IT314 – Software Engineering

Lab: 08

Name: Avinash Baraiya

ID: 202201211

Lab Group: 3

Q1. 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 Class Partitioning Test Cases

Input	Expected Outcome	Reasoning
10, 6, 2010	Valid	Normal valid date
1, 3, 2012	Valid	Valid date after February in leap year
1, 1, 1900	Valid	Valid date at lower year boundary
31, 12, 2015	Valid	Valid date at upper year boundary
33, 5, 2000	Invalid	Invalid day (>31)
0, 8, 2005	Invalid	Invalid day (0)
15, 13, 2001	Invalid	Invalid month (>12)
15, 0, 2001	Invalid	Invalid month (0)
31, 4, 2000	Invalid	Invalid day (April has 30 days)
29, 2, 2001	Invalid	Invalid day (non-leap year February)

Boundary Value Analysis Test Cases

Input	Expected Outcome	Reasoning
1, 1, 1900	Valid	Minimum valid year
31, 12, 2015	Valid	Maximum valid year
31, 12, 1899	Invalid	Year below minimum
1, 1, 2016	Invalid	Year above maximum
29, 2, 2000	Valid	Leap year February boundary
29, 2, 1900	Invalid	Non-leap century year
31, 1, 2000	Valid	Maximum day in 31-day month
30, 4, 2000	Valid	Maximum day in 30-day month
28, 2, 2001	Valid	Maximum day in February non-leap year
1, 1, 2000	Valid	Minimum day in any month

Implementation Code

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

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

int getDaysInMonth(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;
}
```

```
void getPreviousDate(int day, int month, int year) {
    if (year < 1900 || year > 2015 || month < 1 || month > 12 |
|
        day < 1 || day > getDaysInMonth(month, year)) {
        cout << "Invalid date" << endl;
        return;
    }

    day--;
    if (day == 0) {
        month--;
        if (month == 0) {
            month = 12;
            year--;
        }
        day = getDaysInMonth(month, year);
    }

    if (year < 1900) {
        cout << "Invalid date" << endl;
        return;
    }

    cout << "Previous date is: " << day << "/" << month << "/"
<< year << endl;
}

int main() {
    // Test cases execution
    getPreviousDate(1, 1, 2000);
    getPreviousDate(31, 12, 2015);
    getPreviousDate(29, 2, 2000);
    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++;
}
```

Value Present Cases:

- E1: Value present once
- E2: Value present multiple times
- E3: Value not present

Array Edge Cases:

- E4: Empty array
- E5: Value at first/last position

Test Case	Input	Expected Output	Class	Reasoning
TC1	v=4, [1,2,4,5]	2	E1	Single occurrence
TC2	v=3, [3,2,3,3]	0	E2	Multiple occurrences
TC3	v=6, [1,2,3,4]	-1	E3	Value not present
TC4	v=1, []	-1	E4	Empty array
TC5	v=5, [1,2,3,4,5]	4	E5	Value at last position

Boundary Value Analysis:

Test Case	Input	Expected Output	Boundary Condition
BP1	v=1, [1]	0	Single element array, value present
BP2	v=2, [1]	-1	Single element array, value absent
BP3	v=1, [1,2,3]	0	Value at first position
BP4	v=3, [1,2,3]	2	Value at last position
BP5	v=-1, [-2,-1,0]	1	Negative numbers

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 Classes:

- E1: Item appears once
- E2: Item appears multiple times
- E3: Item not present
- E4: Empty array
- E5: Item at boundaries

Test Case	Input	Expected Output	Class	Reasoning
TC1	v=4, [1,4,2,3]	1	E1	Single occurrence
TC2	v=2, [2,3,2,2]	3	E2	Multiple occurrences
TC3	v=5, [1,2,3,4]	0	E3	Value not present
TC4	v=1, []	0	E4	Empty array
TC5	v=3, [3,3,3]	3	E5	All elements are target

Boundary Value Analysis:

Test Case	Input	Expected Output	Boundary Condition
BP1	v=3, [3]	1	Single element, value present
BP2	v=2, [1]	0	Single element, value absent
BP3	v=1, [1,1,1]	3	All elements same as target
BP4	v=5, [1,2,3,4,5]	1	Value at last position
BP5	v=-3, [-3,-3,-2,-1]	2	Negative numbers

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;
```

Equivalence Classes:

- E1: Value at beginning
- E2: Value in middle
- E3: Value at end
- E4: Value not present
- E5: Empty array

Test Case	Input	Expected Output	Class	Reasoning
TC1	v=1, [1,2,3,4,5]	0	E1	Value at start
TC2	v=3, [1,2,3,4,5]	2	E2	Value in middle
TC3	v=5, [1,2,3,4,5]	4	E3	Value at end
TC4	v=6, [1,2,3,4,5]	-1	E4	Value not present
TC5	v=1, []	-1	E5	Empty array

Boundary Value Analysis:

Test Case	Input	Expected Output	Boundary Condition
BP1	v=1, [1]	0	Single element, value present
BP2	v=2, [1]	-1	Single element, value absent
BP3	v=1, [1,2,3]	0	Value at first position
BP4	v=3, [1,2,3]	2	Value at last position
BP5	v=2, [1,2,2,2,3]	1	Multiple occurrences

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 Classes:

- E1: Equilateral triangle
- E2: Isosceles triangle
- E3: Scalene triangle
- E4: Invalid triangle
- E5: Zero/negative sides

Test Case	Input	Expected Output	Class	Reasoning
TC1	5, 5, 5	EQUILATERAL	E1	All sides equal
TC2	5, 5, 3	ISOSCELES	E2	Two sides equal
TC3	3, 4, 5	SCALENE	E3	No sides equal
TC4	1, 1, 3	INVALID	E4	Sum rule violated
TC5	-1, 2, 2	INVALID	E5	Negative side

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 Classes:

- E1: s1 is prefix of s2
- E2: s1 equals s2
- E3: s1 longer than s2
- E4: s1 not a prefix
- E5: Empty string cases

Test Case	Input	Expected Output	Class	Reasoning
TC1	“pre”, “prefix”	true	E1	Valid prefix
TC2	“test”, “test”	true	E2	Equal strings
TC3	“testing”, “test”	false	E3	s1 too long
TC4	“abc”, “def”	false	E4	Not a prefix
TC5	““,”test”	true	E5	Empty string prefix

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 Triangles:
   - E1: Equilateral (all sides equal)
   - E2: Isosceles (exactly two sides equal)
   - E3: Scalene (no sides equal)
   - E4: Right-angled (follows Pythagorean theorem)

2. Invalid Triangles:

   – I1: Sum of two sides ≤ third side

   – I2: One or more sides ≤ 0

b) Test Cases for Equivalence Classes:

Test Case	Input (A,B,C)	Expected O/P	Class	Description
TC1	5.0, 5.0, 5.0	Equilateral	E1	All sides equal
TC2	5.0, 5.0, 3.0	Isosceles	E2	Two sides equal
TC3	3.0, 4.0, 5.0	Right-angled	E4	Pythagorean triple
TC4	4.0, 5.0, 6.0	Scalene	E3	All sides different
TC5	1.0, 1.0, 3.0	Invalid	I1	Violates triangle inequality
TC6	-1.0, 2.0, 2.0	Invalid	I2	Negative side

c) Boundary Test Cases for A + B > C:

Test Case	Input (A,B,C)	Expected Output	Description
BC1	4.0, 5.0, 8.9	Scalene	Just valid
BC2	4.0, 5.0, 9.0	Invalid	Exactly A + B = C
BC3	4.0, 5.0, 9.1	Invalid	Just invalid

d) Boundary Test Cases for A = C (Isosceles):

Test Case	Input (A,B,C)	Expected Output	Description
BC4	5.0, 3.0, 5.0	Isosceles	Perfect isosceles
BC5	5.0, 3.0, 5.001	Scalene	Just beyond isosceles
BC6	5.0, 3.0, 4.999	Scalene	Just below isosceles

e) Boundary Test Cases for A = B = C (Equilateral):

Test Case	Input (A,B,C)	Expected Output	Description
BC7	5.0, 5.0, 5.0	Equilateral	Perfect equilateral
BC8	5.0, 5.0, 5.001	Isosceles	Just beyond equilateral
BC9	5.0, 5.001, 5.0	Isosceles	Slightly unequal sides

f) Boundary Test Cases for Right-angled Triangle:

Test Case	Input (A,B,C)	Expected Output	Description
BC10	3.0, 4.0, 5.0	Right-angled	Perfect right angle
BC11	5.0, 12.0, 13.0	Right-angled	Larger right angle
BC12	3.0, 4.0, 5.001	Scalene	Just beyond right angle

g) Non-triangle Test Cases:

Test Case	Input (A,B,C)	Expected Output	Description
BC13	1.0, 1.0, 2.0	Invalid	Equal to sum
BC14	1.0, 1.0, 2.001	Invalid	Greater than sum
BC15	1.0, 1.0, 1.999	Valid	Less than sum

h) Non-positive Input Test Cases:

Test Case	Input (A,B,C)	Expected Output	Description
BC16	0.0, 1.0, 1.0	Invalid	Zero side
BC17	-1.0, 1.0, 1.0	Invalid	Negative side
BC18	1.0, -0.001, 1.0	Invalid	Small negative value