Name : - Kunj Kheni
ID : - 202201423

IT - 314 Software Engineering
Lab - 8

Question - 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.

The solution of each problem must be given in the format as follows:

Tester Action and Input Data Expected Outcome Equivalence Partitioning

a, b, c An Error message

a-1, b, c Yes

Boundary Value Analysis

a, b, c-1 Yes

Answer : -

To solve this problem, we will apply Equivalence Partitioning (EP) and Boundary Value Analysis (BVA) for test case generation.

1. Equivalence Partitioning and Boundary Value Analysis

Equivalence Partitioning: We divide the input domain into classes of valid and invalid inputs, assuming the same behavior for all inputs within a particular class. For instance:

- Valid days (1-31 depending on the month)
- Invalid days (e.g., 32, 0)
- Valid months (1-12)
- Invalid months (e.g., 0, 13)
- Valid years (1900-2015)
- Invalid years (e.g., 1899, 2016)

Boundary Value Analysis: We focus on the boundaries of these equivalence classes. For instance, valid day boundaries would be:

- For April, 1-30 are valid; hence boundaries would be: 0, 1, 30, 31
- For February in a leap year: 1-29
- For February in a non-leap year: 1-28

2. Test Cases

Equivalence Partitioning (EP)

These test cases cover representative valid and invalid inputs within the partitions.

Input Data	Expected Outcome
32, 5, 2010	Invalid day error
0, 6, 2010	Invalid day error
15, 0, 2010	Invalid month error

15, 13, 2010	Invalid month error
15, 7, 1899	Invalid year error
15, 7, 2016	Invalid year error
15, 4, 2010	Valid (previous date: 14/4/2010)
1, 3, 2000	Valid (previous date: 29/2/2000, leap year)
1, 3, 2001	Valid (previous date: 28/2/2001, non-leap year)

Boundary Value Analysis (BVA)

These test cases focus on the boundary values of the input data.

Input Data	**Expected Outcome**
1, 5, 2010	30/4/2010
1, 1, 2010	31/12/2009
1, 6, 2010	31/5/2010
1, 3, 2010	28/2/2010(non-leap year)
1, 3, 2004	29/2/2004(leap year)
29, 2, 2003	Invalid date
1, 1, 2011	31/12/2010
31, 12, 2015	30/12/2015
1, 8, 2010	31/7/2010
1, 7, 2010	30/6/2010
0, 1, 2010	Invalid day error
32, 1, 2010	Invalid day error
30, 2, 2010	Invalid day error (February)
15, 0, 2010	Invalid month error
15, 13, 2010	Invalid month error

Code : -

```cpp
#include <iostream>
#include <iomanip>
#include <ctime>

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

int daysInMonth(int month, int year)
{
    if (month == 2)
    {
        return isLeapYear(year) ? 29 : 28;
    }
    if (month == 4 || month == 6 || month == 9 || month == 11)
    {
        return 30;
    }
    return 31;
}
```

```cpp
std::string previousDate(int day, int month, int year)

{

    if (month < 1 || month > 12 || day < 1 || year < 1900 || year > 2015)

    {

        return "Invalid date";

    }



    int days_in_prev_month;



    // Handle the case where we need to go to the previous month or year

    if (day == 1)

    {

        if (month == 1)

        { // If it is 1st January, go to the previous year (31st December)

            day = 31;

            month = 12;

            year--;

        }

        else

        { // Go to the last day of the previous month

            month--;

            days_in_prev_month = daysInMonth(month, year);
            day = days_in_prev_month;

        }

    }

    else

    { // Otherwise, just go back one day

        day--;

    }



    // Validate the year is still in range

    if (year < 1900)

    {

        return "Invalid date";

    }



    // Format the previous date as dd/mm/yyyy

    std::ostringstream result;

    result << std::setw(2) << std::setfill('0') << day << "/"

           << std::setw(2) << std::setfill('0') << month << "/"

           << year;



    return result.str();

}
```

```
int main()
{
    // Test cases
    int test_cases[][3] = {
        {32, 5, 2010}, {0, 6, 2010}, {15, 0, 2010}, {15, 13, 2010}, {15, 7,
1899}, {15, 7, 2016}, {15, 4, 2010}, {1, 3, 2000}, {1, 3, 2001}, {1, 1, 1900},
{31, 12, 2015}, {1, 3, 2004}, {1, 3, 2003}, {30, 4, 2010}, {31, 12, 1900}, {1, 2,
2010}, {0, 1, 2010}};

    for (auto &test_case : test_cases)
    {
        int day = test_case[0];
        int month = test_case[1];
        int year = test_case[2];

        std::cout << "Input: " << day << "/" << month << "/" << year
                  << " -> Previous Date: " << previousDate(day, month, year) <<
std::endl;
    }

    return 0;
}
```

Question - 2

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 (EP) Test Cases:

Input Data	Expected Output
linearSearch(7, [5, 3, 7, 9, 10], 5)	Returns index 2
linearSearch(1, [5, 3, 7, 9, 10], 5)	Returns -1
linearSearch(2, [2], 1)	Returns index 0
linearSearch(7, [], 0)	Returns -1

linearSearch(5, [1, 2, 3, 4, 5], 5)	Returns index 4
linearSearch(2147483648, [5, 3, 7, 9, 10], 5)	Returns error (out of range)
linearSearch(1, [2147483648, 3, 7, 9, 10], 5)	Returns error (array element out of range)
linearSearch(1, largeTest, 1000001)	Returns error (array too large)

Boundary Value Analysis (BVA) Test Cases:

Input Data	**Expected Output**
linearSearch(5, [5, 3, 7, 9, 10], 5)	Returns index 0
linearSearch(10, [5, 3, 7, 9, 10], 5)	Returns index 4
linearSearch(7, [5, 3, 7, 9, 10], 5)	Returns index 2
linearSearch(1, [5, 3, 7, 9, 10], 5)	Returns -1
linearSearch(7, [], 0)	Returns -1
linearSearch(2, [2], 1)	Returns index 0
linearSearch(3, [2], 1)	Returns -1
linearSearch(1, largeTest, 1000001)	Returns -1 and prints error: "Array size exceeds limit"
linearSearch(2147483648, [5, 3, 7, 9, 10], 5)	Returns -1 and prints error: "Value of v is out of range"
linearSearch(1, [2147483648, 3, 7, 9, 10], 5)	Returns -1 and prints error: "Array element out of range"

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 (EP) Test Cases:

Input Data	Expected Output
countItem(5, [5, 3, 7, 5, 10], 5)	Returns 2
countItem(6, [1, 2, 3, 4, 5], 5)	Returns 0
countItem(7, [], 0)	Returns 0 (Empty Array)
countItem(2, [2], 1)	Returns 1
countItem(2, [2, 2, 2], 3)	Returns 3
countItem(2147483648, [1, 2, 3, 4, 5], 5)	Returns -1 and prints error: "v out of range"
countItem(1, [2147483648, 3, 7, 9, 10], 5)	Returns -1 and prints error: "Array element out of range"
countItem(1, largeTest, 1000001)	Returns -1 and prints error: "Array size exceeds limit"

Boundary Value Analysis (BVA) Test Cases:

Input Data	Expected Output
countItem(5, [5, 3, 7, 5, 10], 5)	Returns 2
countItem(10, [5, 3, 7, 9, 10], 5)	Returns 1
countItem(7, [5, 3, 7, 9, 10], 5)	Returns 1
countItem(6, [1, 2, 3, 4, 5], 5)	Returns 0
countItem(7, [], 0)	Returns 0
countItem(2, [2], 1)	Returns 1
countItem(3, [2], 1)	Returns 0
countItem(1, largeTest, 1000001)	Returns -1 and prints error: "Array size exceeds limit"
countItem(2147483648, [5, 3, 7, 9, 10], 5)	Returns -1 and prints error: "Value of v is out of range"
countItem(1, [2147483648, 3, 7, 9, 10], 5)	Returns -1 and prints error: "Array element out of range"

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.

```
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 (EP) Test Cases:

Input Data	Expected Output
binarySearch(5, [1, 3, 5, 7, 9], 5)	Returns index 2
binarySearch(5, [1, 3, 5, 7, 9], 5)	Returns index 0
binarySearch(5, [1, 3, 5, 7, 9], 5)	Returns index 4
binarySearch(5, [1, 3, 5, 7, 9], 5)	Returns -1 (not found)
binarySearch(5, [1, 3, 5, 7, 9], 5)	Returns index 3 (one of the occurrences)
binarySearch(5, [], 0)	Returns -1
binarySearch(2, [2], 1)	Returns index 0 (single element matches v)
binarySearch(6, [2], 1)	Returns index -1 (single element but v not present)
binarySearch(2147483648, [1, 2, 3, 4, 5], 5)	Returns index -1 and prints error: "v out of range"
binarySearch(1, [1, 2147483648, 3, 4, 5], 5)	Returns index -1 and prints error: "Array element out of range"

Boundary Value Analysis (BVA) Test Cases:

Input Data	Expected Output
binarySearch(1, [1, 3, 5, 7, 9], 5)	Returns index 0
binarySearch(1, [1, 3, 5, 7, 9], 5)	Returns index 4
binarySearch(1, [1, 3, 5, 7, 9], 5)	Returns index 2
binarySearch(1, [1, 3, 5, 7, 9], 5)	Returns -1
binarySearch(7, [], 0)	Returns -1
binarySearch(2, [2], 1)	Returns index 0
binarySearch(3, [2], 1)	Returns -1
binarySearch(1, largeTest, 1000001)	Returns -1 and prints error: "Array size exceeds limit"
binarySearch(2147483648, [1, 3, 5, 7, 9], 5)	Returns -1 and prints error: "v out of range"
binarySearch(1, [1, 2147483648, 3, 4, 5], 5)	Returns -1 and prints error: "Array element out of range"

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 (EP) Test Cases:

Input Data	Expected Output
triangle(3, 3, 3)	Returns EQUILATERAL (0)
triangle(3, 3, 5)	Returns ISOSCELES (1)
triangle(3, 4, 5)	Returns SCALENE (2)
triangle(1, 2, 3)	Returns INVALID (3)
triangle(0, 0, 0)	Returns INVALID (3)
triangle(-1, -1, -1)	Returns INVALID (3)
triangle(1, 1, 2)	Returns INVALID (3)
triangle(5, 5, 10)	Returns INVALID (3)
triangle(1, 2, -3)	Returns INVALID (3)
triangle(2147483648, 3, 4)	Returns INVALID (3) and prints error: "a out of range"
triangle(1, 3, -2147483649)	Returns INVALID (3) and prints error: "c out of range"
triangle(3, 2147483648, 5)	Returns INVALID (3) and prints error: "b out of range"

Boundary Value Analysis (BVA) Test Cases:

Input Data	Expected Output
triangle(1, 1, 1)	Returns EQUILATERAL (0)
triangle(1, 1, 2)	Returns INVALID (3)
triangle(0, 0, 0)	Returns INVALID (3)
triangle(-1, -1, -1)	Returns INVALID (3)
triangle(1, 1, 2)	Returns INVALID (3)
triangle(3, 4, 5)	Returns SCALENE (2)
triangle(5, 5, 10)	Returns INVALID (3)
triangle(1, 2, -3)	Returns INVALID (3)
triangle(1000000, 1000000, 1000000)	Returns EQUILATERAL (0)
triangle(1000000, 1000000, 2000000)	Returns INVALID (3)
triangle(2147483648, 3, 4)	Returns INVALID (3) and prints error: "a out of range"
triangle(1, 3, -2147483649)	Returns INVALID (3) and prints error: "c out of range"
triangle(3, 2147483648, 5)	Returns INVALID (3) and prints error: "b out of range"

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 (EP) Test Cases:

Input Data	**Expected Output**
prefix("abc", "abcde")	Returns true
prefix("abc", "ab")	Returns false
prefix("abc", "abcdef")	Returns true
prefix("abcde", "abc")	Returns false
prefix("a", "a")	Returns true
prefix("abcd", "abcde")	Returns true
prefix("abcd", "abCde")	Returns false
prefix("a" * 1000000, "a" * 1000000 + "b")	Returns true

prefix("a" * 1000000 + "b", "a" * 1000000)	Returns false
prefix("a" * Integer.MAX_VALUE, "a" * Integer.MAX_VALUE)	May cause OutOfMemoryError or Overflow
prefix("a" * (Integer.MAX_VALUE - 1), "a" * Integer.MAX_VALUE)	Returns true or may cause OutOfMemoryError

Boundary Value Analysis (BVA) Test Cases:

Input Data	**Expected Output**
triangle(1, 1, 1)	Returns EQUILATERAL (0)
triangle(1, 1, 2)	Returns INVALID (3)
triangle(0, 0, 0)	Returns INVALID (3)
triangle(-1, -1, -1)	Returns INVALID (3)
triangle(1, 1, 2)	Returns INVALID (3)
triangle(3, 4, 5)	Returns SCALENE (2)
triangle(5, 5, 10)	Returns INVALID (3)
triangle(1, 2, -3)	Returns INVALID (3)
triangle(1000000, 1000000, 1000000)	Returns EQUILATERAL (0)
triangle(1000000, 1000000, 2000000)	Returns INVALID (3)
triangle(2147483648, 3, 4)	Returns INVALID (3) and prints error: "a out of range"
triangle(1, 3, -2147483649)	Returns INVALID (3) and prints error: "c out of range"
triangle(3, 2147483648, 5)	Returns INVALID (3) and prints error: "b out of range"

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

Valid Triangles:

- **1**: **Equilateral Triangle**: All sides are equal ($A = B = C$).
- **2**: **Isosceles Triangle**: Exactly two sides are equal ($A = B$ or $A = C$ or $B = C$).
- **3**: **Scalene Triangle**: All sides are different ($A \neq B$, $A \neq C$, $B \neq C$).
- **4**: **Right-Angled Triangle**: Satisfies Pythagorean theorem ($A^2 + B^2 = C^2$ or any permutation).

Invalid Triangles:

- **5**: **Non-Triangle Condition**: Fails triangle inequality ($A + B \leq C$, $A + C \leq B$, $B + C \leq A$).
- **6**: **Non-positive Input**: Any side is less than or equal to zero ($A \leq 0$, $B \leq 0$, $C \leq 0$).
- **7**: **Integer Overflow**: Any side exceeds the maximum value for integers (e.g., A > Integer.MAX_VALUE, B > Integer.MAX_VALUE, C > Integer.MAX_VALUE).

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

Test Case	Expected Outcome	Covered Equivalence Class
(3.0, 3.0, 3.0)	Equilateral	1
(4.0, 4.0, 2.0)	Isosceles	2
(5.0, 6.0, 7.0)	Scalene	3
(3.0, 4.0, 5.0)	Right Angled	4
(1.0, 2.0, 3.0)	Not a Triangle	5
(5.0, 1.0, 2.0)	Not a Triangle	5
(0.0, 2.0, 3.0)	Not a Triangle	6
(-1.0, 2.0, 3.0)	Not a Triangle	6
(Integer.MAX_VALUE + 1, 1.0, 1.0)	Not a Triangle or may cause Overflow	7
(2.0, Integer.MAX_VALUE + 1, 2.0)	Not a Triangle or may cause Overflow	7
(1.0, 1.0, Integer.MAX_VALUE + 1)	Not a Triangle or may cause Overflow	7

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

Test Case	Expected Outcome
(2.0, 3.0, 4.0)	Scalene
(2.0, 2.0, 4.0)	Not a Triangle
(1.0, 2.0, 2.9)	Scalene
(0.0, 2.0, 3.0)	Not a Triangle
(Integer.MAX_VALUE + 1, 1.0, 1.0)	Not a Triangle or may cause Overflow

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

Test Case	Expected Outcome
(4.0, 5.0, 4.0)	Isosceles
(5.0, 5.0, 5.0)	Equilateral
(0.0, 5.0, 0.0)	Not a Triangle
(Integer.MAX_VALUE + 1, 1.0, Integer.MAX_VALUE + 1)	Not a Triangle or may cause Overflow

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

Test Case	Expected Outcome
(1.0, 1.0, 1.0)	Equilateral
(1.1, 1.1, 1.1)	Equilateral
(0.0, 0.0, 0.0)	Not a Triangle
(Integer.MAX_VALUE + 1, Integer.MAX_VALUE + 1, Integer.MAX_VALUE + 1)	Not a Triangle or may cause Overflow

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

Test Case	Expected Outcome
(3.0, 4.0, 5.0)	Right Angled
(0.0, 4.0, 4.0)	Not a Triangle
(Integer.MAX_VALUE, Integer.MAX_VALUE, Integer.MAX_VALUE + 1)	Not a Triangle or may cause Overflow

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

Test Case	Expected Outcome
(5.0, 5.0, 10.0)	Not a Triangle
(0.0, 0.0, 0.0)	Not a Triangle
(Integer.MAX_VALUE, Integer.MAX_VALUE, Integer.MAX_VALUE)	Not a Triangle or may cause Overflow

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

Test Case	**Expected Outcome**
(0.0, 0.0, 0.0)	Not a Triangle
(0.0, 2.0, 3.0)	Not a Triangle
(-1.0, -2.0, -3.0)	Not a Triangle
(-1.0, 2.0, 3.0)	Not a Triangle
(-1.0, -2.0, 3.0)	Not a Triangle
(1.0, 2.0, -3.0)	Not a Triangle
(0.0, 0.0, 5.0)	Not a Triangle