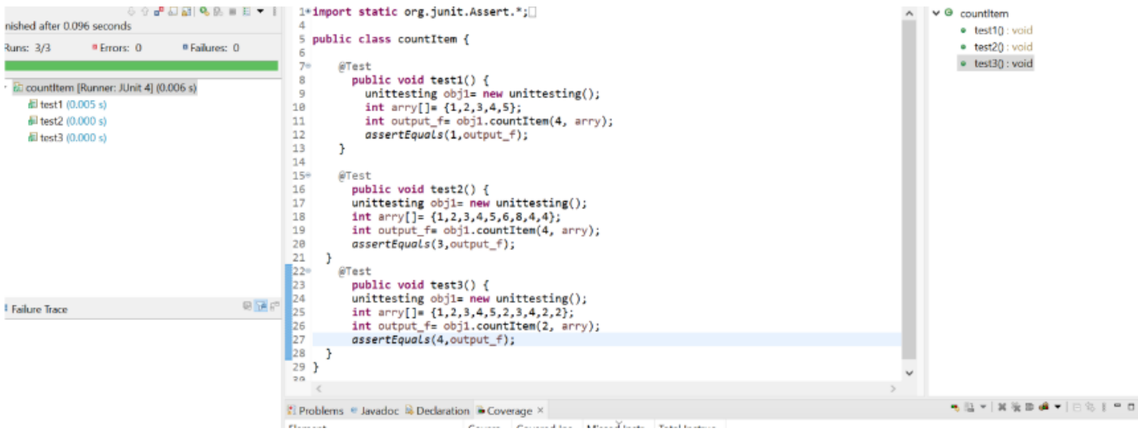
Tester Action and Input Data	Expected Outcome
Test with v as a non-existent value and an empty array a[]	-1
Test with v as a non-existent value and a non-empty array a[]	-1
Test with v as an existent value and an array	-1

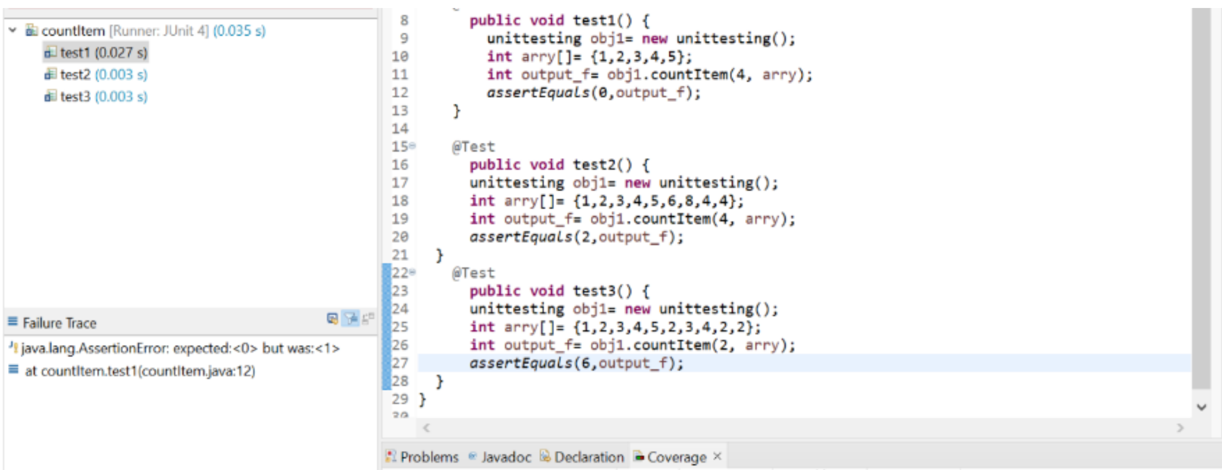
a[] of length 0	
Test with v as an existent value and an array a[] of length 1, where v exists	0
Test with v as an existent value and an array a[] of length 1, where v does not exist	-1
Test with v as an existent value and an array a[] of length greater than 1, where v exists at the beginning of the 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

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 Case in Eclipse:





Equivalence Partitioning:

Tester Action and Input Data	Expected Outcome
Test with v as an existent value and a non-empty array a[] where v exists only once	1
Test with v as a non-existent value and a non-empty array a[]	0

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 an existent value and an array	1

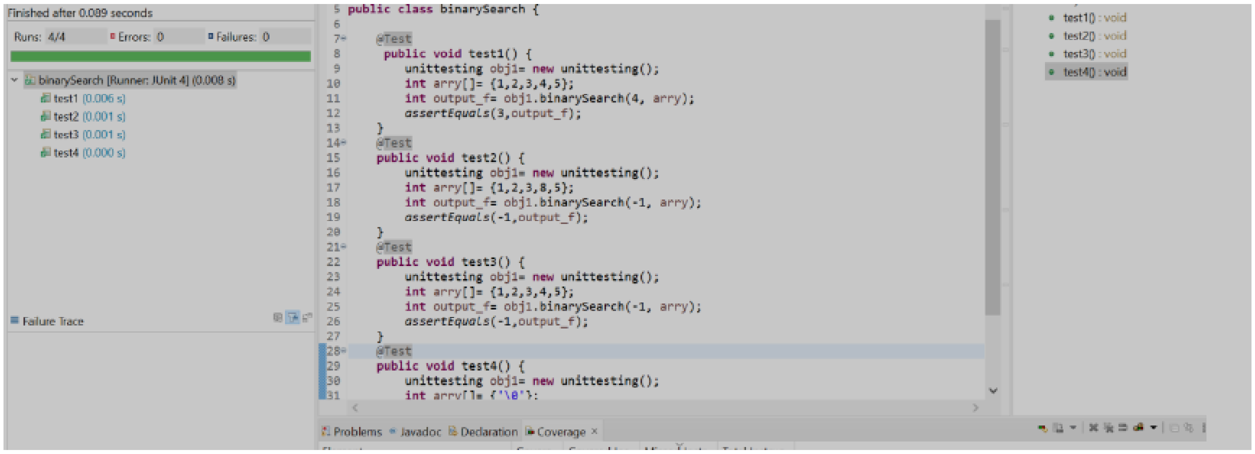
a[] of length 1, where v exists	
Test with v as an existent value and an array a[] of length greater than 1, where v exists at the end of the array	the number of occurrences of v in a[]
Test with v as an existent value and an array a[] of length greater than 1, where v exists in the middle of the array	the number of occurrences of v in a[]

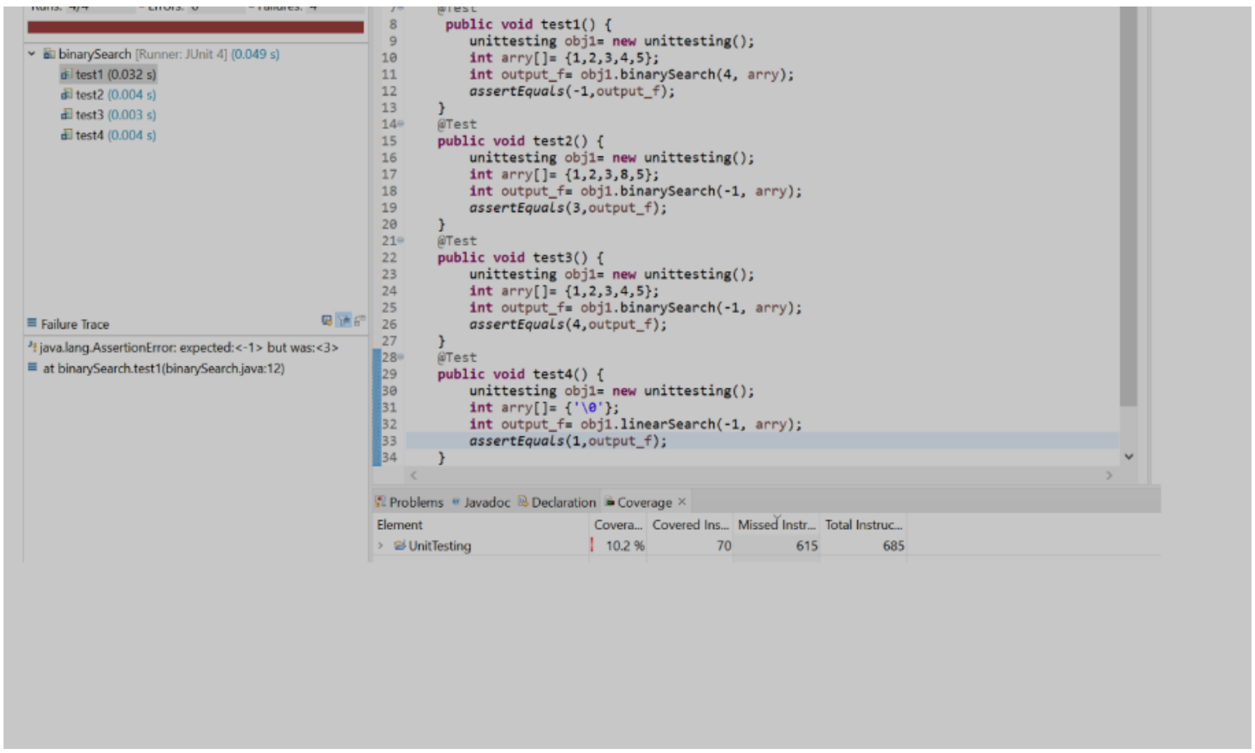
P3. The function binarySearch searches for a value v in an ordered array of integers a. If v appears in the array a, then the function returns an index i, such that a[i] == v; otherwise, -1 is returned. Assumption: the elements in the array are sorted in non-decreasing order.

```
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);
}
```

Test Case in Eclipse:





Equivalence Partitioning:

Tester Action and Input Data	Expected Outcome
v=5, a=[1, 3, 6, 8, 10]	2
v=2, a=[1, 3, 6, 8, 10]	0
v=5,a=[1, 3, 6, 8, 10]	5
v=4, a=[1, 3, 6, 8, 10]	-1
v=11, a=[1, 3, 6, 8, 10]	-1

boundary Value Analysis:

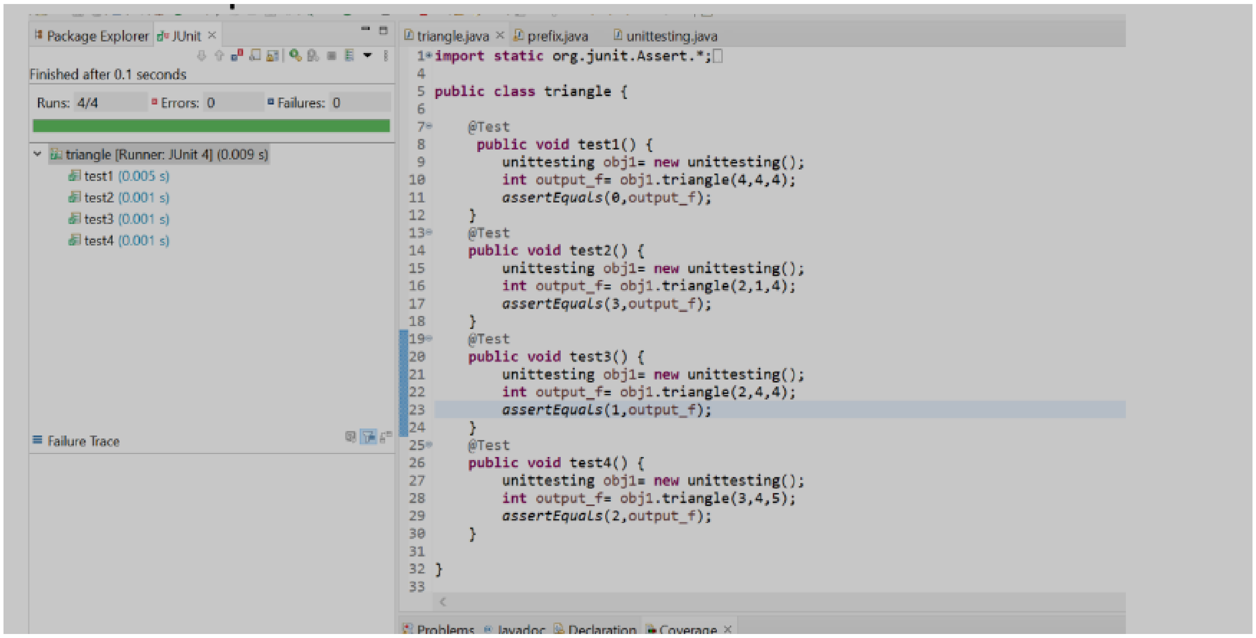
Tester Action and Input Data	Expected Outcome
v=1, a=[1]	0
v=9, a=[9]	0
v=5, a=[]	-1
v=5, a=[5, 7, 9]	0
v=5, a=[1, 3, 5]	2

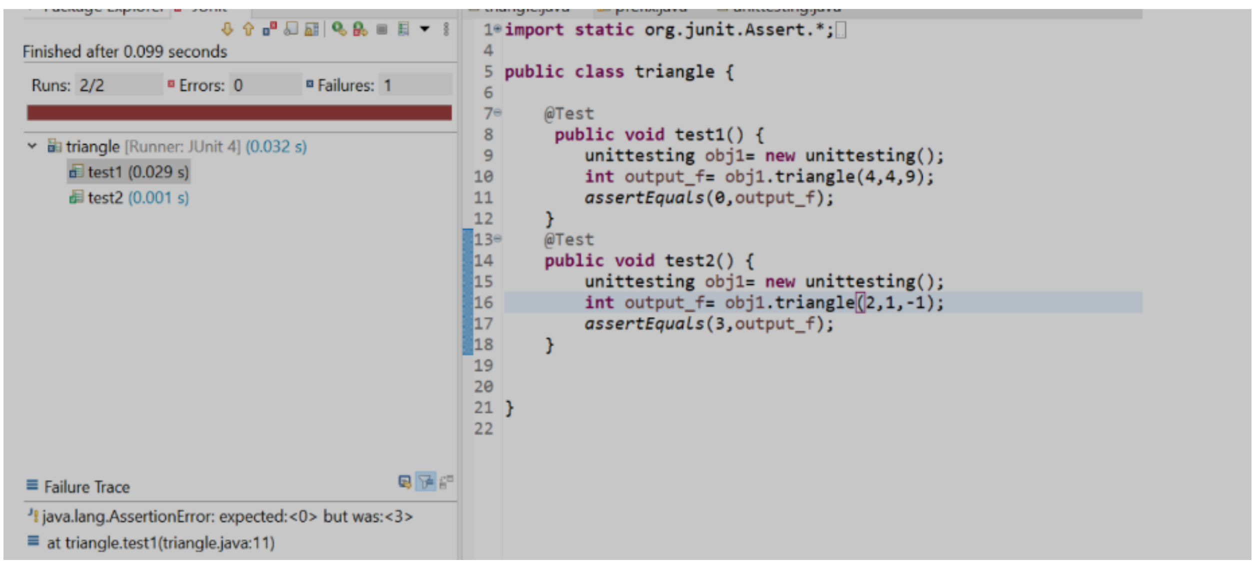
P4. The following problem has been adapted from The Art of Software Testing, by G. Myers (1979). The function triangle takes three integer parameters that are interpreted as the lengths of the sides of a triangle. It returns whether the triangle is equilateral (three lengths equal), isosceles (two lengths 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 Case in Eclipse:





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
Invalid input: a=0, b=1, c=1	INVALID
Invalid input: a=1, b=1, c=0	INVALID

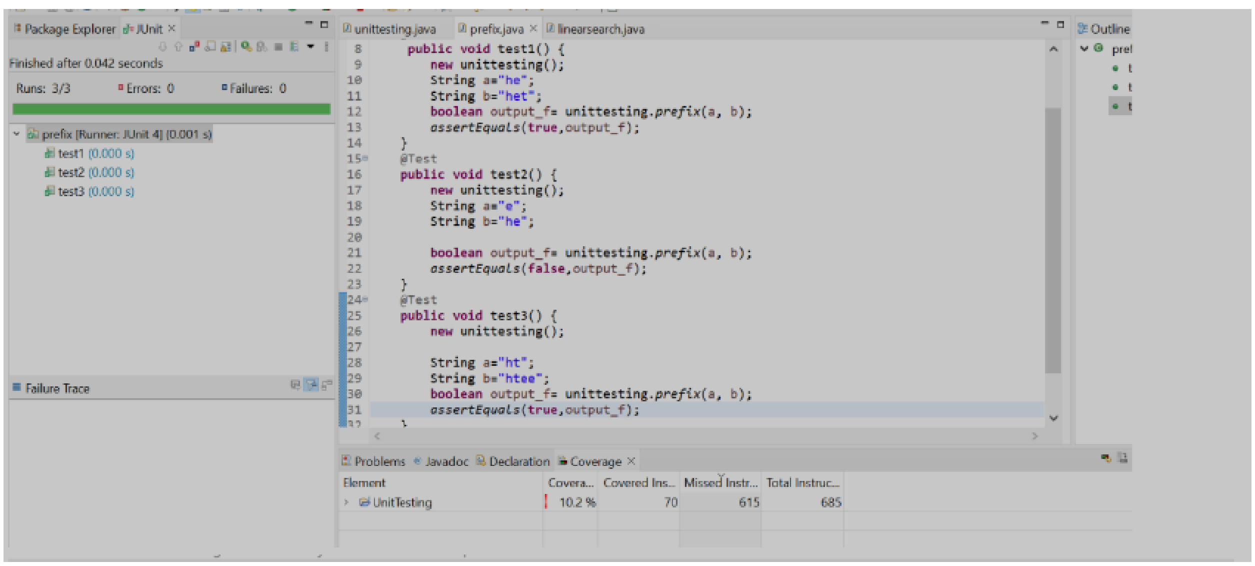
Boundary Value Analysis:

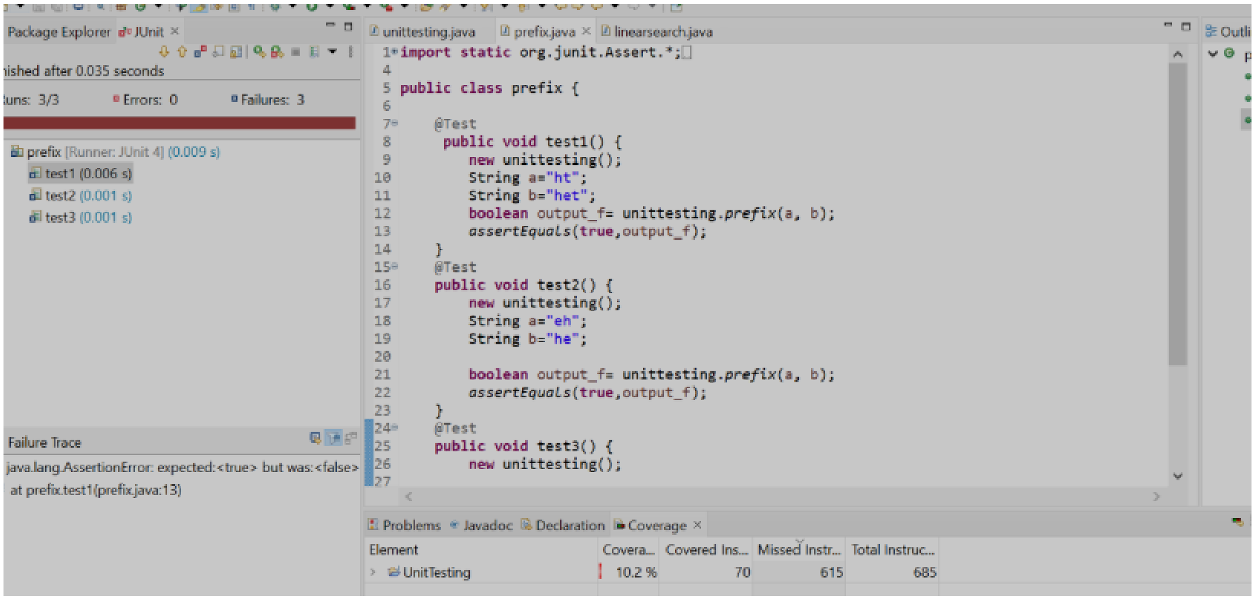
Tester Action and Input Data	**Expected Outcome**
Invalid inputs: a = 0, b = 0, c = 0	INVALID
Invalid inputs: a + b = c or b + c = a or c + a = b (a=3, b=4, c=8)	INVALID
Equilateral triangles: a = b = c = 1	EQUILATERAL
Equilateral triangles: a = b = c = 100	EQUILATERAL
Isosceles triangles: a = b ≠ c = 10	ISOSCELES
Minimum values: a, b, c = 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).

```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;
}
```

Test Case in Eclipse:





Equivalence Partitioning:

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

a. **Identify the equivalence classes for the system**

Tester Action and Input Data	**Expected Outcome**
a = -1, b = 2, c = 3	Invalid input
a = 1, b = 1, c = 1	Equilateral triangle
a = 3, b = 4, c = 5	Scalene right-angled triangle
a = 3, b = 5, c = 4	Scalene right-angled triangle
a = 3, b = 4, c = 6	Not a triangle

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:
Invalid inputs:
a = 0, b = 0, c = 0, a + b = c, b + c = a, c + a = b
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;

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. For the boundary condition A + B > C case (scalene triangle), identify test cases to verify the boundary.
a = Integer.MAX_VALUE, b = Integer.MAX_VALUE, c = 1
a = Double.MAX_VALUE, b = Double.MAX_VALUE, c = Double.MAX_VALUE

d. For the boundary condition A = C case (isosceles triangle), identify test cases to verify the
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 = Integer.MAX_VALUE, b = Integer.MAX_VALUE, c = Integer.MAX_VALUE
a= Double.MAX_VALUE, b = Double.MAX_VALUE, c = Double.MAX_VALUE

f. For the boundary condition A = B = C case (equilateral triangle), identify test cases to verify the boundary.
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. For the boundary condition A2 + B2 = C2 case (right-angle triangle), identify test cases to verify the boundary.
a = 1, b = 2, c = 4
a = 2, b = 4, c = 8

h. For the non-triangle case, identify test cases to explore the boundary.
a = 5, b = 2, c = 7
a = 8, b = 1, c = 6

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

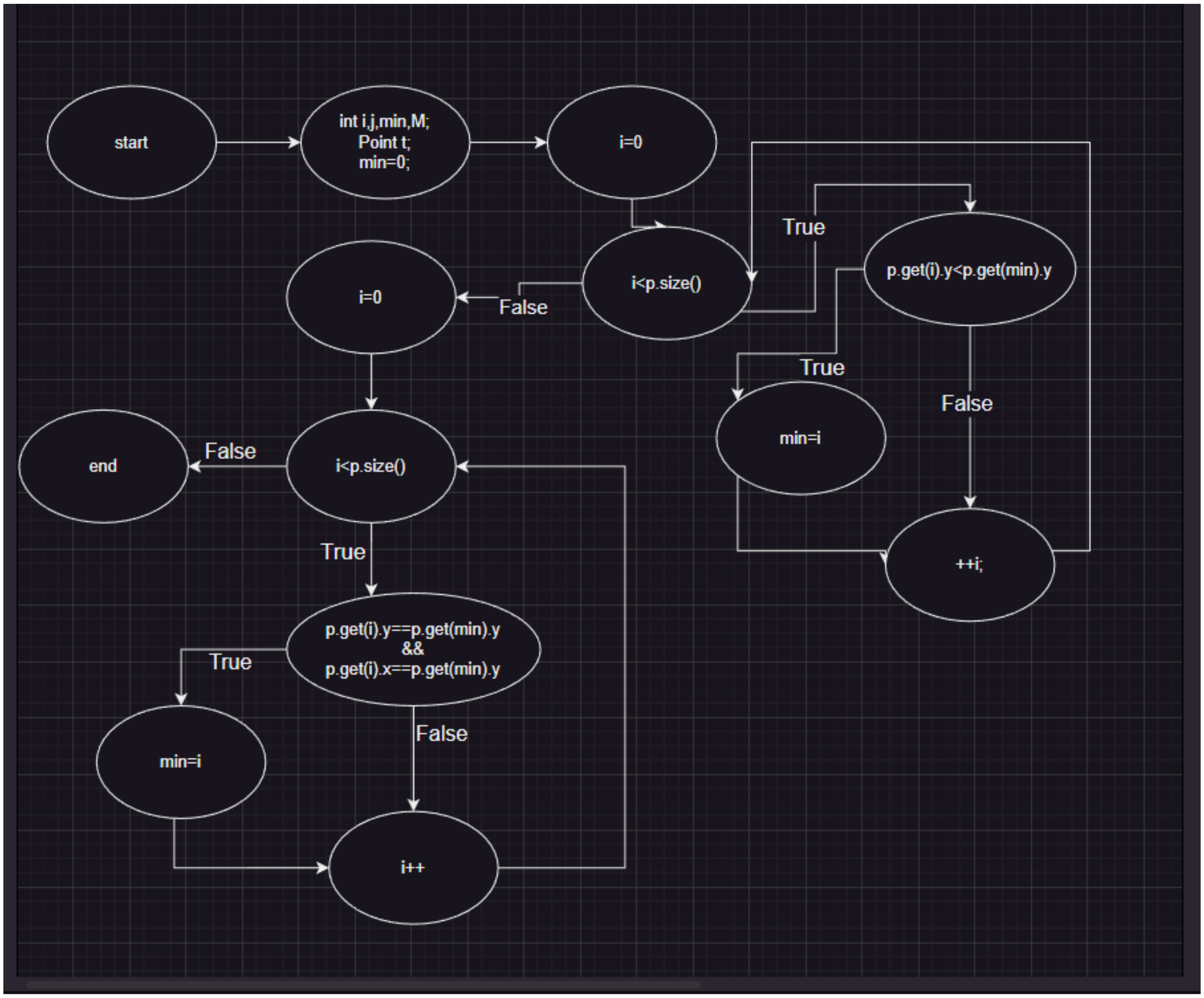
```
Vector doGraham(Vector p) {
        int i,j,min,M;

        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: To achieve statement coverage, we need to make sure that every statement in the code is executed at least once.

- 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

b. Branch Coverage: To achieve branch coverage, we need to make sure that every possible branch in the code is taken at least once

- 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 has 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

c. Basic Condition Coverage: To achieve basic condition coverage, we need to make sure that every basic condition in the code (i.e., every Boolean subexpression) is evaluated as both true and false at least once

- 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.