Software Engineering (IT - 314)

LAB 8 - Black Box Testing

Utsav Kanani - 202201106

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?

1. Year < 1900 (Invalid)
2. 1900<=Year<=2015 (Valid)
3. Year > 2015 (Invalid)
4. Month < 1 (Invalid)
5. 1<=Month<=12 (Valid)
6. Month > 12 (Invalid)
7. Day < 1 (Invalid)
8. 1 <= Day <= 31 (Valid)
9. Day > 31 (Invalid)

Testcase no.	Day	Month	Year	Expected Outcome	Remark
1	15	5	2000	Valid (14,5,2000)	
2	31	12	2015	Valid (30,12,2015)	
3	15	5	1899	Invalid	Year < 1900
4	15	5	2016	Invalid	Year > 2015
5	15	0	2015	Invalid	Month < 1
6	15	13	2015	Invalid	Month > 12
7	30	2	2015	Invalid	Invalid date (February 30)
8	31	4	2015	Invalid	Invalid date (April 31)
9	0	2	2015	Invalid	Day < 1
10	31	4	2015	Invalid	Invalid day for April
11	0	1	2015	Invalid	Day < 1
12	15	12	2016	Invalid	Year > 2015
13	1	13	2015	Invalid	Month > 12

In Boundary Value Analysis, you focus on values at the edges of your partitions:

● Month boundaries:

Test with months like 1 and 12 (valid) and just outside, like 0 and 13 (invalid).

● Day boundaries:

Test with days at the beginning (1) and end (31) and just beyond these values, like 0 and 32.

● Year boundaries:

Test with years 1900 and 2015 (valid), and years just below and above this range.

Testcase no.	Day	Month	Year	Expected Outcome	Remark
1	1	1	1900	Valid (31,12,1899)	
2	31	12	2015	Valid (30,12,2015)	
3	0	1	1900	Invalid	Day < 1
4	32	1	1900	Invalid	Day > 31
5	1	0	2015	Invalid	Month < 1
6	1	13	2015	Invalid	Month > 12
7	0	12	2015	Invalid	Day < 1
8	31	0	2015	Invalid	Month < 1
9	0	0	2015	Invalid	Day < 1 and Month < 1
10	1	1900	1899	Invalid	Year < 1900
11	1	1	2016	Invalid	Year > 2015
12	31	12	2016	Invalid	Year > 2015

● 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.

```
#include <iostream>
#include <string>

using namespace std;

string getPreviousDate(int day, int month, int year) {

    if (year < 1900 || year > 2015) return "Invalid";
    if (month < 1 || month > 12) return "Invalid";
    if (day < 1 || day > 31) return "Invalid";

    if (day > 1) {
        return to_string(day - 1) + "," + to_string(month) + "," +
to_string(year);
    } else {
        if (month > 1) {
            return to_string(31) + "," + to_string(month - 1) + "," +
to_string(year);
        } else {
            return to_string(31) + ",12," + to_string(year - 1);
        }
    }
}
```

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 Class:

- Array contains the value
- Array does not contain the value

Equivalence Partitioning:

Input data	Description	Expected Outcome
v = 3, a = {1, 2, 3, 4, 5}	v is present in array a[]	2 (valid)
v = 6, a = {1, 2, 3, 4, 5}	v is not present in array a[]	-1 (valid)
v = 1, a = {}	Array a[] is empty	-1 (valid)
v = 2, a = {1, 2, 2, 3, 4}	v is present multiple times in a[]	1 (valid)

v = 5, a = null	a[] is null	Error
v = 3, a = {1, 'a', 3, 4}	Array a[] contains non-integer values	Error
v = 'b', a = {1, 2, 3, 4,5}	v is a non-integer value	Error

Boundary Value Analysis:

Input data	Description	Expected Outcome
v = 5, a = {5}	Array has only one element and v is present	0
v = 3, a = {5}	Array has only one element and v is not present	-1
v = 10, a = {1, 2, 3, 4, 5}	v is not in the array but outside the range	-1
v = 5, a = {}	Empty array	-1

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 Class:

- Array contains the value multiple times.
- Array does not contain the value.

Equivalence Partitioning:

Input data	Description	Expected Outcome
v = 3, a = {1, 2, 3, 4, 5}	v is present 1 time in array a[]	1 (valid)
v = 6, a = {1, 2, 3, 4, 5}	v is not present in array a[]	0 (valid)
v = 1, a = {}	Array a[] is empty	0 (valid)
v = 2, a = {1, 2, 2, 3, 4}	v is present 2 times in a[]	2 (valid)
v = 5, a = null	a[] is null	Error
v = 3, a = {1, 'a', 3, 4}	Array a[] contains non-integer values	Error
v = 'b', a = {1, 2, 3, 4,5}	v is a non-integer value	Error

Boundary Value Analysis:

Input data	Description	Expected Outcome
v = 5, a = {5,5,5,5,5}	Array has all element same and v is present	5
v = 3, a = {5}	Array has only one element and v is not present	0
v = 3, a = {6,6,3,7,3,5,3}	v is present in the array at different places	3
v = 5, a = {}	Empty array	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.

```
#include <iostream>

int binarySearch(int v, int a[], int size) {
    int lo = 0;
    int hi = size - 1;

    while (lo <= hi) {
        int mid = (lo + hi) / 2;
        if (v == a[mid]) {
            return mid; // Found the value, return index
        } 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
}
```

Equivalence Class:

- Array is sorted
- Array is not sorted
- Array contains the value
- Array does not contains the value

Equivalence Partitioning:

Input data	Description	Expected Outcome
v = 3, a = {1, 2, 3, 4, 5}	v is present in array a[]	2 (valid)
v = 6, a = {1, 2, 3, 4, 5}	v is not present in array a[]	-1 (valid)
v = 1, a = {}	Array a[] is empty	-1 (valid)
v = 2, a = {1, 2, 2, 3, 4}	v is present 2 times in a[]	2 (valid) (can change according to array)
v = 5, a = null	a[] is null	Error
v = 3, a = {1, 'a', 3, 4}	Array a[] contains non-integer values	Error
v = 'b', a = {1, 2, 3, 4,5}	v is a non-integer value	Error
v = 2, a = {4,5, 3, 1}	Unsorted array	Inconsistent binary search
v = 3, a = {1, 'a', 3, 4}	Array a[] contains non-integer values	Error

v = 'b', a = {1, 2, 3, 4,5}	v is a non-integer value	Error

Boundary Value Analysis:

Input data	Description	Expected Outcome
v = 5, a = {5,5,5,5,5}	Array has all element same and v is present	(a.length-1)/2
v = 3, a = {5}	Array has only one element and v is not present	-1
v = 5, a = {5}	Array has only one element and v is not present	0
v = 2, a = {4,5, 3, 1}	Unsorted array	Inconsistent binary search
v = 5, a = {}	Empty array	-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).

```
#include <iostream>

const int EQUILATERAL = 0;
const int ISOSCELES = 1;
```

```
const int SCALENE = 2;
const int INVALID = 3;

int triangle(int a, int b, int c) {
    // Check for invalid triangle condition
    if (a >= b + c || b >= a + c || c >= a + b) {
        return INVALID;
    }
    // Check for equilateral triangle
    if (a == b && b == c) {
        return EQUILATERAL;
    }
    // Check for isosceles triangle
    if (a == b || a == c || b == c) {
        return ISOSCELES;
    }
    // If none of the above, it is scalene
    return SCALENE;
}
```

Equivalence Class:

- Triangle is Equilateral.
- Triangle is Isosceles.
- Triangle is Scalene.
- Triangle is Invalid.

Equivalence Partitioning:

Input data	Description	Expected Outcome
a = 3, b = 3, c = 3	All sides are equal (equilateral triangle)	0
a = 3, b = 4, c = 3	Two sides are equal (Isosceles triangle)	1
a = 3, b = 4, c = 5	No sides are equal (Scalene	2

	triangle)	
a = 3, b = 3, c = 6	Invalid Triangle	3
a = 0, b = 0, c = 0	Invalid Triangle	3
a = 0, b = 5, c = 0	Invalid Triangle	3
a = 0, b = 7, c =6	Invalid Triangle	3
a = 5, b = -7, c =6	Invalid Triangle	3
a = ‘a’, b = 3, c = 3	Side a is not an integer	Error

Boundary Value Analysis:

Input data	Description	Expected Outcome
a = 1, b = 1, c = 1	Minimum Valid Equilateral	0
a = 1, b = 2, c = 2	Minimum Valid Isosceles	1
a = 2, b = 3, c = 4	Minimum Valid Scalene	2
a = 3, b = 3, c = 6	Invalid Triangle	3
a = 0, b = 7, c =6	One side is zero	3
a = 0, b = 5, c = 0	Two side are zero	3
a = 1, b = 2, c =3	Impossible outcome	3

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 Class:

- S1 is longer than S2
- S1 is valid prefix of S2
- S1 is not a valid prefix of S2

Equivalence Partitioning:

Input data	Description	Expected Outcome
s1 = "pre", s2 = "prefix"	s1 is a valid prefix of s2	TRUE
s1 = "preT", s2 = "prefix"	s1 is not a prefix of s2	TRUE
s1 = "", s2 = "prefix"	s1 is a empty string	TRUE
s1 = "prE", s2 = "prefix"	s1 is a not valid prefix of s2 (Case sensitive check()	FALSE

s1 = "prefixmain", s2 = "prefix"	s1 is longer than s2 prefix impossible	FALSE

Boundary Value Analysis:

Input data	Description	Expected Outcome
s1 = "p", s2 = "prefix"	s1 is a min positive length	TRUE
s1 = "prefix", s2 = "prefix"	s1 is a max positive length	TRUE
s1 = "prefixx", s2 = "prefix"	s1 exceeds s2 length by 1 till then it completely matches s2	FALSE
s1 = "", s2 =”"	s1 and s2 both are empty string	TRUE

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:

1. Valid Equivalence Classes
2. Equilateral Triangle
    a. All sides are equal.
3. Isosceles Triangle

    a. Exactly two sides are equal.
4. Scalene Triangle
    a. All sides are different.
5. Right-Angled Triangle
    a. Satisfies the Pythagorean theorem ($A^2 + B^2 = C^2$).
6. Invalid Equivalence Classes
7. Non-Triangle
    a. The sum of the lengths of any two sides is not greater than the third side.
8. Negative Values
    a. One or more sides have negative lengths.
9. Zero Values
    a. One or more sides are zero.

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)

Side Lengths	Description	Triangle Class
2.0, 2.0, 2.0	All sides are equal	Equilateral Triangle
3.0, 3.0, 4.0	Two sides are equal	Isosceles Triangle
7.0, 9.0, 10.0	All sides are different	Scalene Triangle
3.0, 4.0, 5.0	Satisfies the Pythagorean theorem	Right-Angled Triangle
1.0, 2.0, 3.0	The sum of the two shorter sides is equal to the longest	Non-Triangle
-1.0, 2.0, 3.0	One side is negative	Negative Values
0.0, 1.0, 1.0	One side is zero	Zero Values

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

Input Data	Description	Covered Class
2.0, 3.0, 4.0	Valid scalene triangle	Scalene Triangle
2.0, 2.5, 5.0	Just below boundary	Non-Triangle
1.0, 1.0, 2.0	Exactly on the boundary	Non-Triangle

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

Input Data	Description	Covered Class
5.0, 5.0, 3.0	Two sides equal, valid isosceles	Isosceles Triangle
3.0, 3.0, 6.0	Just below boundary	Non-Triangle
2.0, 2.0, 4.0	Exactly on the boundary	Non-Triangle

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

Input Data	Description	Covered Class
3.0, 3.0, 3.0	All sides equal, valid equilateral	Equilateral Triangle
2.0, 2.0, 2.0	Valid equilateral	Equilateral Triangle
2.0, 2.0, 3.0	Just isosceles	Isosceles Triangle

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

Input Data	Description	Covered Class
3.0, 4.0, 5.0	Valid right-angled triangle	Right-Angled Triangle
5.0, 12.0, 13.0	Valid right-angled triangle	Right-Angled Triangle

2.0, 2.0, 3.0	Not a right-angled triangle	Not Right-Angled

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

Input Data	Description	Covered Class
1.0, 2.0, 3.0	Sum of two sides equals the third	Non-Triangle
2.0, 5.0, 3.0	Valid triangle	Scalene
10.0, 1.0, 1.0	Impossible lengths	Non-Triangle

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

Input Data	Description	Covered Class
0.0, 0.0, 0.0	All sides are zero	Non-Triangle
-1.0, 2.0, 3.0	One side is negative	Non-Triangle
2.0, 0.0, 2.0	One side is zero	Non-Triangle