
Dhirubhai Ambani Institute of Information and Communication Technology

Gandhinagar, Gujarat

**Name : Ghori Zeel Jivrajbhai**

**Student ID : 202201287**

**Course : IT 314 Software Engineering**

**Professor : Saurabh Tiwari**

**Semester : Autumn 2024**

# Lab 8 : Functional_Testing_Black_box

**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 Partitioning and Boundary Value Analysis Test Cases

### Equivalence Classes:

| Equivalence Class Number | Equivalence Class | Valid/Invalid | Description |
|---|---|---|---|
| 1 | 1 <= day <= 28 | Valid | Valid day range for all months |
| 2 | Day = 29 | Valid | Valid for all months; for February, it should be a leap year |
| 3 | Day = 30 | Valid | Valid for all months except February |
| 4 | Day = 31 | Valid | Valid for months with 31 days (January, March, etc.) |
| 5 | Day < 1 | Invalid | Invalid for any month |
| 6 | Day > 31 | Invalid | Invalid for any month |
| 7 | 1 <= month <= 12 | Valid | Valid range for month |

| | | | |
|---|---|---|---|
| 8 | Month > 12 | Invalid | Invalid range for month |
| 9 | Month < 1 | Invalid | Invalid range for month |
| 10 | 1900 <= year <= 2015 | Valid | Valid year range |
| 11 | Year < 1900 | Invalid | Invalid year range |
| 12 | Year > 2015 | Invalid | Invalid year range |
| 13 | Year divisible by 4 but not 100, or divisible by 400 | Valid | Valid leap year |
| 14 | Year divisible by 100 but not by 400 | Valid | Invalid leap year |

## Test Cases:

| Test Case | Day | Month | Year | Expected Output | BVA/ EP case | Equivalent Class Covered | Equivalent Class Covered number |
|---|---|---|---|---|---|---|---|
| Case 1 | 15 | 6 | 2010 | 14/06/2010 | BVA: Day - Intermediate | Valid day (1 <= day <= 31, valid month) | 1, 7, 10 |
| Case 2 | 1 | 3 | 2000 | 29/02/2000 | BVA: Day - Minimum | Leap year transition (Valid leap year) | 1, 7, 13 |
| Case 3 | 9 | 5 | 2020 | Invalid Date | BVA: Day - Minimum - 1 | Invalid day (day < 1) | 5, 7, 10 |
| Case 4 | 30 | 2 | 2019 | Invalid Date | BVA: Day - Maximum + 1 (February with 30 days) | Invalid day (February with 30 days) | 6, 7, 10 |
| Case 5 | 31 | 12 | 2015 | 30/12/2015 | BVA: Day - Maximum | Valid day (1 <= day <= 31, valid month) | 4, 7, 10 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Case 6 | 1 | 1 | 1900 | 31/12/1899 | BVA: Month - Minimum | Valid day, month, and year | 1, 7, 10 |
| Case 7 | 19 | 0 | 2005 | Invalid Date | BVA: Month - Minimum - 1 | Invalid month (month < 1) | 1, 8, 10 |
| Case 8 | 30 | 6 | 2011 | 29/06/2011 | BVA: Month - Intermediate | Valid day, valid month | 3, 7, 10 |
| Case 9 | 28 | 12 | 2001 | 27/12/2001 | BVA: Month - Maximum | Valid day (1 <= day <= 28, valid month) | 1, 7, 10 |
| Case 10 | 29 | 13 | 2004 | Invalid Date | BVA: Month - Maximum + 1 | Invalid month (month > 12) | 2, 9, 10 |
| Case 11 | 31 | 4 | 1999 | Invalid Date | EP: April with 31 Days | Invalid day (April with 31 days) | 4, 7, 10 |
| Case 12 | 29 | 2 | 2004 | 28/02/2004 | EP: Valid leap year | Valid leap year | 2, 7, 13 |
| Case 13 | 29 | 2 | 2005 | Invalid Date | EP: Non-leap year (February with 28 days) | Invalid day (non-leap year) | 2, 7, 14 |
| Case 14 | 1 | 12 | 2015 | 30/11/2015 | BVA: Year - Maximum | Valid year, valid month, valid day | 1, 7, 10 |
| Case 15 | 31 | 13 | 2014 | Invalid Date | BVA: Year - Maximum + 1 | Invalid month (month > 12) | 4, 9, 10 |
| Case 16 | 28 | 2 | 1990 | 27/02/1990 | BVA: Year - Minimum | Valid day, valid year, valid month | 1, 7, 10 |
| Case 17 | 3 | 11 | 1989 | Invalid Date | BVA: Year - Minimum - 1 | Invalid year (year < 1900) | 1, 7, 11 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Case 18 | 1 | 3 | 2004 | 29/02/2004 | EP: Leap Year Transition | Valid leap year | 2, 7, 13 |
| Case 19 | 1 | 3 | 2005 | 28/02/2005 | EP: Non-Leap Year | Valid non-leap year | 1, 7, 14 |
| Case 20 | 31 | 9 | 2006 | Invalid Date | EP: September with 31 Days | Invalid day (September with 31 days) | 4, 7, 10 |

## Code:

```
#include <iostream>
using namespace std;

// Function to check if a year is a leap year
bool isLeapYear(int year) {
    return (year % 4 == 0 && year % 100 != 0) || (year % 400 == 0);
}

// Function to get the number of days in a month
int getDaysInMonth(int month, int year) {
    switch (month) {
        case 1: case 3: case 5: case 7: case 8: case 10: case 12:
            return 31;
        case 4: case 6: case 9: case 11:
            return 30;
        case 2:
            return isLeapYear(year) ? 29 : 28;
        default:
            return -1;  // Invalid month
    }
}

// Function to print the previous date
void previousDate(int day, int month, int year) {
    if (month < 1 || month > 12 || year < 1900 || year > 2015 || day < 1 || day > 31) {
        cout << "Invalid Date" << endl;
        return;
    }

    int daysInMonth = getDaysInMonth(month, year);
```

```
    if (daysInMonth == -1 || day > daysInMonth) {
        cout << "Invalid Date" << endl;
        return;
    }

    if (day > 1) {
        day--;
    } else {
        if (month == 1) {  // If it's January, move to December of the previous year
            month = 12;
            year--;
            day = 31;
        } else {
            month--;
            day = getDaysInMonth(month, year);
        }
    }

    // Check if the computed year is valid
    if (year < 1900 || year > 2015) {
        cout << "Invalid Date" << endl;
    } else {
        cout << day << "/" << (month < 10 ? "0" : "") << month << "/" << year << endl;
    }
}

// Function to run test cases
void runTestCases() {
    cout << "Running test cases..." << endl;

    // Test Case 1
    cout << "Test Case 1: ";
    previousDate(15, 6, 2010); // Expected Output: 14/06/2010

    // Test Case 2
    cout << "Test Case 2: ";
    previousDate(1, 3, 2000);  // Expected Output: 29/02/2000 (Leap Year)

    // Test Case 3
    cout << "Test Case 3: ";
    previousDate(0, 5, 2020);  // Expected Output: Invalid Date (Day = 0)

    // Test Case 4
    cout << "Test Case 4: ";
```

```
previousDate(30, 2, 2019); // Expected Output: Invalid Date (Feb with 30 days)

// Test Case 5
cout << "Test Case 5: ";
previousDate(31, 12, 2015); // Expected Output: 30/12/2015

// Test Case 6
cout << "Test Case 6: ";
previousDate(1, 1, 1900);  // Expected Output: 31/12/1899

// Test Case 7
cout << "Test Case 7: ";
previousDate(19, 0, 2005); // Expected Output: Invalid Date (Month = 0)

// Test Case 8
cout << "Test Case 8: ";
previousDate(30, 6, 2011); // Expected Output: 29/06/2011

// Test Case 9
cout << "Test Case 9: ";
previousDate(28, 12, 2001); // Expected Output: 27/12/2001

// Test Case 10
cout << "Test Case 10: ";
previousDate(29, 13, 2004); // Expected Output: Invalid Date (Month = 13)

// Test Case 11
cout << "Test Case 11: ";
previousDate(31, 4, 1999); // Expected Output: Invalid Date (April with 31 days)

// Test Case 12
cout << "Test Case 12: ";
previousDate(29, 2, 2004); // Expected Output: 28/02/2004 (Leap Year)

// Test Case 13
cout << "Test Case 13: ";
previousDate(29, 2, 2005); // Expected Output: Invalid Date (Non-Leap Year)

// Test Case 14
cout << "Test Case 14: ";
previousDate(1, 12, 2015); // Expected Output: 30/11/2015

// Test Case 15
cout << "Test Case 15: ";
```

```
    previousDate(31, 13, 2014); // Expected Output: Invalid Date (Month = 13)

    // Test Case 16
    cout << "Test Case 16: ";
    previousDate(28, 2, 1990); // Expected Output: 27/02/1990

    // Test Case 17
    cout << "Test Case 17: ";
    previousDate(3, 11, 1989); // Expected Output: Invalid Date (Year < 1900)

    // Test Case 18
    cout << "Test Case 18: ";
    previousDate(1, 3, 2004); // Expected Output: 29/02/2004 (Leap Year Transition)

    // Test Case 19
    cout << "Test Case 19: ";
    previousDate(1, 3, 2005); // Expected Output: 28/02/2005 (Non-Leap Year)

    // Test Case 20
    cout << "Test Case 20: ";
    previousDate(31, 9, 2006); // Expected Output: Invalid Date (September with 31 Days)
}

int main() {
    runTestCases();
    return 0;
}
```

## 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.

**1. Equivalence Partitioning (EP):**

- **EP1:** The array includes the value v (valid, return index).
  - Example: `a = [3, 5, 7, 9]`, `v = 7`, Expected output: 2
- **EP2:** The array does not include the value v (invalid, return -1).
  - Example: `a = [3, 5, 7, 9]`, `v = 8`, Expected output: `-1`
- **EP3:** The array is empty.
  - Example: `a = [ ]`, `v = 3`, Expected output: `-1`
- **EP4:** The array contains the value v more than once (valid, return index of first occurrence).
  - Example: `a = [3, 5, 7, 7, 9]`, `v = 7`, Expected output: 2

**2. Boundary Value Analysis (BVA):**

- **BVA1:** Test with an array of length 1 (v matches the element).
  - Example: `a = [5]`, `v = 5`, Expected output: `0`
- **BVA2:** Test with an array of length 1 (v does not match the element).
  - Example: `a = [5]`, `v = 3`, Expected output: `-1`
- **BVA3:** Test with an array of larger size.
  - Example: `a = [3, 4, 6, 7, 9, 11, 13]`, `v = 6`, Expected output: 2
- **BVA4:** Test when v matches the first element.
  - Example: `a = [3, 5, 7, 9]`, `v = 3`, Expected output: `0`
- **BVA5:** Test when v matches the last element.
  - Example: `a = [3, 5, 7, 9]`, `v = 9`, Expected output: 3

| Test Case | Class covered | Input(a,v) | Expected Output | Reason |
|---|---|---|---|---|
| case1 | EP1 | [3, 5, 7, 9], 7 | 2 | Array contains v, and the first occurrence is at index 2. |
| case2 | EP2 | [3, 5, 7, 9], 8 | -1 | Array does not contain v, so −1 is returned. |
| case3 | EP3 | [], 3 | -1 | Array is empty, so v cannot be found, returning −1. |
| case4 | EP4 | [3, 5, 7, 7, 9], 7 | 2 | Array contains v more than once, and the first occurrence is at index 2. |
| case5 | BVA1 | [5], 5 | 0 | Single element array where v matches the only element at index 0. |
| case6 | BVA2 | [5], 3 | -1 | Single element array where v does not match the only element, so −1 is returned. |
| case7 | BVA3 | [3, 4, 6, 7, 9, 11, 13], 6 | 2 | Array contains v in the middle, at index 2. |
| case8 | BVA4 | [3, 5, 7, 9], 3 | 0 | v matches the first element in the array, so index 0 is returned. |
| case9 | BVA5 | [3, 5, 7, 9], 9 | 3 | v matches the last element in the array, so index 3 is returned. |

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

### 1. Equivalence Partitioning (EP)

- **EP1**: The array includes the value v (valid case).
  - **Example**: `a = [3, 5, 7, 9, 7]`, `v = 7`, **Expected Output**: 2 (7 appears twice).
- **EP2**: The array does not include the value v (invalid case).
  - **Example**: `a = [3, 5, 7, 9]`, `v = 8`, **Expected Output**: `0` (8 does not appear).
- **EP3**: The array is empty.
  - **Example**: `a = []`, `v = 3`, **Expected Output**: `0` (no elements to count).
- **EP4**: The array contains the value v more than once.
  - **Example**: `a = [3, 5, 7, 7, 9]`, `v = 7`, **Expected Output**: 2 (7 appears twice).

### 2. Boundary Value Analysis (BVA)

- **BVA1**: Test with an array of length 1 (v matches the element).
  - **Example**: `a = [5]`, `v = 5`, **Expected Output**: 1 (5 appears once).
- **BVA2**: Test with an array of length 1 (v does not match the element).
  - **Example**: `a = [5]`, `v = 3`, **Expected Output**: `0` (3 does not appear).
- **BVA3**: Test with an array of larger size.
  - **Example**: `a = [3, 4, 6, 7, 9, 11, 13]`, `v = 6`, **Expected Output**: 1 (6 appears once).
- **BVA4**: Test when v matches the first element.
  - **Example**: `a = [3, 5, 7, 9]`, `v = 3`, **Expected Output**: 1 (3 appears once).
- **BVA5**: Test when v matches the last element.
  - **Example**: `a = [3, 5, 7, 9]`, `v = 9`, **Expected Output**: 1 (9 appears once).

| Test Case | Class covered | Input(a,v) | Expected Output | Reason |
|---|---|---|---|---|
| case1 | EP1 | `[3, 5, 7, 9, 7], 7` | 2 | Array contains v, and v appears twice. |
| case2 | EP2 | `[3, 5, 7, 9], 8` | 0 | Array does not contain v, so `0` is returned. |
| case3 | EP3 | `[ ], 3` | 0 | Array is empty, so no elements to count, returning `0` |
| case4 | EP4 | `[3, 5, 7, 7, 9], 7` | 2 | Array contains v more than once, so it appears twice. |
| case5 | BVA1 | `[5], 5` | 1 | Single element array where v matches the only element, so it appears once. |
| case6 | BVA2 | `[5], 3` | 0 | Single element array where v does not match the only element, so `0` is returned. |
| case7 | BVA3 | `[3, 4, 6, 7, 9, 11, 13], 6` | 1 | Array contains v once, so it appears once. |
| case8 | BVA4 | `[3, 5, 7, 9], 3` | 1 | v matches the first element in the array, so it appears once. |
| case9 | BVA5 | `[3, 5, 7, 9], 9` | 1 | v matches the last element in the array, so it appears once. |

**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.**

1. **Equivalence Partitioning:**

EP1: The array contains v.
EP2: The array is sorted.
EP3: The array is not sorted. (invalid)
EP4: The array does not contain v.
EP5: The array is empty.
EP6: The array contains v multiple times.

2. **Boundary Value Analysis (BVA):**

BVA1: Test with an array of length 1.
BVA2: Test when v matches the first or last element.
BVA3: Test when v matches the middle element

| Test Case | Class covered | Input(a,v) | Expected Output | Reason |
|---|---|---|---|---|
| case1 | EP1 | [1, 2, 3, 4, 5], 3 | 2 | Array contains v, and v appears once. |
| case2 | EP2 | [1, 2, 3, 4, 5], 6 | -1 | Array does not contain v. |
| case3 | EP3 | [5, 4, 3, 2, 1], 3 | -1 | Array is not sorted (invalid). |
| case4 | EP4 | [1, 1, 2, 3, 3, 4, 5], 3 | 2 | Array contains v, and v appears multiple times. |
| case5 | EP5 | [ ], 1 | -1 | Array is empty. |
| case6 | EP6 | [1, 2, 3, 4, 5], 1 | 0 | v matches the first element. |
| case7 | EP6 | [1, 2, 3, 4, 5], 5 | 4 | v matches the last element. |

| | | | | |
|---|---|---|---|---|
| case8 | BVA1 | [5], 5 | 0 | Array length is 1 and v matches the only element. |
| case9 | BVA1 | [5], 10 | -1 | Array length is 1 and v does not match the only element. |
| case10 | BVA2 | [1, 2, 3, 4, 5], 3 | 2 | v matches the middle element. |
| case11 | BVA3 | [1, 2, 3, 4], 2 | 1 | v matches the middle element in an even-length array. |

**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).**

1. **Equivalence Partitioning:**

EP1: The triangle is equilateral.
EP2: The triangle is isosceles.
EP3: The triangle is scalene.
EP4: Invalid triangle configurations.
EP5: Side length of zero (invalid).
EP6: Negative side lengths (invalid).
EP7: Another equilateral configuration.
EP8: Another isosceles configuration.

2. **Boundary Value Analysis (BVA):**

BVA1: Focuses on the minimum valid lengths and scenarios where triangles cannot be formed.
BVA2: Checks lengths that are just under and above the triangle inequality threshold.
BVA3: Valid lengths resulting in different triangle types.

| Test Case | Class covered | Input(a,b,c) | Expected Output | Reason |
|---|---|---|---|---|
| case1 | EP1 | 3, 3, 3 | Equilateral | All sides are equal. |
| case2 | EP2 | 3, 3, 4 | Isosceles | Two sides are equal. |
| case3 | EP3 | 3, 4, 5 | Scalene | All sides are different. |
| case4 | EP4 | 1, 2, 3 | Invalid | Sides do not satisfy triangle inequality (1 + 2 ≤ 3). |
| case5 | EP5 | `0,1,1` | Invalid | Side length is zero (not valid for a triangle). |
| case6 | EP6 | `-1,2,2` | Invalid | Side length is negative (not valid for a triangle). |
| case7 | EP7 | `4,4,4` | Equilateral | All sides are equal (same as case1, just a repeat). |
| case8 | EP8 | 5, 5, 1 | Isosceles | Two sides are equal, one is different. |
| case9 | BVA1 | 1, 1, 1 | Equilateral | Minimum valid lengths for an equilateral triangle. |
| case10 | BVA1 | 1, 1, 2 | Invalid | Minimum lengths where sides cannot form a triangle. |
| case11 | BVA2 | 2, 2, 3 | Isosceles | Lengths just below the triangle inequality threshold. |
| case12 | BVA2 | 3, 3, 6 | Invalid | Sides do not satisfy triangle inequality (3 + 3 ≤ 6). |
| case13 | BVA3 | 2, 3, 4 | Scalene | Valid lengths that form a scalene triangle. |
| case14 | BVA4 | 3, 3, 2 | Isosceles | Valid lengths that can also be rearranged. |

**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).**

1. **Equivalence Partitioning**

EP1: s1 is a prefix of s2
EP2: s1 is equal to s2
EP3: s1 is not a prefix of s2
EP4: s1 is an empty string and s2 is non-empty
EP5: s1 is non-empty and s2 is an empty string
EP6: Both s1 and s2 are empty strings

2. **Boundary Value Analysis**

BVA1: s1 is an empty string and s2 is also empty
BVA2: s1 is an empty string and s2 is non-empty.
BVA3: s1 is the same length as s2 but not a prefix
BVA4: s1 is a single character prefix of s2
BVA5: s1 is longer than s2
BVA6: s1 is a prefix at the boundary length of s2
BVA7: s1 is the same as s2

| Test Case | Class covered | Input (s1, s2) | Expected Output | Reason |
|---|---|---|---|---|
| case1 | EP1 | ("pre", "prefix") | True | s1 is a prefix of s2. |
| case2 | EP2 | ("equal", "equal") | True | s1 is equal to s2. |
| case3 | EP3 | ("non", "prefix") | False | s1 is not a prefix of s2. |
| case4 | EP4 | ("", "notempty") | True | An empty string is a prefix of any non-empty string. |

| | | | | |
|---|---|---|---|---|
| case5 | EP5 | ("nonempty", "") | False | A non-empty string cannot be a prefix of an empty string. |
| case6 | EP6 | ("", "") | True | Both strings are empty, so s1 is considered a prefix of s2. |
| case7 | BVA1 | ("", "") | True | Both strings are empty; should return true. |
| case8 | BVA2 | ("", "abc") | True | An empty string is a prefix of any non-empty string. |
| case9 | BVA3 | ("abc", "xyz") | False | s1 is the same length as s2 but not a prefix. |
| case10 | BVA4 | ("a", "abc") | True | s1 is a single character that matches the start of s2. |
| case11 | BVA5 | ("abcde", "abc") | False | s1 is longer than s2, so it cannot be a prefix. |
| case12 | BVA6 | ("abc", "abcd") | True | s1 is a prefix at the boundary length of s2. |
| case13 | BVA7 | ("same", "same") | True | Both strings are the same, so s1 is a prefix of s2. |

**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 Class | Description |
|---|---|
| EP1 | Equilateral triangle (all sides equal) |
| EP2 | Isosceles triangle (two sides equal) |

| | |
|---|---|
| EP3 | Scalene triangle (all sides unequal) |
| EP4 | Right-angled triangle (satisfies Pythagorean theorem) |
| EP5 | Invalid triangle (fails triangle inequality) |
| EP6 | Non-triangle case (sides are zero or negative) |
| EP7 | Non-positive input (any side length <= 0) |

**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 | Class covered | Input (A,B,C) | Expected Output | Reason |
|---|---|---|---|---|
| case1 | EP1 | 3.0, 3.0, 3.0 | Equilateral | All sides are equal. |
| case2 | EP2 | 3.0, 3.0, 4.0 | Isosceles | Two sides are equal. |
| case3 | EP3 | 3.0, 4.0, 5.0 | Scalene | All sides are different. |
| case4 | EP4 | 3.0, 4.0, 5.0 | Right-angled | Satisfies Pythagorean theorem ($3^2 + 4^2 = 5^2$). |
| case5 | EP5 | `1.0, 2.0, 3.0` | Invalid | Fails triangle inequality (1 + 2 <= 3). |
| case6 | EP6 | `0.0, 1.0, 2.0` | Invalid | One side is zero, not a triangle. |
| case7 | EP7 | `-1.0, 1.0, 1.0` | Invalid | Negative side length, not valid. |

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

| Test Case | Description | Input (A,B,C) | Expected Output |
|---|---|---|---|
| case1 | Just above the boundary | 3.0, 4.0, 5.0 | Scalene |
| case2 | Exactly on the boundary | 2.0, 3.0, 4.0 | Scalene |
| case3 | Just below the boundary | 1.0, 2.0, 2.9 | Scalene |

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

| Test Case | Description | Input (A,B,C) | Expected Output |
|---|---|---|---|
| case1 | Just above the boundary | 3.0, 4.0, 3.0 | Isosceles |
| case2 | Exactly on the boundary | 3.0, 4.0, 3.0 | Isosceles |
| case3 | Just below the boundary | 2.0, 3.0, 2.0 | Isosceles |

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

| Test Case | Description | Input (A,B,C) | Expected Output |
|---|---|---|---|
| case1 | Just above the boundary | 3.0, 3.0, 3.0 | Equilateral |
| case2 | Exactly on the boundary | 2.0, 2.0, 2.0 | Equilateral |
| case3 | Just below the boundary | 1.0, 1.0, 1.0 | Equilateral |

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

| Test Case | Description | Input (A,B,C) | Expected Output |
|---|---|---|---|
| case1 | Just above the boundary | 3.0, 4.0, 5.0 | right-angled |
| case2 | Exactly on the boundary | 3.0, 4.0, 5.0 | right-angled |
| case3 | Just below the boundary | 2.0, 2.0, 2.9 | Invalid |

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

| Test Case | Description | Input (A,B,C) | Expected Output |
|---|---|---|---|
| case1 | Sides sum equal to a side | 1.0, 2.0, 3.0 | Invalid |
| case2 | One side greater than sum of others | 1.0, 1.0, 3.0 | Invalid |
| case3 | Negative lengths | -1.0, 2.0, 3.0 | Invalid |

### h) For non-positive input, identify test points.

| Test Case | Description | Input (A,B,C) | Expected Output |
|---|---|---|---|
| case1 | Zero length | 0.0, 1.0, 1.0 | Invalid |
| case2 | All sides zero | 0.0, 0.0, 0.0 | Invalid |
| case3 | One negative length | 1.0, 1.0, -1.0 | Invalid |