IT 314
SOFTWARE ENGINEERING

Sakshi Shah
202201281

Software Testing

Lab Session - Functional Testing (Black-Box)

To design a program for determining the previous date based on given input values (day, month, year), we can utilize Equivalence Partitioning (EP) and Boundary Value Analysis (BVA) to create a robust set of test cases.

1. Test Case Identification

We identify valid and invalid partitions based on the input ranges:

- Valid Input Partitions:
  - Valid dates (e.g., `1, 1, 1900` to `31, 12, 2015` considering leap years)
- Invalid Input Partitions:
  - Invalid month (e.g., `1, 13, 2000`, `1, 0, 2000`)
  - Invalid day (e.g., `32, 1, 2000`, `0, 1, 2000`)
  - Invalid year (e.g., `1, 1, 1899`, `1, 1, 2016`)

Boundary Value Analysis (BVA)

We focus on the boundaries of valid ranges:

- Valid Boundaries:
  - `1, 1, 1900` (first valid date)
  - `31, 12, 2015` (last valid date)
- Invalid Boundaries:
  - `0, 1, 1900` (below lower boundary)
  - `1, 13, 2000` (above upper boundary for month)
  - `32, 1, 2000` (above upper boundary for day)
  - `1, 1, 1899` (below lower boundary for year)

- `1, 1, 2016` (above upper boundary for year)

Test Suite

Equivalence Partitioning Test Cases

Tester Action and Input Data	Expected Outcome
1, 1, 1900	31,12,1899
15,5,2000	14,5,2000
1, 1, 2015	31, 12, 2014
29,2,2000	18,2,2000
1,1,2016	Error Message
0, 1, 2000	Error Message
1, 13, 2000	Error Message
31, 12, 2015	30, 12, 2015

Boundary Value Analysis Test Cases

Tester Action and Input Data	Expected Outcome
1, 1, 1900	31, 12, 1899
31, 12, 2015	30, 12, 2015
0, 1, 1900	Error Message
1, 0, 2000	Error Message

1, 13, 2000	Error Message
32, 1, 2000	Error Message
1, 1, 1899	Error Message
1, 1, 2016	Error Message

2. Program Implementation and Execution

```
def previous_date(day, month, year):
    # Check for valid input ranges
    if year < 1900 or year > 2015:
        return "Error: Invalid Year"
    if month < 1 or month > 12:
        return "Error: Invalid Month"
    if day < 1 or day > 31:
        return "Error: Invalid Day"

    # Handle the logic for the previous date
    if day == 1:
        month -= 1
        if month == 0:
            month = 12
            year -= 1
        day = get_last_day_of_month(month, year)
    else:
        day -= 1

    return day, month, year
```

```
def get_last_day_of_month(month, year):
    if month in [1, 3, 5, 7, 8, 10, 12]:
        return 31
    elif month in [4, 6, 9, 11]:
        return 30
    elif month == 2:
        if year % 4 == 0 and (year % 100 != 0 or year % 400 == 0):
            return 29
        return 28

# Run test cases
test_cases = [
    (1, 1, 1900), (15, 5, 2000), (1, 1, 2015), (29, 2, 2000),
    (29, 2, 2001), (1, 1, 2016), (31, 12, 2015), (0, 1, 2000),
    (1, 13, 2000), (32, 1, 2000), (1, 1, 1899), (1, 1, 2016)
]

for day, month, year in test_cases:
    print(previous_date(day, month, year))
```

Q2.
P1

```
#include <stdio.h>

int linearSearch(int v, int a[], int length) {
    for (int i = 0; i < length; i++) {
        if (a[i] == v) {
            return i; // Return the index if value is found
        }
    }
    return -1; // Return -1 if value is not found
```

}

Equivalence Partitioning Test Cases

Input	Expected Output
`{3, 5, 2, 4, 5, 1}, 5`	`1`
`{3, 5, 2, 4, 5, 1}, 3`	`0`
`{3, 5, 2, 4, 5, 1}, 1`	`5`
`{3, 5, 2, 4, 5, 1}, 10`	`-1`
`[], 5`	`-1`

Boundary Value Analysis Test Cases

Input	Expected Output
`{3}, 3`	`0`
`{3}, 5`	`-1`
`{3, 5}, 3`	`0`
`{3, 5}, 5`	`1`
`{3, 5}, 2`	`-1`
`{1, 2, ..., 1000}, 500`	`499`
`{1, 2, ..., 1000}, 1001`	`-1`

P2

```
#include <stdio.h>

int countItem(int v, int a[], int length) {
    int count = 0;
    for (int i = 0; i < length; i++) {
```

```
        if (a[i] == v) {
            count++; // Increment count for each match
        }
    }
    return count; // Return total count
}
```

Equivalence Partitioning Test Cases

Input	Expected Output
`{3, 5, 2, 4, 5, 1, 5}, 5`	`3`
`{3}, 3`	`1`
`{3, 5, 2, 4, 5, 1}, 10`	`0`
`[], 5`	`0`

Boundary Value Analysis Test Cases

Input	Expected Output
`{3}, 3`	`1`
`{3}, 5`	`0`
`{3, 3}, 3`	`2`
`{3, 5}, 3`	`1`
`{3, 5}, 2`	`0`
`{1, 1, 1, 2, 3, 1}, 1`	`4`
`{1, 2, ..., 1000}, 500`	`0`

P3

```
#include <stdio.h>

int binarySearch(int v, int a[], int length) {
    int lo = 0;
    int hi = length - 1;
```

```
    while (lo <= hi) {
        int mid = (lo + hi) / 2; // Calculate mid index
        if (v == a[mid]) {
            return mid; // Value found
        } else if (v < a[mid]) {
            hi = mid - 1; // Search left half
        } else {
            lo = mid + 1; // Search right half
        }
    }
    return -1; // Value not found
}
```

Equivalence Partitioning Test Cases

Input	Expected Output
`{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}, 7`	`6`
`{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}, 1`	`0`
`{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}, 10`	`9`
`{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}, 0`	`-1`
`{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}, 11`	`-1`
`[], 5`	`-1`

Boundary Value Analysis Test Cases

Input	Expected Output
`{3}, 3`	`0`
`{3}, 5`	`-1`
`{1, 2}, 1`	`0`
`{1, 2}, 2`	`1`
`{1, 2}, 0`	`-1`
`{1, 2, 3}, 2`	`1`
`{1, 2, ..., 1000}, 500`	`499`
`{1, 2, ..., 1000}, 0`	`-1`
`{1, 2, ..., 1000}, 1001`	`-1`

P4

```
#include <stdio.h>

#define EQUILATERAL 0
#define ISOSCELES 1
#define SCALENE 2
#define INVALID 3

int triangle(int a, int b, int c) {
    // Check for invalid triangle conditions
    if (a <= 0 || b <= 0 || c <= 0 || a >= b + c || b >= a + c || c >= a + b) {
        return INVALID;
    }
    // Check for equilateral triangle
    if (a == b && b == c) {
        return EQUILATERAL;
    }
    // Check for isosceles triangle
    if (a == b || a == c || b == c) {
```

```
        return ISOSCELES;
    }
    // Otherwise, it's a scalene triangle
    return SCALENE;
}
```

Equivalence Partitioning Test Cases

Input	Expected Output
`(3, 3, 3)`	`EQUILATERAL`
`(3, 3, 2)`	`ISOSCELES`
`(3, 4, 5)`	`SCALENE`
`(1, 2, 3)`	`INVALID`
`(3, 4, -5)`	`INVALID`
`(0, 4, 5)`	`INVALID`

Boundary Value Analysis Test Cases

Input	Expected Output
`(1, 1, 1)`	`EQUILATERAL`
`(1, 1, 2)`	`INVALID`
`(2, 2, 3)`	`ISOSCELES`
`(2, 3, 4)`	`SCALENE`
`(-1, 2, 3)`	`INVALID`
`(0, 2, 3)`	`INVALID`

P5

```
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 Partitioning Test Cases

Input	Expected Output
`("pre", "prefix")`	`true`
`("same", "same")`	`true`
`("test", "best")`	`false`
`("longer", "short")`	`false`
`("", "not empty")`	`true`
`("not empty", "")`	`false`

Boundary Value Analysis Test Cases

Input	Expected Output
`("", "hello")`	`true`
`("hello", "")`	`false`
`("abc", "abc")`	`true`
`("abc", "abcd")`	`true`
`("abc", "abx")`	`false`

P6

a) Identify the Equivalence Classes

Valid Input Classes:

1. **Equilateral Triangle**: All three sides are equal (A = B = C).
2. **Isosceles Triangle**: Exactly two sides are equal (A = B or B = C or A = C).

3. **Scalene Triangle**: All sides are different, and they satisfy the triangle inequality (A + B > C, A + C > B, B + C > A).
4. **Right-Angled Triangle**: Satisfies the Pythagorean theorem ($A^2 + B^2 = C^2$ or similar).

Invalid Input Classes:

1. **Non-Triangle**: The lengths do not satisfy the triangle inequality (e.g., A + B ≤ C).
2. **Non-Positive Inputs**: One or more sides are zero or negative (A ≤ 0, B ≤ 0, C ≤ 0).

b) Identify Test Cases

Test Case	Expected Outcome	Equivalence Class
(3.0, 3.0, 3.0)	"Equilateral"	Equilateral Triangle
(4.0, 4.0, 2.0)	"Isosceles"	Isosceles Triangle
(3.0, 4.0, 5.0)	"Scalene"	Scalene Triangle
(5.0, 12.0, 13.0)	"Right-Angled"	Right-Angled Triangle
(1.0, 2.0, 3.0)	"Not a triangle"	Non-Triangle
(-1.0, 2.0, 3.0)	"Invalid input"	Non-Positive Input
(0.0, 2.0, 3.0)	"Invalid input"	Non-Positive Input

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

Boundary Test Cases

1. **Just valid**: (3.0, 4.0, 5.0) — Expected: "Scalene"
2. **Just invalid**: (2.0, 2.0, 4.0) — Expected: "Not a triangle"
3. **Boundary case**: (1.0, 1.0, 2.0) — Expected: "Not a triangle"
4. **Valid but close**: (2.0, 3.0, 4.0) — Expected: "Scalene"

d) Boundary Condition A = C (Isosceles Triangle)

Boundary Test Cases

1. **Just valid**: (3.0, 3.0, 4.0) — Expected: "Isosceles"
2. **Just invalid**: (3.0, 4.0, 3.0) — Expected: "Isosceles"
3. **Exactly equal**: (5.0, 5.0, 5.0) — Expected: "Equilateral" (to ensure it does not misclassify)
4. **Invalid**: (2.0, 2.0, 5.0) — Expected: "Not a triangle"

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

Boundary Test Cases

1. **All sides equal**: (3.0, 3.0, 3.0) — Expected: "Equilateral"
2. **Small equal sides**: (0.1, 0.1, 0.1) — Expected: "Equilateral"
3. **Invalid but equal sides**: (1.0, 1.0, 2.0) — Expected: "Not a triangle"
4. **Non-positive**: (0.0, 0.0, 0.0) — Expected: "Invalid input"

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

Boundary Test Cases

1. **Exact right triangle**: (3.0, 4.0, 5.0) — Expected: "Right-Angled"
2. **Not right-angled but close**: (3.0, 4.0, 6.0) — Expected: "Scalene"
3. **Another right triangle**: (5.0, 12.0, 13.0) — Expected: "Right-Angled"
4. **Just invalid**: (6.0, 8.0, 10.0) — Expected: "Scalene" (not a right angle)

g) Non-Triangle Cases

Test Cases

1. **Zero sum**: (1.0, 1.0, 2.0) — Expected: "Not a triangle"
2. **Negative side**: (-1.0, 1.0, 1.0) — Expected: "Invalid input"

3. **Two positive, one negative**: (2.0, -1.0, 3.0) — Expected: "Invalid input"
4. **Positive and zero**: (0.0, 1.0, 1.0) — Expected: "Invalid input"

h) Non-Positive Input Cases

Test Cases

1. **All sides negative**: (-1.0, -2.0, -3.0) — Expected: "Invalid input"
2. **One side zero**: (0.0, 2.0, 3.0) — Expected: "Invalid input"
3. **One side negative**: (1.0, 2.0, -3.0) — Expected: "Invalid input"
4. **All sides zero**: (0.0, 0.0, 0.0) — Expected: "Invalid input"