LAB 8

202201458

Srushti N Makwana

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.

1. Equivalence Partitioning (EP)

Identified Equivalence Classes:

- **Valid dates (should return a valid previous date)**:
  - Any valid date (e.g., 2nd March 2000)
- **Invalid dates (should return an error message)**:
  - Invalid month (e.g., month = 0 or month = 13)
  - Invalid day for the month (e.g., day = 32 in any month, day = 30 in February, day = 29 in non-leap years)
  - Invalid year (e.g., year < 1900 or year > 2015)

Tester Action and Input Data	Expected Outcome
`day = 1, month = 1, year = 2000`	`31, 12, 1999`
`day = 15, month = 5, year = 2000`	`14, 5, 2000`
`day = 1, month = 2, year = 2000`	`29, 2, 2000`
`day = 31, month = 12, year = 2015`	`30, 12, 2015`
`day = 0, month = 1, year = 2000`	Error message (invalid day)
`day = 32, month = 1, year = 2000`	Error message (invalid day)
`day = 1, month = 0, year = 2000`	Error message (invalid month)
`day = 1, month = 13, year = 2000`	Error message (invalid month)
`day = 1, month = 1, year = 1899`	Error message (invalid year)
`day = 1, month = 1, year = 2016`	Error message (invalid year)

2. Boundary Value Analysis (BVA)

Identified Boundary Values:

- Days at the start and end of months (1 and 31)
- Leap year boundary (February 29)
- Year boundaries (1900 and 2015)

Tester Action and Input Data	Expected Outcome
`day = 0, month = 1, year = 2000`	Error message (invalid day)
`day = 1, month = 1, year = 2000`	`31, 12, 1999`
`day = 2, month = 1, year = 2000`	`1, 1, 2000`
`day = 30, month = 1, year = 2000`	`29, 1, 2000`

`day = 31, month = 1, year = 2000`	`30, 1, 2000`
`day = 32, month = 1, year = 2000`	Error message (invalid day)
`day = 1, month = 0, year = 2000`	Error message (invalid month)
`day = 31, month = 12, year = 2000`	`30, 12, 2000`
`day = 31, month = 13, year = 2000`	Error message (invalid month)
`day = 1, month = 1, year = 1899`	Error message (invalid year)
`day = 1, month = 1, year = 1900`	`31, 12, 1899`
`day = 31, month = 12, year = 2015`	`30, 12, 2015`
`day = 31, month = 12, year = 2016`	Error message (invalid year)
`day = 15, month = 5, year = 2000`	`14, 5, 2000`
`day = 1, month = 3, year = 2000`	`29, 2, 2000`
`day = 1, month = 3, year = 2001`	`28, 2, 2001`
`day = 1, month = 4, year = 2000`	`31, 3, 2000`
`day = 1, month = 1, year = 2015`	`31, 12, 2014`
`day = 30, month = 6, year = 2015`	`29, 6, 2015`
`day = 1, month = 2, year = 1900`	`31, 1, 1900`

Program Execution and Outcome Verification

```
def is_leap_year(year):
    return (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0)

def previous_date(day, month, year):
    if year < 1900 or year > 2015 or month < 1 or month > 12 or day < 1:
        return "Invalid date"

    # Days in each month (assuming non-leap year)
    days_in_month = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]

    # Adjust February for leap years
    if is_leap_year(year):
```

```
        days_in_month[1] = 29

    if day > days_in_month[month - 1]:
        return "Invalid date"  # Invalid day for the month

    if day == 1 and month == 1:
        return (31, 12, year - 1)  # New Year transition

    if day == 1:
        month -= 1
        if month == 0:
            month = 12
            year -= 1
        day = days_in_month[month - 1]

    else:
        day -= 1

    return (day, month, year)

# Test the function with the test cases
test_cases = [
    (1, 1, 2000),      # Expected: (31, 12, 1999)
    (1, 3, 2015),      # Expected: (28, 2, 2015)
    (1, 2, 2000),      # Expected: (31, 1, 2000)
    (1, 2, 2001),      # Expected: (31, 1, 2001)
    (1, 3, 2015),      # Expected: (28, 2, 2015)
    (31, 12, 2015),    # Expected: (30, 12, 2015)
    (29, 2, 2000),     # Expected: (28, 2, 2000)
    (30, 4, 2015),     # Expected: (29, 4, 2015)
    (31, 4, 2015),     # Expected: Invalid date
    (1, 13, 2015),     # Expected: Invalid date
    (1, 2, 1899),      # Expected: Invalid date
    (32, 1, 2000),     # Expected: Invalid date
    (29, 2, 2015),     # Expected: (28, 2, 2015)
    (0, 1, 2015),      # Expected: Invalid date
    (1, 1, 2016)       # Expected: Invalid date
]

for day, month, year in test_cases:
    print(f"Input: {day}, {month}, {year} => Output: {previous_date(day, month, year)}")
```

Q.2. Programs

P1.

Equivalence Partitioning Test Cases

Test Case	Input Value	Input Array	Expected Output
1	3	[3, 1, 2, 4]	0
2	1	[3, 1, 2, 4]	1
3	4	[3, 1, 2, 4]	3
4	0	[3, 1, 2, 4]	-1
5	5	[3, 1, 2, 4]	-1
6	10	[3, 1, 2, 4]	-1

Boundary Value Analysis Test Cases

Test Case	Input Value	Input Array	Expected Output
7	1	[]	-1
8	1	[1]	0
9	2	[1]	-1
10	1	[1, 2]	0
11	2	[1, 2]	1
12	3	[1, 2]	-1
13	1	[1] + [2]*9998 + [3]	0
14	3	[1] + [2]*9998 + [3]	9999
15	4	[1] + [2]*9998 + [3]	-1

Program Execution and Outcome Verification

```
def linear_search(v, a):
    for i in range(len(a)):
        if a[i] == v:
            return i
    return -1

# Test suite
```

```
test_cases = [
    (5, [1, 2, 3, 4, 5]),        # Expected: 4 (found at index 4)
    (1, [1, 2, 3, 4, 5]),        # Expected: 0 (found at index 0)
    (6, [1, 2, 3, 4, 5]),        # Expected: -1 (not found)
    (2, [2, 2, 2, 2, 2]),        # Expected: 0 (first occurrence)
    (7, []),                     # Expected: -1 (empty array)
    (3, [3, 1, 4, 1, 5, 9]),     # Expected: 0 (found at index 0)
]

# Execute test cases
for v, array in test_cases:
    result = linear_search(v, array)
    print(f"Input: {v}, Array: {array} => Output: {result}")
```

P2.

1. Equivalence Partitioning Test Cases

Test Case	Input Value	Input Array	Expected Output
1	3	[3, 1, 2, 4]	1
2	1	[3, 1, 2, 4]	1
3	4	[3, 1, 2, 4]	1
4	2	[3, 1, 2, 4]	1
5	5	[3, 1, 2, 4]	0
6	0	[3, 1, 2, 4]	0
7	1	[1, 1, 1, 1]	4
8	1	[]	0

2. Boundary Value Analysis Test Cases

Test Case	Input Value	Input Array	Expected Output
9	1	[]	0
10	1	[1]	1

11	2	[1]	0
12	1	[1, 2]	1
13	2	[1, 2]	1
14	3	[1, 2]	0
15	1	[1] + [2]*9999	1
16	2	[1] + [2]*9999	9999

Program Execution and Outcome Verification

```
def count_item(v, a):
    count = 0
    for i in range(len(a)):
        if a[i] == v:
            count += 1
    return count

# Test suite
test_cases = [
    (3, [1, 2, 3, 4, 3, 5]),     # Expected: 2 (3 appears twice)
    (1, [1, 1, 1, 1, 1]),        # Expected: 5 (1 appears five times)
    (2, [2, 2, 2, 3, 4]),        # Expected: 3 (2 appears three times)
    (5, [1, 2, 3, 4]),           # Expected: 0 (5 does not appear)
    (0, []),                     # Expected: 0 (empty array)
    (4, [4, 4, 4, 4, 4]),        # Expected: 5 (4 appears five times)
    (10, [1, 2, 3, 4, 5, 10]),   # Expected: 1 (10 appears once)
]

# Execute test cases
for v, array in test_cases:
    result = count_item(v, array)
    print(f"Input: {v}, Array: {array} => Output: {result}")
```

P3.

1. Equivalence Partitioning Test Cases

Test Case	Input Value	Input Array	Expected Output
1	3	[1, 2, 3, 4, 5]	2

2	1	[1, 2, 3, 4, 5]	0
3	5	[1, 2, 3, 4, 5]	4
4	4	[1, 2, 3, 4, 5]	3
5	0	[1, 2, 3, 4, 5]	-1
6	6	[1, 2, 3, 4, 5]	-1
7	3	[1, 1, 1, 1, 1]	1
8	2	[]	-1

2. Boundary Value Analysis Test Cases

Test Case	Input Value	Input Array	Expected Output
9	1	[]	-1
10	1	[1]	0
11	2	[1]	-1
12	1	[1, 2]	0
13	2	[1, 2]	1
14	3	[1, 2]	-1
15	1	[1] + [2]*9999	0
16	2	[1] + [2]*9999	9999

Program Execution and Outcome Verification

```
def binary_search(v, a):
    lo, hi = 0, len(a) - 1
    while lo <= hi:
        mid = (lo + hi) // 2
        if v == a[mid]:
            return mid
        elif v < a[mid]:
            hi = mid - 1
        else:
            lo = mid + 1
    return -1
```

```
# Test suite
test_cases = [
    (5, [1, 2, 3, 4, 5]),        # Expected: 4 (found at index 4)
    (1, [1, 2, 3, 4, 5]),        # Expected: 0 (found at index 0)
    (6, [1, 2, 3, 4, 5]),        # Expected: -1 (not found)
    (2, [1, 2, 2, 2, 3]),        # Expected: 1 (first occurrence)
    (0, [1, 2, 3, 4]),           # Expected: -1 (0 does not appear)
    (3, [1, 2, 3, 4, 5, 6]),     # Expected: 2 (found at index 2)
    (10, [1, 2, 3, 4, 5, 10]),   # Expected: 5 (found at index 5)
]

# Execute test cases
for v, array in test_cases:
    result = binary_search(v, array)
    print(f"Input: {v}, Array: {array} => Output: {result}")
```

P4.

1. Equivalence Partitioning Test Cases

Test Case	Input Values (a, b, c)	Expected Output
1	(3, 3, 3)	0 (EQUILATERAL)
2	(3, 3, 2)	1 (ISOSCELES)
3	(3, 2, 3)	1 (ISOSCELES)
4	(2, 3, 3)	1 (ISOSCELES)
5	(3, 4, 5)	2 (SCALENE)
6	(5, 4, 3)	2 (SCALENE)
7	(3, 3, 4)	1 (ISOSCELES)
8	(1, 1, 2)	3 (INVALID)
9	(1, 2, 3)	3 (INVALID)
10	(0, 0, 0)	3 (INVALID)

11	(-1, -1, -1)	3 (INVALID)

2. Boundary Value Analysis Test Cases

Boundary Value Analysis focuses on values at the edges of valid and invalid ranges. Here are the test cases based on BVA:

Test Case	Input Values (a, b, c)	Expected Output
12	(1, 1, 1)	0 (EQUILATERAL)
13	(2, 2, 1)	1 (ISOSCELES)
14	(1, 2, 2)	1 (ISOSCELES)
15	(3, 4, 7)	3 (INVALID)
16	(3, 5, 2)	2 (SCALENE)
17	(3, 3, 6)	3 (INVALID)
18	(1, 0, 0)	3 (INVALID)
19	(1000000, 1000000, 1000000)	0 (EQUILATERAL)
20	(1, 1, 2)	3 (INVALID)

Program Execution and Outcome Verification

```
def triangle(a, b, c):
    if a >= b + c or b >= a + c or c >= a + b:
        return "INVALID"
    if a == b and b == c:
        return "EQUILATERAL"
    if a == b or a == c or b == c:
        return "ISOSCELES"
    return "SCALENE"

# Test suite for triangle function
triangle_test_cases = [
    (3, 3, 3),      # Expected: "EQUILATERAL"
    (3, 4, 5),      # Expected: "SCALENE"
    (2, 2, 3),      # Expected: "ISOSCELES"
```

```
    (1, 1, 2),        # Expected: "INVALID"
    (5, 5, 10),       # Expected: "INVALID"
    (0, 1, 1),        # Expected: "INVALID"
    (3, 3, 2),        # Expected: "ISOSCELES"
]

# Execute test cases for triangle function
for a, b, c in triangle_test_cases:
    result = triangle(a, b, c)
    print(f"Input: ({a}, {b}, {c}) => Output: {result}")
```

P5.

1. Equivalence Partitioning Test Cases

Equivalence Partitioning divides the input data into valid and invalid classes. Here are the test cases based on EP:

Test Case	Input Values (s1, s2)	Expected Output
1	("abc", "abcdef")	TRUE
2	("abc", "ab")	FALSE
3	("abc", "abcxyz")	TRUE
4	("abcd", "abc")	FALSE
5	("", "abc")	TRUE
6	("abc", "")	FALSE
7	("abc", "ABC")	FALSE
8	("", "")	TRUE

2. Boundary Value Analysis Test Cases

Boundary Value Analysis focuses on values at the edges of valid and invalid ranges. Here are the test cases based on BVA:

Test Case	Input Values (s1, s2)	Expected Output
9	("a", "a")	TRUE

10	("a", "b")	FALSE
11	("a", "aa")	TRUE
12	("aa", "a")	FALSE
13	("abc", "abc")	TRUE
14	("abc", "abcd")	TRUE
15	("abcd", "abcde")	FALSE
16	("abc", "ab")	FALSE

Program Execution and Outcome Verification

```
def prefix(s1, s2):
    if len(s1) > len(s2):
        return False
    for i in range(len(s1)):
        if s1[i] != s2[i]:
            return False
    return True

# Test suite for prefix function
prefix_test_cases = [
    ("hello", "hello world"),     # Expected: True (s1 is a prefix of s2)
    ("hello", "world hello"),     # Expected: False (s1 is not a prefix of s2)
    ("hi", "hi there"),           # Expected: True (s1 is a prefix of s2)
    ("hi", "hello"),              # Expected: False (s1 is not a prefix of s2)
    ("", "any string"),           # Expected: True (empty string is a prefix)
    ("not", ""),                  # Expected: False (not a prefix of empty string)
]

# Execute test cases for prefix function
for s1, s2 in prefix_test_cases:
    result = prefix(s1, s2)
    print(f"Input: ({s1}, {s2}) => Output: {result}")
```

P6.

a) Identify the Equivalence Classes

1. **Valid Triangles:**
   - Equilateral Triangle: A=B=CA = B = CA=B=C
   - Isosceles Triangle: Two sides equal (e.g., A=BA = BA=B, A=CA = CA=C, or B=CB = CB=C)
   - Scalene Triangle: All sides different (e.g., A≠B≠CA \neq B \neq CA=B=C)
   - Right Triangle: A2+B2=C2A^2 + B^2 = C^2A2+B2=C2 (one right angle)
2. **Invalid Triangles:**
   - Non-triangle: A+B≤CA + B \leq CA+B≤C or any side is non-positive
   - Non-positive values: A≤0A \leq 0A≤0, B≤0B \leq 0B≤0, or C≤0C \leq 0C≤0

b) Identify Test Cases to Cover Equivalence Classes

Test Case	Input Values (A, B, C)	Expected Output	Equivalence Class
1	(3.0, 3.0, 3.0)	"Equilateral"	Equilateral Triangle
2	(3.0, 3.0, 2.0)	"Isosceles"	Isosceles Triangle
3	(3.0, 4.0, 5.0)	"Scalene"	Scalene Triangle
4	(5.0, 12.0, 13.0)	"Right angled"	Right Triangle
5	(1.0, 2.0, 3.0)	"Not a triangle"	Non-triangle
6	(0.0, 1.0, 2.0)	"Not a triangle"	Non-positive input
7	(-1.0, 1.0, 1.0)	"Not a triangle"	Non-positive input

c) Boundary Conditions for Scalene Triangle (A + B > C)

Test Case	Input Values (A, B, C)	Expected Output
8	(3.0, 4.0, 6.0)	"Scalene"
9	(3.0, 4.0, 7.0)	"Not a triangle"
10	(3.0, 4.0, 5.0)	"Scalene"

d) Boundary Condition for Isosceles Triangle (A = C)

Test Case	Input Values (A, B, C)	Expected Output
11	(5.0, 5.0, 3.0)	"Isosceles"
12	(3.0, 5.0, 3.0)	"Isosceles"
13	(3.0, 4.0, 3.0)	"Isosceles"

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

Test Case	Input Values (A, B, C)	Expected Output
14	(4.0, 4.0, 4.0)	"Equilateral"
15	(1.0, 1.0, 1.0)	"Equilateral"

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

Test Case	Input Values (A, B, C)	Expected Output
16	(3.0, 4.0, 5.0)	"Right angled"
17	(5.0, 12.0, 13.0)	"Right angled"
18	(8.0, 15.0, 17.0)	"Right angled"
19	(1.0, 1.0, 1.414)	"Right angled"

g) Test Cases for Non-Triangle

Test Case	Input Values (A, B, C)	Expected Output
20	(1.0, 2.0, 3.0)	"Not a triangle"
21	(1.0, 1.0, 2.0)	"Not a triangle"
22	(3.0, 3.0, 7.0)	"Not a triangle"

h) Test Cases for Non-Positive Input

Test Case	Input Values (A, B, C)	Expected Output

23	(0.0, 1.0, 1.0)	"Not a triangle"
24	(-1.0, 1.0, 1.0)	"Not a triangle"
25	(1.0, -1.0, 1.0)	"Not a triangle"
26	(1.0, 1.0, -1.0)	"Not a triangle"