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

Name :- Aryan Solanki
ID :- 202201239

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.

Input :- Triple of day, month and year

Input ranges :-
1 <= month <= 12
1 <= day <= 31
1900 <= year <= 2015

Output :- : “Previous date” or “Invalid date”

<u>**Test Suite** :-</u>
Equivalence Partitioning Valid Partitions:
•Normal days (not month end or year end)
•Month end (not year end)
•Year end (December 31)
• Leap year February 29 Invalid Partitions:
• Invalid month (< 1 or > 12)
• Invalid day (< 1 or > max days in month)
• Invalid year (< 1900 or > 2015)
• Invalid day for specific month (e.g., February 30)

Boundary Value Analysis:
• First day of year: January 1, YYYY
• Last day of year: December 31, YYYY
• First day of month: DD 1, MM
• Last day of month: DD 30/31, MM (28/29 for February)
•Minimum valid year: 1900
•Maximum valid year: 2015

Equivalence classes :-

No.	**Value**	**Expected Outcome**
E1	1 <= day<= 31	Valid
E2	day < 1	Invalid
E3	day > 31	Invalid
E4	1 <= month <= 12	Valid
E5	month < 1	Invalid
E6	month > 12	Invalid
E7	1900 <= year <= 2015	Valid
E8	year < 1900	Invalid
E9	year > 2015	Invalid

Test Cases :-

No.	**Input Values (date, month, year)**	**Equivalence Classes Covered**	**Expected Outcome**
1	(16, 8, 2013)	E1, E4, E7	(15, 8, 2013)
2	(16, 8, 2025)	E1, E4, E9	Invalid year
3	(16, 15, 2013)	E1, E6, E7	Invalid month
4	(35, 8, 2013)	E3, E4, E7	Invalid day
5	(44, -8, 2013)	E3, E6, E7	Invalid day and month
6	(16, 17, 2026)	E1, E6, E9	Invalid month and

			year
7	(11, 8, 1900)	E1, E4, E7	(10, 8, 1900)
8	(29, 2, 2000)	E1, E4, E7	(28, 2, 2000)
9	(1, 5, 2011)	E1, E4, E7	(30, 4, 2011)
10	(1, 1, 2012)	E1, E4, E7	(31, 12, 2011)
11	(1, 1, 1899)	E1, E4, E8	Invaild year
12	(30, 9, 1900)	E1, E4, E7	(29, 9, 1900)
13	(0, 8, 2013)	E2, E4, E7	Invalid day
14	(11, 0, 2013)	E1, E5, E7	Invalid month
15	(11, 0, 1880)	E1, E5, E8	Invalid month and year
16	(0, 0, 2013)	E2, E5, E7	Invalid day and month
17	(11, 6, 2000)	E1, E4, E7	(10, 6, 2000)
18	(29, 2, 2011)	E1, E4, E7	Invalid day
19	(10, 6, 2000)	E1, E4, E7	(9, 6, 2000)
20	(0, 0, 1800)	E2, E5, E8	Invalid day, month and year

C++ Code :-

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

class Date {
private:
    int day, month, year;
```

```
    // Array to store number of days in each month
    int daysInMonth[12] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30,
31};

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

    string validateDate() {
        bool isValidDay = true;
        bool isValidMonth = true;
        bool isValidYear = true;

        // Check year range
        if (year < 1900 || year > 2015) {
            isValidYear = false;
        }

        // Check month range
        if (month < 1 || month > 12) {
            isValidMonth = false;
        }

        // Check day range
        if (day < 1) {
            isValidDay = false;
        }
        else {
            // Adjust February days for leap year
            if (month >= 1 && month <= 12) {  // Only check if month is
valid
                if (month == 2 && isLeapYear(year)) {
                    daysInMonth[1] = 29;
                } else {
                    daysInMonth[1] = 28;
                }

                // Check if day is valid for the given month
                if (day > daysInMonth[month - 1]) {
```

```
                    isValidDay = false;
                }
            }
        }

        // Return appropriate error message based on combinations
        if (!isValidDay && !isValidMonth && !isValidYear) {
            return "InvalidDateMonthYear";
        }
        else if (!isValidDay && !isValidMonth) {
            return "InvalidDateMonth";
        }
        else if (!isValidDay && !isValidYear) {
            return "InvalidDateYear";
        }
        else if (!isValidMonth && !isValidYear) {
            return "InvalidMonthYear";
        }
        else if (!isValidDay) {
            return "InvalidDate";
        }
        else if (!isValidMonth) {
            return "InvalidMonth";
        }
        else if (!isValidYear) {
            return "InvalidYear";
        }

        return "Valid";
    }

public:
    Date(int d, int m, int y) : day(d), month(m), year(y) {}

    void getPreviousDate() {
        string validationResult = validateDate();

        if (validationResult != "Valid") {
            cout << validationResult << endl;
            return;
```

```
        }

        // Calculate previous date
        day--;

        // If day becomes 0, go to previous month
        if (day == 0) {
            month--;

            // If month becomes 0, go to previous year
            if (month == 0) {
                month = 12;
                year--;
            }

            // Set day to last day of previous month
            if (month == 2 && isLeapYear(year)) {
                day = 29;
            } else {
                day = daysInMonth[month - 1];
            }
        }

        // Print the previous date
        cout << "(" << day << "," << month << "," << year << ")" << endl;
    }
};

int main() {
    int day, month, year;
    char choice = 'y';

    while (choice == 'y' || choice == 'Y') {
        cout << "Enter date (day month year): ";
        cin >> day >> month >> year;

        Date date(day, month, year);
        date.getPreviousDate();

        cout << "\nDo you want to check another date? (y/n): ";
```

```
        cin >> choice;
    }

    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.

Equivalence Classes :-

No.	**Value**
E1	v belongs to array a
E2	v does not belongs to array a

Test Cases :-

TesterAction and InputData	**ExpectedOutcome**
EquivalencePartitioning	
v = 5 , a = [1,2,3,4,5]	4
v = 11 , a = [1,2,3,4,5]	-1
v = 3 , a = [3]	0
v = 0 , a = [1,2]	-1
v = 6 , a = [3,4,5,6]	3
Boundary Value Analysis	

v = 1 , a = [1,4,5,6]	0
v = 6 , a = [3,4,5,6]	3
v = 4 , a = [4,5,6]	0
v = 5 , a = null	An error message
v = {1,2,3} , a = [3,4,5,6]	An error message

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

Test Cases :-

Tester Action and Input Data	ExpectedOutcome
Equivalence Partitioning	
v = 5 , a = [1,2,3,4,5]	1
v = 11 , a = [1,2,3,4,5,11,11]	2
v = 3 , a = [1,2,4,5]	0
v = 0 , a = [0,0,0,0,1,2]	4
v = 6 , a = []	0
Boundary Value Analysis	
v = 1 , a = [1,1,1]	3
v = 6 , a = [3,4,5,6]	1
v = 4 , a = [4,5,6]	1
v = 5 , a = null	An error message

v = {1,2,3} , a = [3,4,5,6]	An error message

Generating test cases using boundary value analysis is not possible.Because, element can either exist in an array or not, there cannot be a boundary value possible.

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.

Since Assumption, it is mentioned that an array is assumed to be sorted in non-decreasing order, it is not required to make a equivalence class that represents if an array is sorted or not.

Test Cases :-

Tester Action and Input Data	ExpectedOutcome
Equivalence Partitioning	
v = 5 , a = [1,2,3,4,5]	4
v = 11 , a = [1,3,5,7,9,11]	5
v = 3 , a = [1,2,4,5]	-1
v = 0 , a = [5,6,7,8]	-1
v = 6 , a = null	An error message
Boundary Value Analysis	
v = 1 , a = [1]	0
v = 6 , a = [3,4,5,6]	3
v = 4 , a = [4,5,6]	0

v = 0 , a = [4,5,6]	-1
v = 5 , a = null	An error message
v = {1,2,3} , a = [3,4,5,6]	An error message

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).

Types of Triangle :-
EQUILATERAL (three lengths equal)
ISOSCELES (two lengths equal)
SCALENE (no lengths equal)
INVALID (impossible lengths)

Test Cases :-

Tester Action and Input Data	ExpectedOutcome
Equivalence Partitioning	
a = 3, b = 3 , c = 3	EQUILATERAL
a = 3, b = 3 , c = 4	ISOSCELES
a = 3, b = 4 , c = 5	SCALENE
a = 2, b = 10, c = 2	INVALID
a = 1, b = 2 , c = 3	INVALID
a = 10, b = 2 , c = 3	INVALID

Boundary Value Analysis	
a = 2, b = 2 , c = 2	EQUILATERAL
a = 1, b = 1 , c = 3	ISOSCELES
a = 2, b = 2 , c = 3	ISOSCELES
a = 5, b = 5 , c = 10	INVALID
a = 5, b = 12 , c = 13	SCALENE

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).

Test Cases :-

Tester Action and Input Data	**Expected Outcome**
Equivalence Partitioning	
s1 = “abc” , s2 = “abcdef”	True
s1 = “ab” , s2 = “cde”	False
s1 = “wxy” , s2 = “zasd”	False
s1 = “mnop” , s2 = “mnopqrst”	True
s1 = “aryan” , s2 = “aryansolanki”	True
Boundary Value Analysis	
s1 = “b” , s2 = “b”	True
s1 = “a” , s2 = “b”	False

s1 = “” , s2 = “”	True
s1 = “xy” , s2 = “z”	False

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

No.	**Value**
E1	Scalene triangle(all sides are different lengths)
E2	Isosceles triangle(two sides are equal)
E3	Equilateral triangle(all sides are equal)
E4	Right angled triangle(Satisfies Pythagorean theorem)
E5	Non triangle (sum of any two sides is less than the third side)
E6	Non positive input

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)

Test Case	Equivalence Class
A = 3.0, B = 4.0 , C= 5.0	E1
A = 3.0, B = 3.0 , C= 5.0	E2
A = 4.0, B = 4.0 , C= 4.0	E3
A = 3.0, B = 4.0 , C= 5.0	E4
A = 1.0, B = 2.0 , C= 3.0	E5
A = -1.0, B = 4.0 , C= 5.0	E6

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

Test Case	Description
A = 1.0, B = 2.0 , C= 3.0	Minimum possible scalene triangle
A = 100.0, B = 100.1 , C= 200.0	Maximum possible scalene triangle

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

Test Case	Description
A = 3.0, B = 4.0 , C= 3.0	Minimum possible isosceles triangle
A = 100.0, B = 200.0 , C= 100.0	Maximum possible isosceles triangle

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

Test Case	Description
A = 1.0, B = 1.0 , C= 1.0	Minimum possible equilateral triangle
A = 100.0, B = 100.0 , C= 100.0	Maximum possible equilateral triangle

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

Test Case	Description
A = 3.0, B = 4.0 , C= 5.0	Minimum possible right-angled triangle
A = 100.0, B = 100.0 , C= 141.42	Maximum possible right-angled triangle

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

Test Case	Description
A = 1.0, B = 2.0 , C= 3.0	Minimum non-triangle case
A = 100.0, B = 100.0 , C= 200.0	Maximum non-triangle case

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

Test Case	Description

A = 0.0, B = 1.0 , C= 2.0	Zero input
A = -1.0, B = 2.0 , C= 3.0	Negative input