IT-314 LAB 07

Software engineering

Name : Nisarg Ashok Kumar Jadav
ID : 202001010

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?

Tester Action and Input Data	Expected Outcome
Equivalence Partitioning	
Input (2, 3, 2010)	Previous date: (1, 3, 2010)
Input (32, 13, 2010)	Invalid date
Input (29, 2, 2011)	Invalid date
Input (29, 2, 2012)	Previous date: (28, 2, 2012)

Boundary Value Analysis	
Input (1, 3, 2010)	Previous date: (28, 2, 2010)
Input (31, 3, 2010)	Previous date: (30, 3, 2010)
Input (1, 1, 2010)	Previous date: (31, 12, 2009)
Input (31, 12, 2010)	Previous date: (30, 12, 2010)
Input (1, 3, 1900)	Previous date: (28, 2, 1900)
Input (31, 12, 2015)	Previous date: (30, 12, 2015)
Input (1, 1, 1900)	Invalid date
Input (31, 12, 2014)	Previous date: (30, 12, 2014)

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.

Tester Action and Input Data	Expected Outcome
Equivalence Partitioning	
`linearSearch(3, [1, 2, 3, 4])`	2
`linearSearch(5, [1, 2, 3, 4])`	-1
`linearSearch(3, [])`	-1
`linearSearch(3, [3, 3, 3])`	0
`linearSearch(0, [1, 2, 3])`	-1
Boundary Value Analysis	
`linearSearch(1, [1, 2, 3, 4])`	0
`linearSearch(4, [1, 2, 3, 4])`	3
`linearSearch(1, [1])`	0
`linearSearch(2, [1])`	-1
`linearSearch(0, [0])`	0
`linearSearch(0, [1])`	-1

CODE:

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

    public static void main(String[] args) {
        System.out.println("Test cases for Equivalence Partitioning: ");
        int v1 = 7;
        int[] a1 = {2, 4, 7, 9, 11};
        System.out.println("\nTest Case 1: " + linearSearch(v1, a1)); // Expected : 2

        int v2 = 6;
        int[] a2 = {3, 8, 12, 15};
        System.out.println("\nTest Case 2: " + linearSearch(v2, a2)); // Expected : -1

        int v3 = 10;
        int[] a3 = {10, 20, 30, 40};
        System.out.println("\nTest Case 3: " + linearSearch(v3, a3)); // Expected : 0

        System.out.println("\nTest cases for Boundary Value Analysis: ");
        int v4 = 5;
        int[] a4 = {};
        System.out.println("\nTest Case 4: " + linearSearch(v4, a4)); // Expected : -1

        int v5 = 25;
        int[] a5 = {10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85, 90, 95,
100};
        System.out.println("\nTest Case 5: " + linearSearch(v5, a5)); // Expected : 3
```

```
        int v6 = 200;
        int[] a6 = {50, 100, 150, 250, 300, 350, 400, 450, 500};
        System.out.println("\nTest Case 6: " + linearSearch(v6, a6)); // Expected : -1

        int v7 = 60;
        int[] a7 = {30, 40, 50, 60};
        System.out.println("\nTest Case 7: " + linearSearch(v7, a7)); // Expected : 3

        int v8 = 5;
        int[] a8 = {5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85, 90, 95,
100};
        System.out.println("\nTest Case 8: " + linearSearch(v8, a8)); // Expected : 0
    }
}
```

OUTPUT:

Test cases for Equivalence Partitioning:

Test Case 1: 2
Test Case 2: -1
Test Case 3: 0
Test cases for Boundary Value Analysis:

Test Case 4: -1
Test Case 5: 3
Test Case 6: -1
Test Case 7: 3
Test Case 8: 0

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

Tester Action and Input Data	Expected Outcome
Equivalence Partitioning	
`countItem(3, [1, 2, 3, 4])`	1
`countItem(5, [1, 2, 3, 4])`	0
`countItem(3, [])`	0
`countItem(3, [3, 3, 3])`	3
`countItem(0, [1, 2, 3])`	0
Boundary Value Analysis	
`countItem(1, [1])`	1
`countItem(2, [1])`	0
`countItem(0, [0])`	1
`countItem(0, [1])`	0

CODE:

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

Output:

Test cases for Equivalence Partitioning:

Test Case 1: 1
Test Case 2: 0
Test Case 3: 1

Test cases for Boundary Value Analysis:

Test Case 4: 0
Test Case 5: 1
Test Case 6: 7
Test Case 7: 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.**

Tester Action and Input Data	Expected Outcome
Equivalence Partitioning	
`binarySearch(3, [1, 2, 3, 4])`	2
`binarySearch(5, [1, 2, 3, 4])`	-1
`binarySearch(3, [])`	-1
`binarySearch(0, [1, 2, 3])`	-1
Boundary Value Analysis	
`binarySearch(1, [1])`	0
`binarySearch(2, [1])`	-1
`binarySearch(1, [1, 2])`	0
`binarySearch(2, [1, 2])`	1
`binarySearch(0, [0])`	0
`binarySearch(0, [1])`	-1

CODE:

```
public static 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);
}
```

OUTPUT:

Test cases for Equivalence Partitioning:

Test Case 1: 4
Test Case 2: -1
Test Case 3: 0

Test cases for Boundary Value Analysis:

Test Case 4: -1
Test Case 5: 0
Test Case 6: 4
Test Case 7: -1

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).

Tester Action and Input Data	Expected Outcome
Equivalence Partitioning	
`triangle(3, 3, 3)`	`EQUILATERAL`
`triangle(3, 4, 5)`	`SCALENE`
`triangle(3, 3, 5)`	`ISOSCELES`
`triangle(1, 2, 3)`	`INVALID`
Boundary Value Analysis	
`triangle(1, 1, 1)`	`EQUILATERAL`
`triangle(1, 1, 2)`	`ISOSCELES`
`triangle(1, 2, 2)`	`ISOSCELES`
`triangle(2, 2, 3)`	`ISOSCELES`
`triangle(1, 2, 3)`	`INVALID`

CODE:

```
public 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);
}
```

OUTPUT:

Test cases for Equivalence Partitioning:

Test Case 1: Equilateral triangle
Test Case 2: Isosceles triangle
Test Case 3: Scalene triangle
Test Case 4: Invalid triangle
Test Case 5: Invalid triangle

Test cases for Boundary Value Analysis:

Test Case 6: Equilateral triangle
Test Case 7: Isosceles triangle
Test Case 8: Isosceles triangle
Test Case 9: Isosceles triangle
Test Case 10: Scalene triangle
Test Case 11: Scalene triangle
Test Case 12: Invalid triangle
Test Case 13: Invalid triangle
Test Case 14: Invalid triangle

Tester Action and Input Data	**Expected Outcome**
Test with matching prefix	True
prefix("hello", "helloworld")	
Test with non-matching prefix	False
prefix("hello", "goodbye")	
Test with empty prefix	True
prefix("", "hello")	
Test with empty string	False
prefix("hello", "")	
Test with same string	True
prefix("hello", "hello")	
Test with one-letter prefix	True
prefix("h", "hello")	
Test with one-letter non-matching	False
prefix("h", "goodbye")	

CODE:

```
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;
}
```

Output:

Test cases for Equivalence Partitioning:

Test Case 1: true
Test Case 2: false
Test Case 3: true
Test Case 4: NullPointerException
Test Case 5: NullPointerException
Test Case 6: NullPointerException

Test cases for Boundary Value Analysis:

Test Case 7: true
Test Case 8: true
Test Case 9: true
Test Case 10: false
Test Case 11: false

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

Equivalence classes:

- Non-positive input: Any side of the triangle is less than or equal to 0.
- Non-triangle: The sum of any two sides is less than or equal to the third side.
- Equilateral triangle: All three sides are equal.
- Isosceles triangle: Two sides are equal and the third side is different.
- Scalene triangle: All three sides are different.
- Right-angle triangle: The square of one side is equal to the sum of the squares of the other two sides.

b) Identify test cases to cover the identified equivalence classes. Also, explicitly mention which test case would cover which equivalence class.

Tester Action and Input Data	**Expected Outcome**	**Equivalence Class**
`triangle(-1.0, 2.0, 3.0)`	Invalid input	Non-positive input
`triangle(1.0, 2.0, 3.0)`	Not a triangle	Non-triangle
`triangle(3.0, 3.0, 3.0)`	Equilateral	Equilateral triangle
`triangle(3.0, 3.0, 4.0)`	Isosceles	Isosceles triangle

`triangle(3.0, 4.0, 5.0)`	Scalene	Scalene triangle
`triangle(3.0, 4.0, 5.0)`	Right-angle	Right-angle triangle

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

Tester Action and Input Data	Expected Outcome
`triangle(2.99, 3.01, 6.0)`	Not a triangle
`triangle(3.01, 3.01, 6.0)`	Scalene

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

Tester Action and Input Data	Expected Outcome
`triangle(3.99, 4.01, 4.01)`	Isosceles
`triangle(4.01, 4.01, 4.01)`	Equilateral

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

Tester Action and Input Data	**Expected Outcome**
`triangle(3.99, 3.99, 3.99)`	Equilateral
`triangle(3.99, 3.99, 4.01)`	Isosceles

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

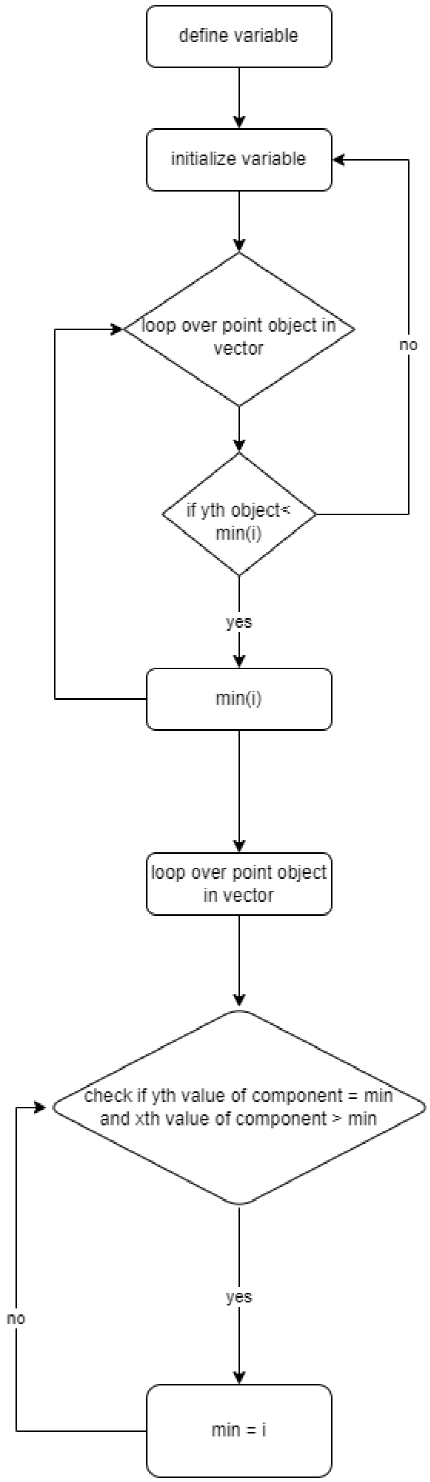
Tester Action and Input Data	Expected Outcome
`triangle(3.0, 4.0, 5.0)`	Right-angle
`triangle(5.0, 4.0, 3.0)`	Right-angle

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

Tester Action and Input Data	**Expected Outcome**
`triangle(1.99, 2.01, 4.0)`	Not a triangle
`triangle(2.01, 2.01, 4.0)`	Scalene

Section B

1) control flow graph



2)

A. Statement Coverage:

Test Case	Input	Expected Output
1	`p = []`	Empty vector
2	`p = [(1,1)]`	Vector with single point
3	`p = [(1,1), (2,2)]`	Vector with two points
4	`p = [(1,1), (2,2), (3,1)]`	Vector with three points
5	`p = [(1,1), (2,2), (3,1), (4,3)]`	Vector with four point

B. Branch Coverage:

Test Case	Input	Expected Output
1	`p = []`	Empty vector
2	`p = [(1,1)]`	Vector with single point
3	`p = [(1,1), (2,2)]`	Vector with two points
4	`p = [(1,1), (2,2), (3,1)]`	Vector with three points
5	`p = [(1,1), (2,2), (3,1), (4,3)]`	Vector with four points
6	`p = [(1,2), (3,1), (2,1)]`	Vector with three points in different order

C. Basic Condition Coverage:

Test Case	Input	Expected Output
1	p = []	Empty vector
2	p = [(1,1)]	Vector with single point
3	p = [(1,1), (2,2)]	Vector with two points
4	p = [(1,1), (2,2), (3,1)]	Vector with three points
5	p = [(1,1), (2,2), (3,1), (4,3)]	Vector with four points
6	p = [(1,2), (3,1), (2,1)]	Vector with three points in different order
7	p = [(1,1), (1,1), (1,1)]	Vector with three identical points
8	p = [(1,1), (2,2), (1,1)]	Vector with two identical points
9	p = [(1,1), (1,2), (2,1)]	Vector with two points with same y component