

Lab8:

Functional Testing (Black-Box)

Jaimin Prajapati

23th October, 2024

Q.1. Consider a program for determining the previous date. Its input is triple of day, month and year with the following ranges $1 \leq \text{month} \leq 12$, $1 \leq \text{day} \leq 31$, $1900 \leq \text{year} \leq 2015$. 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.

The solution of each problem must be given in the format as follows:

Tester Action and Input Data	Expected Outcome
Equivalence Partitioning	
a, b, c	An Error message
a-1, b, c	Yes
Boundary Value Analysis	
a, b, c-1	Yes

Answer :

Making equivalence classes :

EC	Range	Description
EC1	DAY < 1	INVALID
EC2	DAY > 31	INVALID
EC3	$1 \leq \text{DAY} \leq 28$	Valid for february (no leap year)
EC4	$1 \leq \text{DAY} \leq 29$	Valid for february (leap year)
EC5	$1 \leq \text{DAY} \leq 30$	Valid for month with 30 day
EC6	$1 \leq \text{DAY} \leq 31$	Valid for month with 31 day

EC7	MONTH < 1	INVALID
EC8	MONTH > 12	INVALID
EC9	1 <= MONTH <= 12	VALID
EC10	DAY > 29 IN FEBRUARY	INVALID
EC11	DAY > 30 IN 30 DAY MONTH	INVALID
EC12	YEAR < 1900	INVALID
EC13	YEAR > 2015	INVALID
EC14	1900 <= YEAR <= 2015	VALID
EC15	YEAR % 4 == 0 AND [YEAR % 100 != 0 OR YEAR % 400 == 0]	LEAP YEAR

Generation equivalence classes test cases:

TEST CASES	DAY	MONTH	YEAR	EXPECTED OUTPUT	DESCRIPTION	EC
TC1	31	6	1984	ERROR	INVALID	6,9,11
TC2	29	2	2000	29/2/2000	VALID	4,9,14
TC3	8	2	2005	8/2/2005	VALID	3,9,14
TC4	21	10	2008	21/10/2004	VALID	6,9,14
TC5	12	13	1800	ERROR	INVALID	6,8,12
TC6	32	8	2005	ERROR	INVALID	2,9,14
TC7	10	24	2010	ERROR	INVALID	6,8,14
TC8	50	50	2050	ERROR	INVALID	2,8,13
TC9	12	12	2012	12/12/2012	VALID	6,9,14
TC10	50	1	2016	ERROR	INVALID	2,9,13

Boundary cases:

From the given ranges of the day, month and year we can find the following cases:

- For days 1, 28/29 (for february(with or without leap year) , 30 and 31 days months
- For month 1 and 12
- For year 1980 and 2015

TEST CASES	DAY	MONTH	YEAR	EXPECTED OUTPUT	DESCRIPTION	EC
TC11	1	8	2000	1/8/2000	Start of 31 days	6,9,14
TC12	31	10	2010	31/10/2010	End of 31 days	6,9,14
TC13	29	2	2008	29/2/2008	Leap year	3,9,14,15
TC14	29	2	2010	ERROR	Not leap year	6,9,14
TC15	1	3	1990	1/3/1990	Start of 30 days	6,9,14
TC16	30	8	2005	30/8/2005	End of 30 days	5,9,14
TC17	12	8	1900	12/8/1900	Start of year	6,8,14
TC18	13	5	2015	13/5/2015	End of year	6,9,14
TC19	31	31	2031	ERROR	INVALID	6,8,13

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);
}
```

Functional Test Cases Table:

Test Case ID	Method	Tester Action and Input Data	Expected Outcome
E1	Equivalence Partitioning	<code>linearSearch(1, [1, 2, 3, 4])</code>	0
E2	Equivalence Partitioning	<code>linearSearch(0, [1, 2, 3, 4])</code>	-1 (NOT FOUND)
E3	Equivalence Partitioning	<code>linearSearch(5, [])</code>	Error: array is empty.

E4	Equivalence Partitioning	linearSearch(9.4, [1, 2, 3, 4])	Error: Invalid input type.
E5	Equivalence Partitioning	linearSearch("jai", [1, 2, 3, 4])	Error: Invalid input type.
E6	Equivalence Partitioning	linearSearch(NULL, [1, 2, 3, 4])	Error: Invalid input type.
E7	Equivalence Partitioning	linearSearch(-1, NULL)	Error: Input array is null.
B1	Boundary Value Analysis	linearSearch(4, [1, 2, 3, 4])	3
B2	Boundary Value Analysis	linearSearch(0, [0, 2, 3, 4])	0
B3	Boundary Value Analysis	linearSearch(32, [1, 2, 3, 4])	-1 (NOT FOUND)
B4	Boundary Value Analysis	linearSearch(5, [5])	0



B5	Boundary Value Analysis	linearSearch(45, [5])	-1 (NOT FOUND)
----	-------------------------------	-----------------------	-------------------

MODIFIED CODE :-

```
#include <iostream>

#include <vector>

#include <limits>

#include <type_traits>
```

```
using namespace std;
```

```
int countItem(int v, const vector<int>& a) {
    int count = 0;
    for (size_t i = 0; i < a.size(); i++) {
        if (a[i] == v)
            count++;
    }
    return count;
}
```

```
int main() {
```

```
vector<int> myArray1 = {1, 2, 3, 4, 3, 3};

vector<int> myArray2 = {};

vector<int> myArray3 = {-2, -1, 0, 1};

int inputValue;

while (true) {

    cout << "Enter a value to count (or -999 to exit): ";

    cin >> inputValue;

    if (cin.fail() || cin.peek() != '\n') {

        cout << "Error: Invalid input type." << endl;

        cin.clear();

        cin.ignore(numeric_limits<streamsize>::max(), '\n');

        continue;

    }

    if (inputValue == -999)

        break;

    cout << "Count in myArray1: " << countItem(inputValue, myArray1) << endl;

    cout << "Count in myArray2: " << countItem(inputValue, myArray2) << endl;

    cout << "Count in myArray3: " << countItem(inputValue, myArray3) << 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);  
}
```

Functional Test Cases Table:

Test Case ID	Method	Tester Action and Input Data	Expected Outcome
E1	Equivalence Partitioning	countItem(4, [1, 2, 3, 4, 4, 3])	2
E2	Equivalence Partitioning	countItem(8, [1, 2, 3, 4, 3, 3])	0
E3	Equivalence Partitioning	countItem(1, [])	0

E4	Equivalence Partitioning	countItem(-1, [-2, -1, 0, 1])	1
E5	Equivalence Partitioning	countItem("jai", [1, 2, 3, 4, 5, 5])	Error: Invalid input type.
E6	Equivalence Partitioning	countItem('p', [1, 2, 3, 4, 5, 5])	Error: Invalid input type.
E7	Equivalence Partitioning	countItem(3.14, [1, 2, 3, 4, 3, 3])	Error: Invalid input type.
E8	Equivalence Partitioning	countItem(NULL, [1, 2, 3, 4, 3, 3])	Error: Invalid input type.
E9	Equivalence Partitioning	countItem(-1, NULL)	Error: Input array is null.
B1	Boundary Value Analysis	countItem(1, [1, 2, 3, 4, 3, 1])	2
B2	Boundary Value Analysis	countItem(4, [1, 2, 3, 4, 3, 3])	1
B3	Boundary Value Analysis	countItem(0, [1, 2, 3, 4, 3, 3])	0



B4	Boundary Value Analysis	countItem(5, [5])	1
B5	Boundary Value Analysis	countItem(2, [3])	0
B6	Boundary Value Analysis	countItem(0,[1,1,1,1])	0

MODIFIED CODE :-

```
#include <iostream>

#include <vector>

#include <limits>

#include <type_traits>
```

```
using namespace std;
```

```
int countItem(int v, const vector<int>& a) {
    int count = 0;
    for (size_t i = 0; i < a.size(); i++) {
        if (a[i] == v)
            count++;
    }
```

```
return count;
}

int main() {

    vector<int> myArray1 = {1, 2, 3, 4, 3, 3};

    vector<int> myArray2 = {};

    vector<int> myArray3 = {-2, -1, 0, 1};

    int inputValue;

    while (true) {

        cout << "Enter a value to count (or -999 to exit): ";

        cin >> inputValue;

        if (cin.fail() || cin.peek() != '\n') {

            cout << "Error: Invalid input type." << endl;

            cin.clear();

            cin.ignore(numeric_limits<streamsize>::max(), '\n');

            continue;

        }

        if (inputValue == -999)

            break;

    }

}
```



```
    cout << "Count in myArray1: " << countItem(inputValue, myArray1) << endl;

    cout << "Count in myArray2: " << countItem(inputValue, myArray2) << endl;

    cout << "Count in myArray3: " << countItem(inputValue, myArray3) << 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.

```
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);
}
```

Functional Test Cases Table:

Test Case ID	Method	Tester Action and Input Data	Expected Outcome
E1	Equivalence Partitioning	binarySearch(4, [1, 2, 3, 4, 5])	3
E2	Equivalence Partitioning	binarySearch(10, [1, 2, 3, 4, 5])	-1 (NOT FOUND)
E3	Equivalence Partitioning	binarySearch(1, [])	Error: Input array is empty.
E4	Equivalence Partitioning	binarySearch(-2, [-3, -2, -1, 0, 1])	1
E5	Equivalence Partitioning	binarySearch("MIN", [1, 2, 3, 4, 5])	Error: Invalid input type.
E6	Equivalence Partitioning	binarySearch(3.14, [1, 2, 3, 4, 5])	Error: Invalid input type.
E7	Equivalence Partitioning	binarySearch(NULL, [1, 2, 3, 4, 5])	Error: Invalid input type.

B1	Boundary Value Analysis	binarySearch(0, [0, 2, 3, 4, 5])	0
B2	Boundary Value Analysis	binarySearch(5, [1, 2, 3, 4, 5])	4
B3	Boundary Value Analysis	binarySearch(100, [1, 2, 3, 4, 5])	-1
B4	Boundary Value Analysis	binarySearch(3, [3])	0
B5	Boundary Value Analysis	binarySearch(2, [3])	-1
B6	Boundary Value Analysis	binarySearch(10, NULL)	-1

MODIFIED CODE :-

```

#include <iostream>

#include <vector>


#include <limits>

#include <type_traits>

using namespace std;

```

```
int binarySearch(int v, const vector<int>& a) {  
  
    int lo = 0, hi = a.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;  
    }  
  
    return -1;  
}  
  
int main() {  
  
    vector<int> myArray1 = {1, 2, 3, 4, 5};  
  
    vector<int> myArray2 = {};  
  
    vector<int> myArray3 = {-3, -2, -1, 0, 1};  
  
    int inputValue;  
  
    while (true) {  
  
        cout << "Enter a value to search (or -999 to exit): ";
```

```
cin >> inputValue;

if (cin.fail() || cin.peek() != '\n') {
    cout << "Error: Invalid input type." << endl;
    cin.clear();
    cin.ignore(numeric_limits<streamsize>::max(), '\n');
    continue;
}

if (inputValue == -999)
    break;

cout << "Index in myArray1: " << binarySearch(inputValue, myArray1) << endl;
cout << "Index in myArray2: " << binarySearch(inputValue, myArray2) << endl;
cout << "Index in myArray3: " << binarySearch(inputValue, myArray3) << endl;
}

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 (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);
}
```

Functional Test Cases Table:

Test Case ID	Method	Tester Action and Input Data	Expected Outcome
E1	Equivalence Partitioning	triangle(5, 5, 5)	0 (EQUILATERAL)
E2	Equivalence Partitioning	triangle(5, 2, 5)	1 (ISOSCELES)
E3	Equivalence Partitioning	triangle(8, 9, 10)	2 (SCALENE)

E4	Equivalence Partitioning	triangle(5, 1, 6)	3 (INVALID)
E5	Equivalence Partitioning	triangle(-1, -1, -1)	3 (INVALID)
E6	Equivalence Partitioning	triangle(0, 0, 0)	3 (INVALID)
E7	Equivalence Partitioning	triangle(1.5, 2, 2)	Error: Invalid input type.
E8	Equivalence Partitioning	triangle("PRA", 2, 2)	Error: Invalid input type.
E9	Equivalence Partitioning	triangle('O', 2, 2)	Error: Invalid input type.
E10	Equivalence Partitioning	triangle(3, 4, -5)	3 (INVALID)
E11	Equivalence Partitioning	triangle(1, 1, 50)	3 (INVALID)
E12	Equivalence Partitioning	triangle(NULL, 2, 2)	Error: Invalid input type.

B1	Boundary Value Analysis	triangle(10, 10, 10)	0 (EQUILATERAL)
B2	Boundary Value Analysis	triangle(2, 2, 1)	1 (ISOSCELES)
B3	Boundary Value Analysis	triangle(3, 3, 4)	1 (ISOSCELES)
B4	Boundary Value Analysis	triangle(1, 2, 3)	3 (INVALID)

MODIFIED CODE :-

```
#include <iostream>
```

```
#include <limits>
```

```
#include <type_traits>
```


```
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 side1, side2, side3;  
    while (true) {  
        cout << "Enter three sides of a triangle (or -999 to exit): ";  
        cin >> side1 >> side2 >> side3;  
  
        if (cin.fail() || cin.peek() != '\n') {  
            cout << "Error: Invalid input type." << endl;  
            cin.clear();  
            cin.ignore(numeric_limits<streamsize>::max(), '\n');  
            continue;  
        }  
    }  
}
```

```

    }

    if (side1 == -999 || side2 == -999 || side3 == -999)

        break;

    int result = triangle(side1, side2, side3);

    if (result == INVALID)

        cout << "Triangle is invalid." << endl;

    else if (result == EQUILATERAL)

        cout << "Triangle is equilateral." << endl;

    else if (result == ISOSCELES)

        cout << "Triangle is isosceles." << endl;

    else

        cout << "Triangle is scalene." << 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).

```

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;
}

```

Functional Test Cases Table:

Test Case ID	Method	Tester Action and Input Data	Expected Outcome
E1	Equivalence Partitioning	prefix("JAI", "JAIMIN")	true
E2	Equivalence Partitioning	prefix("pre", "test")	false
E3	Equivalence Partitioning	prefix("long", "longer")	true
E4	Equivalence Partitioning	prefix("", "samplestr")	true
E5	Equivalence Partitioning	prefix("data", "")	false

E6	Equivalence Partitioning	prefix("abcd", "abc")	false
E7	Equivalence Partitioning	prefix("123", "12345")	true
E8	Equivalence Partitioning	prefix("123456", "123")	false
E9	Equivalence Partitioning	prefix("1a3", "123")	false
E10	Equivalence Partitioning	prefix("abc", null)	Error: Invalid input type.
E11	Equivalence Partitioning	prefix(null, "abc")	Error: Invalid input type.
E12	Equivalence Partitioning	prefix("", null)	Error: Invalid input type.
E13	Equivalence Partitioning	prefix(null, null)	Error: Invalid input type.
B1	Boundary Value Analysis	prefix("string", "string")	true



B2	Boundary Value Analysis	prefix("ab", "abcabcabc")	true
B3	Boundary Value Analysis	prefix("", "")	true
B4	Boundary Value Analysis	prefix("longdata", "ongdata")	false
B5	Boundary Value Analysis	prefix("11", "111111")	true

MODIFIED CODE :-

```
#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 firstString, secondString;
```

```
    while (true) {
```

```
        cout << "Enter two strings (or 'exit' to quit): ";
```

```
        getline(cin, firstString);
```

```
        if (firstString == "exit")
```

```
            break;
```

```
        getline(cin, secondString);
```

```
        if (secondString == "exit")
```

```
            break;
```

```
        if (firstString.empty() || secondString.empty()) {
```

```
            cout << "Error: Strings cannot be empty." << endl;
```

```
            continue;
```

```
        }
```

```
        bool result = prefix(firstString, secondString);
```

```
        if (result) {
```

```
            cout << "The first string is a prefix of the second string." << endl;
```

```
        } else {
```

```
cout << "The first string is not a prefix of the second string." << 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

1. Valid Inputs:

Class 1: Scalene Triangle ($A \neq B \neq C$)

Class 2: Isosceles Triangle ($A = B$ OR $A = C$ OR $B = C$, any two side will be same)

Class 3: Equilateral Triangle ($A = B = C$, all sides will same)

Class 4: Right-Angled Triangle ($A^2 + B^2 = C^2$ OR any permutation)

2. Abstract Inputs:

Class 5: Non-Triangle ($A + B \leq C$ OR any permutation)

Class 6: invalid inputs ($A < 0$ OR $B < 0$ OR $C < 0$ OR strings etc)

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)

Test Case ID	Tester Action and Input Data	Expected Outcome	Equivalence Class
E1	triangle(5.0, 5.0, 5.0)	Equilateral	Class 1
E2	triangle(2.0, 2.0, 3.0)	Isosceles	Class 2
E3	triangle(8.0, 9.0, 10.0)	Scalene	Class 3
E4	triangle(3.0, 4.0, 7.0)	Non-Triangle	Class 5
E5	triangle(1.0, 1.0, 2.0)	Non-Triangle	Class 5
E6	triangle(0.0, 0.0, 2.0)	Non-Triangle	Class 5
E7	triangle(-1.0, 2.0, 2.0)	Error: Invalid input type.	Class 6
E8	triangle("abc", 2.0, 2.0)	Error: Invalid input type.	Class 6
E9	triangle(3.0, 4.0, 5.0)	Right-Angled	Class 4

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

Test Case ID	Tester Action and Input Data	Expected Outcome
B1	triangle(4.0, 5.0, 6.0)	Scalene
B2	triangle(2.0, 3.0, 5.0)	Non-Triangle
B3	triangle(3.0, 3.0, 5.0)	Non-Triangle

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

Test Case ID	Tester Action and Input Data	Expected Outcome
B4	triangle(5.0, 4.0, 5.0)	Isosceles
B5	triangle(3.0, 2.0, 3.0)	Isosceles

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

Test Case ID	Tester Action and Input Data	Expected Outcome
B6	triangle(6.0, 6.0, 6.0)	Equilateral
B7	triangle(0.0, 0.0, 0.0)	Non-Triangle

f) For the boundary condition $A^2 + B^2 = C^2$ case (right-angle triangle), identify test cases to verify the boundary.

Test Case ID	Tester Action and Input Data	Expected Outcome
B8	triangle(3.0, 4.0, 5.0)	Right-Angled
B9	triangle(5.0, 12.0, 13.0)	Right-Angled
B10	triangle(1.0, 1.0, 1.414)	Right-Angled
B11	triangle(1.0, 2.0, 2.236)	Non-Triangle

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

Test Case ID	Tester Action and Input Data	Expected Outcome
N1	triangle(1.0, 1.0, 20.0)	Non-Triangle
N2	triangle(5.0, 10.0, 4.0)	Non-Triangle

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

Test Case ID	Tester Action and Input Data	Expected Outcome
P1	triangle(-1.0, 2.0, 2.0)	Error: Invalid input type.
P2	triangle(0.0, 0.0, 2.0)	Non-Triangle
P3	triangle(-1.0, -2.0, -2.0)	Error: Invalid input type.