IT - 314

SOFTWARE ENGINEERING

Lab Assignment: 8

Student Name : Jay Goyani

Student ID : 202201163

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:

Equivalence class partitioning :

Equivalence Class	Description	Valid / Invalid
1	month < 1	Invalid
2	1 <= month <= 12	Valid
3	month > 12	Invalid
4	day < 1	Invalid
5	1 ≤ day ≤ 31	Valid
6	Day > 31	Invalid

7	year < 1900	Invalid
8	1900 ≤ year ≤ 2015	Valid
9	year > 2015	Invalid

Boundary Value Analysis:

No.	Test Data	Expected Outcome	Classes Covered
1	01/01/1900	Valid	2, 5, 8
2	31/12/1900	Valid	2, 5, 8
3	01/01/2015	Valid	2, 5, 8
4	31/12/2015	Valid	2, 5, 8
5	00/01/2015	Invalid	4
6	01/00/2015	Invalid	1
7	32/12/2015	Invalid	6
8	01/13/2015	Invalid	3
9	31/12/1899	Invalid	7
10	31/12/2016	Invalid	9

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.

```
#include <iostream>
```

```
using namespace std;

string prev_date(int day, int month, int year) {
    if (month < 1 || month > 12 || year < 1900 || year > 2015 || day
< 1 || day > 31) {
        return "Invalid";
    }
    return "Valid";
}
```

Q2. 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 Description:

- E1: The array is empty.
- E2: The value v is present in the array.
- E3: The value v is not present in the array.
- E4: The array contains only one element which is equal to v.
- E5: The array contains more than one element which is equal to v.

Equivalence Partitioning Test cases:

Test Case	Input Data (Array a, Value v)	Expected Outcome	Covered Equivalence Class
TC1	a = [ ], v = 5	-1	E1
TC2	a = [1,2,3,4,5], v = 5	4	E2
TC3	a = [1,2,3,4,6], v = 5	-1	E3
TC4	a = [5], v = 5	0	E4
TC5	a = [4, 5, 5], v = 5	1	E5

Boundary Value Test Cases:

Test Case	Input Data (Array a, Value v)	Expected Outcome	Boundary Condition
TC1	a = [5, 5], v = 5	0	Element appears more than 1 time, Return the first index of the element
TC2	a = [5], v = 6	-1	Single element array, value absent

TC3	a = [1,2,3,4,5], v = 1	0	Value is at the start of the array
TC4	a = [1,2,3,4,5], v = 5	4	Value is at the end of the array
TC5	a = [1,2,3,4,5], v = 6	-1	Value absent but close to elements in the array

Modified Programm:

```
#include <iostream>

using namespace std;

int searchValue(int target, int array[], int size)

{

    for (int i = 0; i < size; i++)

    {

        if (array[i] == target)

            return i;

    }

    return -1;

}

int main()

{
```

```
    int numbers1[] = {10, 20, 30, 40, 50};

    int numbers2[] = {};

    int numbers3[] = {-10, -20, -30};

    cout << "Test 1 (target=30): " << searchValue(30,
numbers1, 5) << endl;   // Output: 2

    cout << "Test 2 (target=60): " << searchValue(60,
numbers1, 5) << endl;   // Output: -1

    cout << "Test 3 (Empty array): " << searchValue(30,
numbers2, 0) << endl; // Output: -1

    cout << "Test 4 (Negative numbers, target=-20): " <<
searchValue(-20, numbers3, 3)

         << endl;
// Output: 1

    cout << "Test 5 (Single element, target=10): " <<
searchValue(10, numbers1, 1) << endl; // Output: 0

    cout << "Test 6 (target=10, First element): " <<
searchValue(10, numbers1, 5) << endl;  // Output: 0

    cout << "Test 7 (target=50, Last element): " <<
searchValue(50, numbers1, 5) << endl;   // Output: 4

    cout << "Test 8 (Empty array): " << searchValue(20,
numbers2, 0) << endl;                // Output: -1

    cout << "Test 9 (target=60, Not found): " <<
searchValue(60, numbers1, 5) << endl;

    // Output: -1

    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);
}
```

P2. The function countItem returns the number of times a value v appears in an array of integers a.

Equivalence Class Description:

1. Array contains multiple occurrences of the value:
2. Array does not contain the value:
3. Empty array:
4. Single element array (element is the value):
5. Single element array (element is not the value):

Test-Case

Test-Case	Expected Outcome	Class
v=1 && a[3]=[1,2,1]	2	1
v=1 && a[3]=[2,3,4]	0	2
v=1 && a=[]	0	3
v=2 && a[1]=[2]	1	4
v=2 && a[1]=[3]	0	5

Boundary Value Analysis :

Test-Case	Expected Outcomes
v=1 && a[3]=[1,2,3]	1
v=1 && a[3]=[2,3,1]	1
v=1 && a=[]	0
v=2 && a[1]=[2]	1
v=2 && a[1]=[3]	0

Modified Programm:

```
#include <iostream>
```

```cpp
using namespace std;

int countItem(int target, int array[], int size)

{

    int count = 0;

    for (int i = 0; i < size; i++)

    {

        if (array[i] == target)

            count++;

    }

    return count;

}

int main()

{

    int a1[] = {1, 2, 1, 4, 1};

    int a2[] = {};

    int a3[] = {-1, -2, -1};

    int a4[] = {2};

    int a5[] = {1};

    cout << "Test 1 (v=1): " << countItem(1, a1, 5) << endl;                    //
output: 3

    cout << "Test 2 (v=6): " << countItem(6, a1, 5) << endl;                     //
output: 0
```

```
    cout << "Test 3 (Empty array): " << countItem(3, a2, 0) << endl;            // output: 0

    cout << "Test 4 (Negative numbers): " << countItem(-1, a3, 3) << endl;
// output: 2

    cout << "Test 5 (Single element): " << countItem(2, a4, 1) << endl;         // output: 1

    cout << "Test 6 (Single element not found): " << countItem(2, a5, 1) <<
endl; // output: 0

    cout << "Test 7 (v=1, First element): " << countItem(1, a1, 5) << endl;     // output: 3

    cout << "Test 8 (v=3, Last element): " << countItem(3, a1, 5) << endl;      // output: 0

    cout << "Test 9 (Empty array): " << countItem(2, a2, 0) << endl;            // output: 0

    cout << "Test 10 (v=4, Not found): " << countItem(4, a1, 5) << endl;        // output: 0

    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 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);
}
```

Equivalence Partitioning (EP):

Equivalence Class Description:

- E1: The array is empty.
- E2: The value v is present in the array.
- E3: The value v is not present in the array.

- E4: The array contains only one element which is equal to v.
- E5: The array contains only one element which is not equal to v.

Equivalence Class Test Cases:

Test Case	Input Data (Array a, Value v)	Expected Outcome	Covered Equivalence Class
TC1	a = [ ], v = 5	-1	E1
TC2	a = [1,2,3,4,5], v = 5	4	E2
TC3	a = [1,2,3,4,6], v = 5	-1	E3
TC4	a = [5], v = 5	0	E4
TC5	a = [4], v = 5	-1	E5

Boundary Value Analysis (BVA):

Boundary Conditions:

- The array size is at its minimum size, either empty or contains just one element.
- The value is at the start, middle, or end of the array.
- The value is not present but close to elements in the array.

Boundary Value Test Cases

Test Case	Input Data (Array a, Value v)	Expected Outcome	Boundary Condition
TC1	a = [5], v = 5	0	Single element array, value present

TC2	a = [5], v = 6	-1	Single element array, value absent
TC3	a = [1,2,3,4,5], v = 1	0	Value is at the start of the array
TC4	a = [1,2,3,4,5], v = 3	2	Value is in the middle of the array
TC5	a = [1,2,3,4,5], v = 5	4	Value is at the end of the array
TC6	a = [1,2,3,4,5], v = 6	-1	Value absent but close to elements in the array

Modified Programm:

```
#include <iostream>
#include <vector>
using namespace std;
int binarySearch(const vector<int> &a, int v)
{
   int left = 0;
   int right = a.size() - 1;
   while (left <= right)
   {
      int mid = left + (right - left) / 2;
```

```
        if (a[mid] == v)
        {
            return mid; // Value found at index mid
        }
        else if (a[mid] < v)
        {
            left = mid + 1; // Search in the right half
        }
        else
        {
            right = mid - 1; // Search in the left half
        }
    }
    return -1; // Value not found
}
int main()
{
    // Test cases
    vector<int> arr1 = {};                  // Empty array
    vector<int> arr2 = {1, 2, 3, 5, 6};     // Value is present
    vector<int> arr3 = {1, 2, 3, 4, 6};     // Value is not present
```

```
    vector<int> arr4 = {5};                    // Single element, value present
    vector<int> arr5 = {3};                    // Single element, value not present

    cout << "TC1: " << binarySearch(arr1, 5) << endl; // output -1
    cout << "TC2: " << binarySearch(arr2, 5) << endl; // output 3
    cout << "TC3: " << binarySearch(arr3, 5) << endl; // output -1
    cout << "TC4: " << binarySearch(arr4, 5) << endl; // output 0
    cout << "TC5: " << binarySearch(arr5, 5) << endl; // output -1
    return 0;
}
```

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 , isosceles (two lengths equal), scalene , or invalid.

```
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 Test cases:

No.	Test Description	Input (a,b,c)	Output
1.	Equilateral triangle (all sides equal)	(3, 3, 3)	EQUILATERAL
2.	Isosceles triangle (two sides equal)	(3, 3, 4)	ISOSCELES
3.	Scalene triangle (no sides equal)	(3, 4, 5)	SCALENE
4.	Invalid triangle (sum of two sides ≤ third)	(1, 2, 3)	INVALID
5.	Equilateral triangle with larger values	(1000, 1000, 1000)	EQUILATERAL
6.	(-1, 3, 4)	(-1, 3, 4)	INVALID

Boundary Value Analysis Test cases:

No.	Test Description	Input(a,b,c)	Output

1.	Invalid triangle (one side zero)	(0, 3, 4)	INVALID
2.	Isosceles triangle at boundary value	(2, 2, 1)	ISOSCELES
3.	Scalene triangle at boundary value	{2,3,4}	SCALENE
4.	Invalid triangle at boundary (sides sum = third)	(5, 5, 10)	INVALID

Modified Programm:

```
#include <iostream>

using namespace std;

const char *triangle(int a, int b, int c)

{

   if (a <= 0 || b <= 0 || c <= 0 || a + b <= c || a + c <= b || b + c <= a)

   {

      return "Invalid";

   }

   if (a == b && b == c)

   {

      return "Equilateral";

   }

   if (a == b || b == c || a == c)

   {
```

```
        return "Isosceles";
    }
    return "Scalene";
}
int main()
{
    cout << "Test 1: " << triangle(3, 3, 3) << endl;  // Output: Equilateral
    cout << "Test 2: " << triangle(4, 4, 5) << endl;  // Output: Isosceles
    cout << "Test 3: " << triangle(3, 4, 5) << endl;  // Output: Scalene
    cout << "Test 4: " << triangle(1, 2, 3) << endl;  // Output: Invalid
    cout << "Test 5: " << triangle(-1, 2, 3) << endl; // Output: Invalid
    cout << "Test 6: " << triangle(0, 2, 2) << endl;  // Output: Invalid
    cout << "Test 7: " << triangle(1, 1, 1) << endl;  // Output: Equilateral
    cout << "Test 8: " << triangle(1, 1, 2) << endl;  // Output: Invalid
    cout << "Test 9: " << triangle(-1, 1, 1) << endl; // Output: Invalid
    cout << "Test 10: " << triangle(0, 1, 1) << endl; // Output: Invalid
    cout << "Test 11: " << triangle(2, 2, 2) << endl; // Output: Equilateral
    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).

```
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):

Equivalence Class Description:

- E1: s1 is longer than s2 (impossible to be a prefix).
- E2: s1 is a valid prefix of s2.

- E3: s1 is not a prefix of s2.
- E4: s1 is an empty string (edge case).
- E5: s2 is an empty string (edge case).

Equivalence Class Test Cases:

Test Case	Input Data (s1, s2)	Expected Outcome	Covered Equivalence Class
TC1	"abcdef", "abc"	false	E1
TC2	"abc", "abcdef"	true	E2
TC3	"xyz", "abcdef"	false	E3
TC4	"", "abcdef"	true	E4
TC5	"abc", ""	false	E5

Boundary Value Analysis (BVA):

Boundary Conditions:

- Length of s1 is greater than the length of s2.
- s1 is an empty string, s2 is a non-empty string.
- s2 is an empty string, s1 is non-empty.
- s1 equals s2.

Boundary Value Test Cases:

Test Case	Input Data (s1, s2)	Expected Outcome	Boundary Condition
TC1	“a”,” ”	false	S2 is empty

TC2	"abcdef", "abcdef"	true	s1 equals s2
TC3	"abc", "abc"	true	Shorter but equal strings
TC4	"", ""	true	Both strings are empty

Modified Programme:

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

// Function to check if s1 is a prefix of s2
bool isPrefix(const string& prefixStr, const string& fullStr) {
    // If the prefix is longer than the full string, it can't be a prefix
    if (prefixStr.length() > fullStr.length()) {
        return false;
    }

    // Compare characters of both strings
    for (size_t i = 0; i < prefixStr.length(); i++) {
        if (prefixStr[i] != fullStr[i]) {
```

```
            return false; // Characters do not match
        }
    }
    return true; // All characters match; prefix is valid
}

int main() {
    // Equivalence Partitioning Test Cases
    cout << "Test Case 1: " << (isPrefix("abcdef", "abc") ? "true" : "false") << endl;
// Expected output: false
    cout << "Test Case 2: " << (isPrefix("abc", "abcdef") ? "true" : "false") << endl;
// Expected output: true
    cout << "Test Case 3: " << (isPrefix("xyz", "abcdef") ? "true" : "false") << endl;
// Expected output: false
    cout << "Test Case 4: " << (isPrefix("", "abcdef") ? "true" : "false") << endl;   //
Expected output: true
    cout << "Test Case 5: " << (isPrefix("abc", "") ? "true" : "false") << endl;      //
Expected output: false

    // Boundary Value Test Cases
    cout << "Test Case 6: " << (isPrefix("a", "") ? "true" : "false") << endl;        //
Expected output: false
```

```
    cout << "Test Case 7: " << (isPrefix("abcdef", "abcdef") ? "true" : "false") <<
endl; // Expected output: true

    cout << "Test Case 8: " << (isPrefix("abc", "abc") ? "true" : "false") << endl;
// Expected output: true

    cout << "Test Case 9: " << (isPrefix("", "") ? "true" : "false") << endl;       //
Expected output: true



    return 0; // End of the program

}
```

P6: Consider again the triangle classification program (P4) with a slightly different specification: Theprogram reads floating values from the standard input. The three values A, B, and C are interpretedas representing the lengths of the sides of a triangle. The program then prints a message to thestandard 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.

By Equivalence Class:

1. Valid equilateral triangle: All sides are equal.

2. Valid isosceles triangle: Exactly two sides are equal.

3. Valid scalene triangle: All sides are different.

4. Valid right-angled triangle: Follows the Pythagorean theorem.

5. Invalid triangle (non-triangle): Sides do not satisfy triangle inequalities.

6. Invalid input (non-positive values): One or more sides are non-positive.

Equivalence Partitioning Test cases:

No.	Input(a.b,c)	Output	Covered Equivalence class
1.	A = 3.0, B = 3.0, C =3.0	Equilateral	E1
2.	A = 4.0, B = 4.0, C = 5.0	Isosceles	E2
3.	A = 3.0, B = 4.0, C = 5.0	Scalene	E3
4.	A = 3.0, B = 4.0, C = 6.0	Invalid	E5
5.	A = -1.0, B = 2.0, C = 3.0	Invalid	E6
6.	A = 5.0, B = 12.0, C = 13.0	Right-angled	E4

Boundary Value Analysis Test cases:

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

No.	Input(a.b,c)	Output
1.	A = 1.0, B = 1.0, C = 1.9999	Scalene
2.	A = 2.0, B = 3.0, C = 4.0	Scalene

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

No.	Input(a.b,c)	Output
1.	A = 3.0, B = 3.0, C = 4.0	Isosceles
2.	A = 2.0, B = 2.0, C = 3.0	Isosceles
3.	A = 2.0, B = 2.0, C = 2.0	Equilateral

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

No.	Input(a.b,c)	Output

1.	A = 2.0, B = 2.0, C = 2.0	Equilateral
2.	A = 1.9999, B = 1.9999, C = 1.9999	Equilateral

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

No.	Input(a.b,c)	Output
1.	A = 3.0, B = 4.0, C = 5.0	Right-angled
2.	A = 5.0, B = 12.0, C = 13.0	Right-angled

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

No.	Input(a.b,c)	Output
1.	A = 1.0, B = 2.0, C = 3.0	Invalid
2.	A = 1.0, B = 2.0, C = 2.0	Invalid
3.	A = 1.0, B = 1.0, C = 3.0	Invalid

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

No.	Input(a.b,c)	Output
1.	A = 0.0, B = 2.0, C = 3.0	Invalid
2.	A = -1.0, B = -2.0, C = 3.0	Invalid
3.	A = 3.0, B = 0.0, C = 2.0	Invalid

Modified Programm:

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


// Function to classify the triangle based on its sides
const char* classifyTriangle(float sideA, float sideB, float sideC) {
    // Check for valid triangle conditions
    if (sideA <= 0 || sideB <= 0 || sideC <= 0 ||
        sideA + sideB <= sideC ||
        sideA + sideC <= sideB ||
        sideB + sideC <= sideA) {
        return "Invalid";
    }
```

```
    // Check for right-angled triangle using Pythagorean theorem
    if (fabs(pow(sideA, 2) + pow(sideB, 2) - pow(sideC, 2)) < 1e-6 ||
        fabs(pow(sideA, 2) + pow(sideC, 2) - pow(sideB, 2)) < 1e-6 ||
        fabs(pow(sideB, 2) + pow(sideC, 2) - pow(sideA, 2)) < 1e-6) {
        return "Right-angled";
    }

    // Check for equilateral triangle
    if (sideA == sideB && sideB == sideC) {
        return "Equilateral";
    }

    // Check for isosceles triangle
    if (sideA == sideB || sideB == sideC || sideA == sideC) {
        return "Isosceles";
    }

    // If no conditions match, it's scalene
    return "Scalene";
}
```

```cpp
int main() {

    // Test cases to validate triangle classification

    cout << "Test 1: " << classifyTriangle(3.0, 3.0, 3.0) << endl;    // Output: Equilateral

    cout << "Test 2: " << classifyTriangle(4.0, 4.0, 5.0) << endl;    // Output: Isosceles

    cout << "Test 3: " << classifyTriangle(3.0, 4.0, 5.0) << endl;    // Output: Scalene

    cout << "Test 4: " << classifyTriangle(3.0, 4.0, 6.0) << endl;    // Output: Invalid

    cout << "Test 5: " << classifyTriangle(-1.0, 2.0, 3.0) << endl;   // Output: Invalid

    cout << "Test 6: " << classifyTriangle(0.0, 2.0, 2.0) << endl;    // Output: Invalid

    cout << "Test 7: " << classifyTriangle(5.0, 12.0, 13.0) << endl; // Output: Right-angled



    return 0; // End of the program

}
```