IT314: Software Engineering

Lab 8

Roll number: 202201362
Name: Tanay Kewalramani

Q1. Equivalence Partitioning Tests:

Tester Action and Input Data	Expected Outcome
1, 1, 2015	31 Dec 2014
29, 2, 2012	28 Feb 2012
1, 1, 1900	31 Dec 1899
31, 12, 2015	30 Dec 2015
13, 1, 2015	An Error message
1, 0, 2015	An Error message
29, 2, 2016	28 Feb 2016
31, 11, 2015	30 Nov 2015
32, 1, 2015	An Error message

Boundary Analysis:

Tester Action and Input Data	Expected Outcome
1, 1, 1900	31 Dec 1899
1, 1, 2015	31 Dec 2014
1, 12, 2015	30 Nov 2015
1, 2, 2015	31 Jan 2015
29, 2, 2016	28 Feb 2016
31, 1, 2015	30 Dec 2014
31, 12, 2015	30 Nov 2015
0, 1, 2015	An Error message
1, 13, 2015	An Error message

C++ Code:

```
#include <iostream>
#include <iomanip>
#include <stdexcept>

using namespace std;

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

// Function to get the previous date
string previousDate(int day, int month, int year) {
    // Validate the year
    if (year < 1900 || year > 2015) {
        return "An Error message";
    }
    // Validate the month
    if (month < 1 || month > 12) {
        return "An Error message";
    }
    // Validate the day
    if (day < 1 || day > 31) {
        return "An Error message";
    }

    // Days in each month
    int daysInMonth[] = { 31, (isLeapYear(year) ? 29 : 28), 31, 30, 31, 30,
                          31, 31, 30, 31, 30, 31 };

    // Validate the day for the specific month
    if (day > daysInMonth[month - 1]) {
        return "An Error message";
    }

    // Calculate previous date
    if (day > 1) {
        return to_string(day - 1) + " " + to_string(month) + " " + to_string(year);
    } else {
        if (month == 1) {
            return to_string(31) + " 12 " + to_string(year - 1);
        } else {
```

```
            return to_string(daysInMonth[month - 2]) + " " + to_string(month - 1) + " " +
to_string(year);
        }
    }
}

// Test cases
execution void
runTests() {
    int testCases[][3] = {
        {1, 1, 2015},      // Valid
        {29, 2, 2012},     // Valid leap year
        {1, 1, 1900},      // Valid
        {31, 12, 2015},    // Valid
        {13, 1, 2015},     // Invalid month
        {1, 0, 2015},      // Invalid month
        {29, 2, 2016},     // Valid leap year
        {31, 11, 2015},    // Valid
        {32, 1, 2015},     // Invalid day
        {1, 1, 1900},      // Valid
        {1, 1, 2016},      // Invalid year
        {1, 2, 1900},      // Valid
        {0, 1, 2015},      // Invalid day
        {31, 1, 2015},     // Valid
        {31, 2, 2015}      // Invalid day
    };

    string expectedOutcomes[] = {
        "31 1 2015",     // Case 1
        "28 2 2012",     // Case 2
        "31 12 1899",    // Case 3
        "30 12 2015",    // Case 4
        "An Error message", // Case 5
        "An Error message", // Case 6
        "An Error message", // Case 7
        "30 11 2015", // Case 8
        "An Error message", // Case 9
        "31 12 1899", // Case 10
        "An Error message", // Case 11
        "31 1 1900", // Case 12
        "An Error message", // Case 13
        "30 12 2014", // Case 14
        "An Error message" // Case 15
    };
```

```
    for (int i = 0; i < 15; ++i) {
        int day = testCases[i][0];
        int month = testCases[i][1];
        int year = testCases[i][2];
        string result = previousDate(day, month, year);
        cout << "Input: (" << day << ", " << month << ", " << year << ") | "
             << "Expected: " << expectedOutcomes[i] << " | "
             << "Actual: " << result << endl;
    }
}

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

Q2.

1.

Partition Class

Input (v, a)	Expected Outcome	Description
(3, [1, 2, 3, 4])	2	Value exists (at index 2)
(5, [1, 2, 3, 4])	-1	Value does not exist
(1, [1])	0	Value exists in single-element array
(2, [])	-1	Empty array
(0, [0])	0	Value exists (at index 0)
(10, [5, 10, 15])	1	Value exists (at index 1)

Boundary Class

Input (v, a)	Expected Outcome	Description
(1, [1])	0	Single element array, value exists
(2, [1])	-1	Single element array, value does not exist
(3, [1, 2])	-1	Two-element array, value not found
(1, [1, 2])	0	Two-element array, value found at start

(2, [1, 2])	1	Two-element array, value found at end
(5, [])	-1	Empty array

Code

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

using namespace std;

// Function to perform linear search
int linearSearch(int v, const vector<int>& a) {
    for (int i = 0; i < a.size(); ++i) {
        if (a[i] == v) {
            return i;
        }
    }
    return -1;
}

// Function to run test cases
void runTests() {
    vector<pair<int, vector<int>>> testCases = {
        {3, {1, 2, 3, 4}}, // Value exists
        {5, {1, 2, 3, 4}}, // Value does not exist
        {1, {1}},          // Single element array, value exists
        {2, {}},           // Empty array
        {0, {0}},          // Value exists at index 0
        {10, {5, 10, 15}}, // Value exists at index 1
        {2, {1}},          // Single element array, value does not exist
        {3, {1, 2}},       // Two-element array, value not found
        {1, {1, 2}},       // Two-element array, value found at start
        {2, {1, 2}},       // Two-element array, value found at end
        {5, {}}            // Empty array
    };

    vector<int> expectedOutcomes = {
        2, // Case 1
        -1, // Case 2
        0, // Case 3
        -1, // Case 4
        0, // Case 5
        1, // Case 6
```

```
        -1,  // Case 7
        -1,  // Case 8
        0,   // Case 9
        1,   // Case 10
        -1   // Case 11
    };

    for (size_t i = 0; i < testCases.size(); ++i) {
        int value = testCases[i].first;
        const vector<int>& array = testCases[i].second;
        int result = linearSearch(value, array);
        cout << "Input: (" << value << ", [";
        for (const auto& elem : array) {
            cout << elem << " ";
        }
        cout << "]) | Expected: " << expectedOutcomes[i] << " | Actual: " << result << endl;
    }
}

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

2.

Partition Class

Input (v, a)	Expected Outcome	Description
(3, [1, 2, 3, 3, 4])	2	Value exists multiple times
(5, [1, 2, 3, 3, 4])	0	Value does not exist
(1, [1])	1	Value exists in single-element array
(2, [])	0	Empty array
(0, [0])	1	Value exists at index 0
(10, [5, 10, 10, 15])	2	Value exists multiple times

Boundary Class

Input (v, a)	Expected Outcome	Description
(1, [1])	1	Single element array, value exists
(2, [1])	0	Single element array, value does not exist
(3, [1, 2])	0	Two-element array, value not found
(1, [1, 2])	1	Two-element array, value found at start
(2, [1, 2])	1	Two-element array, value found at end
(5, [])	0	Empty array

Code

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

using namespace std;

// Function to count occurrences of value v in array a
int countItem(int v, const vector<int>& a) {
    int count = 0;
    for (int i = 0; i < a.size(); i++) {
        if (a[i] == v) {
            count++;
        }
    }
    return count;
}

// Function to run test cases
void runTests() {
    vector<pair<int, vector<int>>> testCases = {
        {3, {1, 2, 3, 3, 4}}, // Value exists multiple times
        {5, {1, 2, 3, 3, 4}}, // Value does not exist
        {1, {1}},             // Single element array, value exists
        {2, {}},              // Empty array
        {0, {0}},             // Value exists at index 0
        {10, {5, 10, 10, 15}}, // Value exists multiple times
        {2, {1}},             // Single element array, value does not exist
        {3, {1, 2}},          // Two-element array, value not found
        {1, {1, 2}},          // Two-element array, value found at start
```

```
        {2, {1, 2}},            // Two-element array, value found at end
        {5, {}}                 // Empty array
    };

    vector<int> expectedOutcomes = {
        2, // Case 1
        0, // Case 2
        1, // Case 3
        0, // Case 4
        1, // Case 5
        2, // Case 6
        0, // Case 7
        0, // Case 8
        1, // Case 9
        1, // Case 10
        0    // Case 11
    };

    for (size_t i = 0; i < testCases.size(); ++i) {
        int value = testCases[i].first;
        const vector<int>& array = testCases[i].second;
        int result = countItem(value, array);
        cout << "Input: (" << value << ", [";
        for (const auto& elem : array) {
            cout << elem << " ";
        }
        cout << "]) | Expected: " << expectedOutcomes[i] << " | Actual: " << result << endl;
    }
}

int main() {
    runTests();
```

3.

Partition Class

Input (v, a)	Expected Outcome	Description
(3, [1, 2, 3, 4])	2	Value exists (at index 2)
(5, [1, 2, 3, 4])	-1	Value does not exist
(1, [1])	0	Value exists in single-element array
(2, [])	-1	Empty array
(0, [0])	0	Value exists (at index 0)
(10, [5, 10, 15])	1	Value exists (at index 1)

Boundary Class

Input (v, a)	Expected Outcome	Description
(1, [1])	0	Single element array, value exists
(2, [1])	-1	Single element array, value does not exist
(3, [1, 2])	-1	Two-element array, value not found
(1, [1, 2])	0	Two-element array, value found at start
(2, [1, 2])	1	Two-element array, value found at end
(5, [])	-1	Empty array

Code

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

using namespace std;

// Function to perform binary search
int binarySearch(int v, const vector<int>& a) {
    int lo = 0;
    int 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; // Value not found
}

// Function to run test cases
void runTests() {
    vector<pair<int, vector<int>>> testCases = {
        {3, {1, 2, 3, 4}}, // Value exists
        {5, {1, 2, 3, 4}}, // Value does not exist
        {1, {1}},          // Single element array, value exists
        {2, {}},           // Empty array
        {0, {0}},          // Value exists at index 0
        {10, {5, 10, 15}}, // Value exists at index 1
        {2, {1}},          // Single element array, value does not exist
        {3, {1, 2}},       // Two-element array, value not found
        {1, {1, 2}},       // Two-element array, value found at start
        {2, {1, 2}},       // Two-element array, value found at end
        {5, {}}            // Empty array
    };

    vector<int> expectedOutcomes = {
        2, // Case 1
        -1, // Case 2
```

```
        0,   // Case 3
        -1,  // Case 4
        0,   // Case 5
        1,   // Case 6
        -1,  // Case 7
        -1,  // Case 8
        0,   // Case 9
        1,   // Case 10
        -1   // Case 11
    };

    for (size_t i = 0; i < testCases.size(); ++i) {
        int value = testCases[i].first;
        const vector<int>& array = testCases[i].second;
        int result = binarySearch(value, array);
        cout << "Input: (" << value << ", [";
        for (const auto& elem : array) {
            cout << elem << " ";
        }
        cout << "]) | Expected: " << expectedOutcomes[i] << " | Actual: " << result << endl;
    }
}

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

4.

Partition Class

Input (a, b, c)	Expected Outcome	Description
(3, 3, 3)	0	Equilateral triangle
(3, 3, 4)	1	Isosceles triangle
(3, 4, 3)	1	Isosceles triangle
(4, 3, 3)	1	Isosceles triangle
(3, 4, 5)	2	Scalene triangle
(1, 2, 3)	3	Invalid triangle
(1, 1, 2)	3	Invalid triangle

(0, 0, 0)	3	Invalid triangle
(-1, 2, 3)	3	Invalid triangle
(5, 5, 10)	3	Invalid triangle

Boundary Class

Input (a, b, c)	Expected Outcome	Description
(1, 1, 2)	3	Invalid triangle
(1, 2, 2)	1	Isosceles triangle
(2, 2, 2)	0	Equilateral triangle
(3, 4, 5)	2	Scalene triangle
(5, 5, 10)	3	Invalid triangle
(0, 1, 1)	3	Invalid triangle
(-1, -1, -1)	3	Invalid triangle
(0, 0, 0)	3	Invalid triangle

Code

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

using namespace std;

// Constants for triangle
types const int
EQUILATERAL = 0; const int
ISOSCELES = 1; const int
SCALENE = 2; const int
INVALID = 3;

// Function to determine the type of triangle
int triangle(int a, int b, int c) {
    // Check for invalid triangle
    if (a >= b + c || b >= a + c || c >= a + b) {
        return INVALID;
    }
    // Check for equilateral triangle
```

```
    if (a == b && b == c) {
        return EQUILATERAL;
    }
    // Check for isosceles triangle
    if (a == b || a == c || b == c) {
        return ISOSCELES;
    }
    // Otherwise, it's scalene
    return SCALENE;
}

// Function to run test cases
void runTests() {
    vector<tuple<int, int, int>> testCases = {
        {3, 3, 3},    // Equilateral
        {3, 3, 4},    // Isosceles
        {3, 4, 3},    // Isosceles
        {4, 3, 3},    // Isosceles
        {3, 4, 5},    // Scalene
        {1, 2, 3},    // Invalid
        {1, 1, 2},    // Invalid
        {0, 0, 0},    // Invalid
        {-1, 2, 3}, // Invalid
        {5, 5, 10}, // Invalid
        {1, 2, 2},    // Isosceles
        {2, 2, 2},    // Equilateral
        {0, 1, 1},    // Invalid
        {-1, -1, -1}, // Invalid
    };

    vector<int> expectedOutcomes = {
        EQUILATERAL, // Case 1
        ISOSCELES,    // Case 2
        ISOSCELES,    // Case 3
        ISOSCELES,    // Case 4
        SCALENE,      // Case 5
        INVALID,      // Case 6
        INVALID,      // Case 7
        INVALID,      // Case 8
        INVALID,      // Case 9
        INVALID,      // Case 10
        ISOSCELES,    // Case 11
        EQUILATERAL, // Case 12
        INVALID,      // Case 13
```

```
        INVALID      // Case 14
    };

    for (size_t i = 0; i < testCases.size(); ++i) {
        int a, b, c;
        tie(a, b, c) = testCases[i];
        int result = triangle(a, b, c);
        cout << "Input: (" << a << ", " << b << ", " << c << ") | Expected: " << expectedOutcomes[i]
<< " | Actual: " << result << endl;
    }
}

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

5.

Partition Class

Test Case	Input (s1, s2)	Expected Output	Description
TC1	("pre", "prefix")	true	Valid prefix
TC2	("prefix", "prefix")	true	Same strings
TC3	("", "non-empty")	true	Empty string is a prefix
TC4	("long", "longer")	true	Valid prefix
TC5	("longer", "long")	false	s1 is longer than s2
TC6	("pre", "test")	false	Different characters
TC7	("prefix", "pre")	false	s1 is longer than s2
TC8	("", "")	true	Both strings are empty
TC9	("a", "abc")	true	Valid prefix
TC10	("ab", "abc")	true	Valid prefix
TC11	("abc", "abc")	true	Same strings
TC12	("abc", "ab")	false	s1 is longer than s2

Boundary Class

Test Case	Input (s1, s2)	Expected Output	Description
TC1	("", "")	true	Both strings are empty
TC2	("a", "")	false	Single character s1 with empty s2
TC3	("", "a")	true	Empty s1 with non-empty s2
TC4	("a", "a")	true	Both strings are the same (1 char)
TC5	("abc", "abc")	true	Both strings are the same (3 chars)
TC6	("abc", "abcd")	true	s1 is a prefix of s2 (3 chars)
TC7	("abcd", "abc")	false	s1 is longer than s2
TC8	("abc", "ab")	false	s1 is longer than s2
TC9	("long_prefix", "long_prefix_extra")	true	s1 is a valid prefix of s2
TC10	("longer", "long_prefix")	false	s1 is not a prefix of s2
TC11	("pre", "prefix")	true	Valid prefix (both strings non-empty)
TC12	("pre", "pr")	false	s1 is longer than s2

Code

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

using namespace std;

// Constants for triangle
types const int
EQUILATERAL = 0; const int
ISOSCELES = 1; const int
SCALENE = 2; const int
INVALID = 3;
```

```
// Function to determine the type of triangle
int triangle(int a, int b, int c) {
    // Check for invalid triangle
    if (a >= b + c || b >= a + c || c >= a + b) {
        return INVALID;
    }
    // Check for equilateral triangle
    if (a == b && b == c) {
        return EQUILATERAL;
    }
    // Check for isosceles triangle
    if (a == b || a == c || b == c) {
        return ISOSCELES;
    }
    // Otherwise, it's scalene
    return SCALENE;
}

// Function to run test cases
void runTests() {
    vector<tuple<int, int, int>> testCases = {
        {3, 3, 3},    // Equilateral
        {3, 3, 4},    // Isosceles
        {3, 4, 3},    // Isosceles
        {4, 3, 3},    // Isosceles
        {3, 4, 5},    // Scalene
        {1, 2, 3},    // Invalid
        {1, 1, 2},    // Invalid
        {0, 0, 0},    // Invalid
        {-1, 2, 3}, // Invalid
        {5, 5, 10}, // Invalid
        {1, 2, 2},    // Isosceles
        {2, 2, 2},    // Equilateral
        {0, 1, 1},    // Invalid
        {-1, -1, -1}, // Invalid
    };

    vector<int> expectedOutcomes = {
        EQUILATERAL, // Case 1
        ISOSCELES,    // Case 2
        ISOSCELES,    // Case 3
        ISOSCELES,    // Case 4
        SCALENE,      // Case 5
        INVALID,      // Case 6
```

```
        INVALID,      // Case 7
        INVALID,      // Case 8
        INVALID,      // Case 9
        INVALID,      // Case 10
        ISOSCELES,    // Case 11
        EQUILATERAL, // Case 12
        INVALID,      // Case 13
        INVALID       // Case 14
    };

    for (size_t i = 0; i < testCases.size(); ++i) {
        int a, b, c;
        tie(a, b, c) = testCases[i];
        int result = triangle(a, b, c);
        cout << "Input: (" << a << ", " << b << ", " << c << ") | Expected: " << expectedOutcomes[i]
<< " | Actual: " << result << endl;
    }
}

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

6.

a)

Class Type	Example Input	Description
Equilateral	(3.0, 3.0, 3.0)	All sides equal
Isosceles	(3.0, 3.0, 4.0)	Two sides equal
Scalene	(3.0, 4.0, 5.0)	All sides different
Right-Angled	(3.0, 4.0, 5.0)	Fulfills Pythagorean theorem
Invalid: Impossible	(1.0, 2.0, 3.0)	Fails triangle inequality
Invalid: Negative	(-1.0, 2.0, 3.0)	One side negative
Invalid: Zero	(0.0, 1.0, 1.0)	One side zero
Invalid: Non-numeric	(a, b, c)	Input is not a valid number

b)

Test Case	Input (A, B, C)	Expected Output	Equivalence Class Covered
TC1	(3.0, 3.0, 3.0)	"Equilateral triangle"	Equilateral
TC2	(3.0, 3.0, 4.0)	"Isosceles triangle"	Isosceles
TC3	(3.0, 4.0, 5.0)	"Scalene triangle"	Scalene
TC4	(3.0, 4.0, 3.0)	"Isosceles triangle"	Isosceles
TC5	(5.0, 12.0, 13.0)	"Right-angled triangle"	Right-Angled
TC6	(1.0, 2.0, 3.0)	"Invalid triangle"	Invalid: Impossible
TC7	(-1.0, 2.0, 3.0)	"Invalid triangle"	Invalid: Negative
TC8	(0.0, 1.0, 1.0)	"Invalid triangle"	Invalid: Zero
TC9	(a, b, c)	"Invalid input"	Invalid: Non-numeric
TC10	(5.0, 5.0, 10.0)	"Invalid triangle"	Invalid: Impossible

c)

Test Case	Input (A, B, C)	Expected Output	Description
TC1	(3.0, 4.0, 6.0)	"Scalene triangle"	Valid scalene triangle
TC2	(3.0, 4.0, 7.0)	"Invalid triangle"	A+B=7A + B = 7A+B=7 is not greater than CCC
TC3	(5.0, 7.0, 11.0)	"Scalene triangle"	Valid scalene triangle
TC4	(5.0, 7.0, 12.0)	"Invalid triangle"	A+B=12A + B = 12A+B=12 is not greater than CCC
TC5	(2.0, 3.0, 4.0)	"Scalene triangle"	Valid scalene triangle
TC6	(2.0, 3.0, 5.0)	"Invalid triangle"	A+B=5A + B = 5A+B=5 is not greater than CCC

TC7	(1.0, 1.0, 2.0)	"Invalid triangle"	A+B=2A + B = 2A+B=2 is not greater than CCC
TC8	(1.0, 2.0, 3.0)	"Invalid triangle"	A+B=3A + B = 3A+B=3 is not greater than CCC

d)

Test Case	Input (A, B, C)	Expected Output	Description
TC1	(3.0, 4.0, 3.0)	"Isosceles triangle"	Valid isosceles triangle
TC2	(4.0, 4.0, 8.0)	"Invalid triangle"	A+C=6A + C = 6A+C=6 is not greater than BBB
TC3	(5.0, 7.0, 5.0)	"Isosceles triangle"	Valid isosceles triangle
TC4	(5.0, 10.0, 5.0)	"Invalid triangle"	A+C=10A + C = 10A+C=10 is not greater than BBB
TC5	(2.0, 3.0, 2.0)	"Isosceles triangle"	Valid isosceles triangle
TC6	(1.0, 1.0, 2.0)	"Invalid triangle"	A+C=2A + C = 2A+C=2 is not greater than BBB
TC7	(0.0, 1.0, 0.0)	"Invalid triangle"	Zero lengths are invalid
TC8	(1.0, 1.0, 2.0)	"Invalid triangle"	A+C=2A + C = 2A+C=2 is not greater than BBB

e)

Test Case	Input (A, B, C)	Expected Output	Description
TC1	(3.0, 3.0, 3.0)	"Equilateral triangle"	Valid equilateral triangle
TC2	(4.0, 4.0, 8.0)	"Invalid triangle"	A+B=6A + B = 6A+B=6 is not greater than CCC
TC3	(5.0, 5.0, 5.0)	"Equilateral triangle"	Valid equilateral triangle
TC4	(2.0, 2.0, 4.0)	"Invalid triangle"	A+B=4A + B = 4A+B=4 is not greater than CCC

TC5	(1.0, 1.0, 1.0)	"Equilateral triangle"	Valid equilateral triangle
TC6	(0.0, 0.0, 0.0)	"Invalid triangle"	Zero lengths are invalid
TC7	(-1.0, -1.0, -1.0)	"Invalid triangle"	Negative lengths are invalid
TC8	(2.0, 2.0, 2.0)	"Equilateral triangle"	Valid equilateral triangle

f)

Test Case	Input (A, B, C)	Expected Output	Description
TC1	(3.0, 4.0, 5.0)	"Right-angled triangle"	Valid right-angled triangle
TC2	(5.0, 12.0, 13.0)	"Right-angled triangle"	Valid right-angled triangle
TC3	(6.0, 8.0, 10.0)	"Right-angled triangle"	Valid right-angled triangle
TC4	(1.0, 1.0, 1.0)	"Invalid triangle"	Not a right-angled triangle
TC5	(3.0, 4.0, 6.0)	"Invalid triangle"	A2+B2≠C2A^2 + B^2 \neq C^2A2+B2=C2
TC6	(5.0, 5.0, 7.0)	"Invalid triangle"	A2+B2≠C2A^2 + B^2 \neq C^2A2+B2=C2
TC7	(0.0, 0.0, 0.0)	"Invalid triangle"	Zero lengths are invalid
TC8	(8.0, 15.0, 17.0)	"Right-angled triangle"	Valid right-angled triangle
TC9	(7.0, 24.0, 25.0)	"Right-angled triangle"	Valid right-angled triangle
TC10	(10.0, 24.0, 26.0)	"Invalid triangle"	A2+B2≠C2A^2 + B^2 \neq C^2A2+B2=C2

g)

Test Case	Input (A, B, C)	Expected Output	Description
TC1	(1.0, 2.0, 3.0)	"Invalid triangle"	A+B=CA + B = CA+B=C (Degenerate case)
TC2	(2.0, 3.0, 6.0)	"Invalid triangle"	A+B<CA + B < CA+B<C
TC3	(4.0, 5.0, 10.0)	"Invalid triangle"	A+B<CA + B < CA+B<C
TC4	(5.0, 7.0, 12.0)	"Invalid triangle"	A+B<CA + B < CA+B<C

TC5	(3.0, 3.0, 7.0)	"Invalid triangle"	A+B<CA + B < CA+B<C
TC6	(0.0, 1.0, 1.0)	"Invalid triangle"	Zero length is invalid
TC7	(-1.0, 1.0, 1.0)	"Invalid triangle"	Negative length is invalid
TC8	(1.0, 1.0, 2.0)	"Invalid triangle"	A+B=CA + B = CA+B=C (Degenerate case)
TC9	(2.0, 2.0, 5.0)	"Invalid triangle"	A+B<CA + B < CA+B<C
TC10	(10.0, 10.0, 20.0)	"Invalid triangle"	A+B<CA + B < CA+B<C

h)

Test Case	Input (A, B, C)	Expected Output	Description
TC1	(0.0, 1.0, 1.0)	"Invalid triangle"	One side is zero
TC2	(1.0, 0.0, 1.0)	"Invalid triangle"	One side is zero
TC3	(1.0, 1.0, 0.0)	"Invalid triangle"	One side is zero
TC4	(-1.0, 1.0, 1.0)	"Invalid triangle"	One side is negative
TC5	(1.0, -1.0, 1.0)	"Invalid triangle"	One side is negative
TC6	(1.0, 1.0, -1.0)	"Invalid triangle"	One side is negative
TC7	(0.0, 0.0, 0.0)	"Invalid triangle"	All sides are zero
TC8	(-1.0, -1.0, -1.0)	"Invalid triangle"	All sides are negative
TC9	(0.0, -1.0, -1.0)	"Invalid triangle"	One side is zero, others negative
TC10	(-1.0, 0.0, -1.0)	"Invalid triangle"	One side is zero, others negative