Software Engineering

Lab 8

Lab Session - Functional Testing (Black-Box)

Name: Bansi Patel

ID : 202201190

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.

Ans:

Equivalence class:

E1: month range 1<=month<=12 [valid]

E2: month range month<=0 [invalid]

E3: month range month>12 [invalid]

E4: day range 1<=day<=31 [valid]

E5: day range day<=0 [invalid]

E6: day range day>31 [invalid]

E7: year range 1900<=year<=2015 [valid]

E8: year range year<1900 [invalid]

E9: year range year>2015 [invalid]

EQUIVALENCE TEST – CASES:

NO	DAY	MONTH	YEAR	CLASS – COV.	EXPECTED OUTPUT
1	1	12	2015	E1, E4, E7	30-11-2015
2	0	12	2015	E5, E1, E7	INVALID
3	1	13	2015	E2, E4, E7	INVALID
4	1	12	2016	E1, E4, E9	INVALID
5	32	12	2015	E6, E1, E7	INVALID
6	1	0	2015	E2, E4, E7	INVALID
7	1	12	1899	E1, E4, E8	INVALID

BOUNDARY TEST – CASES:

NO	DAY	MONTH	YEAR	EXPECTED OUTPUT
1	1	1	1900	NO PREV DATE
2	2	1	1900	1-1-1900
3	31	12	2015	30-12-15
4	1	3	2016	INVALID
5	29	2	2012	28-2-2012
6	30	4	2010	29-4-2010
7	0	3	1899	INVALID

Program Execution and Outcome Verification

```
def is_leap_year(year):
    return (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0)


def previous_date(day, month, year):
    # Check for valid year, month, day
    if year < 1900 or year > 2015 or month < 1 or month > 12 or day < 1:
        return "Invalid date"


    # Days in each month (assuming non-leap year)
    days_in_month = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
    # Adjust February for leap years
    if is_leap_year(year):
        days_in_month[1] = 29

    # Check for invalid day for the month
    if day > days_in_month[month - 1]:
        return "Invalid date"

    # Handle New Year transition
    if day == 1 and month == 1:
        return (31, 12, year - 1)  # Previous date is December 31 of the previous year

    # Handle end of month cases
    if day == 1:
        month -= 1
        if month == 0:
            month = 12
            year -= 1
        day = days_in_month[month - 1]
    else:
        day -= 1

    return (day, month, year)

# Test the function with test cases
test_cases = [
    (1, 1, 2000),    # Expected: (31, 12, 1999)
    (1, 3, 2015),    # Expected: (28, 2, 2015)
    (1, 2, 2000),    # Expected: (31, 1, 2000)
    (1, 2, 2001),    # Expected: (31, 1, 2001)
    (31, 12, 2015),  # Expected: (30, 12, 2015)
    (29, 2, 2000),   # Expected: (28, 2, 2000)
    (30, 4, 2015),   # Expected: (29, 4, 2015)
    (31, 4, 2015),   # Expected: Invalid date
    (1, 13, 2015),   # Expected: Invalid date
    (1, 2, 1899),    # Expected: Invalid date
    (32, 1, 2000),   # Expected: Invalid date
    (29, 2, 2015),   # Expected: (28, 2, 2015)
    (0, 1, 2015),    # Expected: Invalid date
    (1, 1, 2016)     # Expected: Invalid date
]


# Execute test cases
for day, month, year in test_cases:
    print(f"Input: {day}, {month}, {year} => Output: {previous_date(day, month, year)}")
```

P1. The function linear Search 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);
}
```

E1: v present

E2: v not present

E3: v present at index 0

E4: v present at index n-1

Array: [1,3,-3,2,3,6,8]

V value	Expected output
1	0
90	-1
-2	2
8	6

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);

}
```

E1: count of v = 0

E2: count of v = array.size()

E3: count of v < array.size()

Array and V	Expected output
{1,2,3,4,5} 90	0
{1,1,1,1,1} 1	5
{1,1,2,3,4} 1	2

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;
    }
return(-1);
}
```

E1: v is present in the arrray

E2: v is not present in the array

E3: v is present at index 0

E4: v is present at last index of the array

E5: v is present middle of the array

Array: {1,2,3,4,5}

Array and V	Expected output
2	1
90	-1
1	0
5	4
3	2

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 length sequal), 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);
}
```

E1: Equilateral Triangle when a==b==c

E2: Isosceles Triangle when a==b || a==c || b==c

E3: Scalene Triangle when a NE b NE c

E4: Invalid Triangle when sum of two side <= the third side

E5: Non-positive Lengths

Input[a,b,c]	Expected output
2,2,2	Equilateral
1,2,1	Isosceles
3,2,4	Scalene
1,2,3	Invalid
-1,3,2	Invalid

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;
}
```

E1: s1 is a prefix of s2.

E2: s1 is not a prefix of s2.

E3: s1 is longer than s2.

Input[a,b,c]	Expected output
S1:“abc” S2:”abcdjdlfs”	true
S1:”qwe” S2:”sjewoe”	false
S1:”rtyukjhgfdf” S3:”rty”	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

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.

E1: Equilateral: All sides are equal (A == B == C). [Valid]

E2: Isosceles: Two sides are equal (A == B || A == C || B == C). [Valid]

E3: Scalene: All sides are different ($A \neq B \neq C$). [Valid]

E4: Right-angled: Pythagorean theorem is satisfied ($A^2 + B^2 = C^2$). [Valid]

E5: The sum of any two sides is less than or equal to the third side ($A + B <= C$). [Invalid]

E6: One or more sides have zero or negative lengths. [Invalid]

Input[a,b,c]	Expected output
3,3,3	E1: VALID
4,4,5	E2: VALID
3,4,5	E4: VALID
5,7,10	E3: VALID
1,1,3	E5: INVALID
1,2,3	E6: INVALID

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

A=2, B=3, C=4.99 [VALID]

A=2, B=3, C=5 [BOUNDARY CASE]

A=2, B=3, C=5.001 [INVALID]

Boundary condition for Isosceles Triangle (A=C):

A=4, B=5, C=4.0001 [VALID]

A=4, B=5, C=4 [BOUNDARY CASE]

Boundary condition for Equilateral Triangle (A=B=C):

A=3, B=3, C=3.0001 [VALID]

A=3, B=3, C=3 [BOUNDARY CASE]

Boundary condition for Right angle Triangle (A^2 + B^2 = C^2):

A=3, B=4, C=5 [VALID]

A=1, B=1, C=1.414 [BOUNDARY]

Non valid Triangle:

A=1, B=2, C=3 [INVALID]

Non positive input:

A=-3, B=2, C=4 [NEGATIVE SIDE]

A=0, B=2, C=4 [NOT POSSIBLE]