IT314
Software Engineering

Lab 8

Name: Jainam Patel
Student Id: 202201514

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.

Equivalence Partitioning:

Equivalence Classes:

- Valid day: day∈[1,31]
- Invalid day: day<1 or day>31
- Valid month: month∈[1,12]
- Invalid month: month<1 or month>12
- Valid year: year∈[1900,2015]
- Invalid year: year<1900 or year>2015

ID	Input (Day, Month, Year)	Expected Output	Class
E1	15, 6, 2000	14, 6, 2000	Valid: Normal day
E2	1, 7, 2010	30, 6, 2010	Valid: First day of month
E3	1, 1, 2000	31, 12, 1999	Valid: First day of year

E4	31, 8, 2015	30, 8, 2015	Valid: Last day of 31-day month
E5	30, 4, 2005	29, 4, 2005	Valid: Last day of 30-day month
E6	28, 2, 2001	27, 2, 2001	Valid: Last day of Feb (non-leap)
E7	29, 2, 2012	28, 2, 2012	Valid: Last day of Feb (leap year)
E8	0, 5, 2000	Invalid date	Invalid: Day < 1
E9	32, 5, 2000	Invalid date	Invalid: Day > 31
E10	15, 0, 2000	Invalid date	Invalid: Month < 1
E11	15, 13, 2000	Invalid date	Invalid: Month > 12
E12	10, 10, 1899	Invalid date	Invalid: Year < 1900
E13	10, 10, 2016	Invalid date	Invalid: Year > 2015
E14	31, 4, 2000	Invalid date	Invalid: Day > max for month
E15	29, 2, 2001	Invalid date	Invalid: Feb 29 in non-leap year

Boundary Value Analysis:

ID	Input (Day, Month, Year)	Expected Output	Boundary
B1	1, 1, 1900	31, 12, 1899	Min valid year
B2	31, 12, 2015	30, 12, 2015	Max valid year
B3	1, 1, 1899	Invalid date	Year below min
B4	1, 1, 2016	Invalid date	Year above max
B5	1, 1, 2000	31, 12, 1999	Min valid month
B6	31, 12, 2000	30, 12, 2000	Max valid month

B7	15, 0, 2000	Invalid date	Month below min
B8	15, 13, 2000	Invalid date	Month above max
B9	1, 5, 2000	30, 4, 2000	Min valid day
B10	31, 5, 2000	30, 5, 2000	Max valid day (31-day month)
B11	30, 4, 2000	29, 4, 2000	Max valid day (30-day month)
B12	29, 2, 2000	28, 2, 2000	Max valid day (Feb, leap year)
B13	28, 2, 2001	27, 2, 2001	Max valid day (Feb,non-leap year)
B14	0, 5, 2000	Invalid date	Day below min
B15	32, 5, 2000	Invalid date	Day above max

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.

```
#include <iostream>
#include <string>
#include <vector>
#include <tuple>
using namespace std;

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

string get_previous_date(int day, int month, int year) {
    if (year < 1900 || year > 2015 || month < 1 || month > 12 || day < 1
|| day > 31) {
        return "Invalid date";
    }

    vector<int> days_in_month = {0, 31, 28, 31, 30, 31, 30, 31, 31, 30,
31, 30, 31};
```

```
    if (is_leap_year(year)) {
        days_in_month[2] = 29;
    }

    if (day > days_in_month[month]) {
        return "Invalid date";
    }

    if (day > 1) {
        return to_string(day - 1) + ", " + to_string(month) + ", " +
to_string(year);
    } else if (month > 1) {
        int prev_month = month - 1;
        return to_string(days_in_month[prev_month]) + ", " +
to_string(prev_month) + ", " + to_string(year);
    } else {
        return "31, 12, " + to_string(year - 1);
    }
}

int main() {
    vector<tuple<int, int, int, string>> test_cases = {
        // Equivalence Partitioning
        {15, 6, 2000, "14, 6, 2000"},
        {1, 7, 2010, "30, 6, 2010"},
        {1, 1, 2000, "31, 12, 1999"},
        {31, 8, 2015, "30, 8, 2015"},
        {30, 4, 2005, "29, 4, 2005"},
        {28, 2, 2001, "27, 2, 2001"},
        {29, 2, 2012, "28, 2, 2012"},
        {0, 5, 2000, "Invalid date"},
        {32, 5, 2000, "Invalid date"},
        {15, 0, 2000, "Invalid date"},
        {15, 13, 2000, "Invalid date"},
        {10, 10, 1899, "Invalid date"},
        {10, 10, 2016, "Invalid date"},
        {31, 4, 2000, "Invalid date"},
        {29, 2, 2001, "Invalid date"},

        // Boundary Value Analysis
```

```
        {1, 1, 1900, "31, 12, 1899"},
        {31, 12, 2015, "30, 12, 2015"},
        {1, 1, 1899, "Invalid date"},
        {1, 1, 2016, "Invalid date"},
        {1, 1, 2000, "31, 12, 1999"},
        {31, 12, 2000, "30, 12, 2000"},
        {15, 0, 2000, "Invalid date"},
        {15, 13, 2000, "Invalid date"},
        {1, 5, 2000, "30, 4, 2000"},
        {31, 5, 2000, "30, 5, 2000"},
        {30, 4, 2000, "29, 4, 2000"},
        {29, 2, 2000, "28, 2, 2000"},
        {28, 2, 2001, "27, 2, 2001"},
        {0, 5 , 2000, "Invalid date"},
        {32, 5, 2000, "Invalid date"},
    };

    for (int i = 0; i < test_cases.size(); ++i) {
        int day, month, year;
        string expected;
        tie(day, month, year, expected) = test_cases[i];
        string result = get_previous_date(day, month, year);
        cout << "Test case " << i + 1 << ": ";
        if (result == expected) {
            cout << "Passed" << endl;
        } else {
            cout << "Failed. Expected: " << expected << ", Got: " <<
result << endl;
        }
    }

    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.

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

A. Equivalence Partitioning:

1. Search for a value present in the middle of the array
2. Search for a value present at the beginning of the array
3. Search for a value present at the end of the array
4. Search for a value not present in the array
5. Search in an empty array
6. Search in an array with all elements equal to the search value

Test Case	Search Value	Array	Expected Result
Middle value	3	[1, 2, 3, 4, 5]	2
First value	1	[1, 2, 3, 4, 5]	0
Last value	5	[1, 2, 3, 4, 5]	4
Value not present	6	[1, 2, 3, 4, 5]	-1
Empty array	1	[]	-1
All equal	2	[2, 2, 2, 2, 2]	0

B. Boundary Value Analysis:

1. Search for the first element in the array
2. Search for the last element in the array
3. Search for a value just before the first occurrence of it in the array
4. Search for a value just after the last occurrence of it in the array
5. Search in an array with only one element (matching the search value)
6. Search in an array with only one element (not matching the search value)

Test Case	Search Value	Array	Expected Result
First element	1	[1, 2, 3, 4, 5]	0
Last element	5	[1, 2, 3, 4, 5]	4
Just before first occurrence	2	[1, 3, 2, 4, 5]	2
Just after last occurrence	4	[1, 2, 3, 4, 3, 2, 1]	3
Single element (match)	1	[1]	0
Single element (no match)	2	[1]	-1

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

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

void runTest(const   string& testName, int searchValue, const
vector<int>& arr, int expectedResult) {
    int result = linearSearch(searchValue, arr);
      cout << testName << ": ";
    if (result == expectedResult) {
          cout << "PASSED";
    } else {
          cout << "FAILED (Expected: " << expectedResult << ", Got: " <<
result << ")";
    }
      cout <<   endl;
}

int main() {
    cout << "Equivalence Partitioning Tests:" <<   endl;
    runTest("Middle value", 3, {1, 2, 3, 4, 5}, 2);
    runTest("First value", 1, {1, 2, 3, 4, 5}, 0);
    runTest("Last value", 5, {1, 2, 3, 4, 5}, 4);
    runTest("Value not present", 6, {1, 2, 3, 4, 5}, -1);
    runTest("Empty array", 1, {}, -1);
    runTest("All equal", 2, {2, 2, 2, 2, 2}, 0);

    cout << "\nBoundary Value Analysis Tests:" <<   endl;
    runTest("First element", 1, {1, 2, 3, 4, 5}, 0);
    runTest("Last element", 5, {1, 2, 3, 4, 5}, 4);
    runTest("Just before first occurrence", 2, {1, 3, 2, 4, 5}, 2);
```

```
    runTest("Just after last occurrence", 4, {1, 2, 3, 4, 3, 2, 1}, 3);
    runTest("Single element (match)", 1, {1}, 0);
    runTest("Single element (no match)", 2, {1}, -1);

    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);

}
```

A. Equivalence Partitioning:

1. Count a value present multiple times in the array
2. Count a value present once in the array
3. Count a value not present in the array
4. Count in an empty array
5. Count in an array with all elements equal to the search value
6. Count with a negative search value

Test Case	Search Value	Array	Expected Result
Multiple occurrences	2	[1, 2, 3, 2, 4, 2]	3
Single occurrence	3	[1, 2, 3, 4, 5]	1
No occurrence	6	[1, 2, 3, 4, 5]	0
Empty array	1	[]	0
All equal	2	[2, 2, 2, 2, 2]	5
Negative value	-1	[-1, 2, -1, 4, -1]	3

B. Boundary Value Analysis:

1. Count the first element in the array
2. Count the last element in the array
3. Count in an array with only one element (matching the search value)
4. Count in an array with only one element (not matching the search value)
5. Count in an array with two elements (both matching)
6. Count in an array with two elements (one matching, one not)

Test Case	Search Value	Array	Expected Result
First element	1	[1, 2, 3, 4, 5]	1
Last element	5	[1, 2, 3, 4, 5]	1
Single element (match)	1	[1]	1
Single element (no match)	2	[1]	0
Two elements (both match)	2	[2, 2]	2
Two elements (one match)	2	[1, 2]	1

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

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;
}

void runTest(const  string& testName, int searchValue, const  vector<int>&
arr, int expectedResult) {
    int result = countItem(searchValue, arr);
     cout << testName << ": ";
    if (result == expectedResult) {
         cout << "PASSED";
    } else {
         cout << "FAILED (Expected: " << expectedResult << ", Got: " <<
result << ")";
    }
     cout <<  endl;
}

int main() {
    cout << "Equivalence Partitioning Tests:" <<  endl;
    runTest("Multiple occurrences", 2, {1, 2, 3, 2, 4, 2}, 3);
    runTest("Single occurrence", 3, {1, 2, 3, 4, 5}, 1);
    runTest("No occurrence", 6, {1, 2, 3, 4, 5}, 0);
    runTest("Empty array", 1, {}, 0);
    runTest("All equal", 2, {2, 2, 2, 2, 2}, 5);
    runTest("Negative value", -1, {-1, 2, -1, 4, -1}, 3);

    cout << "\nBoundary Value Analysis Tests:" <<  endl;
    runTest("First element", 1, {1, 2, 3, 4, 5}, 1);
    runTest("Last element", 5, {1, 2, 3, 4, 5}, 1);
    runTest("Single element (match)", 1, {1}, 1);
    runTest("Single element (no match)", 2, {1}, 0);
```

```
    runTest("Two elements (both match)", 2, {2, 2}, 2);
    runTest("Two elements (one match)", 2, {1, 2}, 1);

    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);
}
```

A. Equivalence Partitioning:

1. Search for a value in the middle of the array
2. Search for the first value in the array
3. Search for the last value in the array
4. Search for a value not in the array (less than all elements)
5. Search for a value not in the array (greater than all elements)
6. Search for a value not in the array (between existing elements)
7. Search in an empty array

Test Case	Search Value	Array	Expected Result
Middle value	5	[1, 3, 5, 7, 9]	2
First value	1	[1, 3, 5, 7, 9]	0
Last value	9	[1, 3, 5, 7, 9]	4
Value less than all	0	[1, 3, 5, 7, 9]	-1
Value greater than all	10	[1, 3, 5, 7, 9]	-1
Value between elements	4	[1, 3, 5, 7, 9]	-1
Empty array	5	[]	-1

B. Boundary Value Analysis:

1. Search in an array with only one element (matching)
2. Search in an array with only one element (not matching)
3. Search for a value just less than the middle element
4. Search for a value just greater than the middle element
5. Search in an array with two elements (first matching)
6. Search in an array with two elements (second matching)

Test Case	Search Value	Array	Expected Result
Single element (match)	1	[1]	0
Single element (no match)	2	[1]	-1
Just less than middle	4	[1, 3, 5, 7, 9]	-1
Just greater than middle	6	[1, 3, 5, 7, 9]	-1
Two elements (first match)	1	[1, 3]	0
Two elements (second match)	3	[1, 3]	1

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

int binarySearch(int v, const vector<int>& a) {
    int lo = 0;
    int hi = a.size() - 1;
    while (lo <= hi) {
        int mid = lo + (hi - lo) / 2; // Avoid potential overflow
        if (v == a[mid])
            return mid;
        else if (v < a[mid])
            hi = mid - 1;
        else
            lo = mid + 1;
    }
    return -1;
}

void runTest(const string& testName, int searchValue, const vector<int>&
arr, int expectedResult) {
    int result = binarySearch(searchValue, arr);
    cout << testName << ": ";
```

```
    if (result == expectedResult) {
        cout << "PASSED";
    } else {
        cout << "FAILED (Expected: " << expectedResult << ", Got: " <<
result << ")";
    }
    cout << endl;
}

int main() {
    cout << "Equivalence Partitioning Tests:" << endl;
    runTest("Middle value", 5, {1, 3, 5, 7, 9}, 2);
    runTest("First value", 1, {1, 3, 5, 7, 9}, 0);
    runTest("Last value", 9, {1, 3, 5, 7, 9}, 4);
    runTest("Value less than all", 0, {1, 3, 5, 7, 9}, -1);
    runTest("Value greater than all", 10, {1, 3, 5, 7, 9}, -1);
    runTest("Value between elements", 4, {1, 3, 5, 7, 9}, -1);
    runTest("Empty array", 5, {}, -1);

    cout << "\nBoundary Value Analysis Tests:" << endl;
    runTest("Single element (match)", 1, {1}, 0);
    runTest("Single element (no match)", 2, {1}, -1);
    runTest("Just less than middle", 4, {1, 3, 5, 7, 9}, -1);
    runTest("Just greater than middle", 6, {1, 3, 5, 7, 9}, -1);
    runTest("Two elements (first match)", 1, {1, 3}, 0);
    runTest("Two elements (second match)", 3, {1, 3}, 1);

    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);
}
```

A. Equivalence Partitioning:

1. Equilateral triangle
2. Isosceles triangle (a == b)
3. Isosceles triangle (a == c)
4. Isosceles triangle (b == c)
5. Scalene triangle
6. Invalid triangle (a >= b + c)
7. Invalid triangle (b >= a + c)
8. Invalid triangle (c >= a + b)
9. Invalid triangle (negative sides)
10. Invalid triangle (zero sides)

Test Case	Side a	Side b	Side c	Expected Result
Equilateral	5	5	5	EQUILATERAL (0)
Isosceles (a == b)	5	5	3	ISOSCELES (1)

Isosceles (a == c)	5	3	5	ISOSCELES (1)
Isosceles (b == c)	3	5	5	ISOSCELES (1)
Scalene	3	4	5	SCALENE (2)
Invalid (a >= b + c)	5	2	2	INVALID (3)
Invalid (b >= a + c)	2	5	2	INVALID (3)
Invalid (c >= a + b)	2	2	5	INVALID (3)
Invalid (negative sides)	-1	2	2	INVALID (3)
Invalid (zero sides)	0	2	2	INVALID (3)

B. Boundary Value Analysis:

1. Minimum valid equilateral triangle (1, 1, 1)
2. Minimum valid isosceles triangle (2, 2, 1)
3. Minimum valid scalene triangle (3, 4, 5)
4. Just valid triangle (a + b = c + 1)
5. Just invalid triangle (a + b = c)

Test Case	Side a	Side b	Side c	Expected Result
Minimum equilateral	1	1	1	EQUILATERAL (0)
Minimum isosceles	2	2	1	ISOSCELES (1)
Minimum scalene	3	4	5	SCALENE (2)
Just valid	3	4	6	SCALENE (2)
Just invalid	3	4	7	INVALID (3)

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

const int EQUILATERAL = 0;
const int ISOSCELES = 1;
const int SCALENE = 2;
const int INVALID = 3;

int triangle(int a, int b, int c) {
    if (a <= 0 || b <= 0 || c <= 0 || 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;
}

void runTest(const string& testName, int a, int b, int c, int
expectedResult) {
    int result = triangle(a, b, c);
    cout << testName << ": ";
    if (result == expectedResult) {
        cout << "PASSED";
    } else {
        cout << "FAILED (Expected: " << expectedResult << ", Got: " <<
result << ")";
    }
    cout << endl;
}

int main() {
    cout << "Equivalence Partitioning Tests:" << endl;
    runTest("Equilateral", 5, 5, 5, EQUILATERAL);
    runTest("Isosceles (a == b)", 5, 5, 3, ISOSCELES);
    runTest("Isosceles (a == c)", 5, 3, 5, ISOSCELES);
    runTest("Isosceles (b == c)", 3, 5, 5, ISOSCELES);
    runTest("Scalene", 3, 4, 5, SCALENE);
    runTest("Invalid (a >= b + c)", 5, 2, 2, INVALID);
```

```
    runTest("Invalid (b >= a + c)", 2, 5, 2, INVALID);
    runTest("Invalid (c >= a + b)", 2, 2, 5, INVALID);
    runTest("Invalid (negative sides)", -1, 2, 2, INVALID);
    runTest("Invalid (zero sides)", 0, 2, 2, INVALID);

    cout << "\nBoundary Value Analysis Tests:" << endl;
    runTest("Minimum equilateral", 1, 1, 1, EQUILATERAL);
    runTest("Minimum isosceles", 2, 2, 1, ISOSCELES);
    runTest("Minimum scalene", 3, 4, 5, SCALENE);
    runTest("Just valid", 3, 4, 6, SCALENE);
    runTest("Just invalid", 3, 4, 7, INVALID);

    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;
}
```

A. Equivalence Partitioning:

1. s1 is a prefix of s2
2. s1 is not a prefix of s2
3. s1 is longer than s2
4. s1 and s2 are identical

Test Case	s1	s2	Expected Result
s1 is a prefix of s2	"pre"	"prefix"	TRUE
s1 is not a prefix of s2	"fix"	"prefix"	FALSE
s1 is longer than s2	"prefix"	"pre"	FALSE
s1 and s2 are identical	"test"	"test"	TRUE

B. Boundary Value Analysis:

1. s1 is one character, s2 is one character (matching)
2. s1 is one character, s2 is one character (not matching)
3. s1 is one character shorter than s2 (matching prefix)
4. s1 is one character shorter than s2 (not matching prefix)

Test Case	s1	s2	Expected Result
s1 is one char, s2 is one char (matching)	"a"	"a"	TRUE
s1 is one char, s2 is one char (not matching)	"a"	"b"	FALSE
s1 is one char shorter than s2 (matching prefix)	"tes"	"test"	TRUE
s1 is one char shorter than s2 (not matching prefix)	"tes"	"best"	FALSE

```cpp
#include <iostream>

#include <string>

using namespace std;



class PrefixTest {

public:

    bool prefix(const string& s1, const string& s2) {

        if (s1.length() > s2.length()) {

            return false;

        }

        for (size_t i = 0; i < s1.length(); i++) {

            if (s1[i] != s2[i]) {

                return false;

            }

        }

        return true;

    }
```

```
    void runTest(const string& testName, const string& s1, const string&
s2, bool expectedResult) {

        bool result = prefix(s1, s2);

        cout << testName << ": ";

        if (result == expectedResult) {

            cout << "PASSED" << endl;

        } else {

            cout << "FAILED (Expected: " << (expectedResult ? "true" :
"false")

                    << ", Got: " << (result ? "true" : "false") << ")"
<< endl;

        }

    }



    void runAllTests() {

        cout << "Equivalence Partitioning Tests:" << endl;

        runTest("s1 is a prefix of s2", "pre", "prefix", true);

        runTest("s1 is not a prefix of s2", "fix", "prefix", false);

        runTest("s1 is longer than s2", "prefix", "pre", false);
```

```
        runTest("s1 and s2 are identical", "test", "test", true);

        runTest("s1 is empty", "", "test", true);

        runTest("s2 is empty", "test", "", false);

        runTest("Both s1 and s2 are empty", "", "", true);



        cout << "\nBoundary Value Analysis Tests:" << endl;

        runTest("s1 is one char, s2 is one char (matching)", "a", "a",
true);

        runTest("s1 is one char, s2 is one char (not matching)", "a", "b",
false);

        runTest("s1 is one char shorter than s2 (matching prefix)", "tes",
"test", true);

        runTest("s1 is one char shorter than s2 (not matching prefix)",
"tes", "best", false);

    }

};



int main() {

    PrefixTest tester;

    tester.runAllTests();
```

```
    return 0;


}
```

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 A2 + B2 = C2 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) Equivalence classes:

1. Scalene triangle (all sides different)
2. Isosceles triangle (two sides equal)
3. Equilateral triangle (all sides equal)
4. Right-angled triangle
5. Non-triangle (sum of any two sides ≤ third side)
6. Non-positive input

b) Test cases for each equivalence class:

1. Scalene: (3, 4, 5)
2. Isosceles: (5, 5, 7)
3. Equilateral: (6, 6, 6)
4. Right-angled: (3, 4, 5)
5. Non-triangle: (1, 2, 3)
6. Non-positive input: (0, 4, 5), (-1, 3, 4)

Test Case	Input (A, B, C)	Expected Output
Scalene, Right-angled	3, 4, 5	Scalene triangle, Right-angled triangle
Isosceles	5, 5, 7	Isosceles triangle
Equilateral	6, 6, 6	Equilateral triangle
Non-triangle	1, 2, 3	Not a valid triangle
Non-positive input	0, 4, 5	Not a valid triangle
Non-positive input	-1, 3, 4	Not a valid triangle

c) Boundary condition A + B > C (scalene triangle):

1. (3, 4, 6.99) - Just below the boundary
2. (3, 4, 7) - On the boundary
3. (3, 4, 7.01) - Just above the boundary

d) Boundary condition A = C (isosceles triangle):

1. (5, 4.99, 5) - Just below equality
2. (5, 5, 5) - On the boundary (also equilateral)
3. (5, 5.01, 5) - Just above equality

e) Boundary condition A = B = C (equilateral triangle):

1. (5, 5, 4.99) - Just below equality
2. (5, 5, 5) - On the boundary
3. (5, 5, 5.01) - Just above equality

f) Boundary condition A^2 + B^2 = C^2 (right-angle triangle):

1. (3, 4, 4.99) - Just below right angle
2. (3, 4, 5) - On the boundary (exact right angle)
3. (3, 4, 5.01) - Just above right angle

g) Non-triangle case:

1. (1, 2, 2.99) - Just below triangle formation
2. (1, 2, 3) - On the boundary
3. (1, 2, 3.01) - Just above boundary (valid triangle)

h) Non-positive input:

1. (0, 4, 5) - Zero input
2. (-0.01, 4, 5) - Slightly negative input
3. (-1, 4, 5) - Negative input

Condition	Test Case	Input (A, B, C)	Expected Output
A + B > C	Just below	3, 4, 6.99	Scalene triangle
	On boundary	3, 4, 2007	Not a valid triangle
	Just above	3, 4, 7.01	Not a valid triangle
A = C (Isosceles)	Just below	5, 4.99, 5	Scalene triangle
	On boundary	5, 5, 2005	Equilateral triangle
	Just above	5, 5.01, 5	Isosceles triangle
A = B = C (Equilateral)	Just below	5, 5, 4.99	Isosceles triangle
	On boundary	5, 5, 2005	Equilateral triangle
	Just above	5, 5, 5.01	Isosceles triangle
A^2 + B^2 = C^2 (Right-angled)	Just below	3, 4, 4.99	Scalene triangle
	On boundary	3, 4, 2005	Right-angled triangle
	Just above	3, 4, 5.01	Scalene triangle
Non-triangle	Just below	1, 2, 2.99	Not a valid triangle
	On boundary	1, 2, 2003	Not a valid triangle
	Just above	1, 2, 3.01	Scalene triangle
Non-positive input	Zero input	2000, 4, 0	Not a valid triangle
	Slightly negative	-0.01, 4, 5	Not a valid triangle
	Negative input	-1, 4, 5	Not a valid triangle

```cpp
#include <iostream>
#include <cmath>
#include <iomanip>
using namespace std;

class Triangle {
private:
    double a, b, c;

public:
    Triangle(double side1, double side2, double side3) : a(side1),
b(side2), c(side3) {}

    bool isValid() {
        return (a > 0 && b > 0 && c > 0 &&
                a + b > c && b + c > a && c + a > b);
    }

    bool isScalene() {
        return (a != b && b != c && c != a);
    }

    bool isIsosceles() {
        return (a == b || b == c || c == a);
    }

    bool isEquilateral() {
        return (a == b && b == c);
    }

    bool isRightAngled() {
        double sides[3] = {a, b, c};
        sort(sides, sides + 3);
        return abs(pow(sides[0], 2) + pow(sides[1], 2) - pow(sides[2], 2))
< 1e-6;
    }

    void classify() {
        if (!isValid()) {
            cout << "Not a valid triangle" << endl;
```

```
            return;
        }

        if (isEquilateral()) {
            cout << "Equilateral triangle" << endl;
        } else if (isIsosceles()) {
            cout << "Isosceles triangle" << endl;
        } else if (isScalene()) {
            cout << "Scalene triangle" << endl;
        }

        if (isRightAngled()) {
            cout << "Right-angled triangle" << endl;
        }
    }
};

int main() {
    double a, b, c;
    cout << "Enter three side lengths: ";
    cin >> a >> b >> c;

    Triangle t(a, b, c);
    t.classify();

    return 0;
}
```