Name - Tasmay Patel

Student Id - 202201129

IT314: Lab 8

202201129

Functional Testing (BlackBox)

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 dates or invalid dates. Design the equivalence class test cases?

1. Enlist which set of test cases have been identified using Equivalence Partitioning and Boundary Value Analysis separately.

Equivalence Classes

- E1: 1 <= month <= 12 (Valid)
- E2: month < 1 (Invalid)
- E3: month > 12 (Invalid)
- E4: 1<= Day <= 31(Valid)
- E5: Day < 1 (Invalid)

- E6: Day > 31 (Invalid)
- E7: 1900 <= year <= 2015 (Valid)
- E8: Year < 1900 (Invalid)
- E9: Year > 2015 (Invalid)

Equivalence Test cases

ID	Input(Day,Month,Year)	Expected Output	Classes Covered	Explanation
1	17,9,2004	16,9,2004	E1,E4,E7	My birthdate
2	1,7,2012	30,6,2012	E1,E4,E7	First Day of Month
3	1,1,2000	31,12,1999	E1,E4,E7	First Day of Year
4	31,10,2005	30,10,2005	E1,E4,E7	Last Day of 31-Month
5	30,4,2015	29,4,2015	E1,E4,E7	Last Day of 30-Month
6	28,2,2007	27,2,2007	E1,E4,E7	Last Day of Feb(non leap)
7	28,2,2008	27,2,2008	E1,E4,E7	Last Day of feb (leap)
8	1,3,2008	28,2,2008	E1,E4,E7	First Day of March(leap)
9	1,3,2007	27,2,2007	E1,E4,E7	First Day of March
10	0,5,2000	Invalid date	E5	Invalid day (<1)
11	15,0,2004	Invalid date	E2	Invalid Month (<1)
12	32,1,2004	Invalid date	E6	Invalid Day (>31)
13	15,13,2004	Invalid date	E3	Invalid Month (>12)
14	15,12,1899	Invalid date	E8	Invalid Year (<1899)
15	22,10,2024	Invalid date	E9	Invalid Year (>2015)

202201129

16	29,2,2011	Invalid date	-	Invalid 2011 is not leap year

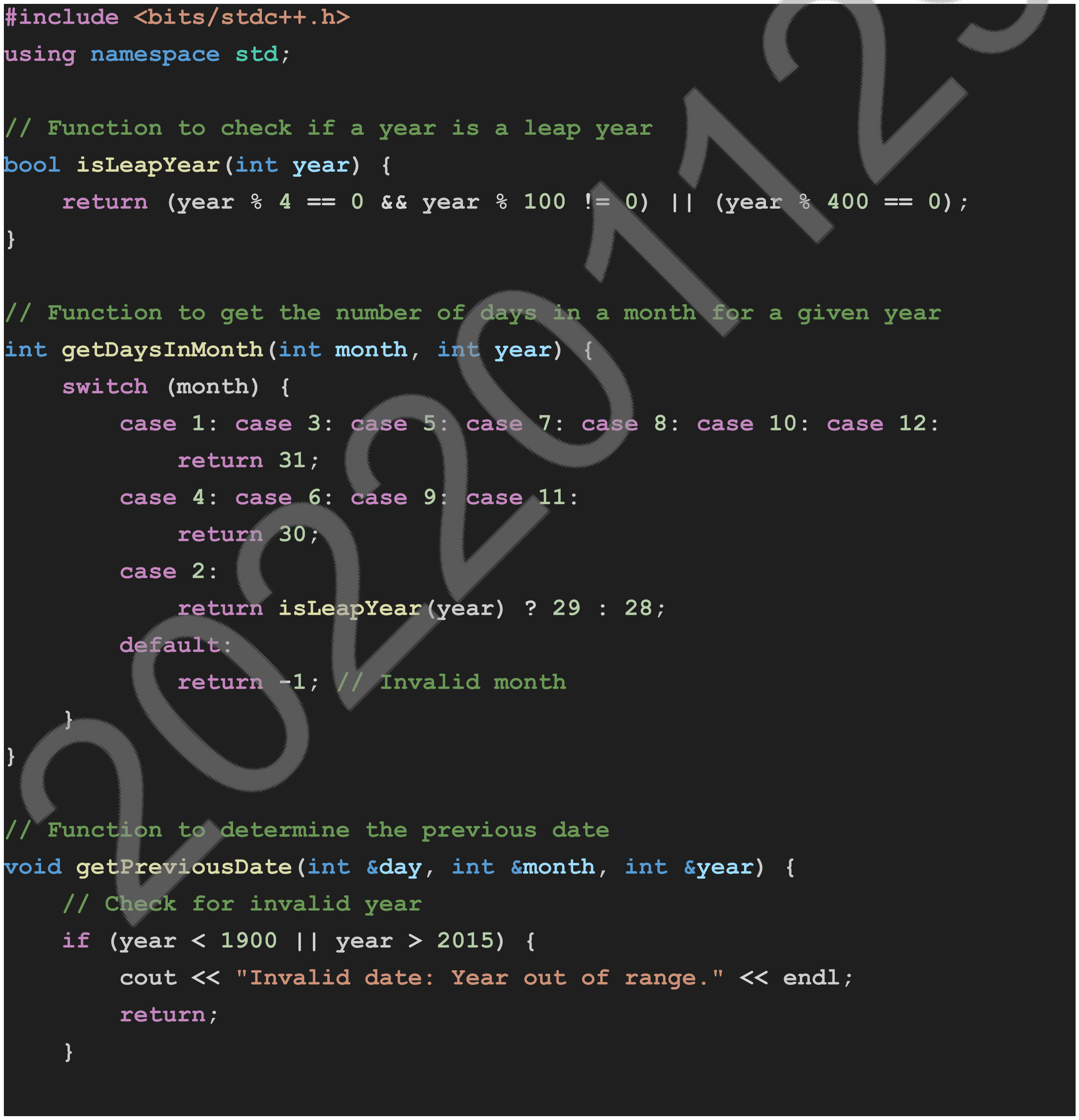
Boundary Analysis Test cases

ID	Input(Day,Month,Year)	Expected Output	Classes Covered	Explanation
1	1,1,1900	31,12,1899	E1,E4,E7	Min valid year
2	31,12,2015	30,12,2015	E1,E4,E7	Max valid year
3	1,1,1899	Invalid date	E8	Year below min
4	1,1,2016	Invalid date	E9	Year above max
5	1,1,2015	31,12,1999	E1,E4,E7	Min valid month
6	31,12,2000	30,12,2000	E1,E4,E7	Max valid month
7	15,0,2000	Invalid date	E2	Month below min
8	15,13,2000	Invalid date	E3	Month above max
9	1,9,2000	31,8,2000	E1,E4,E7	Min valid day
10	31,5,2000	30,5,2000	E1,E4,E7	Max valid day (31-day month)
11	30,9,2004	29,9,2004	E1,E4,E7	Max valid day (30-day month)
12	29,2,2004	28,2,2004	E1,E4,E7	Max valid day (Feb, leap year)
13	28,2,2005	27,2,2005	E6	Max valid day (Feb,non-leap year)
14	0,5,2012	Invalid date	E5	Day below min

20201129

15	32,9,2012	Invalid date	E6	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.



```cpp
#include <bits/stdc++.h>
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 for a given year
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 determine the previous date
void getPreviousDate(int &day, int &month, int &year) {
    // Check for invalid year
    if (year < 1900 || year > 2015) {
        cout << "Invalid date: Year out of range." << endl;
        return;
    }
```

2020129

```
    // Check for invalid month
    if (month < 1 || month > 12) {
        cout << "Invalid date: Month out of range." << endl;
        return;
    }

    // Get the number of days in the current month
    int daysInMonth = getDaysInMonth(month, year);

    // Check for invalid day
    if (day < 1 || day > daysInMonth) {
        cout << "Invalid date: Day out of range." << endl;
        return;
    }

    // Calculate the previous date
    if (day > 1) {
        day--;  // If not the first day of the month, just decrement the
day
    } else {
        // If the first day of the month, move to the last day of the
previous month
        if (month == 1) {  // Special case: January -> December of the
previous year
            month = 12;
            year--;
            day = 31;
        } else {
            month--;
            day = getDaysInMonth(month, year);
        }
    }

    // Output the result
    cout << "Previous date: " << day << "-" << month << "-" << year <<
endl;
}

// Main function to test the previous date program
int main() {
```
20220129

```
    // Test cases based on Equivalence Partitioning and Boundary Value
Analysis
    int testCases[][3] = {
        // Equivalence Partitioning Test Cases
        {17, 9, 2004},     // Valid date
        {1, 7, 2012},      // First day of the month
        {1, 1, 2000},      // First day of the year
        {31, 10, 2005},    // Last day of a 31-day month
        {30, 4, 2015},     // Last day of a 30-day month
        {28, 2, 2007},     // Last day of February (non-leap year)
        {28, 2, 2008},     // Last day of February (leap year)
        {1, 3, 2008},      // First day of March (leap year)
        {1, 3, 2007},      // First day of March (non-leap year)
        {0, 5, 2000},      // Invalid day (day < 1)
        {15, 0, 2004},     // Invalid month (month < 1)
        {32, 1, 2004},     // Invalid day (day > 31)
        {15, 13, 2004},    // Invalid month (month > 12)
        {15, 12, 1899},    // Invalid year (year < 1900)
        {22, 10, 2024},    // Invalid year (year > 2015)
        {29, 2, 2011},     // Invalid date (2011 is not a leap year)

        // Boundary Value Analysis Test Cases
        {1, 1, 1900},      // Minimum valid year
        {31, 12, 2015},    // Maximum valid year
        {1, 1, 1899},      // Year below minimum valid
        {1, 1, 2016},      // Year above maximum valid
        {1, 1, 2015},      // Minimum valid month
        {31, 12, 2000},    // Maximum valid month
        {15, 0, 2000},     // Month below minimum valid
        {15, 13, 2000},    // Month above maximum valid
        {1, 9, 2000},      // Minimum valid day
        {31, 5, 2000},     // Maximum valid day (31-day month)
        {30, 9, 2004},     // Maximum valid day (30-day month)
        {29, 2, 2004},     // Maximum valid day (February, leap year)
        {28, 2, 2005},     // Maximum valid day (February, non-leap year)
        {0, 5, 2012},      // Day below minimum valid
        {32, 9, 2012}      // Day above maximum valid
    };

    int numberOfTestCases = sizeof(testCases) / sizeof(testCases[0]);
```

20201129

```cpp
    // Execute all test cases
    for (int i = 0; i < numberOfTestCases; i++) {
        int day = testCases[i][0];
        int month = testCases[i][1];
        int year = testCases[i][2];

        cout << "Test Case " << (i + 1) << ": Input (" << day << ", " << month << ", " << year << ")" << endl;
        getPreviousDate(day, month, year);
        cout << "--------------------------------" << endl;
    }

    return 0;
}
```

20220129

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.

Equivalence Partitioning

- E1: Search for a value present in the middle of the array
- E2: Search for a value present at the beginning of the array
- E3: Search for a value present at the end of the array
- E4: Search for a value not present in the array
- E5: Search in an empty array
- E6: Search in an array with all elements equal to the search value

202001129

Test Cases	Search Value	Array	Output
E1	3	[1,2,3,4,5]	2
E2	1	[1,2,3,4,5]	0
E3	5	[1,2,3,4,5]	4
E4	6	[1,2,3,4,5]	-1
E5	1	[]	-1
E6	2	[1,1,1,1,1]	0

Boundary Value Analysis

- E1: Search for the first element in the array
- E2: Search for the last element in the array

- E3: Search for a value just before the first occurrence of it in the array
- E4: Search for a value just after the last occurrence of it in the array
- E5: Search in an array with only one element (matching the search value)
- E6: Search in an array with only one element (not matching the search value)

Test Cases	Search Value	Array	Output
E1	1	[1,2,3,4,5]	0
E2	5	[1,2,3,4,5]	4
E3	2	[1,2,3,4,5]	2
E4	4	[1,2,3,4,5]	3
E5	1	[2]	0
E6	2	[2]	-1

```cpp
#include <bits/stdc++.h>
using namespace std;

// Linear Search function
int linearSearch(int v, const vector<int>& a) {
    for (int i = 0; i < a.size(); i++) {
        if (a[i] == v) {
            return i;
        }
    }
    return -1;  // Value not found
}

// Function to run test cases
void runTestCase(int testCaseNum, int searchValue, const vector<int>&
array, int expectedOutput) {
    cout << "Test Case " << testCaseNum << ": ";
    int result = linearSearch(searchValue, array);
    cout << "Search Value = " << searchValue << ", Array = {";
    for (int i = 0; i < array.size(); i++) {
        if (i != 0) cout << ", ";
        cout << array[i];
    }
    cout << "} -> Result: " << result;

    // Check if result matches expected output
    if (result == expectedOutput) {
        cout << " [PASS]" << endl;
    } else {
        cout << " [FAIL]" << endl;
    }
}

// Main function to test all cases
int main() {
    // Equivalence Partitioning Test Cases
    runTestCase(1, 3, {1, 2, 3, 4, 5}, 2);   // E1
    runTestCase(2, 1, {1, 2, 3, 4, 5}, 0);   // E2
    runTestCase(3, 5, {1, 2, 3, 4, 5}, 4);   // E3
    runTestCase(4, 6, {1, 2, 3, 4, 5}, -1);  // E4
    runTestCase(5, 1, {}, -1);               // E5
    runTestCase(6, 1, {1, 1, 1, 1, 1}, 0);   // E6

    // Boundary Value Analysis Test Cases
    runTestCase(7, 1, {1, 2, 3, 4, 5}, 0);   // E1 (First element)
    runTestCase(8, 5, {1, 2, 3, 4, 5}, 4);   // E2 (Last element)
    runTestCase(9, 2, {1, 2, 3, 4, 5}, 1);   // E3 (Just before the first
occurrence)
    runTestCase(10, 4, {1, 2, 3, 4, 5}, 3);  // E4 (Just after the last
occurrence)
    runTestCase(11, 2, {2}, 0);              // E5 (Array with one
element, matches search value)
    runTestCase(12, 1, {2}, -1);             // E6 (Array with one
element, does not match)

    return 0;
```

202201129

```
}
```

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

Equivalence Partitioning

- E1: Count a value present multiple times in the array
- E2: Count a value present once in the array
- E3: Count a value not present in the array
- E4: Count in an empty array
- E5: Count in an array with all elements equal to the search value
- E6: Count with a negative search value

Test Cases	Search Value	Array	Output
E1	2	[1,2,2,4,5]	2
E2	1	[1,2,3,4,5]	1
E3	6	[1,2,3,4,5]	0
E4	6	[]	0
E5	1	[1,1,1]	3
E6	-2	[-2,1,-2,-2,1]	3

Boundary Value Analysis

- E1: Count the first element in the array
- E2: Count the last element in the array
- E3: Count in an array with only one element (matching the search value)

20201129

- E4: Count in an array with only one element (not matching the search value)
- E5: Count in an array with two elements (both matching)
- E6: Count in an array with two elements (one matching, one not)

Test Cases	Search Value	Array	Output
E1	1	[1,2,2,4,5]	1
E2	5	[1,2,3,4,5,5]	2
E3	8	[8]	1
E4	6	[8]	0
E5	5	[5,5]	2
E6	1	[1,2]	1

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

// Function to count the number of times value v appears in the 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 runTestCase(int testCaseNum, int searchValue, const vector<int>&
array, int expectedOutput) {
```

20220129

```
    cout << "Test Case " << testCaseNum << ": ";
    int result = countItem(searchValue, array);
    cout << "Search Value = " << searchValue << ", Array = {";
    for (int i = 0; i < array.size(); i++) {
        if (i != 0) cout << ", ";
        cout << array[i];
    }
    cout << "} -> Result: " << result;

    // Check if result matches expected output
    if (result == expectedOutput) {
        cout << " [PASS]" << endl;
    } else {
        cout << " [FAIL]" << endl;
    }
}

// Main function to test all cases
int main() {
    // Equivalence Partitioning Test Cases
    runTestCase(1, 2, {1, 2, 2, 4, 5}, 2);     // E1: Value present
multiple times
    runTestCase(2, 1, {1, 2, 3, 4, 5}, 1);     // E2: Value present once
    runTestCase(3, 6, {1, 2, 3, 4, 5}, 0);     // E3: Value not present
    runTestCase(4, 6, {}, 0);                  // E4: Empty array
    runTestCase(5, 1, {1, 1, 1}, 3);           // E5: Array with all
elements equal to the search value
    runTestCase(6, -2, {-2, 1, -2, -2, 1}, 3);// E6: Count with a negative
search value

    // Boundary Value Analysis Test Cases
    runTestCase(7, 1, {1, 2, 2, 4, 5}, 1);     // E1: Count first element
    runTestCase(8, 5, {1, 2, 3, 4, 5, 5}, 2); // E2: Count last element
    runTestCase(9, 8, {8}, 1);                 // E3: Count in array with
one matching element
    runTestCase(10, 6, {8}, 0);                // E4: Count in array with
one non-matching element
    runTestCase(11, 5, {5, 5}, 2);             // E5: Count in array with
two matching elements
```

20220129

```
    runTestCase(12, 1, {1, 2}, 1);          // E6: Count in array with
one matching and one non-matching element

    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 are sorted in non-decreasing order.

Equivalence Partitioning

- E1: Search for a value in the middle of the array
- E2: Search for the first value in the array
- E3: Search for the last value in the array
- E4: Search for a value not in the array (less than all elements)
- E5: Search for a value not in the array (greater than all elements)
- E6: Search for a value not in the array (between existing elements)
- E7: Search in an empty array

Test Cases	Search Value	Array	Output
E1	3	[1,2,3,4,5]	2
E2	1	[1,2,3,4,5]	0
E3	5	[1,2,3,4,5]	4
E4	0	[1,2,3,4,5]	-1
E5	6	[1,2,3,4,5]	-1

20201129

E6	-2	[1,2,3,4,5]	-1
E7	1	[]	-1

Boundary Value Analysis

- E1: Search in an array with only one element (matching)
- E2: Search in an array with only one element (not matching)
- E3: Search for a value just less than the middle element
- E4: Search for a value just greater than the middle element
- E5: Search in an array with two elements (first matching)
- E6: Search in an array with two elements (second matching)

Test Cases	Search Value	Array	Output
E1	3	[3]	1
E2	1	[3]	-1
E3	2	[1,2,3,4,5]	-1
E4	4	[1,2,3,4,5]	-1
E5	1	[1,2]	0
E6	2	[1,2]	1

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

// Function to perform binary search on an ordered array a for the value v
int binarySearch(int v, const vector<int>& a) {
    int lo = 0;
    int hi = a.size() - 1;

    while (lo <= hi) {
```

20201129

```
        int mid = (lo + hi) / 2;

        if (v == a[mid]) {
            return mid; // Value found
        } else if (v < a[mid]) {
            hi = mid - 1; // Search left half
        } else {
            lo = mid + 1; // Search right half
        }
    }
    return -1; // Value not found
}

// Function to run test cases
void runTestCase(int testCaseNum, int searchValue, const vector<int>&
array, int expectedOutput) {
    cout << "Test Case " << testCaseNum << ": ";
    int result = binarySearch(searchValue, array);
    cout << "Search Value = " << searchValue << ", Array = {";
    for (int i = 0; i < array.size(); i++) {
        if (i != 0) cout << ", ";
        cout << array[i];
    }
    cout << "} -> Result: " << result;

    // Check if result matches expected output
    if (result == expectedOutput) {
        cout << " [PASS]" << endl;
    } else {
        cout << " [FAIL]" << endl;
    }
}

// Main function to test all cases
int main() {
    // Equivalence Partitioning Test Cases
    runTestCase(1, 3, {1, 2, 3, 4, 5}, 2);   // E1: Search for a value in
the middle
    runTestCase(2, 1, {1, 2, 3, 4, 5}, 0);   // E2: Search for the first
value
```

202201129

```
    runTestCase(3, 5, {1, 2, 3, 4, 5}, 4);   // E3: Search for the last
value
    runTestCase(4, 0, {1, 2, 3, 4, 5}, -1);   // E4: Search for a value
not in the array (less than all)
    runTestCase(5, 6, {1, 2, 3, 4, 5}, -1);   // E5: Search for a value
not in the array (greater than all)
    runTestCase(6, -2, {1, 2, 3, 4, 5}, -1);  // E6: Search for a value
not in the array (between existing)
    runTestCase(7, 1, {}, -1);                 // E7: Search in an empty
array

    // Boundary Value Analysis Test Cases
    runTestCase(8, 3, {3}, 0);                 // E1: One-element array
(matching)
    runTestCase(9, 1, {3}, -1);                // E2: One-element array
(not matching)
    runTestCase(10, 2, {1, 2, 3, 4, 5}, 1);   // E3: Just less than middle
element
    runTestCase(11, 4, {1, 2, 3, 4, 5}, 3);   // E4: Just greater than
middle element
    runTestCase(12, 1, {1, 2}, 0);             // E5: Two-element array
(first matching)
    runTestCase(13, 2, {1, 2}, 1);             // E6: Two-element array
(second matching)

    return 0;
}
```

20201129

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).

Equivalence Partitioning

- E1: Equilateral triangle
- E2: Isosceles triangle (a == b)
- E3: Isosceles triangle (b == c)
- E4: Isosceles triangle (a == c)
- E5: Scalene triangle
- E6: Invalid triangle (a >= b + c)
- E7: Invalid triangle (b >= a + c)
- E8: Invalid triangle (c >= a + b)
- E9: Invalid triangle (negative sides)
- E10: Invalid triangle (zero sides)

Test Cases	A	B	C	Output
E1	3	3	3	0
E2	2	2	3	1
E3	3	2	2	1
E4	4	2	4	1

20201129

E5	2	3	4	2
E6	3	1	2	3
E7	1	3	2	3
E8	2	1	3	3
E9	-3	-1	-2	3
E10	0	1	0	3

Boundary Value Analysis

- E1: Minimum valid equilateral triangle (1, 1, 1)
- E2: Minimum valid isosceles triangle (1, 2, 2)
- E3: Minimum valid scalene triangle (2, 3, 4)
- E4: Just valid triangle (a + b = c + 1)
- E5: Just invalid triangle (a + b = c)

Test Cases	A	B	C	Output
E1	1	1	1	0
E2	1	2	2	1
E3	2	3	4	2
E4	3	4	6	2
E5	1	2	3	3

```
#include <bits/stdc++.h>
using namespace std;

const int EQUILATERAL = 0;
const int ISOSCELES = 1;
```

20201129

```cpp
const int SCALENE = 2;
const int INVALID = 3;

int triangle(int a, int b, int c) {
    // Check for invalid triangles
    if (a <= 0 || b <= 0 || c <= 0)  // Negative or zero sides are invalid
        return INVALID;
    if (a >= b + c || b >= a + c || c >= a + b)  // Sum of two sides must
be greater than the third
        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 is a scalene triangle
    return SCALENE;
}

void runEquivalencePartitioningTests() {
    cout << "Equivalence Partitioning Test Cases:" << endl;

    // Test case E1
    cout << "Test E1: " << triangle(3, 3, 3) << " (Expected: 0)" << endl;

    // Test case E2
    cout << "Test E2: " << triangle(2, 2, 3) << " (Expected: 1)" << endl;

    // Test case E3
    cout << "Test E3: " << triangle(3, 2, 2) << " (Expected: 1)" << endl;

    // Test case E4
    cout << "Test E4: " << triangle(4, 2, 4) << " (Expected: 1)" << endl;

    // Test case E5
    cout << "Test E5: " << triangle(2, 3, 4) << " (Expected: 2)" << endl;
```

20201129

```
    // Test case E6
    cout << "Test E6: " << triangle(3, 1, 2) << " (Expected: 3)" << endl;

    // Test case E7
    cout << "Test E7: " << triangle(1, 3, 2) << " (Expected: 3)" << endl;

    // Test case E8
    cout << "Test E8: " << triangle(2, 1, 3) << " (Expected: 3)" << endl;

    // Test case E9
    cout << "Test E9: " << triangle(-3, -1, -2) << " (Expected: 3)" <<
endl;

    // Test case E10
    cout << "Test E10: " << triangle(0, 1, 0) << " (Expected: 3)" << endl;
}

void runBoundaryValueAnalysisTests() {
    cout << "Boundary Value Analysis Test Cases:" << endl;

    // Test case E1
    cout << "Test E1: " << triangle(1, 1, 1) << " (Expected: 0)" << endl;

    // Test case E2
    cout << "Test E2: " << triangle(1, 2, 2) << " (Expected: 1)" << endl;

    // Test case E3
    cout << "Test E3: " << triangle(2, 3, 4) << " (Expected: 2)" << endl;

    // Test case E4
    cout << "Test E4: " << triangle(3, 4, 6) << " (Expected: 2)" << endl;

    // Test case E5
    cout << "Test E5: " << triangle(1, 2, 3) << " (Expected: 3)" << endl;
}

int main() {
    runEquivalencePartitioningTests();
    runBoundaryValueAnalysisTests();
```

202201129

```
    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).

Equivalence Partitioning

- E1: s1 is a prefix of s2
- E2: s1 is not a prefix of s2
- E3: s1 is longer than s2
- E4: s1 and s2 are identical

Test Cases	S1	S2	Output
E1	“Hell”	“Hello”	TRUE
E2	“helo”	“Hello”	FALSE
E3	“hellish”	“Hello”	FALSE
E4	“Hello”	“Hello”	TRUE

20201129

Boundary Value Analysis

- E1: s1 is one character, s2 is one character (matching)
- E2: s1 is one character, s2 is one character (not matching)
- E3: s1 is one character shorter than s2 (matching prefix)
- E4: s1 is one character shorter than s2 (not matching prefix)

Test Cases	S1	S2	Output
E1	“H”	“H”	TRUE
E2	“T”	“H”	FALSE
E3	“Hell”	“Hello”	TRUE
E4	“Fell”	“Hello”	FALSE

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

bool prefix(string s1, string s2) {
    if (s1.length() > s2.length())
        return false;

    for (int i = 0; i < s1.length(); i++) {
        if (s1[i] != s2[i])
            return false;
    }
    return true;
}

void runEquivalencePartitioningTests() {
    cout << "Equivalence Partitioning Test Cases:" << endl;

    // Test case E1
    cout << "Test E1: " << (prefix("Hell", "Hello") ? "TRUE" : "FALSE") <<
" (Expected: TRUE)" << endl;

    // Test case E2
    cout << "Test E2: " << (prefix("helo", "Hello") ? "TRUE" : "FALSE") <<
" (Expected: FALSE)" << endl;
```

20201129

```
    // Test case E3
    cout << "Test E3: " << (prefix("hellish", "Hello") ? "TRUE" : "FALSE")
<< " (Expected: FALSE)" << endl;

    // Test case E4
    cout << "Test E4: " << (prefix("Hello", "Hello") ? "TRUE" : "FALSE")
<< " (Expected: TRUE)" << endl;
}

void runBoundaryValueAnalysisTests() {
    cout << "Boundary Value Analysis Test Cases:" << endl;

    // Test case E1
    cout << "Test E1: " << (prefix("H", "H") ? "TRUE" : "FALSE") << "
(Expected: TRUE)" << endl;

    // Test case E2
    cout << "Test E2: " << (prefix("T", "H") ? "TRUE" : "FALSE") << "
(Expected: FALSE)" << endl;

    // Test case E3
    cout << "Test E3: " << (prefix("Hell", "Hello") ? "TRUE" : "FALSE") <<
" (Expected: TRUE)" << endl;

    // Test case E4
    cout << "Test E4: " << (prefix("Fell", "Hello") ? "TRUE" : "FALSE") <<
" (Expected: FALSE)" << endl;
}

int main() {
    runEquivalencePartitioningTests();
    runBoundaryValueAnalysisTests();

    return 0;
}
```

20201129

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.

20220129

a) Equivalence classes

- E1: Scalene triangle (all sides different)
- E2: Isosceles triangle (two sides equal)
- E3: Equilateral triangle (all sides equal)
- E4: Right-angled triangle
- E5: Non-triangle (sum of any two sides ≤ third side)
- E6: Non-positive input

b) Test cases for each equivalence class

- E1: Scalene: (2, 3, 4)
- E2: Isosceles: (5, 5, 7)
- E3: Equilateral: (2, 2, 2)
- E4: Right-angled: (3, 4, 5)
- E5: Non-triangle: (1, 2, 3)
- E6: Non-positive input: (0, -1, 2)

Test Cases	A	B	C	Output
E1	2	3	4	Scalene Triangle
E2	5	5	7	Isosceles Triangle
E3	2	2	2	Equilateral Triangle
E4	3	4	5	Right Angled Triangle
E5	1	2	3	Non - Triangle
E6	0	-1	2	Non - Triangle

202201129

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

- E1: (3, 4, 6.99) - Just below the boundary
- E2: (3, 4, 7) - On the boundary
- E3: (3, 4, 7.01) - Just above the boundary

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

- E4: (5, 4.99, 5) - Just below equality
- E5: (5, 5, 5) - On the boundary (also equilateral)
- E6: (5, 5.01, 5) - Just above equality

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

- E7: (5, 5, 4.99) - Just below equality
- E8: (5, 5, 5) - On the boundary
- E9: (5, 5, 5.01) - Just above equality

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

- E10: (3, 4, 4.99) - Just below right angle
- E11: (3, 4, 5) - On the boundary (exact right angle)
- E12: (3, 4, 5.01) - Just above right angle

g) Non-triangle case:

- E13: (1, 2, 2.99) - Just below triangle formation
- E14: (1, 2, 3) - On the boundary
- E15: (1, 2, 3.01) - Just above boundary (valid triangle)

h) Non-positive input:

- E16: (0, 0, 0) - Zero input
- E17: (-1, 2, 5) - Slightly negative input

20201129

- E18: (-1, -9, -1) - Negative input

Condition	Test Cases	Input (A,B,C)	Output
A + B > C	E1	3, 4, 6.99	Scalene Triangle
	E2	3, 4, 7	Non - Triangle
	E3	3, 4, 7.1	Non - Triangle
A = C	E4	5, 4.99, 5	Isosceles Triangle
	E5	5, 5, 5	Equilateral Triangle
	E6	5, 5, 5.1	Isosceles Triangle
A = B = C	E7	5, 5, 4.99	Scalene Triangle
	E8	5, 5, 5	Equilateral Triangle
	E9	5, 5, 5.01	Scalene Triangle
A^2 + B^2 = C^2	E10	3, 4, 4.99	Scalene Triangle
	E11	3, 4, 5	Right - Angled Triangle
	E12	3, 4, 5.01	Scalene Triangle
Non - Triangle	E13	1, 2, 2.99	Non - Triangle
	E14	1, 2, 3	Non - Triangle
	E15	1, 2, 3.01	Scalene Triangle
Non - Positive Input	E16	0, 0, 0	Non - Triangle
	E17	-1, 2, 5	Non - Triangle
	E18	-1, -9, -1	Non - Triangle

20220129

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

// Function to classify the triangle based on sides A, B, and C
string classifyTriangle(double A, double B, double C) {
    // Check for non-positive inputs
    if (A <= 0 || B <= 0 || C <= 0) {
        return "Non - Triangle";
    }

    // Check for non-triangle cases (sum of two sides less than or equal
to the third)
    if (A + B <= C || A + C <= B || B + C <= A) {
        return "Non - Triangle";
    }

    // Check for equilateral triangle
    if (A == B && B == C) {
        return "Equilateral Triangle";
    }

    // Check for right-angled triangle (Pythagoras theorem)
    if (abs(A * A + B * B - C * C) < 1e-2 || abs(A * A + C * C - B * B) <
1e-2 || abs(B * B + C * C - A * A) < 1e-2) {
        return "Right - Angled Triangle";
    }

    // Check for isosceles triangle (two sides equal)
    if (A == B || B == C || A == C) {
        return "Isosceles Triangle";
    }

    // Otherwise, it's a scalene triangle
    return "Scalene Triangle";
}

// Function to run test cases and print results
void runTestCases() {
```

20201129

```
    cout << "Test Cases and Outputs:\n" << endl;

    // Equivalence Partitioning Tests
    cout << "E1 (Scalene): " << classifyTriangle(2, 3, 4) << " (Expected:
Scalene Triangle)" << endl;
    cout << "E2 (Isosceles): " << classifyTriangle(5, 5, 7) << "
(Expected: Isosceles Triangle)" << endl;
    cout << "E3 (Equilateral): " << classifyTriangle(2, 2, 2) << "
(Expected: Equilateral Triangle)" << endl;
    cout << "E4 (Right-angled): " << classifyTriangle(3, 4, 5) << "
(Expected: Right - Angled Triangle)" << endl;
    cout << "E5 (Non-Triangle): " << classifyTriangle(1, 2, 3) << "
(Expected: Non - Triangle)" << endl;
    cout << "E6 (Non-positive): " << classifyTriangle(0, -1, 2) << "
(Expected: Non - Triangle)" << endl;

    cout << "\nBoundary Condition Tests:\n" << endl;

    // Boundary condition A + B > C (scalene triangle)
    cout << "E1 (Just below boundary): " << classifyTriangle(3, 4, 6.99)
<< " (Expected: Scalene Triangle)" << endl;
    cout << "E2 (On boundary): " << classifyTriangle(3, 4, 7) << "
(Expected: Non - Triangle)" << endl;
    cout << "E3 (Just above boundary): " << classifyTriangle(3, 4, 7.01)
<< " (Expected: Scalene Triangle)" << endl;

    // Boundary condition A = C (isosceles triangle)
    cout << "E4 (Just below equality): " << classifyTriangle(5, 4.99, 5)
<< " (Expected: Isosceles Triangle)" << endl;
    cout << "E5 (On boundary - equilateral): " << classifyTriangle(5, 5,
5) << " (Expected: Equilateral Triangle)" << endl;
    cout << "E6 (Just above equality): " << classifyTriangle(5, 5.01, 5)
<< " (Expected: Isosceles Triangle)" << endl;

    // Boundary condition A = B = C (equilateral triangle)
    cout << "E7 (Just below equality): " << classifyTriangle(5, 5, 4.99)
<< " (Expected: Scalene Triangle)" << endl;
    cout << "E8 (On boundary): " << classifyTriangle(5, 5, 5) << "
(Expected: Equilateral Triangle)" << endl;
```

20220129

```
    cout << "E9 (Just above equality): " << classifyTriangle(5, 5, 5.01) << " (Expected: Scalene Triangle)" << endl;

    // Boundary condition A^2 + B^2 = C^2 (right-angle triangle)
    cout << "E10 (Just below right angle): " << classifyTriangle(3, 4, 4.99) << " (Expected: Scalene Triangle)" << endl;
    cout << "E11 (On boundary - right angle): " << classifyTriangle(3, 4, 5) << " (Expected: Right - Angled Triangle)" << endl;
    cout << "E12 (Just above right angle): " << classifyTriangle(3, 4, 5.01) << " (Expected: Scalene Triangle)" << endl;

    // Non-triangle case
    cout << "E13 (Just below triangle formation): " << classifyTriangle(1, 2, 2.99) << " (Expected: Non - Triangle)" << endl;
    cout << "E14 (On boundary): " << classifyTriangle(1, 2, 3) << " (Expected: Non - Triangle)" << endl;
    cout << "E15 (Just above boundary - valid triangle): " << classifyTriangle(1, 2, 3.01) << " (Expected: Scalene Triangle)" << endl;

    // Non-positive input
    cout << "E16 (Zero input): " << classifyTriangle(0, 0, 0) << " (Expected: Non - Triangle)" << endl;
    cout << "E17 (Slightly negative input): " << classifyTriangle(-1, 2, 5) << " (Expected: Non - Triangle)" << endl;
    cout << "E18 (Negative input): " << classifyTriangle(-1, -9, -1) << " (Expected: Non - Triangle)" << endl;
}

int main() {
    // Run all the test cases
    runTestCases();
    return 0;
}
```

20201129