Lab8-Functional Testing (Black-Box)

Jish Chanchapra
202201501

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?

- **By Equivalence Class**:-

1. (Month < 1, Day < 1, Year < 1900)
2. (Month < 1, Day < 1, 1900<=Year<=2015)
3. (Month < 1, Day < 1, Year > 2015)
4. (Month < 1, 1<=Day<=31 , Year < 1900)
5. (Month < 1,1<=Day<=31,1900<=Year<=2015)
6. (Month < 1,1<=Day<=31, Year > 2015)
7. (Month < 1, Day >31, Year < 1900)
8. (Month < 1, Day >31, 1900<=Year<=2015)
9. (Month < 1, Day > 31, Year > 2015)
10. (1<=Month<=12, Day < 1, Year < 1900)
11. (1<=Month<=12, Day < 1,1900<=Year<=2015)
12. (1<=Month<=12, Day < 1, Year > 2015)
13. (1<=Month<=12, 1<=Day<=31, Year < 1900)
14. (1<=Month<=12, 1<=Day<=31, 1900<=Year<=2015)
15. (1<=Month<=12, 1<=Day<=31, Year > 2015)
16. (1<=Month<=12, Day >31, Year < 1900)
17. (1<=Month<=12, Day >31, 1900<=Year<=2015)
18. (1<=Month<=12, Day >31, Year > 2015)
19. (Month > 12, Day < 1, Year < 1900)

20. (Month > 12, Day < 1, 1900<=Year<=2015)
21. (Month > 12, Day < 1, Year > 2015)
22. (Month > 12,1<=Day<=31, Year < 1900)
23. (Month > 12, 1<=Day<=31, 1900<=Year<=2015)
24. (Month > 12, 1<=Day<=31, Year > 2015)
25. (Month > 12, Day > 31, Year < 1900)
26. (Month > 12, Day >31, 1900<=Year<=2015)
27. (Month > 12, Day >31, Year > 2015)

- **Test-case:**

Test-Case	**valid-Invalid**	**Included Class**
Month=0,Day=0, Year=1899	Invalid	1
Month=0,Day=0, Year=1900	Invalid	2
Month=0,Day=0, Year=2016	Invalid	3
Month=0,Day=1, Year=1899	Invalid	4
Month=0,Day=1, Year=1900	Invalid	5
Month=0,Day=1, Year=2016	Invalid	6
Month=0,Day=31, Year=1899	Invalid	7

Month=0,Day=31, Year=1900	Invalid	8
Month=0,Day=31, Year=2016	Invalid	9
Month=1,Day=0, Year=1890	Invalid	10
Month=1,Day=0, Year=1900	valid	11
Month=2,Day=0, Year=2018	Invalid	12
Month=1,Day=1, Year=1895	Invalid	13
Month=11,Day=27, Year=1900	valid	14
Month=10,Day=31, Year=2019	Invalid	15
Month=11,Day=32, Year=1896	Invalid	16
Month=1,Day=32, Year=1900	Invalid	17
Month=1,Day=34, Year=2019	Invalid	18

Month=13,Day=0, Year=1899	Invalid	19
Month=14,Day=0, Year=1900	Invalid	20
Month=15,Day=0, Year=2019	Invalid	21
Month=16,Day=1, Year=1888	Invalid	22
Month=17,Day=1, Year=1902	Invalid	23
Month=15,Day=1, Year=2019	Invalid	24
Month=14,Day=31 ,Year=1885	Invalid	25
Month=13,Day=32 ,Year=1905	Invalid	26
Month=13,Day=33 ,Year=2045	Invalid	27

- **Boundary Value Analysis :**

Test-Case	**Valid/Invalid**
Month=13	Invalid
Month=0	Invalid
Day=0	Invalid
Day=32	Invalid
Year=1899	Invalid
Year=2016	Invalid
Month=1,Day=1,year=1900	Valid
Month=1,Day=1,year=2015	Valid
Month=1,Day=31,year=1900	Valid
Month=1,Day=31,year=2015	Valid
Month=12,Day=1,year=1900	Valid
Month=12,Day=1,year=2015	Valid
Month=12,Day=31,year=1900	Valid
Month=12,Day=31,year=2015	Valid

- **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 <tuple>

using namespace std;

string prev_date(int d, int m, int y) {
  if (m < 1 || m > 12 || y < 1900 || y > 2015 d < 1 || d > 31) {
    return "Invalid";
  }

   return "Valid";
}
```

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.

- **By Equivalence Class**:-

    1. Array contains 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	**Valid/Invalid**	**Class**
v=3 && a[2]=[3,4]	Valid	1
v=5 && a[2]=[3,4]	Invalid	2
v=6 && a[]	Invalid	3
v=6 && a[1]=[6]	Valid	4
v=6 && a[1]=[7]	Invalid	5

- **Boundary Value Analysis :**

Test-Case	**Valid/Invalid**
v=1 && a[3]=[1,2,3]	Valid
v=3 && a[3]=[1,2,3]	Valid
v=3 &&a[]	Invalid
v=3 && a[1]=[3]	Valid
v=4 && a[3]=[1,2,3]	Invalid

- **Modified Programm && their output Besides of test-case :**

```
#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.

- **By Equivalence Class**:-

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**	**Class**

	Outcome	
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 && their output Besides of test-case :**

```
#include <iostream>
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.

- **By Equivalence Class**:-

2. The array is empty.
3. The value v is present in the array.
4. The value v is not present in the array.
5. The array contains only one element which is equal to v.
6. 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 Test Cases**

Input Data (Array a, Value v)	Expected Outcome	Boundary Condition
a = [5], v = 5	0	Single element array, value present
a = [5], v = 6	-1	Single element array, value absent
a = [1,2,3,4,5], v = 1	0	Value is at the start of the array
a = [1,2,3,4,5], v = 3	2	Value is in the middle of the array
a = [1,2,3,4,5], v = 5	4	Value is at the end of the array
a = [1,2,3,4,5], v = 6	-1	Value absent but close to elements in the array

- **Modified Programm && their output Besides of test-case :**

```cpp
#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 (three lengths equal), isosceles (two lengths equal), scalene (no lengths equal), or invalid (impossible lengths).

- **By Equivalence Class:**

1. Valid equilateral triangle:
2. Valid isosceles triangle:
3. Valid scalene triangle:
4. Invalid triangle (sum of any two sides must be greater than the third):
5. Negative lengths:
6. Zero lengths:

- **Test-Case:**

Test-Case	**Expected Outcomes**
a=3 ,b=3,c=3	Equilateral
a=4,b=4,c=5	Isosceles
a=3,b=4,c=5	Scalene
a=1,b=2,c=3	Invalid

a=-1,b=3,c=4	Invalid
a=0,b=3,c=4	Invalid

- **Boundary Value Analysis :**

Test-Case	**Expected Outcomes**
a = 2, b = 2, c = 2	Equilateral
a = 1, b = 1, c = 2	Invalid
a = -1, b = 1, c = 1	Invalid
a = 0, b = 1, c = 1	Invalid
a = 1, b = 1, c = 1	Equilateral

- **Modified Programm && their output Besides of test-case :**

```
#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";
}
```

```cpp
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).

- **By Equivalence Class:**

1. s1 is longer than s2 (impossible to be a prefix).
2. E2: s1 is a valid prefix of s2.
3. E3: s1 is not a prefix of s2.
4. E4: s1 is an empty string (edge case).
5. E5: s2 is an empty string (edge case).

- **Test-Case:**

Input Data (s1, s2)	Expected Outcome	Covered Equivalence Class
"abcdef", "abc"	false	E1
"abc", "abcdef"	true	E2
"xyz", "abcdef"	false	E3
"", "abcdef"	true	E4
"abc", ""	false	E5

- **Boundary Value Test Cases:**

Input Data (s1, s2)	Expected Outcome	Boundary Condition
“a”,” ”	false	S2 is empty
"abcdef", "abcdef"	true	s1 equals s2
"abc", "abc"	true	Shorter but equal strings

"", ""	true	Both strings are empty

- **Modified Programm && their output Besides of test-case :**

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

using namespace std;

bool prefix(string s1, string s2) {
    if (s1.length() > s2.length()) {
        return false;
    }
    for (int i = 0; i < s1.length(); i++) {
        if (s1[i] != s2[i]) {
            return false;
        }
    }
    return true;
}

int main() {
    // Equivalence Partitioning Test Cases
    cout << "TC1: " << (prefix("abcdef", "abc") ? "true" : "false") <<
endl; // output false
    cout << "TC2: " << (prefix("abc", "abcdef") ? "true" : "false") <<
endl; // output true
    cout << "TC3: " << (prefix("xyz", "abcdef") ? "true" : "false") <<
endl; // output false
    cout << "TC4: " << (prefix("", "abcdef") ? "true" : "false") << endl;
// output true
    cout << "TC5: " << (prefix("abc", "") ? "true" : "false") << endl;
// output false

    // Boundary Value Test Cases
```

```
    cout << "TC6: " << (prefix("a", "") ? "true" : "false") << endl;
// output false
    cout << "TC7: " << (prefix("abcdef", "abcdef") ? "true" : "false") <<
endl; // output true
    cout << "TC8: " << (prefix("abc", "abc") ? "true" : "false") << endl;
// output true
    cout << "TC9: " << (prefix("", "") ? "true" : "false") << endl;
// output true

    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:

2. **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.

- **Test-Case:**

Test-Case	Output	Class
A = 3.0, B = 3.0, C = 3.0	Equilateral	1
A = 4.0, B = 4.0, C = 5.0	Isosceles	2
A = 3.0, B = 4.0, C = 5.0	Scalene	3
A = 3.0, B = 4.0, C = 6.0	Invalid	5
A = -1.0, B = 2.0, C = 3.0	Invalid	6
A = 5.0, B = 12.0, C = 13.0	Right-angled	4

- **Boundary Conditions:**

c) Boundary Conditions for A + B > C (Scalene Triangle)

Test-Case	Output
A = 1.0, B = 1.0, C = 1.9999	Scalene
A = 2.0, B = 3.0, C = 4.0	Scalene

d) Boundary Conditions for A = C (Isosceles Triangle)

Test-Case	Output
A = 3.0, B = 3.0, C = 4.0	Isosceles

A = 2.0, B = 2.0, C = 3.0	Isosceles
A = 2.0, B = 2.0, C = 2.0	Equilateral

E) Boundary Conditions for A = B = C (Equilateral Triangle)

Test-Case	**Output**
A = 2.0, B = 2.0, C = 2.0	Equilateral
A = 1.9999, B = 1.9999, C = 1.9999	Equilateral

f) Boundary Conditions for $A^2 + B^2 = C^2$ (Right-Angle Triangle)

Test-Case	**Output**
A = 3.0, B = 4.0, C = 5.0	Right-angled
A = 5.0, B = 12.0, C = 13.0	Right-angled

g) Test Cases for Non-Triangle Case

Test-Case	**Output**
A = 1.0, B = 2.0, C = 3.0	Invalid
A = 1.0, B = 2.0, C = 2.0	Invalid
A = 1.0, B = 1.0, C = 3.0	Invalid

h) Test Cases for Non-Positive Input

Test-Case	Output
A = 0.0, B = 2.0, C = 3.0	Invalid
A = -1.0, B = -2.0, C = 3.0	Invalid
A = 3.0, B = 0.0, C = 2.0	Invalid

- **Modified Programm && their output Besides of test-case :**

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

const char* classifyTriangle(float A, float B, float C) {
  if (A <= 0 || B <= 0 || C <= 0 || A + B <= C || A + C <= B || B + C <= A) {
    return "Invalid";
  }
  if (fabs(pow(A, 2) + pow(B, 2) - pow(C, 2)) < 1e-6 ||
    fabs(pow(A, 2) + pow(C, 2) - pow(B, 2)) < 1e-6 ||
    fabs(pow(B, 2) + pow(C, 2) - pow(A, 2)) < 1e-6) {
    return "Right-angled";
  }

  if (A == B && B == C) {
    return "Equilateral";
  }

  if (A == B || B == C || A == C) {
    return "Isosceles";
  }

  return "Scalene";
}
```

```
int main() {
   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;
}
```