SOFTWARE ENGINEERING

IT 314

LAB 8

GROUP : 01

NAME : NIPURNA PATEL

STUDENT ID : 202201061

[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.**

➢ ***Equivalence Partitioning:***

- Valid classes
    - 1<= month <=12
    - 1<= day <=31
    - 1900 <= year <=2015

- Invalid classes
    - Month < 1
    - Month > 12
    - Day < 1
    - Day > 31
    - Year < 1900
    - Year > 2015

Test cases for equivalence partitioning:

Input data	**Expected output**	**Category**
EP 1 : (1, 1, 2000)	(31, 12, 1999)	Valid
EP 2 : (20, 12, 2012)	(19, 12, 2012)	Valid
EP 3 : (29, 2, 2000)	(28, 2, 2000)	Valid leap year
EP 4 : (29, 2, 2003)	Error	Invalid leap year
EP 5 : (32, 4, 2010)	Invalid date	Invalid date

EP 6 : (5, 13, 1999)	Invalid month	Invalid month
EP 7 : (7, 7, 1888)	Invalid year	Invalid year
EP 8 : (9, 8, 2016)	Invalid year	Invalid year
EP 9 : (3, 6, 1998)	(2, 6, 1998)	Valid

➢ ***Boundary value analysis :***

- Day boundaries
    - Day >=1 & Day <=31

- Month boundaries
    - Month >=1 & Month <=12

- Year boundaries
    - Year <=1900 & Year >=2015

Boundary value analysis :

Input data	**Expected output data**	**Category**
(1, 1, 2000)	(31, 12, 1999)	Valid
(1, 1, 1900)	(31, 12, 1899)	Valid
(32, 3, 2010)	Invalid date	Invalid date
(13, 13, 2014)	Invalid year	Invalid year
(1, 2, 1899)	Invalid year	Invalid year
(29, 2, 2001)	Invalid leap year	Invalid leap year
(1, 12, 1900)	(30, 11, 1899)	Valid
(1, 1, 2015)	(31, 12, 2014)	Valid
(1, 3, 2001)	(28, 2, 2001)	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.

```
int linearSearch(int v, int a[]) {
    int i = 0;
    while (i < a.length) {
        if (a[i] == v) {
            return i;
        }
        i++;
    }
    return -1;
}
```

➢ **Equivalence Class Partitioning:**

- Valid classes
    - The value v exists in the array.
    - The value v does not exist in the array.
    - The value v appears multiple times in the array.
    - The array is empty.

- Invalid classes
    - Invalid input type. (eg. string)

Test cases for Equivalence Class Partitioning:

	Input data	**Output data**	**Description**
1.	a = [1, 2, 3, 4, 5] v = 1	0	Value found at first index.
2.	a = [1, 2, 3, 4, 5] v = 5	4	Value found at last index.
3.	a = [1, 2, 3, 4, 5] v = 3	2	Value found in the middle of the array.
4.	a = []	-1	Empty array

	v = 3		
5.	a = [1, 2, 3, 4, 5] v = 7	-1	Value not found
6.	a = [1, 2, 1, 3, 4] v = 1	0	More than 1 occurrence of element.

➢ **Boundary value analysis :**

- Value found at the boundaries of the array (first, last)
- Array size (0, 1, 2)

Test cases for Boundary value analysis:

	Input data	**Expected output**	**Description**
1.	a = [1] v = 1	0	First and only element
2.	a = [1] v = 2	-1	Single element, value not found
3.	a = [1, 2, 3] v = 1	0	Value found at first index.
4.	a = [1, 2, 3] v = 3	2	Value found at the last index.
5.	a = [1, 3, 5, 7] v = 2	-1	Value not found
6.	a = [1, 3, 5, 7] v = 0	-1	Value below the range
7.	a = [1, 3, 5, 7] v = 10	-1	Value above the range

(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;
}
```

➢ **Equivalence Class Partitioning:**

- The value appears more than once in the array.
- The value appears exactly once.
- The value does not exist in the array.
- The array contains no elements.
- The array contains the same value repeated.

	Input data	**Expected output**	**Description**
1.	a = [1, 2, 3,2, 4] v = 2	2	Value found multiple times
2.	a = [1, 2, 3, 4, 5] v = 3	1	Value found once
3.	a = [1, 2, 3, 4, 5] v = 7	0	Value not found
4.	a = [] v = 2	0	Empty array
5.	a = [1, 1, 1, 1] v = 1	4	All elements are the same

➢ **Boundary value analysis :**

- Test around the boundaries of array sizes (e.g., sizes 0, 1, and 2).
- Check counting for the first and last elements in small arrays.
- Values just below the smallest and just above the largest element in the array.

Test cases for Boundary value analysis:

	Input data	Output data	Description
1.	a = [1] v = 1	1	Single element found
2.	a = [1] v = 2	0	Single element not found
3.	a = [1, 2, 3] v = 1	1	Value found in the first position
4.	a = [1, 2, 3] v = 3	1	Value found in the last position
5.	a = [1, 3] v = 2	0	Value not found in two-element array
6.	a = [1, 2, 3, 4] v = 0	0	Value below the range
7.	a = [1, 2, 3, 4] v = 6	0	Value above the range

(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.

```
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 Class Partitioning :**

- The value exists in the array.
- The value does not exist in the array.
- The value is the first element in the array.
- The value is the last element in the array.
- The value is the middle element in the array.
- Value smaller than the smallest element.
- Value larger than the largest element.

Test cases for Equivalence Class Partitioning :

	Input data	**Output data**	**Description**
1.	a = [1, 2, 4, 5, 6] v = 1	0	Value found at first index
2.	a = [1, 2, 4, 5, 6] v = 6	4	Value found at last index
3.	a = [1, 2, 4, 5, 6] v = 3	-1	Value not found
4.	a = [1, 2, 4, 5, 6] v = 4	2	Value found at middle index
5.	a = [] v = 1	-1	Empty array

➢ **Boundary value analysis :**

- Test cases around the boundaries of array sizes (e.g., size 0, 1, and 2).
- Check searching for the first and last elements of a two-element array.
- Values just below the smallest and just above the largest element in the array.

Test cases for boundary value analysis:

	Input data	**Expected Output data**	**Description**
1.	a = [1] v = 1	0	Single element found

2.	a = [1] v = 0	-1	Single element not found
3.	a = [1, 2] v = 1	0	Value found in the first position
4.	a = [1, 2] v = 2	1	Value found in the last position
5.	a = [1, 2] v = 3	-1	Value not found in two-element array
6.	a = [1, 2, 3] v = 0	-1	Value below the range
7.	a = [1, 2,3] v = 5	-1	Value above the range
8.	a = [1, 5, 3, 2, 8] v = 5	Invalid	Unsorted array

(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;
}
```

➢ **Equivalence Class Partitioning :**

- All three sides are equal.
- Two sides are equal.
- All sides are different.
- Impossible lengths. (a + b > c)
- At least one side is negative.
- One side is zero.

	Input data	**Expected output**	**Description**
1.	(3, 3, 3)	0 (valid)	Equilateral triangle
2.	(3, 3, 4)	1 (valid)	Isosceles triangle
3.	(3, 4, 5)	2 (valid)	Scalene triangle
4.	(1, 2, 3)	Invalid triangle	(a + b > c)
5.	(-1, 2, 3)	Invalid triangle	Negative length not possible
6.	(0, 1, 2)	Invalid triangle	Zero length

➢ **Boundary value analysis :**

- Smallest valid sides. (1, 1, 1)
- Slightly violating triangle inequality.
- One side is zero.
- One side is negative.

	Input data	**Expected output**	**Description**
1.	(1, 1, 1)	0 (Valid)	Minimum valid equilateral triangle
2.	(1, 2, 3)	Invalid triangle	Minimum invalid triangle (a + b > c)
3.	(2, 3, 0)	Invalid triangle	Zero length

4.	(2, -1, 2)	Invalid triangle	Invalid length (negative)

(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 class partitioning :**

- s1 is the prefix of s2.
- s1 is equal to the s2.
- s1 is not a prefix of s2.
- s1 is longer than s2.

	Input data	**Expected output data**	**Description**
1.	("pre", "prefix")	true	s1 is a valid prefix of s2
2.	("same", "same")	true	s1 is equal to s2
3.	("not", "prefix")	false	s1 is not a prefix of s2
4.	("longer", "short")	false	s1 is longer than s2
5.	("pre", "short")	false	s1 is not a prefix of s2

➢ **Boundary value analysis:**

- s1 has one character and s2 has one character
    - Both characters can either match or differ

- s1 has one character and s2 has multiple characters
    - This includes both matching and non-matching scenarios.

	Input data	**Expected output data**	**Description**
1.	("a", "ab")	true	Single character as a valid prefix
2.	("b", "a")	false	Single character not a prefix
3.	("abc", "abcd")	true	Multi-character prefix
4.	("abc", "ab")	false	s1 is longer than s2
5.	("abc", "abc")	true	Both strings are identical

(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

- Valid equivalence classes
    - **E1 : Equilateral Triangle**: All sides are equal
    - **E2 : Isosceles Triangle**: Two sides are equal
    - **E3 : Scalene Triangle**: All sides are different (a + b > c)
    - **E4 : Right-Angled Triangle**: Follows Pythagorean theorem

- Invalid equivalence classes
    - **E5 : Non-Triangle**: The values cannot form a triangle (a + b <= c)
    - **E6 : Non-Positive Input**: Any side length that is zero or negative

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)

	Input data	**Expected output**	**Description**	**Equivalence Class**
1.	(3.0, 3.0, 3.0)	"Equilateral"	All sides are equal	E1
2.	(4.0, 4.0, 5.0)	"Isosceles"	Two sides equal	E2
3.	(3.0, 4.0, 5.0)	"Scalene"	All sides different and satisfy triangle inequality	E3
4.	(5.0, 12.0, 13.0)	"Right-Angled"	Satisfies Pythagorean theorem	E4
5.	(1.0, 2.0, 3.0)	Invalid	Fails triangle inequality	E5
6.	(0.0, 4.0, 5.0)	Invalid	Zero length	E6
7.	(-1.0, 2.0, 3.0)	Invalid	Non-positive length	E6
8	(0.0, 0.0, 0.0)	Invalid	All sides are zero	E6

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

	Input data	**Expected output**	**Description**	**Boundary condition**
1.	(2.0, 3.0, 4.0)	"Scalene"	Sides are different and satisfy a + b > c	Boundary: a + b > c
2.	(2.0, 2.0, 4.0)	Invalid	Fails triangle inequality (a + b = c)	Boundary: a + b = c
3.	(2.0, 1.0, 5.0)	Invalid	Fails triangle inequality (a + b < c)	Boundary: a + b < c

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

	Input data	**Expected output**	**Description**	**Boundary condition**

1.	(3.0, 4.0, 3.0)	"Isosceles"	Two sides are equal	Boundary: a = c
2.	(3.0, 4.0, 5.0)	“Scalene"	All sides different	Boundary: Not equal
3.	(3.0, 3.0, 5.0)	"Isosceles"	Two sides are equal	Boundary: a = b

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

	Input data	**Expected output**	**Description**	**Boundary condition**
1.	(3.0, 3.0, 3.0)	"Equilateral"	All sides are equal	Boundary: a = b = c
2.	(3.0, 3.0, 2.9)	"Isosceles"	Almost equilateral, two sides equal	Boundary: a = b
3.	(3.0, 2.9, 2.9)	Invalid	Fails triangle inequality	Boundary: a + b > c

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

	Input data	**Expected output**	**Description**	**Boundary condition**
1.	(3.0, 4.0, 5.0)	"Right-Angled"	Pythagorean triple	Boundary: $A^2 + B^2 = C^2$
2.	(5.0, 12.0, 13.0)	"Right-Angled"	Pythagorean triple	Boundary: $A^2 + B^2 = C^2$
3.	(3.0, 3.0, 4.0)	Invalid	Does not satisfy Pythagoras theorem	Boundary: $A^2 + B^2 > C^2$

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

	Input data	**Expected output**	**Description**	**Boundary condition**
1.	(1.0, 1.0, 3.0)	Invalid	Fails triangle inequality	Boundary: a + b

				= c
2.	(1.0, 1.0, 2.0)	Invalid	Fails triangle inequality	Boundary: a + b = c
3.	(5.0, 5.0, 11.0)	Invalid	Fails triangle inequality	Boundary: a + b < c

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

	Input data	**Expected output**	**Description**	**Boundary condition**
1.	(0.0, 4.0, 5.0)	Invalid	Zero length	Boundary: a ≤ 0
2.	(-1.0, 2.0, 3.0)	Invalid	Non-positive length	Boundary: a ≤ 0
3.	(2.0, 0.0, 3.0)	Invalid	Non-positive length	Boundary: b ≤ 0
4.	(2.0, 3.0, -1.0)	Invalid	Non-positive length	Boundary: c ≤ 0