LAB8

Maahi Shah - 202201419

Q1. Problem Definition

The program will take three inputs: day, month, and year. It should validate these inputs and return the previous date or an error message for invalid dates.

Equivalence Partitioning

1. **Valid Dates**: Combinations of day, month, and year that form valid dates.
2. **Invalid Dates**: Combinations that do not conform to the rules (e.g., out-of-range values).

Boundary Value Analysis

1. **Lower Boundaries**: The lowest valid values for day, month, and year.
2. **Upper Boundaries**: The highest valid values for day, month, and year.
3. **Transition Values**: Values that are just inside or outside of valid ranges.

Test Cases

Here are the test cases organized into equivalence classes and boundary values:

Tester Action and Input Data	Expected Outcome	Category
Previous date with input (1, 1, 1900)	(31, 12, 1899)	Valid Date
Previous date with input (1, 2, 2000)	(31, 1, 2000)	Valid Date
Previous date with input (1, 3, 2015)	(29, 2, 2015)	Valid Date
Previous date with input (1, 4, 2000)	(31, 3, 2000)	Valid Date
Previous date with input (1, 5, 2015)	(30, 4, 2015)	Valid Date
Previous date with input (1, 12, 2015)	(30, 11, 2015)	Valid Date
Previous date with input (1, 6, 1900)	(31, 5, 1900)	Valid Date
Previous date with input (1, 13, 2000)	An Error message	Invalid Month
Previous date with input (32, 1, 2000)	An Error message	Invalid Day

Previous date with input (1, 1, 1899)	An Error message	Invalid Year
Previous date with input (1, 2, 2016)	An Error message	Invalid Year
Previous date with input (0, 1, 2000)	An Error message	Invalid Day
Previous date with input (1, 0, 2000)	An Error message	Invalid Month
Previous date with input (1, 1, 2015)	(31, 12, 2014)	Valid Date
Previous date with input (29, 2, 2016)	(28, 2, 2016)	Valid Date (Leap Year)
Previous date with input (1, 2, 2015)	(31, 1, 2015)	Valid Date
Previous date with input (1, 2, 2001)	(31, 1, 2001)	Valid Date

1. Previous Date Calculation

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

    # Logic for calculating the previous date
    # Assuming the implementation is completed here...
```

```
# Test Cases

print(previous_date(1, 1, 1900))   # Expected: (31, 12, 1899)

print(previous_date(1, 2, 2000))   # Expected: (31, 1, 2000)

print(previous_date(1, 13, 2000))  # Expected: An Error message
(Invalid Month)

print(previous_date(32, 1, 2000))  # Expected: An Error message
(Invalid Day)

print(previous_date(1, 1, 1899))   # Expected: An Error message
(Invalid Year)
```

Equivalence Partitioning Test Cases

Category	Input (day, month, year)	Expected Outcome
Valid Date	(1, 1, 1900)	(31, 12, 1899)
Valid Date	(1, 2, 2000)	(31, 1, 2000)
Valid Date	(1, 3, 2015)	(29, 2, 2015)
Valid Date	(1, 4, 2000)	(31, 3, 2000)
Valid Date	(1, 5, 2015)	(30, 4, 2015)
Valid Date	(1, 12, 2015)	(30, 11, 2015)
Valid Date	(1, 6, 1900)	(31, 5, 1900)

Invalid Month	(1, 13, 2000)	An Error message
Invalid Day	(32, 1, 2000)	An Error message
Invalid Year	(1, 1, 1899)	An Error message
Invalid Year	(1, 2, 2016)	An Error message
Invalid Day	(0, 1, 2000)	An Error message
Invalid Month	(1, 0, 2000)	An Error message

Boundary Value Analysis Test Cases

Boundary Category	Input (day, month, year)	Expected Outcome
Lower Boundary Valid	(1, 1, 1900)	(31, 12, 1899)
Lower Boundary Invalid	(1, 1, 1899)	An Error message
Lower Boundary Invalid	(1, 0, 2000)	An Error message
Lower Boundary Invalid	(0, 1, 2000)	An Error message
Upper Boundary Valid	(1, 12, 2015)	(30, 11, 2015)
Upper Boundary Invalid	(1, 13, 2015)	An Error message
Upper Boundary Invalid	(1, 12, 2016)	An Error message
Transition to Leap Year	(29, 2, 2016)	(28, 2, 2016)
Transition from Leap Year	(1, 3, 2015)	(29, 2, 2015)
Transition from End of Month	(1, 4, 2000)	(31, 3, 2000)

Q2.
P1.

2. Linear Search

```
#include <stdio.h>
```

```
int linearSearch(int v, int a[], int length) {
    if (a == NULL) {
        printf("Error: Null array provided.\n");
        return -1; // Handle error appropriately
    }

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

// Test Cases
int main() {
    int arr1[] = {1, 2, 3, 4, 5};
    printf("%d\n", linearSearch(3, arr1, 5));  // Expected: 2 (3 found
at index 2)
    printf("%d\n", linearSearch(5, arr1, 4));  // Expected: -1 (5 not
found)
    printf("%d\n", linearSearch(3, NULL, 0));  // Expected: Error
message
    return 0;
}
```

Test Cases for `linearSearch`

Equivalence Partitioning

Category	Input (value, array)	Expected Outcome
Valid Search	(3, (int[]){1, 2, 3, 4, 5}, 5)	`2` (3 found at index 2)

Valid Search	(1, (int[]){1, 1, 1, 1}, 4)	`0` (1 found at index 0)
Not Found	(5, (int[]){1, 2, 3, 4}, 4)	`-1` (5 not found)
Empty Array	(3, (int[]){}, 0)	`-1` (empty array)
Null Array	(3, NULL, 0)	Error message

Boundary Value Analysis

Boundary Category	Input (value, array)	Expected Outcome
Single Element	(1, (int[]){1}, 1)	`0` (1 found at index 0)
Not Found Single Element	(2, (int[]){1}, 1)	`-1` (2 not found)
Multiple Elements	(3, (int[]){1, 2, 3, 4, 5}, 5)	`2` (3 found at index 2)
First Element	(1, (int[]){1, 2, 3}, 3)	`0` (1 found at index 0)
Last Element	(3, (int[]){1, 2, 3}, 3)	`2` (3 found at index 2)

P2. Count Items

```
#include <stdio.h>

int countItem(int v, int a[], int length) {
    // Check if the array is NULL or if the length is non-positive
    if (a == NULL) {
        printf("Error: Null array provided.\n");
        return -1; // Handle error appropriately
    }
    if (length <= 0) {
        printf("Error: Invalid length provided.\n");
        return -1; // Handle error appropriately
    }
```

```c
    int count = 0;
    for (int i = 0; i < length; i++) {
        if (a[i] == v) {
            count++;
        }
    }
    return count;
}

int main() {
    // Test cases
    int array1[] = {1, 2, 3, 1, 4, 1};
    int array2[] = {5, 5, 5, 5, 5};
    int array3[] = {0, 1, 2, 3, 4, 5};

    // Test case 1
    printf("Test case 1: %d\n", countItem(1, array1, 6)); // Expected
output: 3 (three 1s in array1)

    // Test case 2
    printf("Test case 2: %d\n", countItem(5, array2, 5)); // Expected
output: 5 (five 5s in array2)

    // Test case 3
    printf("Test case 3: %d\n", countItem(2, array3, 6)); // Expected
output: 1 (one 2 in array3)

    // Test case 4: Testing with a null array
    printf("Test case 4: %d\n", countItem(1, NULL, 6)); // Expected
output: -1 (error for null array)

    // Test case 5: Testing with a zero length
    printf("Test case 5: %d\n", countItem(1, array1, 0)); // Expected
output: -1 (error for invalid length)

    return 0;
}
```

Test Cases for `countItem`

Normal Test Cases

Category	Input (value, array)	Expected Outcome
Normal Case	(3, (int[]){1, 2, 3, 4, 5}, 5)	1 (3 appears once)
Normal Case	(1, (int[]){1, 1, 1, 1}, 4)	4 (1 appears four times)
Normal Case	(5, (int[]){1, 2, 3, 4}, 4)	`0` (5 not found)
Normal Case	(0, (int[]){0, 0, 0, 0}, 4)	4 (0 appears four times)
Normal Case	(2, (int[]){2, 3, 2, 4, 2}, 5)	3 (2 appears three times)

Equivalence Partitioning Test Cases

Category	Input (value, array)	Expected Outcome
Valid Search	(3, (int[]){1, 2, 3, 4, 5}, 5)	1 (3 appears once)
Valid Search	(1, (int[]){1, 1, 1, 1}, 4)	4 (1 appears four times)
Not Found	(5, (int[]){1, 2, 3, 4}, 4)	`0` (5 not found)
Empty Array	(3, (int[]){}, 0)	`0` (empty array)
Null Array	(3, NULL, 0)	Error message

Boundary Value Analysis Test Cases

Boundary Category	Input (value, array)	Expected Outcome
Single Element	(1, (int[]){1}, 1)	1 (1 appears once)
Not Found Single Element	(2, (int[]){1}, 1)	`0` (2 not found)
Multiple Elements	(3, (int[]){1, 2, 3, 4, 5}, 5)	1 (3 appears once)

All Elements Same	(5, (int[]){5, 5, 5, 5}, 4)	4 (5 appears four times)
All Elements Different	(1, (int[]){2, 3, 4}, 3)	0 (1 not found)

3. Binary Search

```
#include <stdio.h>

int binarySearch(int v, int a[], int length) {
    if (a == NULL || length <= 0) {
        printf("Error: Null or empty array provided.\n");
        return -1; // Handle error appropriately
    }

    int lo = 0, hi = length - 1;

    while (lo <= hi) {
        int mid = (lo + hi) / 2;
        if (v == a[mid]) {
            return mid; // Return the index of the found value
        } else if (v < a[mid]) {
            hi = mid - 1;
        } else {
            lo = mid + 1;
        }
    }
    return -1; // Value not found
}

// Test Cases
int main() {
    int arr2[] = {1, 2, 3, 4, 5};
    printf("%d\n", binarySearch(3, arr2, 5)); // Expected: 2 (3 found
at index 2)
```

```
    printf("%d\n", binarySearch(6, arr2, 5)); // Expected: -1 (6 not found)
    return 0;
}
```

Test Cases for `binarySearch`

Normal Test Cases

Category	Input (value, array)	Expected Outcome
Normal Case	(3, (int[]){1, 2, 3, 4, 5}, 5)	`2` (3 found at index 2)
Normal Case	(1, (int[]){1, 2, 3, 4, 5}, 5)	`0` (1 found at index 0)
Normal Case	(5, (int[]){1, 2, 3, 4, 5}, 5)	`4` (5 found at index 4)
Normal Case	(6, (int[]){1, 2, 3, 4, 5}, 5)	`-1` (6 not found)
Normal Case	(2, (int[]){1, 2, 2, 2, 3}, 5)	`1` (first 2 found at index 1)

Equivalence Partitioning Test Cases

Category	Input (value, array)	Expected Outcome
Valid Search	(3, (int[]){1, 2, 3, 4, 5}, 5)	`2` (3 found at index 2)
Valid Search	(1, (int[]){1, 1, 1, 1, 1}, 5)	`0` (1 found at index 0)
Not Found	(5, (int[]){1, 2, 3, 4}, 4)	`-1` (5 not found)
Empty Array	(3, (int[]){}, 0)	`-1` (empty array)
Null Array	(3, NULL, 0)	Error message

Boundary Value Analysis Test Cases

Boundary Category	Input (value, array)	Expected Outcome

Single Element	(1, (int[]){1}, 1)	`0` (1 found at index 0)
Not Found Single Element	(2, (int[]){1}, 1)	`-1` (2 not found)
Multiple Elements	(3, (int[]){1, 2, 3, 4, 5}, 5)	`2` (3 found at index 2)
First Element	(1, (int[]){1, 2, 3}, 3)	`0` (1 found at index 0)
Last Element	(3, (int[]){1, 2, 3}, 3)	`2` (3 found at index 2)

P4.Problem Definition

The `triangle` function takes three integer parameters representing the lengths of the sides of a triangle. It returns:

- `0` for equilateral (all sides equal)
- `1` for isosceles (two sides equal)
- `2` for scalene (all sides different)
- `3` for invalid (the sides cannot form a triangle)

Triangle Classification

```c
#include <stdio.h>

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

int triangle(int a, int b, int c) {
    if (a <= 0 || b <= 0 || c <= 0) {
        return INVALID; // Handle negative or zero lengths
    }
```

```c
    if (a >= b + c || b >= a + c || c >= a + b) {
        return INVALID; // Check triangle inequality
    }
    if (a == b && b == c) {
        return EQUILATERAL; // All sides equal
    }
    if (a == b || a == c || b == c) {
        return ISOSCELES; // Two sides equal
    }
    return SCALENE; // All sides different
}

// Test Cases
int main() {
    printf("%d\n", triangle(3, 3, 3)); // Expected: 0 (Equilateral)
    printf("%d\n", triangle(3, 3, 4)); // Expected: 1 (Isosceles)
    printf("%d\n", triangle(1, 2, 3)); // Expected: 3 (Invalid)
    return 0;
}
```

Test Cases for `triangle`

Normal Test Cases

Category	Input (a, b, c)	Expected Outcome
Equilateral Triangle	(3, 3, 3)	`0` (equilateral)
Isosceles Triangle	(3, 3, 4)	`1` (isosceles)
Isosceles Triangle	(4, 3, 3)	`1` (isosceles)
Scalene Triangle	(3, 4, 5)	`2` (scalene)
Invalid Triangle	(1, 2, 3)	`3` (invalid)

Equivalence Partitioning Test Cases

Category	Input (a, b, c)	Expected Outcome

Valid Equilateral	(5, 5, 5)	`0` (equilateral)
Valid Isosceles	(2, 2, 3)	`1` (isosceles)
Valid Scalene	(2, 3, 4)	`2` (scalene)
Invalid (Zero Length)	(0, 3, 4)	`3` (invalid)
Invalid (Negative)	(-1, 2, 2)	`3` (invalid)
Invalid Triangle	(5, 1, 3)	`3` (invalid)

Boundary Value Analysis Test Cases

Boundary Category	Input (a, b, c)	Expected Outcome
Valid Triangle	(1, 1, 1)	`0` (equilateral)
Valid Triangle	(1, 1, 2)	`1` (isosceles)
Invalid Triangle	(1, 1, 3)	`3` (invalid)
Invalid Triangle	(2, 2, 4)	`3` (invalid)
Valid Scalene	(3, 4, 5)	`2` (scalene)
Valid Scalene	(5, 3, 4)	`2` (scalene)

P5.

5. Prefix Function

```
java
Copy code
public class PrefixTest {
    public static boolean prefix(String s1, String s2) {
        if (s1.length() > s2.length()) {
            return false; // s1 cannot be a prefix if longer
        }
```

```
        for (int i = 0; i < s1.length(); i++) {
            if (s1.charAt(i) != s2.charAt(i)) {
                return false; // Found a mismatch
            }
        }
        return true; // All characters matched
    }

    public static void main(String[] args) {
        System.out.println(prefix("abc", "abcde")); // Expected: true
        System.out.println(prefix("abc", "ab"));    // Expected: false
    }
}
```

Test Cases for `prefix`

Normal Test Cases

Category	Input (s1, s2)	Expected Outcome
Exact Match	("abc", "abcde")	`true`
Partial Match	("ab", "abcde")	`true`
No Match	("abc", "ab")	`false`
Different Characters	("abc", "def")	`false`
Empty Prefix	("", "abcde")	`true`
Empty String Check	("abc", "")	`false`

Equivalence Partitioning Test Cases

Category	Input (s1, s2)	Expected Outcome
Valid Prefix	("abc", "abcde")	`true`
Valid Prefix	("ab", "abcd")	`true`

Not a Prefix	("abc", "abcd")	`false`
Empty Prefix	("", "abcde")	`true`
Longer s1	("abcde", "abc")	`false`

Boundary Value Analysis Test Cases

Boundary Category	Input (s1, s2)	Expected Outcome
Same Length	("abc", "abc")	`true`
Same Length, No Match	("abc", "abd")	`false`
One Character Match	("a", "abc")	`true`
One Character No Match	("b", "abc")	`false`
Single Character Prefix	("a", "a")	`true`
Both Empty	("", "")	`true`

P6. a) Identify the Equivalence Classes

1. **Valid Triangles:**
   - **Equilateral Triangle:** All sides are equal (A = B = C).
   - **Isosceles Triangle:** Two sides are equal (A = B ≠ C, A = C ≠ B, B = C ≠ A).
   - **Scalene Triangle:** All sides are different (A ≠ B, B ≠ C, A ≠ C).
   - **Right-Angled Triangle:** Fulfills the Pythagorean theorem ($A^2 + B^2 = C^2$ or any permutation).
2. **Invalid Triangles:**
   - **Non-Triangle:** Fails the triangle inequality (A + B ≤ C, A + C ≤ B, B + C ≤ A).
   - **Non-Positive Inputs:** Any side length is less than or equal to zero (A ≤ 0, B ≤ 0, C ≤ 0).

b) Identify Test Cases for Equivalence Classes

Equivalence Class	Test Case Input (A, B, C)	Expected Outcome

Equilateral Triangle	(3.0, 3.0, 3.0)	Equilateral
Isosceles Triangle	(3.0, 3.0, 4.0)	Isosceles
Isosceles Triangle	(4.0, 3.0, 3.0)	Isosceles
Scalene Triangle	(3.0, 4.0, 5.0)	Scalene
Right-Angled Triangle	(3.0, 4.0, 5.0)	Right-angled
Non-Triangle	(1.0, 2.0, 3.0)	Non-triangle
Non-Positive Input	(-1.0, 2.0, 2.0)	Non-triangle
Non-Positive Input	(0.0, 2.0, 2.0)	Non-triangle

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

Test Case Input (A, B, C)	Expected Outcome
(2.0, 3.0, 4.0)	Scalene
(3.0, 4.0, 5.0)	Scalene
(1.0, 2.0, 2.9)	Scalene

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

Test Case Input (A, B, C)	Expected Outcome
(3.0, 4.0, 3.0)	Isosceles
(2.0, 1.0, 2.0)	Isosceles

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

Test Case Input (A, B, C)	Expected Outcome
(3.0, 3.0, 3.0)	Equilateral
(1.0, 1.0, 1.0)	Equilateral

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

Test Case Input (A, B, C)	Expected Outcome
(3.0, 4.0, 5.0)	Right-angled
(5.0, 12.0, 13.0)	Right-angled

g) Non-Triangle Case Test Cases

Test Case Input (A, B, C)	Expected Outcome
(1.0, 1.0, 3.0)	Non-triangle
(2.0, 2.0, 5.0)	Non-triangle
(2.0, 1.0, 2.0)	Non-triangle

h) Non-Positive Input Test Cases

Test Case Input (A, B, C)	Expected Outcome
(0.0, 2.0, 2.0)	Non-triangle
(-1.0, 1.0, 1.0)	Non-triangle
(1.0, 0.0, 1.0)	Non-triangle
(1.0, 1.0, -1.0)	Non-triangle