Dhirubhai Ambani
Institute of Information and Communication Technology

Software Engineering - IT314
Assignment

Name : Heman Chauhan

Student ID : 202201267

Task 1

Q.1. 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?

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, 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.

The solution of each problem must be given in the format as follows:

Tester Action and Input Data	***Expected Outcome***
Equivalence Partitioning	
a, b, c	An Error message
a-1, b, c	Yes
Boundary Value Analysis	
a, b, c-1	Yes

1. Test cases using Equivalence Partitioning (EP):

1. Valid Input Combinations (Expected: Returns correct previous date):

2. **Day in the middle of the month and year in range**:

➔ Input: Month = 5, Day = 15, Year = 2010
➔ Input: Month = 11, Day = 10, Year = 1995

3. **Last day of a month**:

➔ Input: Month = 7, Day = 31, Year = 2000
➔ Input: Month = 10, Day = 31, Year = 2015

4. **Month = 12 (December) and valid year**:

➔ Input: Month = 12, Day = 5, Year = 2012
➔ Input: Month = 12, Day = 31, Year = 1999

5. **Leap Year February 29**:

➔ Input: Month = 2, Day = 29, Year = 2000 (Leap year)
➔ Input: Month = 2, Day = 28, Year = 2001 (non-leap year)

2. Invalid Input Combinations (Expected: Error message):

1. **Month out of range**:

➔ Input: Month = 0, Day = 15, Year = 2010
➔ Input: Month = 13, Day = 10, Year = 2020

2. **Day out of range**:

➔ Input: Month = 4, Day = 32, Year = 2005
➔ Input: Month = 11, Day = 0, Year = 1999

3. **Year out of range**:

➔ Input: Month = 6, Day = 15, Year = 1899
➔ Input: Month = 8, Day = 12, Year = 2016

4. **Invalid days in specific months**:

➔ Input: Month = 2, Day = 30, Year = 2010 (February has a maximum of 29 days)
➔ Input: Month = 4, Day = 31, Year = 2007 (April has only 30 days)

2. Test cases using Boundary Value Analysis:

1. Boundary Cases for Month:

2. **Lower Bound (Minimum Month)**:

➔ Input: Month = 1, Day = 1, Year = 1900
➔ Input: Month = 1, Day = 31, Year = 2000

3. **Upper Bound (Maximum Month)**:

➔ Input: Month = 12, Day = 1, Year = 2015
➔ Input: Month = 12, Day = 31, Year = 2015

2. Boundary Cases for Day:

1. **Lower Bound (Minimum Day)**:

➔ Input: Month = 3, Day = 1, Year = 2010
➔ Input: Month = 8, Day = 1, Year = 1901

2. **Upper Bound (Maximum Day)**:

➔ Input: Month = 7, Day = 31, Year = 1999
➔ Input: Month = 12, Day = 31, Year = 2014

3. Boundary Cases for Year:

1. **Lower Bound (Minimum Year)**:

➔ Input: Month = 5, Day = 10, Year = 1900
➔ Input: Month = 1, Day = 1, Year = 1900

2. **Upper Bound (Maximum Year)**:

➔ Input: Month = 8, Day = 23, Year = 2015
➔ Input: Month = 12, Day = 31, Year = 2015

3. Additional Test Cases:

1. **Edge Case for Leap Year Transition**:

➔ Input: Month = 2, Day = 29, Year = 2000 (Leap year, valid day)
➔ Input: Month = 2, Day = 28, Year = 2001 (non-leap year, valid day)
➔ Expected: Returns correct previous date in both cases.

2. **Edge Case for Year 1900 (Minimum)**:

➔ Input: Month = 1, Day = 1, Year = 1900
➔ Expected: Error (there's no date before January 1, 1900).

3. **Transition from December 31 to January 1 of the Next Year**:

➔ Input: Month = 12, Day = 31, Year = 2001
➔ Expected: Previous date is December 30, 2001.

4. **Invalid February 30 Case**:

➔ Input: Month = 2, Day = 30, Year = 2015
➔ Expected: Error, since February has at most 29 days.

List of Test Cases:

Tester Action and Input Data:

Test Action and Input Data	Expected Outcome
Equivalence Partitioning	
Month = 5, Day = 15, Year = 2010	Yes
Month = 11, Day = 10, Year = 1995	Yes
Month = 7, Day = 31, Year = 2000	Yes
Month = 12, Day = 31, Year = 1999	Yes
Month = 0, Day = 15, Year = 2010	An error message
Month = 2, Day = 30, Year = 2010	An error message
Month = 4, Day = 31, Year = 2007	An error message
Boundary Value Analysis	
Month = 1, Day = 1, Year = 1900	Yes
Month = 12, Day = 31, Year = 2015	Yes
Month = 3, Day = 1, Year = 2010	Yes
Month = 7, Day = 31, Year = 1999	Yes
Month = 2, Day = 29, Year = 2000	Yes (Leap Year case)
Month = 2, Day = 28, Year = 2001	Yes (Non-Leap Year case)
Month = 8, Day = 23, Year = 2015	Yes
Additional Edge Cases	
Month = 2, Day = 30, Year = 2015	An error message
Month = 12, Day = 31, Year = 2001	Yes
Month = 1, Day = 1, Year = 1900	An error message (No Previous Date)

Code:

```java
public class PreviousDate {

    // Function to check leap year
    public static boolean isLeapYear(int year) {
        return (year % 4 == 0 && year % 100 != 0) || (year % 400 == 0);
    }

    // Function to return the number of days in a month
    public static int daysInMonth(int month, int year) {
        switch (month) {
            case 2:
                return isLeapYear(year) ? 29 : 28;
            case 4: case 6: case 9: case 11:
                return 30;
            default:
                return 31;
        }
    }

    // Function to calculate the previous date
    public static String previousDate(int day, int month, int year) {
        // Check for invalid year, month, or day
        if (year < 1900 || year > 2015 || month < 1 || month > 12 || day < 1 || day > 31) {
            return "Invalid date";
        }

        int daysInThisMonth = daysInMonth(month, year);

        if (day > daysInThisMonth) {
            return "Invalid date";
        }

        // Handle case where the day is 1
        if (day == 1) {
            // Move to the previous month
            if (month == 1) { // If January, move to December of the previous year
                year -= 1;
                month = 12;
            } else {
                month -= 1;
            }
            day = daysInMonth(month, year);
        } else {
            day -= 1;
        }

        // Check for valid date range
        if (year < 1900 || year > 2015) {
            return "Invalid date";
        }

        return day + "/" + month + "/" + year;
    }

    public static void main(String[] args) {
        // Example: Testing the function
        System.out.println(previousDate(1, 3, 2021));  // Output should be 28/2/2021
    }
}
```

Task 2

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 (EP):

1. Valid Partition:

➔ **Test Case: v exists in the array a**

Input: a = [1, 2, 3, 4, 5], v = 3

Expected Outcome: Returns 2 (the index of 3)

2. Invalid Partitions:

➔ **Test Case: v does not exist in the array a**

Input: a = [1, 2, 3, 4, 5], v = 6

Expected Outcome: Returns -1

➔ **Test Case: The input array a is empty**

Input: a = [], v = 1

Expected Outcome: Returns -1

➔ **Test Case: Input value v is negative or not an integer**

Input: a = [1, 2, 3], v = -1

Expected Outcome: Returns -1

Boundary Value Analysis (BVA):

1. **Lower Boundary:**

➔ **Test Case: Searching for a value at the first index**

Input: a = [1, 2, 3], v = 1

Expected Outcome: Returns 0 (the index of 1)

2. **Upper Boundary:**

➔ **Test Case: Searching for a value at the last index**

Input: a = [1, 2, 3], v = 3

Expected Outcome: Returns 2 (the index of 3)

3. **Boundary Condition with Empty Array:**

➔ **Test Case: Searching in an empty array**

Input: a = [], v = 1

Expected Outcome: Returns -1

4. **Boundary Case of Not Found Value:**

➔ **Test Case: Searching for a value that is just outside the array bounds**

Input: a = [1, 2, 3], v = 4

Expected Outcome: Returns -1

Testing Method	Tester Action and Input Data	Expected Outcome
Equivalence Partitioning		
Valid Input	**a = [1, 2, 3, 4, 5], v = 3**	**2**
Invalid Input	**a = [1, 2, 3, 4, 5], v = 6**	**-1**
Invalid Input	**a = [], v = 1**	**-1**
Invalid Input	**a = [1, 2, 3], v = -1**	**-1**
Boundary Value Analysis		
Lower Boundary	**a = [1, 2, 3], v = 1**	**0**
Upper Boundary	**a = [1, 2, 3], v = 3**	**2**
Empty Array	**a = [], v = 1**	**-1**
Not Found Value	**a = [1, 2, 3], v = 4**	**-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);
```

}

Equivalence Partitioning (EP):

1. **Valid Partitions:**

➔ **Test Case: Value exists in the array multiple times**

Input: a = [1, 2, 3, 2, 4], v = 2

Expected Outcome: Returns 2 (since 2 appears twice)

➔ **Test Case: Value exists in the array exactly once**

Input: a = [1, 2, 3, 4, 5], v = 3

Expected Outcome: Returns 1 (since 3 appears once)

➔ **Test Case: Value does not exist in the array**

Input: a = [1, 2, 3, 4, 5], v = 6

Expected Outcome: Returns 0 (since 6 does not appear)

2. **Invalid Partitions:**

➔ **Test Case: The input array a is empty**

Input: a = [], v = 1

Expected Outcome: Returns 0 (no elements to count)

Boundary Value Analysis (BVA):

1. **Lower Boundary:**

➔ **Test Case: Searching for a value at the first index**

Input: a = [1], v = 1

Expected Outcome: Returns 1 (since 1 appears once)

2. **Upper Boundary:**

➔ **Test Case: Searching for a value at the last index**

Input: a = [1, 2, 3, 4, 5], v = 5

Expected Outcome: Returns 1 (since 5 appears once)

3. Multiple Appearances:

➔ **Test Case: Searching for a value that appears multiple times at the end**

Input: a = [1, 2, 2, 2, 2], v = 2

Expected Outcome: Returns 4 (since 2 appears four times)

4. Boundary Case with Empty Array:

➔ **Test Case: Searching in an empty array**

Input: a = [], v = 1

Expected Outcome: Returns 0

Equivalence Partitioning (EP):

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

Boundary Value Analysis (BVA):

Tester Action and Input Data	Expected Outcome
a = [1], v = 1	1
a = [1, 2, 3, 4, 5], v = 5	1
a = [1, 2, 2, 2, 2], v = 2	4
a = [], v = 1	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.

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 (EP):

1. Valid Partitions:

➔ **Test Case: Value exists in the array**

Input: a = [1, 2, 3, 4, 5], v = 3

Expected Outcome: Returns 2 (the index of 3)

➔ **Test Case: Value exists at the first index**

Input: a = [1, 2, 3, 4, 5], v = 1

Expected Outcome: Returns 0 (the index of 1)

➔ **Test Case: Value exists at the last index**

Input: a = [1, 2, 3, 4, 5], v = 5

Expected Outcome: Returns 4 (the index of 5)

2. Invalid Partitions:

➔ **Test Case: Value does not exist in the array**

Input: a = [1, 2, 3, 4, 5], v = 6

Expected Outcome: Returns -1 (since 6 is not in the array)

➔ **Test Case: Searching for a value less than the smallest element**

Input: a = [1, 2, 3, 4, 5], v = 0

Expected Outcome: Returns -1 (since 0 is not in the array)

➔ **Test Case: Searching for a value greater than the largest element**

Input: a = [1, 2, 3, 4, 5], v = 10

Expected Outcome: Returns -1 (since 10 is not in the array)

➔ **Test Case: The input array a is empty**

Input: a = [], v = 1

Expected Outcome: Returns -1 (no elements to search)

Boundary Value Analysis (BVA):

1. **Lower Boundary:**

➔ **Test Case: Searching for the smallest element**

Input: a = [1], v = 1

Expected Outcome: Returns 0 (the index of 1)

2. **Upper Boundary:**

➔ **Test Case: Searching for the largest element**

Input: a = [1], v = 1

Expected Outcome: Returns 0 (the index of 1)

➔ **Test Case: Searching in an array of size 2**

Input: a = [1, 2], v = 2

Expected Outcome: Returns 1 (the index of 2)

3. **Middle Element Search:**

➔ **Test Case: Searching for a value that is in the middle of a larger array**

Input: a = [1, 2, 3, 4, 5], v = 3

Expected Outcome: Returns 2 (the index of 3)

4. **Boundary Case with Empty Array:**

➔ **Test Case: Searching in an empty array**

Input: a = [], v = 1

Expected Outcome: Returns -1

Equivalence Partitioning (EP):

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

Boundary Value Analysis (BVA):

Tester Action and Input Data	Expected Outcome
a = [1], v = 1	0
a = [1, 2], v = 2	1
a = [1, 2, 3, 4, 5], v = 3	2
a = [], v = 1	-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).

```
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 (EP):

1. Valid Partitions:

➔ **Equilateral Triangle: All three sides are equal.**

Test Case: a = 3, b = 3, c = 3

Expected Outcome: Returns 0 (EQUILATERAL)

➔ **Isosceles Triangle: Two sides are equal.**

Test Case: a = 3, b = 3, c = 2

Expected Outcome: Returns 1 (ISOSCELES)

➔ **Scalene Triangle: All sides are different.**

Test Case: a = 3, b = 4, c = 5

Expected Outcome: Returns 2 (SCALENE)

2. **Invalid Partitions:**

➔ **Invalid Triangle: The sum of any two sides must be greater than the third side.**

Test Case: a = 1, b = 2, c = 3

Expected Outcome: Returns 3 (INVALID)

➔ **Invalid Triangle: Any side must be zero or negative.**

Test Case: a = 0, b = 1, c = 1

Expected Outcome: Returns 3 (INVALID)

➔ **Invalid Triangle: All sides are negative.**

Test Case: a = -1, b = -1, c = -1

Expected Outcome: Returns 3 (INVALID)

Boundary Value Analysis (BVA):

1. **Minimum Valid Triangle:**

➔ **Test Case: The smallest valid triangle**

Input: a = 1, b = 1, c = 1

Expected Outcome: Returns 0 (EQUILATERAL)

2. **Boundary Condition for Invalid Triangle:**

➔ **Test Case: Sum of two sides equals the third side**

Input: a = 1, b = 2, c = 3

Expected Outcome: Returns 3 (INVALID)

➔ **Test Case: One side is zero**

Input: a = 0, b = 2, c = 2

Expected Outcome: Returns 3 (INVALID)

➔ **Test Case: One side is negative**

Input: a = -1, b = 2, c = 2

Expected Outcome: Returns 3 (INVALID)

Equivalence Partitioning (EP):

Tester Action and Input Data	Expected Outcome
a = 3, b = 3, c = 3	0 (EQUILATERAL)
a = 3, b = 3, c = 2	1 (ISOSCELES)
a = 3, b = 4, c = 5	2 (SCALENE)
a = 1, b = 2, c = 3	3 (INVALID)
a = 0, b = 1, c = 1	3 (INVALID)
a = -1, b = -1, c = -1	3 (INVALID)

Boundary Value Analysis (BVA):

Tester Action and Input Data	Expected Outcome
a = 1, b = 1, c = 1	0 (EQUILATERAL)
a = 1, b = 2, c = 3	3 (INVALID)
a = 0, b = 2, c = 2	3 (INVALID)
a = -1, b = 2, c = 2	3 (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;
}
```

Equivalence Partitioning (EP):

1. Valid Partitions:

➔ **Test Case: s1 is an exact match of s2.**

Input: s1 = "hello", s2 = "hello"

Expected Outcome: Returns true (because s1 is a prefix of s2)

➔ **Test Case: s1 is a prefix of s2.**

Input: s1 = "hello", s2 = "hello world"

Expected Outcome: Returns true (because s1 is a prefix of s2)

➔ **Test Case: s1 is an empty string, and s2 is a non-empty string.**

Input: s1 = "", s2 = "hello"

Expected Outcome: Returns true (an empty string is considered a prefix of any string)

2. Invalid Partitions:

➔ **Test Case: s1 is longer than s2.**

Input: s1 = "hello world", s2 = "hello"

Expected Outcome: Returns false (because s1 cannot be a prefix of a shorter string)

➔ **Test Case: s1 is not a prefix of s2.**

Input: s1 = "world", s2 = "hello world"

Expected Outcome: Returns false (because s1 does not match the beginning of s2)

➔ **Test Case: Both strings are empty.**

Input: s1 = "", s2 = ""

Expected Outcome: Returns true (an empty string is a prefix of another empty string)

Boundary Value Analysis (BVA):

1. Minimum Valid Prefix:

➔ **Test Case: s1 has a length of 1, and s2 has the same first character.**

Input: s1 = "h", s2 = "hello"

Expected Outcome: Returns true (because s1 is a prefix of s2)

2. **Minimum Invalid Prefix:**

➔ **Test Case: s1 has a length of 1, and s2 starts with a different character.**

Input: s1 = "w", s2 = "hello"

Expected Outcome: Returns false (because s1 does not match the beginning of s2)

3. **Edge Cases with Similar Lengths:**

➔ **Test Case: s1 is equal to s2, but longer.**

Input: s1 = "hello world", s2 = "hello world"

Expected Outcome: Returns true (exact match)

➔ **Test Case: s1 is shorter than s2, but is not a prefix.**

Input: s1 = "hel", s2 = "world"

Expected Outcome: Returns false (because s1 does not match the beginning of s2)

Equivalence Partitioning (EP):

Tester Action and Input Data	Expected Outcome
s1 = "hello", s2 = "hello"	True
s1 = "hello", s2 = "hello world"	True
s1 = "", s2 = "hello"	True
s1 = "hello world", s2 = "hello"	False
s1 = "world", s2 = "hello world"	False
s1 = "", s2 = ""	True

Boundary Value Analysis (BVA):

Tester Action and Input Data	Expected Outcome
s1 = "h", s2 = "hello"	True
s1 = "w", s2 = "hello"	False
s1 = "hello world", s2 = "hello world"	True
s1 = "hel", s2 = "world"	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
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)
c) For the boundary condition A + B > C case (scalene triangle), identify test cases to verify the boundary.
d) For the boundary condition A = C case (isosceles triangle), identify test cases to verify the boundary.
e) For the boundary condition A = B = C case (equilateral triangle), identify test cases to verify the boundary.
f) For the boundary condition $A_2 + B_2 = C_2$ case (right-angle triangle), identify test cases to verify the boundary.
g) For the non-triangle case, identify test cases to explore the boundary.
h) For non-positive input, identify test points.

(a). Identify the Equivalence Classes for the System:

The equivalence classes for the triangle classification based on the sides A, B, and C can be identified as follows:

1. Valid Equivalence Classes:

➔ **Equilateral Triangle: A=B=C**
➔ **Isosceles Triangle: A=B≠C or A=C≠B or B=C≠A**
➔ **Scalene Triangle: A≠B≠C and satisfies the triangle inequality A+B>C, A+C>B, B+C>A**
➔ **Right-Angled Triangle: A2+B2=C2 permutations thereof (where CCC is the longest side)**

2. Invalid Equivalence Classes:

➔ **Non-Triangle: Any configuration where the triangle inequality is not satisfied (A+B≤C or permutations thereof)**
➔ **Non-positive Input: At least one side is less than or equal to zero**

(b). Identify Test Cases to Cover the Identified Equivalence Classes:

Equivalence Class	Test Case	Input Values	Expected Outcome
Equilateral Triangle	Test case for equilateral triangle	A = 5.0, B = 5.0, C = 5.0	EQUILATERAL (0)
Isosceles Triangle	Test case for isosceles triangle	A = 5.0, B = 5.0, C = 3.0	ISOSCELES (1)
Scalene Triangle	Test case for scalene triangle	A = 3.0, B = 4.0, C = 5.0	SCALENE (2)
Right-Angled Triangle	Test case for right-angled triangle	A = 3.0, B = 4.0, C = 5.0	RIGHT ANGLED (2)
Non-Triangle (invalid)	Test case for non-triangle	A = 1.0, B = 2.0, C = 3.0	INVALID (3)
Non-positive Input	Test case for non-positive input	A = 0.0, B = 2.0, C = 2.0	INVALID (3)
Non-positive Input	Test case for negative input	A = -1.0, B = 2.0, C = 2.0	INVALID (3)

(c). Boundary Condition for A + B > C Case (Scalene Triangle):

Test Case	Input Values	Expected Outcome
Just above the boundary	A = 2.0, B = 3.0, C = 4.0	SCALENE (2)
Exactly at the boundary	A = 3.0, B = 4.0, C = 7.0	INVALID (3)
Just below the boundary	A = 3.0, B = 4.0, C = 8.0	INVALID (3)

(d). Boundary Condition for A = C Case (Isosceles Triangle):

Test Case	Input Values	Expected Outcome
Exactly equal	A = 4.0, B = 2.0, C = 4.0	ISOSCELES (1)
Just above the boundary	A = 4.1, B = 2.0, C = 4.0	ISOSCELES (1)
Just below the boundary	A = 4.0, B = 2.0, C = 3.9	SCALENE (2)

(e) Boundary Condition for A = B = C Case (Equilateral Triangle):

Test Case	Input Values	Expected Outcome
Exactly equal	A = 5.0, B = 5.0, C = 5.0	EQUILATERAL (0)
Just above the boundary	A = 5.1, B = 5.1, C = 5.1	EQUILATERAL (0)
Just below the boundary	A = 4.9, B = 4.9, C = 4.9	EQUILATERAL (0)

(f). Boundary Condition for $A^2 + B^2 = C^2$ Case (Right-Angled Triangle)

Test Case	Input Values	Expected Outcome
Exactly equal	A = 3.0, B = 4.0, C = 5.0	RIGHT ANGLED (2)
Just above the boundary	A = 3.0, B = 4.0, C = 5.1	SCALENE (2)

Just below the boundary	**A = 3.0, B = 4.0, C = 4.9**	**SCALENE (2)**

(g). Non-Triangle Case: Identify Test Cases to Explore the Boundary:

Test Case	Input Values	Expected Outcome
Exactly at the boundary	**A = 1.0, B = 2.0, C = 3.0**	**INVALID (3)**
Just below the boundary	**A = 1.0, B = 2.0, C = 3.1**	**INVALID (3)**
Two sides equal and third side invalid	**A = 1.0, B = 1.0, C = 2.0**	**INVALID (3)**

(h). Non-positive Input: Identify Test Points:

Test Case	Input Values	Expected Outcome
One side zero	**A = 0.0, B = 1.0, C = 1.0**	**INVALID (3)**
One side negative	**A = -1.0, B = 1.0, C = 1.0**	**INVALID (3)**
All sides zero	**A = 0.0, B = 0.0, C = 0.0**	**INVALID (3)**
All sides negative	**A = -1.0, B = -1.0, C = -1.0**	**INVALID (3)**