IT314 Software Engineering
Lab-8

Student-ID:- 202201509 Name:- Ramkumar Patel

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.

Answer :-

(1)

❖ Equivalence Classes

Valid Equivalence Classes

1. Valid Years:
    - Class: 1900 to 2015
    - Test Cases:
        - 1, 1, 1900 → 31, 12, 1899
        - 1, 1, 2015 → 31, 12, 2014
2. Valid Months:
    - Class: 1 to 12
    - Test Cases:
        - 31, 12, 2015 → 30, 12, 2015
        - 29, 2, 2012 → 28, 2, 2012 (leap year)
3. Valid Days:
    - Class: 1 to 31 (depending on the month)
    - Test Cases:
        - 31, 4, 2015 → 30, 4, 2015

Invalid Equivalence Classes

1. Invalid Years:
    - Class: Less than 1900 or greater than 2015
    - Test Cases:
        - 1, 1, 1899 → Error message
        - 1, 1, 2016 → Error message
2. Invalid Months:
    - Class: Less than 1 or greater than 12
    - Test Cases:
        - 0, 5, 2010 → Error message
        - 1, 13, 2010 → Error message
3. Invalid Days:
    - Class: Less than 1 or greater than the number of days in the month
    - Test Cases:
        - 32, 1, 2010 → Error message
        - 31, 4, 2015 (April has only 30 days) → Error message (not included in valid test cases)

Equivalence Partitioning Test Cases

Tester Action and Input Data	Expected Outcome	Reasoning
1, 1, 1900	31, 12, 1899	Valid date, checking transition to previous year.
1, 1, 2015	31, 12, 2014	Valid date, checking transition to previous year.
31, 12, 2015	30, 12, 2015	Valid date, testing end of year.
29, 2, 2012	28, 2, 2012	Valid leap year date, ensuring leap year handling is correct.
31, 4, 2015	30, 4, 2015	Valid date, checking transition within same month.
1, 1, 1899	Error message	Invalid year (below valid range).
0, 5, 2010	Error message	Invalid month (0 is not a valid month).
1, 13, 2010	Error message	Invalid month (13 exceeds valid range).
32, 1, 2010	Error message	Invalid day (32 exceeds valid range).
31, 1, 1899	Error message	Invalid year (below valid range).

Boundary Value Analysis Test Cases

Tester Action and Input Data	Expected Outcome	Reasoning
1, 1, 1900	31, 12, 1899	Lower boundary of year; testing transition to previous year.
1, 1, 2015	31, 12, 2014	Upper boundary of year; testing transition to previous year.
31, 12, 1900	30, 12, 1900	Testing end of year at lower boundary.
1, 12, 2015	30, 11, 2015	Boundary case for December; checking month transition.
1, 1, 1899	Error message	Invalid year (below valid range).
1, 1, 2016	Error message	Invalid year (above valid range).
29, 2, 2015	28, 2, 2015	Non-leap year boundary; checking February's handling.
31, 12, 2015	30, 12, 2015	Valid date at upper boundary; testing end of year.
31, 4, 1900	30, 4, 1900	Valid date at lower boundary of April; testing transition.
1, 12, 1899	Error message	Invalid year (below valid range).

(2) Program for checking verifying test-cases:-

```
public class PreviousDate {
      public static void main(String[] args) {
        // Test suite
        testPreviousDate(1, 1, 1900); // Expected: 31, 12, 1899
        testPreviousDate(1, 1, 2015); // Expected: 31, 12, 2014
        testPreviousDate(31, 12, 2015); // Expected: 30, 12, 2015
        testPreviousDate(29, 2, 2012); // Expected: 28, 2, 2012
        testPreviousDate(31, 4, 2015); // Expected: 30, 4, 2015
        testPreviousDate(1, 1, 1899); // Expected: Error message
```

```
        testPreviousDate(0, 5, 2010); // Expected: Error message
        testPreviousDate(1, 13, 2010); // Expected: Error message
        testPreviousDate(32, 1, 2010); // Expected: Error message
        testPreviousDate(31, 1, 1899); // Expected: Error message
    }

    public static void testPreviousDate(int day, int month, int year) {
        System.out.printf("Input: %d, %d, %d -> Previous Date: %s%n", day,
month, year, getPreviousDate(day, month, year));
    }

    public static String getPreviousDate(int day, int month, int year) {
        if (year < 1900 || year > 2015) {
            return "Error: Invalid year";
        }
        if (month < 1 || month > 12) {
            return "Error: Invalid month";
        }
        if (day < 1 || day > 31) {
            return "Error: Invalid day";
        }

        // Days in each month
        int[] daysInMonth = {31, (isLeapYear(year) ? 29 : 28), 31, 30, 31,
30, 31, 31, 30, 31, 30, 31};

        if (day > daysInMonth[month - 1]) {
            return "Error: Invalid day for the given month";
        }

        // Calculate previous date
        if (day > 1) {
            return String.format("%d, %d, %d", day - 1, month, year);
        } else {
            if (month == 1) {
                return String.format("%d, %d, %d", 31, 12, year - 1);
            } else {
                return String.format("%d, %d, %d", daysInMonth[month - 2],
month - 1, year);
            }
```

```
        }
    }

    public static boolean isLeapYear(int year) {
        return (year % 4 == 0 && year % 100 != 0) || (year % 400 == 0);
    }
}
```

❖ **Output of the Test-Cases:-**

```
Input: 1, 1, 1900 -> Previous Date: 31, 12, 1899
Input: 1, 1, 2015 -> Previous Date: 31, 12, 2014
Input: 31, 12, 2015 -> Previous Date: 30, 12, 2015
Input: 29, 2, 2012 -> Previous Date: 28, 2, 2012
Input: 31, 4, 2015 -> Previous Date: 30, 4, 2015
Input: 1, 1, 1899 -> Previous Date: Error: Invalid year
Input: 0, 5, 2010 -> Previous Date: Error: Invalid day
Input: 1, 13, 2010 -> Previous Date: Error: Invalid month
Input: 32, 1, 2010 -> Previous Date: Error: Invalid day
Input: 31, 1, 1899 -> Previous Date: Error: Invalid year
```

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);
}
```

❖ Test-Cases:-

Tester Action and Input Data	Expected Outcome	Reasoning
3, [1, 2, 3, 4, 5]	2	Value exists in the array.
1, [1, 2, 3, 4, 5]	0	Search for the first element.
5, [1, 2, 3, 4, 5]	4	Search for the last element.
2, [1, 2, 3, 2, 5]	1	Value appears multiple times.
6, [1, 2, 3, 4, 5]	-1	Value not in the array.
1, []	-1	Search in an empty array.
1, [1]	0	Minimum value in a one-element array.
2, [1]	-1	Value not present in a one-element array.
0, [-1, 0, 1]	1	Boundary case with negative and positive.
1000, [1, 2, ..., 1000]	999	Last value in a large array.

❖ Modified Code:-

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

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

```
}

int main() {
    int testCases[][3] = {
        {3, 1, 2, 3, 4, 5}, // Test case 1
        {1, 1, 2, 3, 4, 5}, // Test case 2
        {5, 1, 2, 3, 4, 5}, // Test case 3
        {2, 1, 2, 3, 2, 5}, // Test case 4
        {6, 1, 2, 3, 4, 5}, // Test case 5
        {1, -1},            // Test case 6 (empty array)
        {1, 1},             // Test case 7
        {2, 1},             // Test case 8
        {0, -1, 0, 1},      // Test case 9
        {1000, 1, 2, ..., 1000} // Test case 10 (large array)
    };

    for (const auto& test : testCases) {
        int value = test[0];
        int expectedOutput = test[1];
        int result = linearSearch(value, test + 2, sizeof(test) /
sizeof(test[0]) - 2);
        cout << "Searching for " << value << ": expected " <<
expectedOutput << ", got " << result << 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);
}
```

❖ **Test-Cases:-**

Input Value (v)	Input Array (a)	Expected Output	Description
2	[1, 2, 2, 3, 4]	2	Count a value that appears multiple times.
1	[1, 2, 3, 4, 5]	1	Count a value that appears only once.
6	[1, 2, 3, 4, 5]	0	Count a value that does not exist in the array.
5	[5, 1, 2, 5, 3, 5]	3	Count a value that appears multiple times at start and end.
1	[]	0	Count in an empty array.
1	[1]	1	Count the minimum value in a one-element array.
2	[1]	0	Count a value in a one-element array (value not present).
-2	[-2, -1, -2, -3]	2	Count a value equal to the

			minimum in negative numbers.
1000	[1, 2, ..., 1000]	1	Count the maximum value in a large array.
500	[100, 500, 500, 500]	3	Count a value that appears multiple times in the middle.

❖ Modified Code:-

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

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

int main() {
    int testCases[][6] = {
        {2, 1, 2, 2, 3, 4},
        {1, 1, 2, 3, 4, 5},
        {6, 1, 2, 3, 4, 5},
        {5, 5, 1, 2, 5, 3, 5},
        {1},
        {2, 1},
        {-2, -2, -1, -2, -3},
        {1000, 1, 2, 3, 4, 1000},
        {500, 100, 500, 500}
    };
```

```
    int expectedOutputs[] = {2, 1, 0, 3, 1, 0, 2, 1, 3};

    for (int i = 0; i < sizeof(testCases) / sizeof(testCases[0]);
i++) {
        int value = testCases[i][0];
        int result = countItem(value, testCases[i] + 1,
sizeof(testCases[i]) / sizeof(testCases[i][0]) - 1);
       cout << "Searching for " << value << ": expected " <<
expectedOutputs[i] << ", got " << result << 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.

❖ Test Cases

Tester Action and Input Data	Expected Outcome
Equivalence Partitioning: Valid Inputs	
`binarySearch(3, {1, 2, 3, 4, 5})`	`2`
`binarySearch(6, {1, 2, 3, 4, 5})`	`-1`
`binarySearch(0, {1, 2, 3, 4, 5})`	`-1`
`binarySearch(10, {1, 2, 3, 4, 5})`	`-1`
`binarySearch(1, {1})`	`0`

`binarySearch(2, {1})`	`-1`

Equivalence Partitioning: Invalid Inputs

`binarySearch(3, {})`	`-1`
`binarySearch(3, {5, 1, 3, 2, 4})`	Invalid , array not sorted
`binarySearch(3, {1, null, 3, null})`	Error
`binarySearch(5, {1, "two", 3, "four"})`	Error

Boundary Value Analysis: Valid Inputs

`binarySearch(1, {1, 2, 3, 4, 5})`	`0`
`binarySearch(5, {1, 2, 3, 4, 5})`	`4`
`binarySearch(0, {1, 2, 3, 4, 5})`	`-1`
`binarySearch(6, {1, 2, 3, 4, 5})`	`-1`

Boundary Value Analysis: Invalid Inputs

`binarySearch(1, {})`	`-1`
`binarySearch(2, {1})`	`-1`

❖ **Modified-Code:-**

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


int binarySearch(int v, int a[], int size)

{

    int lo = 0, hi = size - 1;

    while (lo <= hi)

    {

        int mid = (lo + hi) / 2;

        if (v == a[mid])

            return mid;

        else if (v < a[mid])

            hi = mid - 1;

        else

            lo = mid + 1;

    }
}
```

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);
}
```

❖ Test Cases

Input Values (a, b, c)	Expected Output	Description
3, 3, 3	0	Equilateral triangle.
3, 3, 2	1	Isosceles triangle.
3, 4, 5	2	Scalene triangle.
1, 1, 2	3	Invalid triangle (not possible lengths).
5, 5, 5	0	Equilateral triangle (edge case).
2, 2, 3	1	Isosceles triangle (edge case).

2, 3, 4	2	Scalene triangle (edge case).
1, 2, 3	3	Invalid triangle (sum of two sides equals third).
0, 0, 0	3	Invalid triangle (zero length sides).
-1, -1, -1	3	Invalid triangle (negative lengths).

❖ Modified Code:-

```
#include <iostream>
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 >= 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;
}

int main() {
    int testCases[][3] = {
        {3, 3, 3},
        {3, 3, 2},
        {3, 4, 5},
        {1, 1, 2},
        {5, 5, 5},
```

```
        {2, 2, 3},
        {2, 3, 4},
        {1, 2, 3},
        {0, 0, 0},
        {-1, -1, -1}
    };

    int expectedOutputs[] = {
        EQUILATERAL,
        ISOSCELES,
        SCALENE,
        INVALID,
        EQUILATERAL,
        ISOSCELES,
        SCALENE,
        INVALID,
        INVALID,
        INVALID
    };

    for (int i = 0; i < sizeof(testCases) / sizeof(testCases[0]);
i++) {
        int a = testCases[i][0];
        int b = testCases[i][1];
        int c = testCases[i][2];
        int result = triangle(a, b, c);
        cout << "Triangle with sides (" << a << ", " << b << ", " <<
c << "): expected "
             << expectedOutputs[i] << ", got " << result << endl;
    }

    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).

```java
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;
}
```

❖ Test Cases

Input Strings (s1, s2)	Expected Output	Description
"pre" , "prefix"	true	s1 is a prefix of s2.
"hello" , "hello world"	true	s1 is a prefix of s2.
"world" , "hello world"	false	s1 is not a prefix of s2.

"java" , "javascript"	true	s1 is a prefix of s2.
"test" , "testing"	true	s1 is a prefix of s2.
"abc" , "ab"	false	s1 is longer than s2.
"" , "anything"	true	Empty string is a prefix of any string.
"non" , ""	false	Non-empty string cannot be a prefix of an empty string.
"prefix" , "pre"	false	s1 is longer than s2.
"test" , "Test"	false	Case-sensitive check; different cases.

❖ **Modified Code:-**

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

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;
}

int main() {
    string testCases[][2] = {
        {"pre", "prefix"},
```

```
        {"hello", "hello world"},
        {"world", "hello world"},
        {"java", "javascript"},
        {"test", "testing"},
        {"abc", "ab"},
        {"", "anything"},
        {"non", ""},
        {"prefix", "pre"},
        {"test", "Test"}
    };

    bool expectedOutputs[] = {
        true, true, false, true, true, false, true, false, false,
false
    };

    for (size_t i = 0; i < sizeof(testCases) / sizeof(testCases[0]);
i++) {
        string s1 = testCases[i][0];
        string s2 = testCases[i][1];
        bool result = prefix(s1, s2);
        cout << "Prefix check for (" << s1 << ", " << s2 << "):
expected "
             << expectedOutputs[i] << ", got " << result << endl;
    }

    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. **Valid Triangle (General):**
    - **Sides form a triangle (the sum of any two sides is greater than the third).**
2. **Equivalence Class 1 (EC1): Valid triangle where a == b == c (Equilateral triangle).**
   Equivalence Class 2 (EC2): Valid triangle where a == b ≠ c or a ≠ b == c or a == c ≠ b (Isosceles triangle).
   Equivalence Class 3 (EC3): Valid triangle where a ≠ b ≠ c (Scalene triangle).
   Equivalence Class 4 (EC4): Valid right-angle triangle where $A^2 + B^2 = C^2$ (Pythagorean theorem).
3. **Invalid Triangle:**
    - **The sum of two sides is less than or equal to the third.**
4. **Equivalence Class 5 (EC5): Invalid triangle where a + b <= c or a + c <= b or b + c <= a.**

Equivalence Class 6 (EC6): Invalid triangle where one or more sides are zero or negative.

b) Test Case for equivalence Class

Test Case No.	Input (a, b, c)	Expected Output	Equivalence Class Covered
TC1	(3, 3, 3)	EQUILATERAL	EC1
TC2	(4, 4, 2)	ISOSCELES	EC2
TC3	(3, 4, 5)	SCALENE	EC3
TC4	(6, 8, 10)	SCALENE (Right-Angle)	EC4
TC5	(1, 2, 3)	INVALID	EC5
TC6	(1, 1, 2)	INVALID	EC5
TC7	(0, 2, 3)	INVALID	EC6
TC8	(-1, 2, 3)	INVALID	EC6
TC9	(3.0, 3.0, 3.0)	EQUILATERAL	EC1
TC10	(5.0, 5.0, 7.0)	ISOSCELES	EC2
TC11	(4.2, 3.0, 5.0)	SCALENE	EC3

TC12	(6.0, 8.0, 10.0)	RIGHT-ANGLE	EC4
TC13	(1.0, 2.0, 3.0)	INVALID	EC5
TC14	(0.0, 2.0, 2.0)	INVALID	EC6
TC15	(-3.0, 4.0, 5.0)	INVALID	EC6

c) Boundary Condition for A + B > C (Scalene Triangle):

Test Case No.	Input (A, B, C)	Expected Output	Explanation
TC16	(1.0, 1.0, 2.0)	INVALID	A + B = C (invalid boundary)
TC17	(2.0, 3.0, 4.9)	SCALENE	A + B > C (valid boundary)
TC18	(2.0, 3.0, 5.0)	INVALID	A + B = C (invalid boundary)

d) Boundary Condition for A = C (Isosceles Triangle):

Test Case No.	Input (A, B, C)	Expected Output	Explanation
TC19	(3.0, 4.0, 3.0)	ISOSCELES	A = C (valid boundary)

TC20	(3.0, 5.0, 3.0)	ISOSCELES	A = C (valid boundary)
TC21	(3.0, 3.0, 5.0)	ISOSCELES	A = B (valid boundary)

e) Boundary Condition for A = B = C (Equilateral Triangle):

Test Case No.	Input (A, B, C)	Expected Output	Explanation
TC22	(5.0, 5.0, 5.0)	EQUILATERAL	A = B = C (valid boundary)
TC23	(6.0, 6.0, 6.0)	EQUILATERAL	A = B = C (valid boundary)
TC24	(6.1, 6.1, 6.1)	EQUILATERAL	A = B = C (valid boundary)

f) Boundary Condition for $A^2 + B^2 = C^2$ (Right-Angle Triangle):

Test Case No.	Input (A, B, C)	Expected Output	Explanation
TC25	(3.0, 4.0, 5.0)	RIGHT-ANGLE	$A^2 + B^2 = C^2$ (valid boundary)
TC26	(5.0, 12.0, 13.0)	RIGHT-ANGLE	$A^2 + B^2 = C^2$ (valid boundary)
TC27	(8.0, 15.0, 17.0)	RIGHT-ANGLE	$A^2 + B^2 = C^2$ (valid boundary)

g) Non-Triangle Case Boundaries:

Test Case No.	Input (A, B, C)	Expected Output	Explanation
TC28	(1.0, 1.0, 2.0)	INVALID	A + B = C (invalid)
TC29	(2.0, 3.0, 6.0)	INVALID	A + B < C (invalid)

h) Non-Positive Input Test Cases:

Test Case No.	Input (A, B, C)	Expected Output	Explanation
TC30	(0.0, 5.0, 7.0)	INVALID	Non-positive side length (A=0)
TC31	(5.0, 0.0, 7.0)	INVALID	Non-positive side length (B=0)
TC32	(-3.0, 4.0, 5.0)	INVALID	Negative side length (A=-3)