IT314: Software Engineering
Lab 8: Functional Testing (Black-Box)

Name: Harsh Rajwani
Id: 202201027

Q.1. Consider a program for determining the previous date. Its input is triple of day, month and year with the following ranges 1 <= month <= 12, 1 <= day <=31, 1900 <= year <= 2015.The possible output dates would be previous date or invalid date. Design the equivalence class test cases?

Write a set of test cases (i.e., test suite) – specific set of data – to properly test the programs. Your test suite should include both correct and incorrect inputs.
1. Enlist which set of test cases have been identified using Equivalence Partitioning and Boundary Value Analysis separately.
2. Modify your programs such that it runs, and then execute your test suites on the program. While executing your input data in a program, check whether the identified expected outcome (mentioned by you) is correct or not.

EQUIVALENCE PARTITIONING TEST CASES:

Test Case	Input (Day, Month, Year)	Expected Output	Description
1	(1, 1, 2000)	(31, 12, 1999)	Valid: Start of year
2	(15, 3, 2020)	(14, 3, 2020)	Valid: Regular day
3	(1, 5, 2015)	(30, 4, 2015)	Valid: Start of month
4	(29, 2, 2012)	(28, 2, 2012)	Valid: Leap year
5	(1, 2, 1900)	(31, 1, 1900)	Valid: Start of February
6	(0, 1, 2000)	Invalid Date	Invalid: Day out of range
7	(32, 1, 2000)	Invalid Date	Invalid: Day out of range
8	(15, 13, 2000)	Invalid Date	Invalid: Month out of range
9	(29, 2, 2013)	Invalid Date	Invalid: Non-leap year
10	(1, 1, 1899)	Invalid Date	Invalid: Year out of range

BOUNDARY VALUE ANALYSIS TEST CASES:

Test Case	Input (Day, Month, Year)	Expected Output	Description
11	(1, 1, 1900)	(31, 12, 1899)	BVA: First valid year
12	(31, 12, 2015)	(30, 12, 2015)	BVA: End of valid date range
13	(1, 12, 2015)	(30, 11, 2015)	BVA: Start of last month
14	(29, 2, 2016)	(28, 2, 2016)	BVA: Leap year
15	(1, 1, 2016)	Invalid Date	BVA: Year out of range

PROGRAM/CODE:

```
#include <bits/stdc++.h>

using namespace std;

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

// Function to get the previous date
string getPreviousDate(int day, int month, int year) {
   // Check if year, month, and day are within valid ranges
   if (year < 1900 || year > 2015) return "Invalid date";
   if (month < 1 || month > 12) return "Invalid date";

   // Days in each month
   int daysInMonth[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

   // Adjust for leap year in February
   if (isLeapYear(year)) daysInMonth[1] = 29;

   // Check for invalid day
   if (day < 1 || day > daysInMonth[month - 1]) return "Invalid date";

   // Find previous date
   if (day > 1) {
```

```
        day--;
    } else {
        if (month == 1) {  // If it's January, go back to previous year
            day = 31;
            month = 12;
            year--;
        } else {  // Otherwise, go back to the previous month
            month--;
            day = daysInMonth[month - 1];
        }
    }

    // Format the previous date as a string
    return to_string(day) + "-" + to_string(month) + "-" + to_string(year);
}

int main() {
    // Test cases
    int day, month, year;

    // Test Case 1
    day = 1; month = 1; year = 1900;
    cout << "Previous date for " << day << "-" << month << "-" << year << ": " <<
getPreviousDate(day, month, year) << endl;

    // Test Case 2
    day = 15; month = 6; year = 2000;
    cout << "Previous date for " << day << "-" << month << "-" << year << ": " <<
getPreviousDate(day, month, year) << endl;

    // Test Case 3
    day = 31; month = 12; year = 2015;
    cout << "Previous date for " << day << "-" << month << "-" << year << ": " <<
getPreviousDate(day, month, year) << endl;

    // Test Case 4
    day = 1; month = 3; year = 2020;
    cout << "Previous date for " << day << "-" << month << "-" << year << ": " <<
getPreviousDate(day, month, year) << endl;

    // Test Case 5 (Invalid)
    day = 30; month = 2; year = 2012;
    cout << "Previous date for " << day << "-" << month << "-" << year << ": " <<
getPreviousDate(day, month, year) << 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);
}
```

EQUIVALENCE PARTITIONING TEST CASES:

Test Case	Input (v, a[], length)	Expected Output	Description
1	(3, {1, 2, 3, 4, 5}, 5)	2	Value is present (middle)
2	(1, {1, 2, 3, 4, 5}, 5)	0	Value is present (first element)
3	(5, {1, 2, 3, 4, 5}, 5)	4	Value is present (last element)
4	(6, {1, 2, 3, 4, 5}, 5)	-1	Value not present
5	(2, {1, 2, 2, 3, 4}, 5)	1	Value appears multiple times (first index)
6	(3, {}, 0)	-1	Search in an empty array

BOUNDARY VALUE ANALYSIS TEST CASES:

Test Case	Input (v, a[], length)	Expected Output	Description
7	(1, {1}, 1)	0	Single element array, value present
8	(2, {1}, 1)	-1	Single element array, value not present
9	(1, {1, 2}, 2)	0	Two elements, value present at start
10	(2, {1, 2}, 2)	1	Two elements, value present at end
11	(3, {1, 2}, 2)	-1	Two elements, value not present

PROGRAM/CODE:

```
#include <iostream>

int linearSearch(int v, const int a[], int size) {
    for (int i = 0; i < size; i++) {
        if (a[i] == v) {
            return i; // Return the index if the value is found
        }
    }
    return -1; // Return -1 if the value is not found
}


int main() {
    // Example usage
    int array[] = {5, 3, 8, 1, 2, 3, 7};
    int valueToFind = 3;
```

```
    int size = sizeof(array) / sizeof(array[0]); // Calculate the size of the array

    int index = linearSearch(valueToFind, array, size);
    if (index != -1) {
        std::cout << "Value " << valueToFind << " found at index: " << index << std::endl;
    } else {
        std::cout << "Value " << valueToFind << " not found in the array." << std::endl;
    }

    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);

}
```

EQUIVALENCE PARTITIONING TEST CASES:

Test Case	Input (v, a[], length)	Expected Output	Description
1	(2, {1, 2, 2, 3, 4}, 5)	2	Value appears multiple times
2	(1, {1, 2, 3, 4, 5}, 5)	1	Value appears once
3	(6, {1, 2, 3, 4, 5}, 5)	0	Value does not appear
4	(2, {2, 2, 2, 2, 2}, 5)	5	Value appears in all elements
5	(3, {}, 0)	0	Count in an empty array

BOUNDARY VALUE ANALYSIS TEST CASES:

Test Case	Input (v, a[], length)	Expected Output	Description
6	(1, {1}, 1)	1	Single element array, value present
7	(2, {1}, 1)	0	Single element array, value not present
8	(1, {1, 1}, 2)	2	Two elements, value present in both
9	(2, {1, 1}, 2)	0	Two elements, value not present
10	(1, {1, 2, 3}, 3)	1	Three elements, value present once

MODIFIED PROGRAM/CODE:

```
#include <iostream>

int countItem(int v, const int a[], int size) {
    int count = 0;
    for (int i = 0; i < size; i++) {
        if (a[i] == v) {
            count++;
        }
    }
    return count;
}


int main() {
    // Example usage
```

```
    int array[] = {1, 2, 3, 4, 5, 3, 3, 6, 7, 8};

    int valueToCount = 3;

    int size = sizeof(array) / sizeof(array[0]); // Calculate the size of the array



    int count = countItem(valueToCount, array, size);

    std::cout << "Value " << valueToCount << " appears " << count << " times in the
array." << std::endl;



    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);
}
```

EQUIVALENCE PARTITIONING TEST CASES:

Test Case	Input (v, a[], length)	Expected Output	Description
1	(3, {1, 2, 3, 4, 5}, 5)	2	Value is the middle element
2	(1, {1, 2, 3, 4, 5}, 5)	0	Value is the first element

3	(5, {1, 2, 3, 4, 5}, 5)	4	Value is the last element
4	(2, {1, 2, 2, 3, 4}, 5)	1	Value appears multiple times
5	(0, {1, 2, 3, 4, 5}, 5)	-1	Value is less than the smallest element
6	(6, {1, 2, 3, 4, 5}, 5)	-1	Value is greater than the largest element
7	(3, {1, 2, 4, 5}, 4)	-1	Value not present but lies within the range
8	(3, {}, 0)	-1	Search in an empty array

BOUNDARY VALUE ANALYSIS TEST CASES:

Test Case	Input (v, a[], length)	Expected Output	Description
1	(1, {1}, 1)	0	Single element array, value present
2	(2, {1}, 1)	-1	Single element array, value not present
3	(1, {1, 2}, 2)	0	Two elements, value present at start
4	(2, {1, 2}, 2)	1	Two elements, value present at end
5	(3, {1, 2, 3}, 3)	2	Three elements, value present at end

MODIFIED PROGRAM/CODE:

```
#include <iostream>

#include <vector>
```

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

    while (lo <= hi) {
        int mid = lo + (hi - lo) / 2; // To avoid potential overflow
        if (v == a[mid]) {
            return mid; // Value found
        } else if (v < a[mid]) {
            hi = mid - 1; // Search in the left half
        } else {
            lo = mid + 1; // Search in the right half
        }
    }
    return -1; // Value not found
}

int main() {
    // Example usage
    std::vector<int> sortedArray = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int valueToFind = 5;

    int index = binarySearch(valueToFind, sortedArray);
    if (index != -1) {
        std::cout << "Value " << valueToFind << " found at index: " << index << std::endl;
    } else {
```

```
        std::cout << "Value " << valueToFind << " not found in the array." << std::endl;
    }

    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);

}
```

EQUIVALENCE PARTITIONING TEST CASES:

Test Case	Input (a, b, c)	Expected Output	Description
TC1	(3, 3, 3)	EQUILATERAL (0)	All sides equal (equilateral)
TC2	(3, 3, 4)	ISOSCELES (1)	Two sides equal
TC3	(4, 3, 3)	ISOSCELES (1)	Two sides equal
TC4	(3, 4, 3)	ISOSCELES (1)	Two sides equal
TC5	(3, 4, 5)	SCALENE (2)	All sides different

TC6	(1, 2, 3)	INVALID (3)	Invalid triangle (1 + 2 ≤ 3)
TC7	(5, 2, 2)	INVALID (3)	Invalid triangle (5 ≥ 2 + 2)
TC8	(0, 1, 1)	INVALID (3)	Invalid triangle (0 is not valid)

BOUNDARY VALUE ANALYSIS TEST CASES:

Test Case	Input (a, b, c)	Expected Output	Description
TC1	(2, 2, 3)	ISOSCELES (1)	Valid boundary (two equal sides)
TC2	(2, 3, 5)	INVALID (3)	Invalid (2 + 3 ≤ 5)
TC3	(3, 4, 7)	INVALID (3)	Invalid (3 + 4 ≤ 7)
TC4	(3, 3, 6)	INVALID (3)	Invalid (3 + 3 ≤ 6)
TC5	(1, 1, 1)	EQUILATERAL (0)	Minimum valid triangle
TC6	(1, 1, 2)	INVALID (3)	Invalid (1 + 1 ≤ 2)

MODIFIED PROGRAM/CODE:

```
#include <iostream>

#include <cmath> // For fabs function


enum TriangleType {

    EQUILATERAL = 0,

    ISOSCELES = 1,

    SCALENE = 2,

    INVALID = 3

};
```

```
TriangleType triangle(double a, double b, double c) {
    // Check for non-positive lengths
    if (a <= 0 || b <= 0 || c <= 0) {
        return INVALID;
    }
    // Check for triangle inequality
    if (a >= b + c || b >= a + c || c >= a + b) {
        return INVALID;
    }
    // Check for equilateral
    if (fabs(a - b) < 1e-9 && fabs(b - c) < 1e-9) {
        return EQUILATERAL;
    }
    // Check for isosceles
    if (fabs(a - b) < 1e-9 || fabs(a - c) < 1e-9 || fabs(b - c) < 1e-9) {
        return ISOSCELES;
    }
    // If none of the above, it's scalene
    return SCALENE;
}

void testTriangle() {
    // Test cases based on Equivalence Partitioning and Boundary Value Analysis
    std::cout << "Test Case 1: " << (triangle(3, 3, 3) == EQUILATERAL ? "Passed" :
"Failed") << std::endl; // Equilateral
    std::cout << "Test Case 2: " << (triangle(3, 3, 4) == ISOSCELES ? "Passed" :
"Failed") << std::endl; // Isosceles
    std::cout << "Test Case 3: " << (triangle(4, 3, 3) == ISOSCELES ? "Passed" :
```

```
"Failed") << std::endl; // Isosceles

    std::cout << "Test Case 4: " << (triangle(3, 4, 5) == SCALENE ? "Passed" :
"Failed") << std::endl;   // Scalene

    std::cout << "Test Case 5: " << (triangle(1, 2, 3) == INVALID ? "Passed" : "Failed")
<< std::endl;   // Invalid

    std::cout << "Test Case 6: " << (triangle(0, 1, 1) == INVALID ? "Passed" : "Failed")
<< std::endl;     // Non-Triangle

    std::cout << "Test Case 7: " << (triangle(-1, 1, 1) == INVALID ? "Passed" : "Failed")
<< std::endl;    // Non-Triangle

    std::cout << "Test Case 8: " << (triangle(5, 12, 13) == SCALENE ? "Passed" :
"Failed") << std::endl;  // Right Triangle

}

int main() {

    testTriangle();

    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;
}
```

EQUIVALENCE PARTITIONING TEST CASES:

Test Case	Input (s1, s2)	Expected Output	Description
TC1	("abc", "abcde")	true	s1 is a prefix of s2
TC2	("abc", "ab")	false	s1 is longer than s2
TC3	("abc", "abcdef")	true	s1 is a prefix of s2
TC4	("abc", "xyzabc")	false	s1 is not a prefix of s2

TC5	("", "abc")	true	Empty string is a prefix of any string
TC6	("abc", "")	false	Non-empty string is not a prefix of empty string
TC7	("abcdef", "abc")	false	s1 is longer than s2
TC8	("a", "a")	true	Single character match
TC9	("abc", "abcabc")	true	s1 is a prefix of a longer identical string

BOUNDARY VALUE ANALYSIS TEST CASES:

Test Case	Input (s1, s2)	Expected Output	Description
TC1	("", "")	true	Both strings are empty
TC2	("a", "a")	true	Both strings are identical
TC3	("a", "b")	false	Single character mismatch
TC4	("abc", "ab")	false	s1 is longer than s2
TC5	("abc", "abc")	true	s1 and s2 are identical
TC6	("abc", "abcd")	true	s1 is a prefix of s2
TC7	("abcd", "abc")	false	s1 is longer than s2
TC8	("a", "")	false	Non-empty string is not a prefix of empty string

MODIFIED PROGRAM/CODE:

```
#include <iostream>

#include <string>
```

```
class StringUtils {
public:
    static bool prefix(const std::string& s1, const std::string& s2) {
        // Check if s1 is longer than s2
        if (s1.length() > s2.length()) {
            return false;
        }
        // Compare characters
        for (size_t i = 0; i < s1.length(); i++) {
            if (s1[i] != s2[i]) {
                return false;
            }
        }
        return true; // s1 is a prefix of s2
    }
};

int main() {
    // Test cases
    std::cout << StringUtils::prefix("abc", "abcde") << " (Expected: 1)\n"; // true
    std::cout << StringUtils::prefix("abc", "ab") << " (Expected: 0)\n";    // false
    std::cout << StringUtils::prefix("abc", "abcdef") << " (Expected: 1)\n"; // true
    std::cout << StringUtils::prefix("abc", "xyzabc") << " (Expected: 0)\n"; // false
    std::cout << StringUtils::prefix("", "abc") << " (Expected: 1)\n";      // true
    std::cout << StringUtils::prefix("abc", "") << " (Expected: 0)\n";      // false
    std::cout << StringUtils::prefix("abcdef", "abc") << " (Expected: 0)\n"; // false
    std::cout << StringUtils::prefix("a", "a") << " (Expected: 1)\n";        // true
```

```
    std::cout << StringUtils::prefix("abc", "abcabc") << " (Expected: 1)\n"; // true


    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

Ans:

Equivalence Classes for Triangle Types:

Equilateral Triangle: All sides are equal (A = B = C).

Isosceles Triangle: Two sides are equal (A = B, A = C, or B = C).

Scalene Triangle: All sides are different (A ≠ B, B ≠ C, A ≠ C).

Right Triangle: Follows Pythagorean theorem ($A^2 + B^2 = C^2$, assuming C is the longest side).

Invalid Triangle: Does not satisfy triangle inequality (A + B ≤ C, A + C ≤ B, B + C ≤ A).

Non-Triangle: Any side length that is non-positive (A ≤ 0, B ≤ 0, C ≤ 0).

b) Identify test cases to cover the identified equivalence classes. Also, explicitly mention which test case would cover which equivalence class. (Hint: you must need to be ensure that the identified set of test cases cover all identified equivalence classes)

Ans:

Test Case	Input (A, B, C)	Expected Output	Equivalence Class
TC1	(3, 3, 3)	"Equilateral"	Equilateral Triangle
TC2	(3, 3, 4)	"Isosceles"	Isosceles Triangle
TC3	(4, 3, 3)	"Isosceles"	Isosceles Triangle
TC4	(3, 4, 5)	"Scalene"	Scalene Triangle

TC5	(5, 12, 13)	"Right Angled"	Right Triangle
TC6	(1, 2, 3)	"Invalid"	Invalid Triangle
TC7	(1, 1, 2)	"Invalid"	Invalid Triangle
TC8	(0, 1, 1)	"Non-Triangle"	Non-Triangle
TC9	(1, -1, 1)	"Non-Triangle"	Non-Triangle

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

Test Case	Input (A, B, C)	Expected Output	Description
TC1	(2, 3, 4)	"Scalene"	Valid scalene triangle
TC2	(2, 3, 5)	"Invalid"	Just at the boundary (2 + 3 = 5)
TC3	(3, 4, 6)	"Scalene"	Valid scalene triangle

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

Test Case	Input (A, B, C)	Expected Output	Description
TC1	(3, 4, 3)	"Isosceles"	Valid isosceles triangle
TC2	(3, 3, 4)	"Isosceles"	A and C are equal
TC3	(3, 4, 5)	"Scalene"	Not isosceles, covers boundary (A ≠ B)

e) For the boundary condition A = B = C case (equilateral triangle), identify test

cases to verify the boundary.

Test Case	Input (A, B, C)	Expected Output	Description
TC1	(3, 3, 3)	"Equilateral"	Valid equilateral triangle
TC2	(0, 0, 0)	"Non-Triangle"	Non-positive input, also covers boundary
TC3	(1, 1, 1)	"Equilateral"	Minimum valid equilateral triangle

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

Test Case	Input (A, B, C)	Expected Output	Description
TC1	(3, 4, 5)	"Right Angled"	Valid right-angled triangle
TC2	(5, 12, 13)	"Right Angled"	Another valid case
TC3	(1, 1, √2)	"Right Angled"	Testing with decimal

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

Test Case	Input (A, B, C)	Expected Output	Description
TC1	(1, 2, 3)	"Invalid"	1 + 2 ≤ 3
TC2	(1, 1, 2)	"Invalid"	1 + 1 ≤ 2
TC3	(5, 2, 2)	"Invalid"	5 ≥ 2 + 2

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

Test Case	Input (A, B, C)	Expected Output	Description
TC1	(0, 1, 1)	"Non-Triangle"	One side is zero
TC2	(1, -1, 1)	"Non-Triangle"	One side is negative
TC3	(-1, -1, -1)	"Non-Triangle"	All sides are negative