IT – 314

SOFTWARE ENGINEERING

Lab – 7

Name – Rushabh Patel

DAIICT ID – 202001419

Section – A

PROGRAMS

P1.

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

Equivalence Partitioning and Boundary Value Analysis

Equivalence Partitioning:

Tester action and Input Data	Expected Outcome
v is present in a	First occurring index of v
v is NOT present in a	-1

Boundary Value Analysis:

Tester action and Input Data	Expected Outcome
Empty array a	-1
v is present at the first index of a	0
v is present at the last index of a	a-1
v is not present in a	-1

Tests Suits:

Tester action and input data	value to be found (v)	Expected Outcome
Valid partitions:		
[1,3,5,7,9]	3	1
[1,3,5,7,9]	4	-1
[0,2,4,6,8]	5	-1
[0,2,4,6,8]	6	3
Boundary value Analysis:		
[]	9	-1
[9]	9	0
[9]	5	-1
[1,3,4]	1	0
[1,3,4]	4	2
[5,6,7]	5.5	Invalid input
[5,a,7]	5	Invalid input

```java
    @Test
    public void test1_1() {

    programs test = new programs();

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

    int output = test.linearSearch(2, a);
    assertEquals(1,output);
    }

    @Test
    public void test1_2() {

    programs test = new programs();

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

    int output = test.linearSearch(1, a);
    assertEquals(0,output);
    }


    @Test
    public void test1_3() {

    programs test = new programs();

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

    int output = test.linearSearch(7, a);
    assertEquals(-1,output);
    }
```

P2.

```
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 and Boundary Value Analysis

Equivalence Partitioning:

Tester action and Input Data	Expected Outcome
v is present in a	Number of times v appears in a
v is NOT present in a	-1

Boundary Value Analysis:

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

Tests Suits:

Tester action and input data	value to be found (v)	Expected Outcome
Valid partitions:		
[1,3,5,7,9]	3	1
[1,3,3,3,9]	3	3
[0,2,4,6,8]	5	0
Boundary value Analysis:		
[]		-1
[9]	9	0
[9]	5	-1
[1,3,4]	1	1
[4,3,4]	4	2
[5,6,7]	5.5	Invalid input
[5,a,7]	5	Invalid input

```java
    @Test
    public void test2_1() { // no of element p2

    programs test = new programs();

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

    int output = test.countItem(2, a);
    assertEquals(2,output);
    }

    @Test
    public void test2_2() { //no of element p2

    programs test = new programs();

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

    int output = test.countItem(4, a);
    assertEquals(2,output);
    }
    @Test
    public void test2_3() { //no of element p2

    programs test = new programs();

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

    int output = test.countItem(6, a);
    assertEquals(0,output);
    }
```

P3.

```
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 and Boundary Value Analysis

Equivalence Partitioning:

Tester action and Input Data	Expected Outcome
v is present in a	Index of v
v is NOT present in a	-1

Boundary Value Analysis:

Tester action and Input Data	Expected Outcome
Empty array a	-1
v is present at the first index of a	0
v is present at the last index of a	a-1
v is not present in a	-1

Tests Suits:

Tester action and input data	value to be found (v)	Expected Outcome
Valid partitions:		
[1,3,5,7,9]	3	1
[1,3,5,7,9]	4	-1
[0,2,4,6,8]	5	-1
[0,2,4,6,8]	6	3
Boundary value Analysis:		
[]	9	-1
[9]	9	0
[9]	5	-1
[1,3,4]	1	0
[1,3,4]	4	2
[5,6,7]	5.5	Invalid input
[5,a,7]	5	Invalid input

```java
@Test
public void test3_1() { //binary search p3

programs test = new programs();

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

int output = test.binarySearch(2, a);
assertEquals(1,output);
}

@Test
public void test3_2() { //binary search p3

programs test = new programs();

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

int output = test.binarySearch(3, a);
assertEquals(3,output);
}


@Test
public void test3_3() { //binary search p3

programs test = new programs();

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

int output = test.binarySearch(8, a);
assertEquals(-1,output);
}
```

P4.

```
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 and Boundary Value Analysis

Equivalence Partitioning:

Tester action and Input Data	Expected Outcome
a = b = c, where a, b, c are positive integers	Equilateral
a = b < c, where above condition is not satisfied	Isosceles
a + b > c, where above condition(s) is not satisfied	Scalene
a + b < c, where above condition(s) is not satisfied	Invalid

Boundary Value Analysis:

Tester action and Input Data	Expected Outcome
Empty array a	-1
v is present at the first index of a	0
v is present at the last index of a	a-1
v is not present in a	-1

Tests Suits:

Tester action and input data	value to be found (v)
Equivalence partitions:	
a = b = c > 0	Equivalent
a = b > c	isosceles
a < b+c, b < a+c, c < a+b	Scalene
a = b > 0, c = 0	Invalid
a > b + c	Invalid
[0,2,4,6,8]	6
Boundary value Analysis:	
a = 5, b = 5, c = 5	Equilateral
a = 5, b = 5, c = 6	Isosceles
a = 9, b = 9, c = 0	Invalid
a = int_max, b = int_max, c = int_max	Equilateral
a = 10, b = 2, c = 2	Invalid
a = 1, b = 1, c = -1	Invalid
a = int_max, b = int_max, c = int_max + 1	Invalid

```java
    @Test
    public void test4_1() {
    programs test = new programs();
    int output = test.triangle(8,8,8);
    assertEquals(0,output);
    }

    @Test
    public void test4_2() {
    programs test = new programs();
    int output = test.triangle(8,8,10);
    assertEquals(2,output);
    }

    @Test
    public void test4_3() {
    programs test = new programs();
    int output = test.triangle(0,0,0);
    assertEquals(1,output);
    }
```

P5.

```
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 and Boundary Value Analysis

Equivalence Partitioning:

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

Boundary Value Analysis:

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

Tests Suits:

Tester action and input data	value to be found (v)
Valid partitions:	
s1 = “abcd” s2 = “abcd”	True
s1 = “abc” s2 = “abcd”	True
s1 = “abd” s2 = “abcd”	False
s1 = “abcd” s2 = “acd”	False
s1 = “a” s2 = “abcd”	True
Boundary value Analysis:	
s1 = “abcd” s2 = “abcd”	True
s1 = “” s2 = “”	True
s1 = “” s2 = “abcd”	True
s1 = “abcd” s2 = “”	False
s1 = “abc” s2 = “abcd”	True
s1 = “abcd” s2 = “abc”	False
s1 = “abcf” s2 = “abcd”	False

```
    public void test5_1() {
    programs test = new programs();
    boolean output = test.prefix("","nonEmpty");
    assertEquals(true,output);
    }

    @Test
    public void test5_2() {  // example of s1 is prefix of s2
    programs test = new programs();
    boolean output = test.prefix("hello","hello world");
    assertEquals(true,output);
    }

    @Test
    public void test5_3() {  // example of s1 is not prefix of s2
    programs test = new programs();
    boolean output = test.prefix("hello","world hello");
    assertEquals(false,output);
    }
```

P6.

A.

EC1: Invalid inputs (negative or zero values)
EC2: Non-triangle (sum of the two shorter sides is not greater than the longest side)
EC3: Scalene triangle (no sides are equal)
EC4: Isosceles triangle (two sides are equal)
EC5: Equilateral triangle (all sides are equal)
EC6: Right-angled triangle (satisfies the Pythagorean theorem)

B.

Test Cases:
TC1: -1, 0
TC2: 1, 2, 5
TC3: 3, 4, 5
TC4: 5, 5, 7
TC5: 6, 6, 6
TC6: 3, 4, 5
Test case 1 covers class 1, test case 2 covers class 2, test case 3 covers class 3, test case 4 covers class 4, test case 5 covers class 5, and test case 6 covers class 6

C.

2, 3, 6
3, 4, 8
Both test cases have two sides that are shorter than the third side, and should not form a triangle

D.

1, 2, 1
0, 2, 0
5, 6, 5
Both test cases have two sides that are equal, but only test case 2 should form an isosceles triangle, other input are invalid.

E.

5, 5, 5
0, 0, 0
Both test cases have all sides equal, but only test case 1 should form an equilateral triangle, other input are invalid.

F.

3, 4, 5 0, 0, 0 -3, -4, -5 Both test cases satisfy the Pythagorean theorem, but only test case 1 should form right-angled triangle, other input are invalid. Triangle

G.

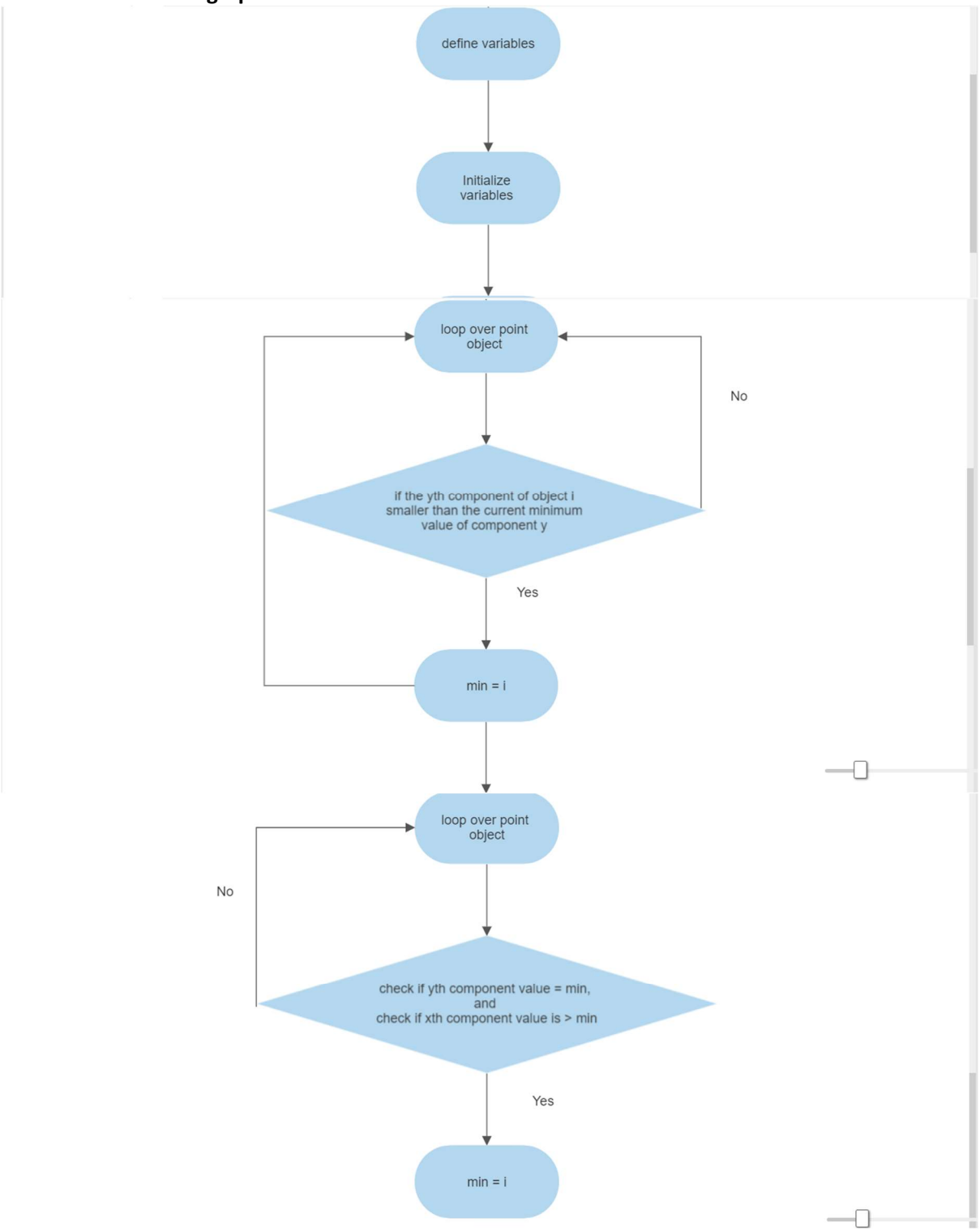
Test cases for the non-triangle case:
TC11 (EC3): A=2, B=2, C=4 (sum of A and B is less than C)

H.

Test points for non-positive input:
TP1 (EC2): A=0, B=4, C=5 (invalid input)
TP2 (EC2): A=-2, B=4, C=5 (invalid input)
Note: Test cases TC1 to TC10 covers all identified equivalence classes.

SECTION B

1. **Control flow graph**



2. **Test sets**

Statement coverage test sets: 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

Branch coverage test sets: 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 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

Basic condition coverage test sets: 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.