Lab8-Functional Testing (Black-Box)

Aman Mangukiya
202201156

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?

Equivalence 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

- **Test-case:**

	Test Data (Day, Month, Year)	Expected Outcome	ivalence Classes Cov
1	15, 7, 2010	valid	2, 5, 8
2	0, 7, 2010	invalid	4
3	32, 7, 2010	valid	6
4	1, 0, 2010	invalid	1
5	15, 13, 2010	invalid	3
6	1, 3, 2016	invalid	9
7	1, 3, 1899	invalid	7
8	1, 1, 1900	valid	2, 5, 8
9	31, 12, 2015	valid	2, 5, 8

- **Boundary Value Analysis :**

Test Data (Day, Month, Year)	Expected Outcome	Equivalence Classes Covered
15, 7, 2022	Invalid	9
0, 7, 2010	Invalid	4
32, 7, 2010	Invalid	6
1, 0, 2010	Invalid	1
15, 13, 2010	Invalid	3
1, 3, 2016	Invalid	9
1, 3, 1899	Invalid	7
1, 1, 1900	Valid	2, 5, 8
31, 12, 2015	Valid	2, 5, 8

- **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 identi ed 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";
}
```

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.

Equivalence Class Description:

- **e1: The array a is empty (Invalid, as there are no elements to search).**
- **e2: The value v exists in the array a (Valid).**
- **e3: The value v does not exist in the array a (Valid).**
- **e4: The array a contains only one element, and v is that element (Valid).**
- **e5: The array a contains only one element, and v is not that element (Valid).**
- **e6: The array a contains multiple elements, and v is at the first position (Valid).**
- **e7: The array a contains multiple elements, and v is at the last position (Valid).**

Equivalence Class Test Cases:	**Expected Outcome**	**Equivalence Classes Covered**
Test Case Input Data (Array a, Value v)	Expected Outcome	Covered Equivalence Class
a = [], v = 5	-1 (Array is empty)	e1
a = [1, 2, 3, 4, 5], v = 3	2 (Value exists in array)	e2
a = [1, 2, 3, 4, 5], v = 6	-1 (Value does not exist)	e3
a = [5], v = 5	0 (Single element array, v exists)	e4
a = [5], v = 3	-1 (Single element array, v does not exist)	e5

a = [5, 10, 15, 20], v = 5	0 (First element matches)	e6
a = [5, 10, 15, 20], v = 20	3 (Last element matches)	e7
31, 12, 2015	Valid	2, 5, 8 (Valid month, day, year)

- **Boundary Value Analysis Test cases:**

Boundary Conditions:

- **b1: The array a is empty.**
- **b2: The array a has one element.**
- **b3: The value v is at the first index of the array.**
- **b4: The value v is at the last index of the array.**
- **b5: The value v does not exist in the array, but the array has values close to v.**

Test Case Input Data (Array a, Value v)	**Expected Outcome**	**Covered Boundary Condition**
a = [], v = 5	-1 (Empty array)	b1
a = [5], v = 5	0 (Single element matches)	b2
a = [5, 10, 15, 20], v = 5	0 (First element matches)	b3
a = [5, 10, 15, 20], v = 20	3 (Last element matches)	b4
a = [5, 10, 15, 20], v = 25	-1 (Value does not exist)	b5

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

```cpp
#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:-**

- e1: The array a is empty (Valid).
- e2: The value v exists in the array a once (Valid).
- e3: The value v exists in the array a multiple times (Valid).
- e4: The value v does not exist in the array a (Valid).
- e5: The array a contains only one element, and v matches that element (Valid).
- e6: The array a contains only one element, and v does not match that element (Valid).
- e7: The array a contains negative values, and v is negative (Valid).
- e8: The array a contains large numbers, and v is a large number (Valid).

- **Test-Case:-**

se Input Data (Array a, V	**Expected Outcome**	**Covered Equivalence Clas**
a = [], v = 5	0 (Array is empty)	e1
a = [1, 2, 3, 4, 5], v = 3	1 (Value exists once)	e2
a = [1, 3, 3, 3, 5], v = 3	3 (Value exists multiple times)	e3
a = [1, 2, 3, 4, 5], v = 6	0 (Value does not exist)	e4
a = [5], v = 5	1 (Single element matches)	e5
a = [5], v = 3	0 (Single element does not match)	e6
a = [-5, -3, -3, -2], v = -3	2 (Value is negative)	e7

- ## Boundary Value Analysis :

Boundary Conditions:

- **b1: The array a is empty.**
- **b2: The array a has one element.**
- **b3: The value v exists only once in the array.**
- **b4: The value v exists multiple times in the array.**
- **b5: The value v does not exist in the array, but there are values close to v.**

Boundary Value Test Cases:		
Case Input Data (Array a, Valu	Expected Outcome	Covered Boundary Condition
a = [], v = 5	0 (Empty array)	b1
a = [5], v = 5	1 (Single element matches)	b2
a = [5, 10, 15, 20], v = 10	1 (Exists only once)	b3
a = [5, 10, 10, 20], v = 10	2 (Exists multiple times)	b4
a = [5, 10, 15, 20], v = 25	0 (Does not exist)	b5

- **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.

Equivalence Class Description:

- e1: Array is empty.
- e2: Array contains one element (the value is present).
- e3: Array contains one element (the value is absent).
- e4: Array contains multiple elements (all negative).
- e5: Array contains multiple elements (all positive).
- e6: Array contains multiple elements (mixed negative and positive).
- e7: Value is less than the smallest element in the array.
- e8: Value is equal to the smallest element in the array.
- e9: Value is equal to the largest element in the array.
- e10: Value is greater than the largest element in the array.
- e11: Value is present in the array (between smallest and largest).
- e12: Value is not present in the array (between smallest and largest).

Test Case Input Data (Array a, Value v)	Expected Outcome	Covered Equivalence Class
([], 5)	-1	e1
([3], 3)	0	e2
([3], 5)	-1	e3
([-3, -2, -1], -2)	1	e4
([1, 2, 3], 2)	1	e5
([-1, 0, 1], 0)	1	e6

([-3, -2, -1], -4)	-1	e7
([-3, -2, -1], -3)	0	e8
([-3, -2, -1], -1)	2	e9
([-3, -2, -1], 0)	-1	e10
([-3, -2, -1], -2.5)	-1	e12
([-3, -2, -1], -2)	1	e11

- ## Boundary Value Test Cases

Boundary Conditions:

- b1: The array contains 0 elements (empty).
- b2: The array contains 1 element (minimum case).
- b3: The array contains 2 elements (smallest non-empty array).
- b4: The value is at the lower boundary (minimum value in the array).
- b5: The value is at the upper boundary (maximum value in the array).
- b6: The value is just outside the boundaries.

Test Case Input Data (Array a, Value v)	Expected Outcome	Covered Boundary Condition
([], 5)	-1	b1
([5], 5)	0	b2
([1, 2], 1)	0	b3
([1, 2], 0)	-1	b4
([1, 2], 2)	1	b5
([1, 2], 3)	-1	b6

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

```
#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).

Equivalence Class Description:

- e1: All three sides are equal (equilateral triangle).
- e2: Two sides are equal, and one side is different (isosceles triangle).
- e3: All three sides are different (scalene triangle).

- e4: The lengths do not form a triangle (invalid).
- e5: One or more sides have lengths of zero (invalid).

- **Test-Case:**

Test Case Input Data (Sides a, b, c)	Expected Outcome
(3, 3, 3)	0
(5, 5, 3)	1
(3, 4, 5)	2
(1, 2, 3)	3
(0, 3, 4)	3
(5, 5, 5)	0

- **Boundary Value Analysis :**

Boundary Conditions:

- b1: All sides equal the minimum valid triangle length (1).
- b2: Two sides equal and one is the smallest invalid triangle length (1, 1, 2).
- b3: One side equal to zero (invalid).
- b4: One side equal to the sum of the other two (invalid).
- b5: One side slightly greater than the sum of the other two (invalid).

Test Case Input Data (Sides a, b, c)	Expected Outcome	Covered Boundary Condition
(1, 1, 1)	0	b1
(1, 1, 2)	3	b2
(0, 1, 1)	3	b3
(2, 2, 4)	3	b4
(2, 3, 6)	3	b5
(3, 5, 2)	2	b3
(2, 2, 3)	1	b1
(3, 3, 5)	1	b2

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

```cpp
#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 pre x (String s1, String s2) returns whether or not the string s1 is a pre x of string s2 (You may assume that neither s1 nor s2 is null).

Equivalence Class Description:

- **e1:** s1 **is an empty string, and** s2 **is non-empty (always true).**
- **e2:** s1 **is a non-empty string that is equal to** s2 **(prefix is equal to string).**
- **e3:** s1 **is a non-empty string that is a proper prefix of** s2 **(some characters match).**
- **e4:** s1 **is a non-empty string that is not a prefix of** s2 **(characters do not match).**
- **e5:** s1 **is longer than** s2 **(cannot be a prefix).**

- **Test-Case:**

Test Case Input Data (s1, s2)	**Expected Outcome**	**Covered Equivalence Class**
("", "hello")	TRUE	e1
("hello", "hello")	TRUE	e2
("he", "hello")	TRUE	e3
("hell", "hello")	TRUE	e3
("world", "hello")	FALSE	e4
("hel", "world")	FALSE	e4
("hello", "world")	FALSE	e4
("hello", "hello world")	TRUE	e3
("hi", "hello")	FALSE	e4
("hello", "hi")	FALSE	e5
("a", "abc")	TRUE	e3
("abc", "a")	FALSE	e5

- **Boundary Value Test Cases:**

Boundary Conditions:

- **b1: s1 is empty, s2 is empty (both are empty).**
- **b2: s1 is a single character, and s2 is a single character (both equal).**
- **b3: s1 is a single character, and s2 is a different single character.**
- **b4: s1 is a single character, and s2 is longer (prefix case).**
- **b5: s1 is longer than s2 (invalid case).**

Test Case Input Data (s1, s2)	Expected Outcome	Covered Boundary Condition
("", "")	TRUE	b1
("a", "a")	TRUE	b2
("a", "b")	FALSE	b3
("a", "abc")	TRUE	b4
("abc", "a")	FALSE	b5
("", "abc")	TRUE	b1
("a", "")	FALSE	b5
("prefix", "prefixSuffix")	TRUE	b4

- **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 classi cation program (P4) with a slightly different speci cation: The program reads oating 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(oat A, oat B, oat 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;
}
```