IT-314 Software Engineering

Assignment 8: Functional Testing (Black Box)

Prof.: Saurabh Tiwari

Name: Tandel Divyakumar (202201469)

October 21, 2024

Question:

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.

.

Program: 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 the previous date or invalid date. Design the equivalence class test cases?

Answer:

1. Equivalence Partitioning (EP):

We divide the input ranges into valid and invalid partitions (equivalence classes) for **day**, **month**, and **year**. Each equivalence class represents a set of inputs that should behave the same way when tested.

- **Day**:
  - Valid range: 1 to 31
  - Invalid ranges: less than 1 (too low) and greater than 31 (too high)
- **Month**:
  - Valid range: 1 to 12
  - Invalid ranges: less than 1 (too low) and greater than 12 (too high)
- **Year**:
  - Valid range: 1900 to 2015
  - Invalid ranges: less than 1900 (too low) and greater than 2015 (too high)

2. Boundary Value Analysis (BVA):

In BVA, we test the boundaries of the input ranges to ensure the program handles edge cases correctly. This includes testing:

- Minimum and maximum valid values for **day**, **month**, and **year**
- Values just outside the valid range (e.g., day = 0, month = 13)

Define the Equivalence Classes:

Equivalence Class	Description	Class Number
Day is valid	1≤day≤31	E1
Day is invalid (low)	day<1	E2
Day is invalid (high)	day>31	E3
Month is valid	1≤month≤12	E4
Month is invalid (low)	month<1	E5
Month is invalid (high)	month>12	E6
Year is valid	1900≤year≤2015	E7
Year is invalid (low)	year<1900	E8
Year is invalid (high)	year>2015	E9

Test Cases Based on Equivalence Partitioning and Boundary Value Analysis:

Input Data (day, month, year)	Expected Outcome	Type of Test	Relevant Equivalence Class(es)
(15, 5, 2000)	14, 5, 2000	EP (Valid)	E1, E4, E7
(32, 5, 2000)	Error: Invalid day	EP (Invalid)	E3, E4, E7
(0, 5, 2000)	Error: Invalid day	EP (Invalid)	E2, E4, E7
(15, 13, 2000)	Error: Invalid month	EP (Invalid)	E1, E6, E7
(15, 0, 2000)	Error: Invalid month	EP (Invalid)	E1, E5, E7

(15, 5, 1899)	Error: Invalid year	EP (Invalid)	E1, E4, E8
(15, 5, 2016)	Error: Invalid year	EP (Invalid)	E1, E4, E9
(1, 5, 2000)	30, 4, 2000	BVA (Boundary)	E1 (min), E4, E7
(31, 12, 2000)	30, 12, 2000	BVA (Boundary)	E1 (max), E4 (max), E7
(0, 5, 2000)	Error: Invalid day	BVA (Boundary)	E2, E4, E7
(15, 12, 2000)	14, 12, 2000	BVA (Boundary)	E1, E4 (max), E7
(15, 0, 2000)	Error: Invalid month	BVA (Boundary)	E1, E5, E7
(15, 5, 2015)	14, 5, 2015	BVA (Boundary)	E1, E4, E7 (max)
(15, 5, 1899)	Error: Invalid year	BVA (Boundary)	E1, E4, E8

Modified Code:

```
#include <iostream>
#include <vector>
#include <tuple>
#include <string>

bool is_leap_year(int year) {
    return (year % 4 == 0 && year % 100 != 0) || (year % 400 == 0);
}

int get_days_in_month(int month, int year) {
    if (month == 2) {
        return is_leap_year(year) ? 29 : 28;
    } else if (month == 4 || month == 6 || month == 9 || month == 11) {
        return 30;
    } else {
        return 31;
    }
```

```cpp
}

std::tuple<bool, std::string> validate_date(int day, int month, int year)
{
    if (year < 1900 || year > 2015) {
        return std::make_tuple(false, "Error: Invalid year");
    }
    if (month < 1 || month > 12) {
        return std::make_tuple(false, "Error: Invalid month");
    }
    if (day < 1 || day > get_days_in_month(month, year)) {
        return std::make_tuple(false, "Error: Invalid day");
    }
    return std::make_tuple(true, "Valid date");
}

std::tuple<int, int, int> previous_date(int day, int month, int year) {
    bool valid;
    std::string message;
    std::tie(valid, message) = validate_date(day, month, year);

    if (!valid) {
        std::cout << message << std::endl;
        return std::make_tuple(-1, -1, -1); // Invalid date placeholder
    }

    if (day > 1) {
        return std::make_tuple(day - 1, month, year);
    } else {
        if (month == 1) {
            return std::make_tuple(31, 12, year - 1);
        } else {
            int previous_month_days = get_days_in_month(month - 1, year);
            return std::make_tuple(previous_month_days, month - 1, year);
        }
    }
}

int main() {
    std::vector<std::tuple<int, int, int>> test_cases = {
```

```
        {15, 5, 2000}, {32, 5, 2000}, {0, 5, 2000}, {15, 13, 2000},
        {15, 0, 2000}, {15, 5, 1899}, {15, 5, 2016}, {1, 5, 2000},
        {31, 12, 2000}, {0, 5, 2000}, {15, 12, 2000}, {15, 0, 2000},
        {15, 5, 2015}, {15, 5, 1899}
    };

    for (const auto& [day, month, year] : test_cases) {
        int prev_day, prev_month, prev_year;
        std::tie(prev_day, prev_month, prev_year) = previous_date(day,
month, year);
        std::cout << "Input: (" << day << ", " << month << ", " << year
                  << ") => Output: (" << prev_day << ", " << prev_month <<
", " << prev_year << ")\n";
    }

    return 0;
}
```

Program 1: 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.

Answer:

Code:

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

Define the Equivalence Classes:

Equivalence Class	Description	Class Number
Value v is present in the array a	The value v exists in a	E1
Value v is not present in the array a	The value v does not exist in a	E2
Array a is empty	The array a is empty	E3
Value v appears multiple times in the array a	The value v appears multiple times in a	E4

Test Cases Based on Equivalence Partitioning and Boundary Value Analysis:

Input Data (array a,value v)	Expected Outcome	Type of Test	Relevant Equivalence Class(es)
([3, 5, 8, 10], 5)	1	EP (Valid)	E1
([3, 5, 8, 10], 15)	-1	EP (Valid)	E2
([], 5)	-1	EP (Empty Array)	E3
([3, 5, 8, 5, 10], 5)	1	EP (Valid, Multiple)	E4
([3], 3)	0	BVA (Boundary)	E1
([3], 10)	-1	BVA (Boundary)	E2

Modified Code:

```
#include <iostream>
#include <vector>

int linearSearch(const std::vector<int>& a, int v) {
    for (int i = 0; i < a.size(); i++) {
        if (a[i] == v) {
            return i;
        }
    }
    return -1;
}

int main() {
    // Test cases
    std::vector<std::pair<std::vector<int>, int>> test_cases = {
        {{3, 5, 8, 10}, 5},     // E1: Value present
        {{3, 5, 8, 10}, 15},    // E2: Value not present
        {{}, 5},                // E3: Empty array
```

```
        {{3, 5, 8, 5, 10}, 5}, // E4: Value present multiple times
        {{3}, 3},              // BVA: Single element, present
        {{3}, 10}              // BVA: Single element, not present
    };

    // Running test cases
    for (const auto& [a, v] : test_cases) {
        int result = linearSearch(a, v);
        std::cout << "Array: { ";
        for (int num : a) {
            std::cout << num << " ";
        }
        std::cout << "}, Value: " << v << " => Output: " << result <<
"\n";
    }

    return 0;
}
```

Program 2: The function countItem returns the number of times a value v appears in an array of integers a.

Answer:

CODE:

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


    }


    return (count);
}
```

Define the Equivalence Classes:

Equivalence Class	Description	Class Number
Value v is present in the array a	The value v exists in a	E1
Value v is not present in the array a	The value v does not exist in a	E2
Array a is empty	The array a is empty	E3
Value v appears multiple times in the array a	The value v appears multiple times in a	E4

Test Cases Based on Equivalence Partitioning and Boundary Value Analysis:

Input Data (array a, value v)	Expected Outcome	Type of Test	Relevant Equivalence Class(es)
([3, 5, 8, 10], 5)	1	EP (Valid)	E1
([3, 5, 8, 10], 15)	0	EP (Valid)	E2
([], 5)	0	EP (Empty Array)	E3
([3, 5, 8, 5, 10], 5)	2	EP (Valid, Multiple)	E4
([3], 3)	1	BVA (Boundary)	E1
([3], 10)	0	BVA (Boundary)	E2

Modified Code:

```
#include <iostream>
#include <vector>

int countItem(const std::vector<int>& a, int v) {
    int count = 0;
    for (int item : a) {
        if (item == v) {
            count++;
        }
    }
    return count;
}

int main() {
    // Test cases
    std::vector<std::pair<std::vector<int>, int>> test_cases = {
        {{3, 5, 8, 10}, 5},     // Value present once
        {{3, 5, 8, 10}, 15},    // Value not present
        {{}, 5},                // Empty array
        {{3, 5, 8, 5, 10}, 5}, // Value present multiple times
```

```
        {{3}, 3},              // Single element, value present
        {{3}, 10}              // Single element, value not present
    };

    // Running test cases
    for (const auto& [a, v] : test_cases) {
        int result = countItem(a, v);
        std::cout << "Array: { ";
        for (int num : a) {
            std::cout << num << " ";
        }
        std::cout << "}, Value: " << v << " => Output: " << result <<
"\n";
    }

    return 0;
}
```

Program 3: 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.

Answer:

CODE:

```
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);
}
```

Define the Equivalence Classes:

Equivalence Class	Description	Class Number
Value v is present in the ordered array a	The value v exists in a	E1
Value v is not present in the ordered array a	The value v does not exist in a	E2
Array a is empty	The array a is empty	E3

Value v is less than the smallest element in the array	The value v is smaller than all elements in a	E4
Value v is greater than the largest element in the array	The value v is larger than all elements in a	E5
Value v appears multiple times in the ordered array a	The value v appears multiple times in a	E6

Test Cases Based on Equivalence Partitioning and Boundary Value Analysis:

Input Data (ordered array a, value v)	Expected Outcome	Type of Test	Relevant Equivalence Class(es)
([1, 3, 5, 7, 9], 5)	2	EP (Valid)	E1
([1, 3, 5, 7, 9], 4)	-1	EP (Valid)	E2
([], 5)	-1	EP (Empty Array)	E3
([1, 3, 5, 7, 9], 1)	0	BVA (Boundary)	E1
([1, 3, 5, 7, 9], 9)	4	BVA (Boundary)	E1
([1, 3, 5, 7, 9], 0)	-1	BVA (Less than Min)	E4
([1, 3, 5, 7, 9], 10)	-1	BVA (Greater than Max)	E5
([1, 3, 5, 5, 5, 7, 9], 5)	2	EP (Valid, Multiple)	E6

Modified Code:

```
#include <iostream>
#include <vector>

int binarySearch(const std::vector<int>& a, int v) {
    int low = 0, high = a.size() - 1;

    while (low <= high) {
```

```
        int mid = low + (high - low) / 2; // Avoid potential overflow with
(low + high) / 2
        if (a[mid] == v) {
            return mid;
        } else if (a[mid] < v) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }

    return -1;
}

int main() {
    // Test cases
    std::vector<std::pair<std::vector<int>, int>> test_cases = {
        {{1, 3, 5, 7, 9}, 5},    // E1: Value present
        {{1, 3, 5, 7, 9}, 4},    // E2: Value not present
        {{}, 5},                 // E3: Empty array
        {{1, 3, 5, 7, 9}, 1},    // BVA: Boundary case, smallest value
        {{1, 3, 5, 7, 9}, 9},    // BVA: Boundary case, largest value
        {{1, 3, 5, 7, 9}, 0},    // BVA: Less than minimum value
        {{1, 3, 5, 7, 9}, 10},   // BVA: Greater than maximum value
        {{1, 3, 5, 5, 5, 7, 9}, 5}  // E6: Value present multiple times
    };

    // Running test cases
    for (const auto& [a, v] : test_cases) {
        int result = binarySearch(a, v);
        std::cout << "Array: { ";
        for (int num : a) {
            std::cout << num << " ";
        }
        std::cout << "}, Value: " << v << " => Output: " << result <<
"\n";
    }

    return 0;
}
```

Program 4: 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).

Answer:

CODE:

```
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);
}
```

Define the Equivalence Classes:

Equivalence Class	Description	Class Number
Three sides are equal	All three sides are the same length (equilateral)	E1
Two sides are equal	Exactly two sides are the same length (isosceles)	E2
All sides are different	All three sides are of different lengths (scalene)	E3
Invalid triangle lengths	The given lengths cannot form a triangle	E4

Test Cases Based on Equivalence Partitioning and Boundary Value Analysis:

Input Data (sides a, b, c)	Expected Outcome	Type of Test	Relevant Equivalence Class(es)
(3, 3, 3)	"Equilateral"	EP (Valid)	E1
(3, 3, 5)	"Isosceles"	EP (Valid)	E2
(3, 4, 5)	"Scalene"	EP (Valid)	E3
(1, 1, 3)	"Invalid"	EP (Invalid)	E4
(0, 1, 1)	"Invalid"	EP (Invalid)	E4
(-1, 1, 1)	"Invalid"	EP (Invalid)	E4
(5, 5, 5)	"Equilateral"	BVA (Boundary)	E1
(2, 2, 3)	"Isosceles"	BVA (Boundary)	E2
(2, 3, 4)	"Scalene"	BVA (Boundary)	E3
(1, 2, 3)	"Invalid"	BVA (Boundary)	E4

Modified Code:

```
#include <iostream>
#include <vector>
#include <tuple>

std::string triangle(int a, int b, int c) {
    if (a <= 0 || b <= 0 || c <= 0) {
        return "Invalid";
    }
    if (a + b <= c || a + c <= b || b + c <= a) {
        return "Invalid";
    }
    if (a == b && b == c) {
        return "Equilateral";
    } else if (a == b || b == c || a == c) {
        return "Isosceles";
```

```
    } else {
        return "Scalene";
    }
}

int main() {
    // Test cases
    std::vector<std::tuple<int, int, int>> test_cases = {
        {3, 3, 3},     // E1: Equilateral
        {3, 3, 5},     // E2: Isosceles
        {3, 4, 5},     // E3: Scalene
        {1, 1, 3},     // E4: Invalid
        {0, 1, 1},     // E4: Invalid
        {-1, 1, 1},    // E4: Invalid
        {5, 5, 5},     // E1: Equilateral
        {2, 2, 3},     // E2: Isosceles
        {2, 3, 4},     // E3: Scalene
        {1, 2, 3}      // E4: Invalid
    };

    // Running test cases
    for (const auto& [a, b, c] : test_cases) {
        std::string result = triangle(a, b, c);
        std::cout << "Sides: (" << a << ", " << b << ", " << c << ") =>
Output: " << result << "\n";
    }

    return 0;
}
```

Program 5: 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).

Answer:

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;
}
```

Define the Equivalence Classes:

Equivalence Class	Description	Class Number
s1 is a prefix of s2	The string s1 matches the beginning of s2	E1
s1 is not a prefix of s2	The string s1 does not match the beginning of s2	E2
s1 is empty	The string s1 is empty	E3
s2 is empty	The string s2 is empty	E4
s1 is longer than s2	The string s1 is longer than s2	E5

Test Cases Based on Equivalence Partitioning and Boundary Value Analysis:

Input Data (s1, s2)	Expected Outcome	Type of Test	Relevant Equivalence Class(es)
("abc", "abcdef")	TRUE	EP (Valid)	E1
("abc", "xyzabc")	FALSE	EP (Valid)	E2
("", "abcdef")	TRUE	EP (Empty s1)	E3
("abc", "")	FALSE	EP (Empty s2)	E4
("abcdef", "abc")	FALSE	EP (s1 longer)	E5
("", "")	TRUE	BVA (Both Empty)	E3, E4

Modified Code:

```cpp
#include <iostream>
#include <string>

bool prefix(const std::string& s1, const std::string& s2) {
    return s2.rfind(s1, 0) == 0;  // Check if s1 is a prefix of s2
}

int main() {
    // Test cases
    std::vector<std::pair<std::string, std::string>> test_cases = {
        {"abc", "abcdef"},     // E1: s1 is a prefix of s2
        {"abc", "xyzabc"},     // E2: s1 is not a prefix of s2
        {"", "abcdef"},        // E3: s1 is empty
        {"abc", ""},           // E4: s2 is empty
        {"abcdef", "abc"},     // E5: s1 is longer than s2
        {"", ""}               // BVA: Both s1 and s2 are empty
    };

    // Running test cases
    for (const auto& [s1, s2] : test_cases) {
```

```
        bool result = prefix(s1, s2);
        std::cout << "s1: '" << s1 << "', s2: '" << s2 << "' => Output: "
<< std::boolalpha << result << "\n";
    }

    return 0;
}
```

Program 6: 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:

1. Identify the equivalence classes for the system
2. 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)
3. For the boundary condition A + B > C case (scalene triangle), identify test cases to verify the boundary.
4. For the boundary condition A = C case (isosceles triangle), identify test cases to verify the boundary.
5. For the boundary condition A = B = C case (equilateral triangle), identify test cases to verify the boundary.
6. For the boundary condition A2 + B2 = C2 case (right-angle triangle), identify test cases to verify the boundary.
7. For the non-triangle case, identify test cases to explore the boundary.
8. For non-positive input, identify test points.

Answer:

Triangle Classification Program:

a) Identify the Equivalence Classes

Equivalence Class	Description	Class Number
Three sides are equal	All three sides are the same length (equilateral)	E1

Two sides are equal	Exactly two sides are the same length (isosceles)	E2
All sides are different	All three sides are of different lengths (scalene)	E3
Right angle triangle	Sides satisfy the condition A2+B2=C2	E4
Invalid triangle lengths	The given lengths cannot form a triangle	E5
Non-positive sides	Any side is non-positive	E6

b) Identify Test Cases to Cover the Identified Equivalence Classes

Input Data (A, B, C)	Expected Outcome	Relevant Equivalence Class(es)
(3.0, 3.0, 3.0)	"Equilateral"	E1
(3.0, 3.0, 5.0)	"Isosceles"	E2
(3.0, 4.0, 5.0)	"Scalene"	E3
(3.0, 4.0, 5.0)	"Right-angled"	E4
(1.0, 1.0, 3.0)	"Invalid"	E5
(0.0, 1.0, 1.0)	"Invalid"	E6
(-1.0, 2.0, 2.0)	"Invalid"	E6
(0.0, 0.0, 0.0)	"Invalid"	E6

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

Input Data (A, B, C)	Expected Outcome	Description
(2.0, 3.0, 4.0)	"Scalene"	Valid scalene triangle
(2.0, 2.0, 3.9)	"Scalene"	Valid scalene triangle

(2.0, 2.0, 4.0)	"Invalid"	Fails triangle inequality

d) Boundary Condition A = C (Isosceles Triangle)

Input Data (A, B, C)	Expected Outcome	Description
(3.0, 4.0, 3.0)	"Isosceles"	Valid isosceles triangle
(3.0, 4.0, 4.0)	"Isosceles"	Valid isosceles triangle
(3.0, 4.0, 2.9)	"Invalid"	Fails triangle inequality

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

Input Data (A, B, C)	Expected Outcome	Description
(3.0, 3.0, 3.0)	"Equilateral"	Valid equilateral triangle
(2.0, 2.0, 2.0)	"Equilateral"	Valid equilateral triangle
(2.9, 2.9, 2.9)	"Equilateral"	Valid equilateral triangle

f) Boundary Condition A2+B2=C2A^2 + B^2 = C^2A2+B2=C2 (Right-Angle Triangle)

Input Data (A, B, C)	Expected Outcome	Description
(3.0, 4.0, 5.0)	"Right-angled"	Valid right-angle triangle
(5.0, 12.0, 13.0)	"Right-angled"	Valid right-angle triangle
(3.0, 4.0, 4.9)	"Invalid"	Fails right-angle condition

g) Non-Triangle Case

Input Data (A, B, C)	Expected Outcome	Description
(1.0, 2.0, 3.0)	"Invalid"	Fails triangle inequality
(5.0, 2.0, 2.0)	"Invalid"	Fails triangle inequality
(10.0, 1.0, 1.0)	"Invalid"	Fails triangle inequality

h) Non-Positive Input

Input Data (A, B, C)	Expected Outcome	Description
(0.0, 1.0, 1.0)	"Invalid"	Non-positive side
(-1.0, 2.0, 2.0)	"Invalid"	Non-positive side
(1.0, 0.0, 1.0)	"Invalid"	Non-positive side
(1.0, 1.0, -1.0)	"Invalid"	Non-positive side

Implementation of the Triangle Classification Program:

```
#include <iostream>
#include <vector>
#include <cmath>
#include <tuple>

std::string triangle_classification(double A, double B, double C) {
    if (A <= 0 || B <= 0 || C <= 0) {
        return "Invalid";
    }

    if (A + B <= C || A + C <= B || B + C <= A) {
        return "Invalid";
    }
```

```
    if (A == B && B == C) {
        return "Equilateral";
    }

    if (A == B || A == C || B == C) {
        return "Isosceles";
    }

    if (std::pow(A, 2) + std::pow(B, 2) == std::pow(C, 2) ||
        std::pow(A, 2) + std::pow(C, 2) == std::pow(B, 2) ||
        std::pow(B, 2) + std::pow(C, 2) == std::pow(A, 2)) {
        return "Right-angled";
    }

    return "Scalene";
}

int main() {
    // Example Test Cases
    std::vector<std::tuple<double, double, double>> test_cases = {
        {3.0, 3.0, 3.0},    // E1: Equilateral
        {3.0, 3.0, 5.0},    // E2: Isosceles
        {3.0, 4.0, 5.0},    // E3: Scalene
        {3.0, 4.0, 5.0},    // E4: Right-angled
        {1.0, 1.0, 3.0},    // E5: Invalid
        {0.0, 1.0, 1.0},    // E6: Invalid
        {-1.0, 2.0, 2.0},   // E6: Invalid
        {0.0, 0.0, 0.0}     // E6: Invalid
    };

    // Running test cases
    for (const auto& [A, B, C] : test_cases) {
        std::string result = triangle_classification(A, B, C);
        std::cout << "Sides: (" << A << ", " << B << ", " << C << ") =>
Output: " << result << "\n";
    }

    return 0;
}
```