IT-314

LAB - 8
FUNCTIONAL TESTING

Name: Anuj k Valambhiya

ID: 20221481

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.

Equivalence Classes:

Equivalence Class	Range / Value	Description
EC1	1 ≤ month ≤ 12	Valid months
EC2	month < 1	Invalid, below minimum month

EC3	month > 12	Invalid, above maximum month
EC4	1 ≤ day ≤ 31	Valid for months with 31 days (Jan, Mar, May, Jul, Aug, Oct, Dec)
EC5	1 ≤ day ≤ 30	Valid for months with 30 days (Apr, Jun, Sep, Nov)
EC6	1 ≤ day ≤ 29	Valid for February in leap years
EC7	1 ≤ day ≤ 28	Valid for February in non-leap years
EC8	day < 1	Invalid, below minimum day
EC9	day > 31	Invalid, above maximum day for any month
EC10	day = 31 in Apr, Jun, Sep, Nov	Invalid, 31st day in a 30-day month
EC11	day = 30 or 31 in February	Invalid, above valid days in February
EC12	1900 ≤ year ≤ 2015	Valid years
EC13	year < 1900	Invalid, below minimum year
EC14	year > 2015	Invalid, above maximum year
EC15	year % 4 == 0, year % 100 != 0 or year % 400 == 0	Leap year condition

Equivalence Class Test Cases:

Test Case	Day	Month	Year	Description	Expected Outcome	Equivalence Classes Identified
TC1	15	5	2010	Valid date	Previous date is 14/5/2010	EC1, EC4, EC12 (Valid month, valid day, valid year)
TC2	31	1	2000	Valid date (January)	Previous date is 30/12/1999	EC1, EC4, EC12 (Valid month, valid day, valid year)
TC3	29	2	2004	Leap year date	Previous date is 28/2/2004	EC1, EC6, EC12, EC15 (Valid leap year, valid February day)
TC4	28	2	2005	Non-leap year date	Previous date is 27/2/2005	EC1, EC7, EC12 (Valid non-leap year, valid February day)
TC5	30	4	2012	Valid date (April)	Previous date is 29/4/2012	EC1, EC5, EC12 (Valid month, valid day for 30-day month, valid year)

TC6	31	9	1998	Invalid day for September	Error message (Invalid date)	EC1, EC10, EC12 (Invalid day for 30-day month, valid year)
TC7	32	5	2011	Invalid day (out of range)	Error message (Invalid date)	EC1, EC9, EC12 (Invalid day, valid month, valid year)
TC8	10	13	2005	Invalid month (out of range)	Error message (Invalid date)	EC3, EC4, EC12 (Invalid month, valid day, valid year)
TC9	15	5	1899	Invalid year (out of range)	Error message (Invalid date)	EC1, EC4, EC13 (Valid month, valid day, invalid year)
TC10	15	5	2016	Invalid year (out of range)	Error message (Invalid date)	EC1, EC4, EC14 (Valid month, valid day, invalid year)

2. Boundary Value Analysis (BVA)

Boundary Value Analysis focuses on testing at the "edges" of valid partitions and beyond the boundary values (i.e. testing near limits).

Boundary Values for Date:

- **Day**: 1, 31 (last valid day for months with 31 days), 30 (last valid day for months with 30 days), 28/29 (February, leap year boundary)
- **Month**: 1 (January), 12 (December)

- **Year**: 1900 (earliest valid year), 2015 (latest valid year

Boundary Value Test Cases:

Test Case	Day	Month	Year	Description	Expected Outcome	Equivalence Classes Identified
TC11	1	1	1900	Boundary: start of year	Previous date is 31/12/1899	EC1, EC4, EC12 (Valid month, valid day, valid year)
TC12	31	12	2015	Boundary: end of year	Previous date is 30/12/2015	EC1, EC4, EC12 (Valid month, valid day, valid year)
TC13	1	2	1900	Boundary: first day of month	Previous date is 31/1/1900	EC1, EC4, EC12 (Valid month, valid day, valid year)
TC14	31	3	2015	Boundary: last day of month	Previous date is 30/3/2015	EC1, EC4, EC12 (Valid month, valid day, valid year)
TC15	29	2	2004	Boundary: leap year February	Previous date is 28/2/2004	EC1, EC6, EC12, EC15 (Valid leap year, valid February day)

TC16	29	2	2005	Boundary: non-leap year February	Previous date is 28/2/2005	EC1, EC7, EC12 (Valid non-leap year, invalid February day)
TC17	30	4	2010	Boundary: end of April (30 days)	Previous date is 29/4/2010	EC1, EC5, EC12 (Valid month, valid day for 30-day month, valid year)
TC18	1	5	2015	Boundary: start of month	Previous date is 30/4/2015	EC1, EC4, EC12 (Valid month, valid day, valid year)
TC19	15	6	2005	Normal middle date	Previous date is 14/6/2005	EC1, EC4, EC12 (Valid month, valid day, valid year)
TC20	1	1	1899	Invalid year (below range)	Error message (Invalid date)	EC1, EC4, EC13 (Valid month, valid day, invalid year)

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

Q.2. 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);
}
```

Functional Test Cases Table:

Test Case ID	Method	Tester Action and Input Data	Expected Outcome

E1	Equivalence Partitioning	`linearSearch(3, [1, 2, 3, 4])`	`2`
E2	Equivalence Partitioning	`linearSearch(5, [1, 2, 3, 4])`	`-1`
E3	Equivalence Partitioning	`linearSearch(3, [])`	`Error: Input array is empty.`
E4	Equivalence Partitioning	`linearSearch(-1, [-2, -1, 0, 1])`	`1`
E5	Equivalence Partitioning	`linearSearch("abc", [1, 2, 3, 4])`	`Error: Invalid input type.`
E6	Equivalence Partitioning	`linearSearch(3.14, [1, 2, 3, 4])`	`Error: Invalid input type.`
E7	Equivalence Partitioning	`linearSearch(NULL, [1, 2, 3, 4])`	`Error: Invalid input type.`
E8	Equivalence Partitioning	`linearSearch(-1, NULL)`	`Error: Input array is null.`

B1	Boundary Value Analysis	`linearSearch(1, [1, 2, 3, 4])`	`0`
B2	Boundary Value Analysis	`linearSearch(4, [1, 2, 3, 4])`	`3`
B3	Boundary Value Analysis	`linearSearch(0, [1, 2, 3, 4])`	`-1`
B4	Boundary Value Analysis	`linearSearch(3, [3])`	`0`
B5	Boundary Value Analysis	`linearSearch(2, [3])`	`-1`

Modified Code:

```
#include <iostream>

#include <vector>

#include <limits>

int linearSearch(int v, const std::vector<int>& a) {

    if (a.empty()) {
```

```
        std::cout << "Error: Input array is empty." << std::endl;
        return -1;
    }
    for (size_t i = 0; i < a.size(); ++i) {
        if (a[i] == v)
            return static_cast<int>(i);
    }
    return -1;
}
int main() {
    std::vector<int> array1 = {1, 2, 3, 4};
    std::vector<int> array2 = {};
    std::vector<int> array3 = {-2, -1, 0, 1};
    int value;
    while (true) {
        std::cout << "Enter a value to search (or -999 to exit): ";
        std::cin >> value;
        if (std::cin.fail() || std::cin.peek() != '\n') {
            std::cout << "Error: Invalid input type." << std::endl;
            std::cin.clear();
            std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
```

```
            continue;
        }
        if (value == -999) break;
        std::cout << "Test Case: " << linearSearch(value, array1) << std::endl;
        std::cout << "Test Case: " << linearSearch(value, array2) << std::endl;
        std::cout << "Test Case: " << linearSearch(value, array3) << std::endl;
    }
    return 0;
}
```

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);
```

Functional Test Cases Table:

Test Case ID	Method	Tester Action and Input Data	Expected Outcome

E1	Equivalence Partitioning	countItem(3, [1, 2, 3, 4, 3, 3])	3
E2	Equivalence Partitioning	countItem(5, [1, 2, 3, 4, 3, 3])	0
E3	Equivalence Partitioning	countItem(1, [])	0
E4	Equivalence Partitioning	countItem(-1, [-2, -1, 0, 1])	1
E5	Equivalence Partitioning	countItem("abc", [1, 2, 3, 4, 3, 3])	Error: Invalid input type.
E6	Equivalence Partitioning	countItem(3.14, [1, 2, 3, 4, 3, 3])	Error: Invalid input type.
E7	Equivalence Partitioning	countItem(NULL, [1, 2, 3, 4, 3, 3])	Error: Invalid input type.
E8	Equivalence Partitioning	countItem(-1, NULL)	Error: Input array is null.
B1	Boundary Value Analysis	countItem(1, [1, 2, 3, 4, 3, 3])	1
B2	Boundary Value Analysis	countItem(4, [1, 2, 3, 4, 3, 3])	1

B3	Boundary Value Analysis	countItem(0, [1, 2, 3, 4, 3, 3])	0
B4	Boundary Value Analysis	countItem(3, [3])	1
B5	Boundary Value Analysis	countItem(2, [3])	0

Modified Code:

```
#include <iostream>

#include <vector>

#include <limits>

#include <type_traits>

int countItem(int v, const std::vector<int>& a) {

    int count = 0;

    for (size_t i = 0; i < a.size(); i++) {

        if (a[i] == v)

            count++;

    }

    return count;

}

int main() {

    std::vector<int> array1 = {1, 2, 3, 4, 3, 3};
```

```
    std::vector<int> array2 = {};
    std::vector<int> array3 = {-2, -1, 0, 1};
    int value;
    while (true) {
        std::cout << "Enter a value to count (or -999 to exit): ";
        std::cin >> value;
        if (std::cin.fail() || std::cin.peek() != '\n') {
            std::cout << "Error: Invalid input type." << std::endl;
            std::cin.clear();
            std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
            continue;
        }
        if (value == -999) break;
        std::cout << "Count in array1: " << countItem(value, array1) << std::endl;
        std::cout << "Count in array2: " << countItem(value, array2) << std::endl;
        std::cout << "Count in array3: " << countItem(value, array3) << std::endl;
    }
    return 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);
}
```

Functional Test Cases Table:

Test Case ID	Method	Tester Action and Input Data	Expected Outcome
E1	Equivalence Partitioning	binarySearch(3, [1, 2, 3, 4, 5])	2
E2	Equivalence Partitioning	binarySearch(6, [1, 2, 3, 4, 5])	-1
E3	Equivalence Partitioning	binarySearch(1, [])	Error: Input array is empty.

E4	Equivalence Partitioning	binarySearch(-1, [-3, -2, -1, 0, 1])	2
E5	Equivalence Partitioning	binarySearch("abc", [1, 2, 3, 4, 5])	Error: Invalid input type.
E6	Equivalence Partitioning	binarySearch(3.14, [1, 2, 3, 4, 5])	Error: Invalid input type.
E7	Equivalence Partitioning	binarySearch(NULL, [1, 2, 3, 4, 5])	Error: Invalid input type.
E8	Equivalence Partitioning	binarySearch(-1, NULL)	Error: Input array is null.
B1	Boundary Value Analysis	binarySearch(1, [1, 2, 3, 4, 5])	0
B2	Boundary Value Analysis	binarySearch(5, [1, 2, 3, 4, 5])	4
B3	Boundary Value Analysis	binarySearch(0, [1, 2, 3, 4, 5])	-1
B4	Boundary Value Analysis	binarySearch(3, [3])	0
B5	Boundary Value Analysis	binarySearch(2, [3])	-1

Modified Code:

```
#include <iostream>
#include <vector>
#include <limits>
#include <type_traits>
int binarySearch(int v, const std::vector<int>& a) {
    int lo = 0, hi = a.size() - 1;
    while (lo <= hi) {
        int mid = (lo + hi) / 2;
        if (v == a[mid])
            return mid;
        else if (v < a[mid])
            hi = mid - 1;
        else
            lo = mid + 1;
    }
    return -1;
}
int main() {
    std::vector<int> array1 = {1, 2, 3, 4, 5};
    std::vector<int> array2 = {};
    std::vector<int> array3 = {-3, -2, -1, 0, 1};
```

```
    int value;
    while (true) {
        std::cout << "Enter a value to search (or -999 to exit): ";
        std::cin >> value;
        if (std::cin.fail() || std::cin.peek() != '\n') {
            std::cout << "Error: Invalid input type." << std::endl;
            std::cin.clear();
            std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
            continue;
        }
        if (value == -999) break;
        std::cout << "Index in array1: " << binarySearch(value, array1) << std::endl;
        std::cout << "Index in array2: " << binarySearch(value, array2) << std::endl;
        std::cout << "Index in array3: " << binarySearch(value, array3) << std::endl;
    }
    return 0;
}
```

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);
}
```

Functional Test Cases Table:

Test Case ID	Method	Tester Action and Input Data	Expected Outcome
E1	Equivalence Partitioning	triangle(3, 3, 3)	0 (EQUILATERAL)
E2	Equivalence Partitioning	triangle(2, 2, 3)	1 (ISOSCELES)
E3	Equivalence Partitioning	triangle(3, 4, 5)	2 (SCALENE)

E4	Equivalence Partitioning	triangle(1, 1, 2)	3 (INVALID)
E5	Equivalence Partitioning	triangle(0, 1, 1)	3 (INVALID)
E6	Equivalence Partitioning	triangle(-1, 2, 2)	3 (INVALID)
E7	Equivalence Partitioning	triangle(1.5, 2, 2)	Error: Invalid input type.
E8	Equivalence Partitioning	triangle("abc", 2, 2)	Error: Invalid input type.
E9	Equivalence Partitioning	triangle(3, 4, -5)	3 (INVALID)
E10	Equivalence Partitioning	triangle(NULL, 2, 2)	Error: Invalid input type.
B1	Boundary Value Analysis	triangle(1, 1, 1)	0 (EQUILATERAL)
B2	Boundary Value Analysis	triangle(2, 2, 1)	1 (ISOSCELES)

B3	Boundary Value Analysis	triangle(2, 2, 2)	0 (EQUILATERAL)
B4	Boundary Value Analysis	triangle(3, 3, 4)	1 (ISOSCELES)
B5	Boundary Value Analysis	triangle(3, 4, 3)	1 (ISOSCELES)
B6	Boundary Value Analysis	triangle(1, 2, 3)	3 (INVALID)

Modified Code:

```
#include <iostream>

#include <limits>

#include <type_traits>

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;
}

int main() {
    int a, b, c;

    while (true) {
        std::cout << "Enter three sides of a triangle (or -999 to exit): ";
        std::cin >> a >> b >> c;
        if (std::cin.fail() || std::cin.peek() != '\n') {
            std::cout << "Error: Invalid input type." << std::endl;
            std::cin.clear();
            std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
            continue;
```

```
    }
    if (a == -999 || b == -999 || c == -999) break;
    int result = triangle(a, b, c);
    if (result == INVALID)
        std::cout << "Triangle is invalid." << std::endl;
    else if (result == EQUILATERAL)
        std::cout << "Triangle is equilateral." << std::endl;
    else if (result == ISOSCELES)
        std::cout << "Triangle is isosceles." << std::endl;
    else
        std::cout << "Triangle is scalene." << std::endl;
  }
  return 0;
}
```

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;
}
```

Functional Test Cases Table:

Test Case ID	Method	Tester Action and Input Data	Expected Outcome
E1	Equivalence Partitioning	prefix("pre", "prefix")	true
E2	Equivalence Partitioning	prefix("pre", "test")	false
E3	Equivalence Partitioning	prefix("long", "longer")	true
E4	Equivalence Partitioning	prefix("", "anystring")	true
E5	Equivalence Partitioning	prefix("string", "")	false

E6	Equivalence Partitioning	prefix("abc", "abcd")	true
E7	Equivalence Partitioning	prefix("abcd", "abc")	false
E8	Equivalence Partitioning	prefix("123", "12345")	true
E9	Equivalence Partitioning	prefix("abc", "1ab")	false
E10	Equivalence Partitioning	prefix("prefix", "pre")	false
E11	Equivalence Partitioning	prefix("abc", null)	Error: Invalid input type.
E12	Equivalence Partitioning	prefix(null, "abc")	Error: Invalid input type.
E13	Equivalence Partitioning	prefix("", null)	Error: Invalid input type.
E14	Equivalence Partitioning	prefix(null, null)	Error: Invalid input type.
B1	Boundary Value Analysis	prefix("abc", "abc")	true

B2	Boundary Value Analysis	prefix("abc", "abcabc")	true
B3	Boundary Value Analysis	prefix("abc", "ab")	false
B4	Boundary Value Analysis	prefix("", "")	true
B5	Boundary Value Analysis	prefix("longprefix", "prefix")	false

Modified Code:

```
import java.util.Scanner;

public class PrefixChecker {
    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;
    }
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        while (true) {
```

```
        System.out.print("Enter two strings (or 'exit' to quit): ");
        String s1 = scanner.nextLine();
        if (s1.equalsIgnoreCase("exit")) break;
        String s2 = scanner.nextLine();
        if (s2.equalsIgnoreCase("exit")) break;
        if (s1 == null || s2 == null) {
            System.out.println("Error: Invalid input type.");
            continue;
        }
        if (s1.length() == 0 || s2.length() == 0) {
            System.out.println("Error: Strings cannot be empty.");
            continue;
        }
        boolean result = prefix(s1, s2);
        if (result) {
            System.out.println("The first string is a prefix of the second string.");
        } else {
            System.out.println("The first string is not a prefix of the second string.");
        }
    }
    scanner.close();
```

```
  }

}
```

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

1. **Valid Inputs:**
    - Class 1: **Equilateral Triangle** (A = B = C)
    - Class 2: **Isosceles Triangle** (A = B or A = C or B = C, but not all equal)
    - Class 3: **Scalene Triangle** (A ≠ B ≠ C)
    - Class 4: **Right-Angled Triangle** ($A^2 + B^2 = C^2$ or any permutation)
2. **Abstract Inputs:**
    - Class 5: **Non-Triangle** (A + B ≤ C or any permutation)
    - Class 6: **invalid input types** (A < 0, B < 0, or C < 0,strings,characters etc)

b) Identify Test Cases for Equivalence Classes

Test Case ID	Tester Action and Input Data	Expected Outcome	Equivalence Class
E1	triangle(3.0, 3.0, 3.0)	Equilateral	Class 1
E2	triangle(2.0, 2.0, 3.0)	Isosceles	Class 2
E3	triangle(3.0, 4.0, 5.0)	Scalene	Class 3
E4	triangle(3.0, 4.0, 6.0)	Non-Triangle	Class 5
E5	triangle(1.0, 1.0, 2.0)	Non-Triangle	Class 5
E6	triangle(0.0, 2.0, 2.0)	Non-Triangle	Class 5
E7	triangle(-1.0, 2.0, 2.0)	Error: Invalid input type.	Class 6
E8	triangle("abc", 2.0, 2.0)	Error: Invalid input type.	Class 6
E9	triangle(3.0, 4.0, 5.0)	Right-Angled	Class 4

c) Boundary Condition A + B > C (Scalene Triangle)

Test Case ID	Tester Action and Input Data	Expected Outcome
B1	triangle(2.0, 3.0, 4.0)	Scalene
B2	triangle(2.0, 3.0, 5.0)	Non-Triangle

B3	triangle(3.0, 3.0, 5.0)	Non-Triangle

d) Boundary Condition A = C (Isosceles Triangle)

Test Case ID	Tester Action and Input Data	Expected Outcome
B4	triangle(3.0, 4.0, 3.0)	Isosceles
B5	triangle(3.0, 2.0, 3.0)	Isosceles

e) Boundary Condition A = B = C (Equilateral Triangle)

Test Case ID	Tester Action and Input Data	Expected Outcome
B6	triangle(3.0, 3.0, 3.0)	Equilateral
B7	triangle(0.0, 0.0, 0.0)	Non-Triangle

f) Boundary Condition $A^2 + B^2 = C^2$ (Right-Angle Triangle)

Test Case ID	Tester Action and Input Data	Expected Outcome
B8	triangle(3.0, 4.0, 5.0)	Right-Angled
B9	triangle(5.0, 12.0, 13.0)	Right-Angled
B10	triangle(1.0, 1.0, 1.414)	Right-Angled
B11	triangle(1.0, 2.0, 2.236)	Non-Triangle

g) Non-Triangle Case Test Cases

Test Case ID	Tester Action and Input Data	Expected Outcome
N1	triangle(1.0, 1.0, 3.0)	Non-Triangle
N2	triangle(5.0, 10.0, 4.0)	Non-Triangle

h) Non-Positive Input Test Cases

Test Case ID	Tester Action and Input Data	Expected Outcome
P1	triangle(-1.0, 2.0, 2.0)	Error: Invalid input type.
P2	triangle(0.0, 0.0, 2.0)	Non-Triangle
P3	triangle(1.0, -2.0, 2.0)	Error: Invalid input type.