# IT314: Software Engineering

# Lab-7

**Name: Parva Dhami**
**ID: 202001117**
**Date: 13/04/2023**

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

**a. Equivalence Partitioning Test Cases:**

| Tester Action and Input Data | Expected Outcome |
|---|---|
| Valid input: day=1, month=1, year=1900 | Invalid date |
| Valid input: day=31, month=12, year=2015 | Previous date |
| Invalid input: day=0, month=6, year=2000 | Error message |
| Invalid input: day=14, month=7, year=2050 | Error message |
| Invalid input: day=29, month=2, year=2005 | Error message |

**b. Boundary Value Analysis Test Cases:**

| Tester Action and Input Data | Expected Outcome |
|---|---|
| Valid input: day=1, month=1, year=1900 | Invalid date |
| Valid input: day=31, month=12, year=2015 | Previous date |
| Invalid input: day=0, month=14, year=2000 | Error message |

| | |
|---|---|
| **Invalid input: day=32, month=6, year=1870** | **Error message** |
| **Invalid input: day=29, month=2, year=2000** | **Error message** |
| **Valid input: day=1, month=6, year=2000** | **Previous date** |
| **Invalid input: day=0, month=6, year=2000** | **Error message** |
| **Valid input: day=31, month=5, year=2000** | **Previous date** |
| **Valid input: day=15, month=6, year=2000** | **Previous date** |
| **Invalid input: day=31, month=4, year=1998** | **Error message** |

## 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.**

```
import org.junit.Test;
import static org.junit.Assert.*;

public class LinearSearchTest {

  @Test
  public void testExistingValue() {
    int[] arr = {1, 2, 3, 4, 5, 6};
    int index = linearSearch(3, arr);
    assertEquals(2, index);
  }
```

```java
    @Test
    public void testNonExistingValue() {
      int[] arr = {1, 2, 3, 4, 5, 6};
      int index = linearSearch(6, arr);
      assertEquals(-1, index);
    }

    @Test
    public void testFirstElement() {
      int[] arr = {1, 2, 3, 4, 5, 6};
      int index = linearSearch(1, arr);
      assertEquals(0, index);
    }

    @Test
    public void testLastElement() {
      int[] arr = {1, 2, 3, 4, 5, 6};
      int index = linearSearch(5, arr);
      assertEquals(4, index);
    }

    @Test
    public void testEmptyArray() {
      int[] arr = {};
      int index = linearSearch(1, arr);
      assertEquals(-1, index);
    }

    @Test
    public void testNullArray() {
      int[] arr = null;
      int index = linearSearch(1, arr);
      assertEquals(-1, index);
    }
  }
```

# Equivalence Partitioning:

| Tester Action and Input Data | Expected Outcome |
|---|---|

| | |
|---|---|
| Test with v as a non-existent value and an empty array a[ ] | -1 |
| Test with v as a non-existent value and a non-empty array a[ ] | -1 |
| Test with v as an existent value and an empty array a[ ] | -1 |
| Test with v as an existent value and a non-empty array a[ ] where v exists | Index of v in a [ ] |
| Test with v as an existent value and a non-empty array a[ ] where v does not exist | -1 |

## Boundary Value Analysis:

| Tester Action and Input Data | Expected Outcome |
|---|---|
| Test with v as a non-existent value and an empty array a[ ] | -1 |
| Test with v as a non-existent value and a non-empty array a[ ] | -1 |
| Test with v as an existent value and an array a[ ] of length 0 | -1 |
| Test with v as an existent value and an array a[ ] of length 1, where v exists | 0 |
| Test with v as an existent value and an array a[ ] of length 1, where v does not exist | -1 |

| | |
|---|---|
| Test with v as an existent value and an array a[ ] of length greater than 1, where v exists at the beginning of the array | 0 |
| Test with v as an existent value and an array a[ ] of length greater than 1, where v exists at the end of the array | the last index where v is found |
| Test with v as an existent value and an array a[ ] of length greater than 1, where v exists in the middle of the array | the index where v is found |

## P2: The function countItem returns the number of times a value v appears in an array of integers a.

## Equivalence Partitioning:

| Tester Action and Input Data | Expected Outcome |
|---|---|
| Test with v as a non-existent value and an empty array a[ ] | 0 |
| Test with v as an existent value and an empty array a[ ] | 0 |
| Test with v as a non-existent value and a non-empty array a[ ] | 0 |
| Test with v as an existent value and a non-empty array a[ ] where v exists multiple times | the number of occurrences of v in a[ ] |

| | |
|---|---|
| Test with v as an existent value and a non-empty array a[ ] where v exists only once | 1 |

## Boundary Value Analysis:

| Tester Action and Input Data | Expected Outcome |
|---|---|
| Test with v as a non-existent value and an empty array a[ ] | 0 |
| Test with v as a non-existent value and a non-empty array a[ ] | 0 |
| Test with v as an existent value and an array a[ ] of length 0 | 0 |
| Test with v as an existent value and an array a[ ] of length 1, where v exists | 1 |
| Test with v as an existent value and an array a[ ] of length 1, where v does not exist | 0 |
| Test with v as an existent value and an array a[ ] of length greater than 1, where v exists at the beginning of the array | the number of occurrences of v in a[ ] |
| Test with v as an existent value and an array a[ ] of length greater than 1, where v exists at the end of the array | the number of occurrences of v in a[ ] |
| Test with v as an existent value and an array a[ ] of length greater than 1, where v exists in the middle of the array | the number of occurrences of v in a[ ] |

## 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.**

## Equivalence Partitioning:

| Tester Action and Input Data | Expected Outcome |
|---|---|
| v=5, a=[1, 3, 5, 7, 9] | 2 |
| v=1, a=[1, 3, 5, 7, 9] | 0 |
| v=9, a=[1, 3, 5, 7, 9] | 4 |
| v=4, a=[1, 3, 5, 7, 9] | -1 |
| v=11, a=[1, 3, 5, 7, 9] | -1 |

## Boundary Value Analysis:

| Tester Action and Input Data | Expected Outcome |
|---|---|
| v=1, a=[1] | 0 |
| v=9, a=[9] | 0 |
| v=5, a=[] | -1 |

| | |
|---|---|
| v=5, a=[5, 7, 9] | 0 (smallest element in the array) |
| v=5, a=[1, 3, 5] | 2 (largest element in the 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).**

## Boundary Value Analysis:

| Tester Action and Input Data | Expected Outcome |
|---|---|
| Invalid inputs: a = 0, b = 0, c = 0 | INVALID |
| Invalid inputs: a + b = c or b + c = a or c + a = b (a=5, b=4, c=11) | INVALID |
| Equilateral triangles: a = b = c = 9 | EQUILATERAL |
| Isosceles triangles: a = b ≠ c = 10 | ISOSCELES |
| Isosceles triangles: a ≠ b = c = 8 | ISOSCELES |
| Isosceles triangles: a = c ≠ b = 23 | ISOSCELES |
| Scalene triangles: a = b + c - 1 | SCALENE |

| | |
|---|---|
| Scalene triangles: b = a + c - 1 | SCALENE |
| Scalene triangles: c = a + b - 1 | SCALENE |
| Maximum values: a, b, c = Integer.MAX_VALUE | INVALID |
| Minimum values: a, b, c = Integer.MIN_VALUE | INVALID |

## Equivalence Partitioning:

| Tester Action and Input Data | Expected Outcome |
|---|---|
| Valid input: a=3, b=3, c=3 | EQUILATERAL |
| Valid input: a=4, b=4, c=5 | ISOSCELES |
| Valid input: a=5, b=4, c=3 | SCALENE |
| Invalid input: a=0, b=0, c=0 | INVALID |
| Invalid input: a=-1, b=2, c=3 | INVALID |
| Valid input: a=1, b=1, c=1 | EQUILATERAL |
| Valid input: a=2, b=2, c=1 | ISOSCELES |
| Valid input: a=3, b=4, c=5 | SCALENE |
| Invalid input: a=0, b=1, c=1 | INVALID |

| | |
|---|---|
| Invalid input: a=1, b=0, c=1 | INVALID |
| Invalid input: a=1, b=1, c=0 | INVALID |

## 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).**

## Equivalence Partitioning:

| Tester Action and Input Data | Expected Outcome |
|---|---|
| Valid Inputs: s1 = "hello", s2 = "hello world" | true |
| Valid Inputs: s1 = "a", s2 = "abc" | true |
| Invalid Inputs: s1 = "", s2 = "hello world" | false |
| Invalid Inputs: s1 = "world", s2 = "hello world" | false |

## Boundary Value Analysis:

| Tester Action and Input Data | Expected Outcome |
|---|---|
| s1 = "", s2 = "abc" | False |
| s1 = "ab", s2 = "abc" | True |

| | |
|---|---|
| s1 = "abc", s2 = "ab" | False |
| s1 = "a", s2 = "ab" | True |
| s1 = "aaaaaaaaaaaaaaaaaaaaaa", s2 = "aaaaaaaaaaaaaaaaaaaaab" | True |
| s1 = "abc", s2 = "abc" | True |
| s1 = "b", s2 = "c" | False |
| s1 = " ", s2 = " " | True |
| s1 = "a", s2 = "b" | False |
| s1 = "a", s2 = " " | False |

## 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) Equivalence Classes:**

| Tester Action and Input Data | Expected Outcome |
|---|---|
| a = 0, b = 2, c = 3 | Invalid Input |

| | |
|---|---|
| a = 3, b = 4, c = 5 | Scalene Right angled triangle |
| a = 3, b = 5, c = 4 | Scalene right angled triangle |
| a = 5, b = 3, c = 4 | Scalene |
| a = 2, b = 2, c = 3 | Isosceles triangle |
| a = 3, b = 4, c = 9 | Not a triangle |

**b) Test cases:**

**Invalid inputs: a = 0, b = 0, c = 0, a + b = c, b + c = a, c + a = b Invalid inputs: a = -1, b = 1, c = 1, a + b = c Equilateral triangles: a = b = c = 1, a = b = c = 100 Isosceles triangles: a = b = 10, c = 5; a = c = 10, b = 3; b = c = 10, a = 6 Scalene triangles: a = 4, b = 5, c = 6; a = 10, b = 11, c = 13 Right angled triangle: a = 3, b = 4, c = 5; a = 5, b = 12, c = 13 Non-triangle: a = 1, b = 2, c = 3 Non-positive input: a = -1, b = -2, c = -3**
**c) Boundary condition A + B > C:**

**a = Integer.MAX_VALUE, b = Integer.MAX_VALUE, c = 1 a = Double.MAX_VALUE, b = Double.MAX_VALUE, c = Double.MAX_VALUE**
**d) Boundary condition A = C:**

**a = Integer.MAX_VALUE, b = 2, c = Integer.MAX_VALUE a = Double.MAX_VALUE, b = 2.5, c = Double.MAX_VALUE**
**e) Boundary condition A = B = C:**

**a = Integer.MAX_VALUE, b = Integer.MAX_VALUE, c = Integer.MAX_VALUE a = Double.MAX_VALUE, b = Double.MAX_VALUE, c = Double.MAX_VALUE**
**f) Boundary condition A^2 + B^2 = C^2:**

**a = Integer.MAX_VALUE, b = Integer.MAX_VALUE, c = Integer.MAX_VALUE a = Double.MAX_VALUE, b = Double.MAX_VALUE, c = Math.sqrt(Math.pow(Double.MAX_VALUE, 2) + Math.pow(Double.MAX_VALUE, 2))**
**g) Non-triangle:**

**a = 1, b = 2, c = 4 a = 2, b = 4, c = 8**
**h) Non-positive input:**

**a = -1, b = -2, c = -3 a = 0, b = 1, c = 2**

# Section B:

1. **Control Flow Graph (CFG):**

```
       +-----+
        | i = 1|
        +-----+
           |
           |
           v
+---------------+
| i < p.size()   |
+---------------+
     | true
     |
     v
+-----------------------+
| ((Point)p.get(i)).y < ((Point)p.get(min)).y |
+-----------------------+
    | true         | false
    |              |
    v              v
+---------------+    +---------------+
| min = i       |    | i++           |
+---------------+    +---------------+
     |                    |
     |                    |
     v                    v
+---------------+    +---------------+
| i < p.size()   |    | return doStack(p) |
+---------------+    +---------------+
     | false              |
     |                    |
     v                    v
+------------------------+
| ((Point)p.get(i)).y == ((Point)p.get(min)).y |
+------------------------+
        | true             | false
        |                  |
        v                  v
+------------------------+  +---------------+
```

```
| ((Point)p.get(i)).x > ((Point)p.get(min)).x|
+-----------------------+  +--------------+
        | true            | false
        |                 |
        v                 v
+---------------+    +---------------+
| min = i       |    | i++           |
+---------------+    +---------------+
        |                 |
        |                 |
        v                 v
+---------------+    +---------------+
| i < p.size()   |    | return doStack(p) |
+---------------+    +---------------+
        | false             |
        |                   |
        v                   v
+--------------------------+
| return doStack(p)        |
+--------------------------+
```

2. **Test sets for each coverage criterion:**

**a. Statement Coverage:**

- **Test 1: p = {new Point(0, 0), new Point(1, 1)}**
- **Test 2: p = {new Point(0, 0), new Point(1, 0), new Point(2, 0)}**

**b. Branch Coverage:**

- **Test 1: p = {new Point(0, 0), new Point(1, 1)}**
- **Test 2: p = {new Point(0, 0), new Point(1, 0), new Point(2, 0)}**
- **Test 3: p = {new Point(0, 0), new Point(1, 0), new Point(1, 1)}**

**c. Basic Condition Coverage:**

- **Test 1: p = {new Point(0, 0), new Point(1, 1)}**
- **Test 2: p = {new Point(0, 0), new Point(1, 0), new Point(2, 0)}**
- **Test 3: p = {new Point(0, 0), new Point(1, 0), new Point(1, 1)}**
- **Test 4: p = {new Point(0, 0), new Point(1, 0), new Point(0, 1)}**
- **Test 5: p = {new Point(0, 0), new Point(0, 1), new Point(1, 1)}**