**Lab-08**

# IT-314 Software Engineering

Name**-** Meet Sarvan
ID- 202201218

## Q1.

1. Enlist which set of test cases have been identified using Equivalence Partitioning and Boundary Value Analysis separately.

**Ans:-**

## ❖ Equivalence Partitioning Test Cases

### >> Valid Date Equivalence Classes:

1. **Normal Day Scenario:**
   - Given Date: (10, 3, 2012)
   - Expected Output: (9, 3, 2012)
2. **Leap Year February End Scenario:**
   - Given Date: (29, 2, 2004)
   - Expected Output: (28, 2, 2004)
3. **Non-Leap Year February End Scenario:**
   - Given Date: (1, 3, 2011)
   - Expected Output: (28, 2, 2011)
4. **Month-End Scenario:**
   - Given Date: (31, 5, 2015)
   - Expected Output: (30, 5, 2015)
5. **30-Day Month Scenario:**
   - Given Date: (30, 11, 2015)
   - Expected Output: (29, 11, 2015)

### >> Invalid Date Equivalence Classes:

1. **Invalid Day (Zero):**
   - Given Date: (0, 12, 2015)
   - Expected Output: Error message (Invalid day)
2. **Invalid Day (Negative):**
   - Given Date: (-5, 6, 2015)
   - Expected Output: Error message (Invalid day)
3. **Invalid Month (Zero):**
   - Given Date: (15, 0, 2015)
   - Expected Output: Error message (Invalid month)
4. **Invalid Month (Negative):**

- Given Date: (15, -2, 2015)
- Expected Output: Error message (Invalid month)

5. **Invalid Year (Future Year):**
    - Given Date: (1, 1, 2026)
    - Expected Output: Error message (Year in the future)

## >> Boundary Value Analysis Test Cases

1. **Day Before the First of the Month:**
    - Given Date: (1, 4, 2015)
    - Expected Output: (31, 3, 2015)
2. **Last Day of February (Non-Leap Year):**
    - Given Date: (28, 2, 2012)
    - Expected Output: (27, 2, 2012)
3. **Last Day of February (Leap Year):**
    - Given Date: (29, 2, 2012)
    - Expected Output: (28, 2, 2012)
4. **Day Boundary (31st Day):**
    - Given Date: (31, 1, 2015)
    - Expected Output: (30, 1, 2015)
5. **Year Lower Boundary:**
    - Given Date: (1, 1, 1900)
    - Expected Output: (31, 12, 1899)
6. **Year Upper Boundary:**
    - Given Date: (1, 1, 2015)
    - Expected Output: (31, 12, 2014)
7. **Maximum Day for Months with 30 Days:**
    - Given Date: (30, 9, 2015)
    - Expected Output: (29, 9, 2015)
8. **Last Valid Date for the Year:**
    - Given Date: (31, 12, 2015)
    - Expected Output: (30, 12, 2015)

**>> Equivalence Partitioning Test Cases:**

| Tester Action and Input Data | Expected Outcome |
|---|---|
| (10, 3, 2012) | (9, 3, 2012) |
| (29, 2, 2004) | (28, 2, 2004) |
| (1, 3, 2011) | (28, 2, 2011) |
| (31, 5, 2015) | (30, 5, 2015) |
| (30, 11, 2015) | (29, 11, 2015) |
| (0, 12, 2015) | An Error message |
| (-5, 6, 2015) | An Error message |
| (15, 0, 2015) | An Error message |
| (15, -2, 2015) | An Error message |
| (1, 1, 2026) | An Error message |

**>> Boundary Value Analysis Test Cases:**

| Tester Action and Input Data | Expected Outcome |
|---|---|
| (1, 4, 2015) | (31, 3, 2014) |
| (28, 2, 2012) | (27, 2, 2012) |
| (29, 2, 2012) | (28, 2, 2012) |
| (31, 1, 2015) | (30, 1, 2015) |
| (1, 1, 1900) | (31, 12, 1899) |
| (1, 1, 2015) | (31, 12, 2014) |
| (30, 9, 2015) | (29, 9, 2015) |

|  |  |
|---|---|
| (31, 12, 2015) | (30, 12, 2015) |

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.

**Ans:-**

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

bool leapYearCheck(int year) {
    return (year % 4 == 0 && year % 100 != 0) || (year % 400 == 0);
}

string calculatePreviousDate(int day, int month, int year) {
    if (year < 1900 || year > 2015 || month < 1 || month > 12 || day < 1 || day > 31) {
        return "Date is not valid";
    }

    int daysInMonths[] = {31, leapYearCheck(year) ? 29 : 28, 31, 30, 31, 30, 31, 31,
30, 31, 30, 31};

    if (day > daysInMonths[month - 1]) {
        return "Date is not valid";
    }

    if (day > 1) {
        return to_string(day - 1) + "/" + to_string(month) + "/" + to_string(year);
    } else {
        if (month == 1) {
            return "31/12/" + to_string(year - 1);
        } else {
            return to_string(daysInMonths[month - 2]) + "/" + to_string(month - 1) + "/"
+ to_string(year);
        }
    }
}

int main() {
    int day, month, year;

    cout << "Enter the day: ";
    cin >> day;
    cout << "Enter the month: ";
    cin >> month;
    cout << "Enter the year: ";
    cin >> year;

    string result = calculatePreviousDate(day, month, year);
    cout << "Previous date is: " << result << endl;

    return 0;
}
```

## Testing the Program

You can manually enter the values using the previously mentioned test cases. Below are some examples you can use for testing:

| Test Case Input | Expected Output |
|---|---|
| (15, 8, 2022) | "14/08/2022" |
| (29, 2, 2000) | "28/2/2000" |
| (1, 3, 2013) | "28/2/2013" |
| (31, 12, 2024) | "30/12/2024" |
| (1, 4, 2023) | "31/3/2023" |
| (15, -1, 2015) | "Invalid date" |
| (35, 1, 2017) | "Invalid date" |
| (31, 4, 2015) | "Invalid date" |
| (29, 2, 2015) | "Invalid date" |

## Checking Outcomes:-

For each input, determine whether the output matches the expected result. Run the program using the specified day, month, and year for each test case. After execution, compare the program’s output with the anticipated outcome:

1. If the output aligns with the expected result, the test case is regarded as successful.
2. If there is a mismatch, the test case is considered unsuccessful.

# Q2.

## P1)

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

## Equivalence Classes:

**Class 1**: Empty array → Output: `-1`

**Class 2**: Value exists (first occurrence at index 0) → Output: `0`

**Class 3**: Value exists (first occurrence at index n, n > 0) → Output: `n`

**Class 4**: Value does not exist in the array → Output: `-1`

**Class 5**: Value exists with duplicates → Output: Index of the first occurrence.

## Test Cases :

Here's a table summarizing the test cases for the `linearSearch` function, including the input, expected outcome, and the equivalence class each case covers:

| Input (v, a) | Expected Output | Covers Equivalence Class |
|---|---|---|
| (9, []) | -1 | 1 |
| (7, [7, 8, 9]) | 0 | 2 |
| (8, [1, 2, 8]) | 2 | 3 |
| (11, [3, 5, 7, 9]) | -1 | 4 |
| (6, [4, 6, 6, 3, 6]) | 1 | 5 |

| | | |
|---|---|---|
| (12, [12, 15, 16]) | 0 | 2 |
| (5, [2, 5, 5, 7]) | 1 | 5 |
| (19, [14, 15, 19]) | 2 | 3 |
| (13, [1, 2, 3, 13]) | 3 | 3 |
| (0, [0, 2, 4, 6]) | 0 | 2 |

This table provides a clear overview of the test cases, the inputs provided, the expected outputs, and the corresponding equivalence classes each test case covers.

## P2)

```
int countItem(int v, int a[])
{
        int count = 0;
        for (int i = 0; i &lt; a.length; i++)
        {
                if (a[i] == v)
                count++;
        }
        return (count);
}
```

## Equivalence Classes:

**Class 1**: Empty array → Output: `0`
**Class 2**: Value exists once → Output: `1`

**Class 3**: Value exists multiple times → Output: Count of occurrences

(e.g., `n` if `v` appears `n` times)

**Class 4**: Value does not exist → Output: `0`

**Class 5**: All elements are equal to `v` → Output: Length of the array

## Test Cases :

| Input (v, a) | Expected Output | Covers Equivalence Class |
|---|---|---|
| (7, []) | 0 | 1 |
| (6, [3, 6, 9]) | 1 | 2 |
| (10, [1, 2, 4]) | 0 | 4 |
| (2, [2, 2, 2, 2]) | 4 | 5 |
| (4, [4, 4, 4, 5]) | 3 | 3 |
| (12, [5, 6, 8]) | 0 | 4 |
| (11, [11, 11, 11]) | 3 | 5 |
| (5, [0, 1, 5, 5]) | 2 | 3 |
| (6, [6, 7, 8, 9]) | 1 | 2 |
| (3, [1, 2, 3, 3, 3]) | 3 | 3 |

## P3)

```
int binarySearch(int v, int a[])
{
int lo,mid,hi;
lo = 0;
hi = a.length-1;
while (lo &lt;= hi)
{
mid = (lo+hi)/2;
if (v == a[mid])
return (mid);
else if (v &lt; a[mid])
hi = mid-1;
else
lo = mid+1;

}
return(-1);
}
```

## Equivalence Classes :

**Class 1**: Empty array → Output: `-1`

**Class 2**: Value exists at the first index → Output: Index `0`
**Class 3**: Value exists at a middle index → Output: Index of `v`

**Class 4**: Value exists at the last index → Output: Index of last occurrence

**Class 5**: Value does not exist (less than smallest element) → Output: `-1`

**Class 6**: Value does not exist (greater than largest element) → Output: `-1`

**Class 7**: Value does not exist (between two elements) → Output: `-1`

**Class 8**: Value exists with duplicates → Output: Index of any occurrence of v.

## Test Cases :

| Input (v, a) | Expected Output | Covers Equivalence Class |
|---|---|---|
| (8, []) | -1 | 1 |
| (3, [3, 4, 5, 6]) | 0 | 2 |
| (9, [2, 5, 9, 12]) | 2 | 3 |
| (6, [1, 6, 8]) | 1 | 3 |
| (10, [2, 5, 9]) | -1 | 6 |
| (4, [1, 2, 3, 7]) | -1 | 7 |
| (15, [3, 7, 9, 12]) | -1 | 6 |
| (5, [5, 5, 6, 7]) | 0 | 8 |
| (1, [1, 2, 2, 3]) | 0 | 2 |

## P4)

```
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 &gt;= b+c || b &gt;= a+c || c &gt;= a+b)
                return(INVALID);
        if (a == b &amp;&amp; b == c)
                return(EQUILATERAL);
        if (a == b || a == c || b == c)
                return(ISOSCELES);

        return(SCALENE);
}
```

## Equivalence Classes :

**Class 1**: Invalid triangle (non-positive sides) → Output: `INVALID`

**Class 2**: Invalid triangle (triangle inequality not satisfied) → Output: `INVALID`

**Class 3**: Equilateral triangle (all sides equal) → Output: `EQUILATERAL`

**Class 4**: Isosceles triangle (two sides equal) → Output: `ISOSCELES`

**Class 5**: Scalene triangle (all sides different) → Output: `SCALENE`

## Test Cases :

| Input (a, b, c) | Expected Outcome | Covers Equivalence Class |
|---|---|---|
| (0, 3, 5) | INVALID | 1 |
| (1, -2, 4) | INVALID | 1 |
| (4, 4, 4) | EQUILATERAL | 3 |
| (5, 5, 8) | ISOSCELES | 4 |
| (6, 8, 10) | SCALENE | 5 |
| (1, 2, 3) | INVALID | 2 |

|  |  |  |
|---|---|---|
| (2, 2, 5) | ISOSCELES | 4 |
| (3, 4, 8) | INVALID | 2 |
| (6, 7, 9) | SCALENE | 5 |
| (4, 7, 11) | INVALID | 2 |

## P5)

```
public static boolean prefix(String s1, String s2)
{
        if (s1.length() &gt; s2.length())
        {
                return false;
        }
        for (int i = 0; i &lt; s1.length(); i++)
        {
                if (s1.charAt(i) != s2.charAt(i))
                {
                        return false;
                }
        }
        return true;
}
```

## Equivalence Classes :

**Class 1**: s1 is longer than s2 → Output: `false`
**Class 2**: s1 is an exact prefix of s2 → Output: `true`
**Class 3**: s1 is a partial prefix of s2 → Output: `false`
**Class 4**: s1 is empty → Output: `true`

**Class 5**: s2 is empty and s1 is not → Output: `false`
**Class 6**: s1 is equal to s2 → Output: `true`

## Test Cases:

| Input (s1, s2) | Expected Outcome | Covers Equivalence Class |
|---|---|---|
| ("long", "short") | false | 1 |
| ("hello", "hello!") | true | 2 |
| ("greeting", "") | false | 5 |
| ("ab", "abc") | true | 6 |
| ("longerPrefix", "short") | false | 1 |
| ("abc", "abcde") | true | 2 |
| ("match", "mat") | false | 3 |
| ("empty", "empty") | true | 6 |

## P6)

### a) Identifying the Equivalence Classes

**>> Valid Triangle Types:**

**Equilateral Triangle**: Side A = Side B = Side C
**Isosceles Triangle**: Side A = Side B, or Side A = Side C, or Side B = Side C
**Scalene Triangle**: All sides unequal (A ≠ B ≠ C)

**Right-Angled Triangle**: $A^2 + B^2 = C^2$ (Pythagorean theorem) or its permutations

**>> Invalid Triangle Cases:**

**Not a Triangle**: $A + B \leq C$, $A + C \leq B$, or $B + C \leq A$
**Non-positive Input**: Any side A, B, or C is less than or equal to zero

## b) Test Cases Covering the Identified Equivalence Classes:

| Input (A, B, C) | Expected Output | Equivalence Classes Covered |
|---|---|---|
| (4.0, 4.0, 4.0) | Equilateral Triangle | Equilateral Triangle |
| (5, 5, 8) | Isosceles Triangle | Isosceles Triangle |
| (5, 8, 9) | Scalene Triangle | Scalene Triangle |
| (6, 8, 10) | Right-Angled Triangle | Right-Angled Triangle |
| (1, 2, 8) | Not a Triangle | Not a Triangle |
| (0, 6, 7) | Invalid | Non-positive Input |

## c) Boundary Condition $A + B > C$ (Scalene Triangle):

| Input (A, B, C) | Expected Output |
|---|---|

| | |
|---|---|
| (4, 5, 6) | Scalene Triangle |
| (6, 7, 12) | Scalene Triangle |
| (6, 7, 13) | Not a Triangle |
| (5, 7, 11) | Scalene Triangle |

## d) Boundary Condition A = C (Isosceles Triangle):

| Input (A, B, C) | Expected Output |
|---|---|
| (7.0, 8.0, 7.0) | Isosceles Triangle |
| (39.0, 40.0, 40.0) | Isosceles Triangle |
| (5, 9, 14) | Not a Triangle |
| (9, 9, 9) | Equilateral Triangle |

## e) Boundary Condition A = B = C (Equilateral Triangle):

| Input (A, B, C) | Expected Output |
|---|---|
| (6, 6, 6) | Equilateral Triangle |
| (3, 3, 3) | Equilateral Triangle |

| | |
|---|---|
| (7, 8, 14) | Not a Triangle |
| (6, 9, 13) | Scalene Triangle |

## f) Boundary Condition $A^2 + B^2 = C^2$ (Right-Angled Triangle):

| Input (A, B, C) | Expected Output |
|---|---|
| (6, 8, 10) | Right-Angled Triangle |
| (5, 12, 13) | Right-Angled Triangle |
| (6, 9, 14) | Not a Triangle |
| (7, 10, 12) | Scalene Triangle |

## g) Non-Triangle Case:

| Input (A, B, C) | Expected Output |
|---|---|
| (5, 6, 7) | Scalene Triangle |
| (1.0, 2.0, 3.0) | Not a Triangle |
| (10.0, 1.0, 1.0) | Not a Triangle |
| (6, 8, 14) | Scalene Triangle |

## h) Non-Positive Input Case:

| Input (A, B, C) | Expected Output |
|---|---|
| (0.0, 1.0, 1.0) | Invalid |
| (5, 7, -3) | Invalid |
| (1.0, 0.0, 1.0) | Invalid |
| (-4, 6, 9) | Invalid |