

IT314

Software Engineering

Lab 8

Functional Testing (Black-Box)



Name: Maulik Kansara

StudentId: 202201442

Lab Group: Group 5

Q.1. Consider a program for determining the previous date. Its input is triple of day, month and year with the following ranges  $1 \leq \text{month} \leq 12$ ,  $1 \leq \text{day} \leq 31$ ,  $1900 \leq \text{year} \leq 2015$ . 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.

**Answer:**

**Equivalence Classes:**

Valid Input Classes:

- Day:  $1 \leq \text{day} \leq 31$
- Month:  $1 \leq \text{month} \leq 12$
- Year:  $1900 \leq \text{year} \leq 2015$

Invalid Input Classes:

- Day:  $\text{day} < 1$  or  $\text{day} > 31$
- Month:  $\text{month} < 1$  or  $\text{month} > 12$
- Year:  $\text{year} < 1900$  or  $\text{year} > 2015$

**Boundary Classes:**

Valid Boundaries:

- Day: Minimum = 1, Maximum = 31
- Month: Minimum = 1, Maximum = 12
- Year: Minimum = 1900, Maximum = 2015

Invalid Boundaries:

- Day: Minimum - 1 (0), Maximum + 1 (32)

- Month: Minimum - 1 (0), Maximum + 1 (13)
- Year: Minimum - 1 (1899), Maximum + 1 (2016)

**Test Cases:**

Test Case	Day	Month	Year	Expected Result	Description
1	1	1	1900	Previous date (31-12-1899)	Minimum valid values for day, month, and year
2	15	7	2005	Previous date (14-7-2005)	Valid values between minimum and maximum
3	31	12	2015	Previous date (30-12-2015)	Maximum valid values for day, month, and year
4	0	5	2010	Invalid date	Invalid day (minimum - 1)
5	32	8	2012	Invalid date	Invalid day (maximum + 1)
6	30	4	1980	Previous date (29-4-1980)	Maximum valid day in a 30-day month
7	1	3	2000	Previous date (29-2-2000)	Minimum valid day in a leap year,

					rollover to February
8	1	5	2010	Previous date (30-4-2010)	Minimum valid day in May, rollover to April
9	1	1	2016	Invalid date	Year exceeding maximum valid value
10	15	0	2010	Invalid date	Invalid month (minimum - 1)
11	15	13	2000	Invalid date	Invalid month (maximum + 1)
12	29	2	1900	Invalid date	Invalid leap year handling (1900 is not a leap year)

**Equivalence Partitioning Test Cases:** 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12

**Boundary Value Analysis Test Cases:** 1, 3, 4, 5, 6, 7, 9, 10, 11, 12

**C++ Code:**

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

// Function to check if a year is a leap year
bool isLeapYear(int year) {
    if (year % 4 == 0) {
        if (year % 100 == 0) {
            if (year % 400 == 0) return true;
            else return false;
        }
    }
}
```

```

    }
    return true;
}
return false;
}

// Function to get the number of days in a month
int getDaysInMonth(int month, int year) {
    switch (month) {
        case 1: case 3: case 5: case 7: case 8: case 10: case 12:
            return 31;
        case 4: case 6: case 9: case 11:
            return 30;
        case 2:
            return (isLeapYear(year) ? 29 : 28);
        default:
            return -1; // Invalid month
    }
}

// Function to calculate the previous date
bool getPreviousDate(int day, int month, int year, int &prevDay, int
&prevMonth, int &prevYear) {
    // Handle invalid dates
    if (year < 1900 || year > 2015 || month < 1 || month > 12 || day < 1
|| day > getDaysInMonth(month, year)) {
        return false; // Invalid date
    }

    // If it's the first day of the month, go to the last day of the
previous month
    if (day == 1) {
        if (month == 1) {
            prevDay = 31;
            prevMonth = 12;
            prevYear = year - 1;
        } else {
            prevMonth = month - 1;
            prevDay = getDaysInMonth(prevMonth, year);
            prevYear = year;
        }
    }
}

```

```

    }
    } else {
        // Otherwise, just subtract one day
        prevDay = day - 1;
        prevMonth = month;
        prevYear = year;
    }
    return true;
}

// Function to execute and print results for each test case
void runTestCase(int testCaseID, int day, int month, int year, string
expectedResult) {
    int prevDay, prevMonth, prevYear;
    bool result = getPreviousDate(day, month, year, prevDay, prevMonth,
prevYear);

    if (!result) {
        cout << "Test Case " << testCaseID << ": Invalid date - Expected:
" << expectedResult << endl;
    } else {
        cout << "Test Case " << testCaseID << ": Previous Date is " <<
prevDay << "-" << prevMonth << "-" << prevYear
        << " - Expected: " << expectedResult << endl;
    }
}

int main() {
    // Running the test cases
    runTestCase(1, 1, 1, 1900, "Previous date (31-12-1899)");
    runTestCase(2, 15, 7, 2005, "Previous date (14-7-2005)");
    runTestCase(3, 31, 12, 2015, "Previous date (30-12-2015)");
    runTestCase(4, 0, 5, 2010, "Invalid date");
    runTestCase(5, 32, 8, 2012, "Invalid date");
    runTestCase(6, 30, 4, 1980, "Previous date (29-4-1980)");
    runTestCase(7, 1, 3, 2000, "Previous date (29-2-2000)");
    runTestCase(8, 1, 5, 2010, "Previous date (30-4-2010)");
    runTestCase(9, 1, 1, 2016, "Invalid date");
    runTestCase(10, 15, 0, 2010, "Invalid date");
    runTestCase(11, 15, 13, 2000, "Invalid date");
}

```

```

runTestCase(12, 29, 2, 1900, "Invalid date");

return 0;
}

```

## Output:

The code output matches with Expected Output.

```

6.cpp > main()
58 void runTestCase(int testCaseID, int day, int month, int year, string expectedResult) {
59     cout << "Test Case " << testCaseID << " : Invalid date - Expected: " << expectedResult << endl;
64 } else {
65     cout << "Test Case " << testCaseID << " : Previous Date is " << prevDay << "-" << prevMonth << "-" << prevYear
66         << " - Expected: " << expectedResult << endl;
67 }
68 }
69
70 int main() {
71     // Running the test cases
72     runTestCase(1, 1, 1, 1900, "Previous date (31-12-1899)");
73     runTestCase(2, 15, 7, 2005, "Previous date (14-7-2005)");
74     runTestCase(3, 31, 12, 2015, "Previous date (30-12-2015)");
75     runTestCase(4, 0, 5, 2010, "Invalid date");
76     runTestCase(5, 32, 8, 2012, "Invalid date");
77     runTestCase(6, 30, 4, 1980, "Previous date (29-4-1980)");
78     runTestCase(7, 1, 3, 2000, "Previous date (29-2-2000)");
79     runTestCase(8, 1, 5, 2010, "Previous date (30-4-2010)");
80     runTestCase(9, 1, 1, 2016, "Invalid date");
81     runTestCase(10, 15, 0, 2010, "Invalid date");
82     runTestCase(11, 15, 13, 2000, "Invalid date");
83     runTestCase(12, 29, 2, 1900, "Invalid date");
84
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  COMMENTS
Test Case 1: Previous Date is 31-12-1899 - Expected: Previous date (31-12-1899)
Test Case 2: Previous Date is 14-7-2005 - Expected: Previous date (14-7-2005)
Test Case 3: Previous Date is 30-12-2015 - Expected: Previous date (30-12-2015)
Test Case 4: Invalid date - Expected: Invalid date
Test Case 5: Invalid date - Expected: Invalid date
Test Case 6: Previous Date is 29-4-1980 - Expected: Previous date (29-4-1980)
Test Case 7: Previous Date is 29-2-2000 - Expected: Previous date (29-2-2000)
Test Case 8: Previous Date is 30-4-2010 - Expected: Previous date (30-4-2010)
Test Case 9: Invalid date - Expected: Invalid date
Test Case 10: Invalid date - Expected: Invalid date
Test Case 11: Invalid date - Expected: Invalid date
Test Case 12: Invalid date - Expected: Invalid date

```

## Q.2. 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.

```

int linearSearch(int v, int a[])
{
    int i = 0;
    while (i < a.length)
    {
        if (a[i] == v)
            return(i);
    }
}

```

```
i++;  
}  
return (-1);  
}
```

**Answer:**

### Equivalence Class:

Valid Inputs:

- Integer value  $v$  exists in the array.
- Integer value  $v$  does not exist in the array.
- The array contains one element, which may or may not match  $v$ .
- The array is empty.

Invalid Inputs:

- Non-integer value (e.g., character, string, floating-point).
- Array containing non-integer types (e.g., floating-point numbers, characters).

### Test Cases:

Test Case	$v$ (value)	Array ( $a[]$ )	Expected Result	Description
1	3	{1, 2, 3, 4, 5}	2	$v$ found in the middle of the array
2	10	{10,20,30,40,50}	0	$v$ found at the first index
3	50	{10, 20, 30, 40, 50}	4	$v$ found at the last index
4	100	{10,20,30,40,50}	-1	$v$ is not in the array
5	0	{}	-1	Searching in an empty array, returns -1
6	1	{1}	0	Array contains only one element, which matches $v$



7	'a'	{1}	Error / Invalid Input	Searching with a non-integer (char) value, should result in error
8	3.14	{1,2,3}	Error / Invalid Input	Searching with a floating-point value, should result in error
9	"test"	{1,2}	Error / Invalid Input	Array contains floating-point values, should result in error
10	3	{1.1, 2.2, 3.3}	Error / Invalid Input	Array contains floating-point values, should result in error
11	3	{'a', 'b', 'c'}	Error / Invalid Input	Array contains characters instead of integers, should result in error

**Equivalence Partitioning Test Cases:** 1,2,3,4,5,6,7,8,9,10,11

**Boundary Value Analysis Test Cases:** 5,6,7,8,9,10,11

**C++ Code:**

```
#include <iostream>
#include <vector>
#include <string>
#include <variant>
#include <typeinfo>

// Type alias for test case input, allowing int or other types
using TestInput = std::variant<int, char, double, std::string>;

// Function to perform linear search
```

```

int linearSearch(int v, const std::vector<int>& a) {
    for (int i = 0; i < a.size(); i++) {
        if (a[i] == v) {
            return i; // Return the index if found
        }
    }
    return -1; // Return -1 if not found
}

void testLinearSearch() {
    struct TestCase {
        TestInput v; // The search value, can be int or other types
        std::vector<int> a; // The array
        int expected; // Expected result
        std::string description; // Description of the test case
    };

    std::vector<TestCase> testCases = {
        {3, {1, 2, 3, 4, 5}, 2, "v found in the middle of the array"},
        {10, {10, 20, 30, 40, 50}, 0, "v found at the first index"},
        {50, {10, 20, 30, 40, 50}, 4, "v found at the last index"},
        {100, {10, 20, 30, 40, 50}, -1, "v is not in the array"},
        {0, {}, -1, "Searching in an empty array, returns -1"},
        {1, {1}, 0, "Array contains only one element, which matches v"},
        {'a', {}, -1, "Error / Invalid Input (character)"}, // Invalid input
        {3.14, {}, -1, "Error / Invalid Input (floating-point)"}, // Invalid input
        {"test", {}, -1, "Error / Invalid Input (string)"}, // Invalid input
    };

    for (const auto& testCase : testCases) {
        // Check if the input is a valid integer before performing the search
        if (std::holds_alternative<int>(testCase.v)) {
            int result = linearSearch(std::get<int>(testCase.v), testCase.a);
            std::cout << "Test Case: " << testCase.description << "\n";
            std::cout << "Expected: " << testCase.expected << ", Got: " << result << "\n\n";
        } else {
            // Handle invalid inputs
            std::cout << "Test Case: " << testCase.description << "\n";
        }
    }
}

```

```

        std::cout << "Expected: Error / Invalid Input, Got: Invalid Input\n\n";
    }
}

int main() {
    testLinearSearch();
    return 0;
}

```

## Output:

The code output matches with the expected output.

```

20 void testLinearSearch() {
25     std::string description; // Description of the test case
26 };
27
28     std::vector<TestCase> testCases = {
29         {3, {1, 2, 3, 4, 5}, 2, "v found in the middle of the array"},
30         {10, {10, 20, 30, 40, 50}, 0, "v found at the first index"},
31         {50, {10, 20, 30, 40, 50}, 4, "v found at the last index"},
32         {100, {10, 20, 30, 40, 50}, -1, "v is not in the array"},
33         {0, {}, -1, "Searching in an empty array, returns -1"},
34         {1, {1}, 0, "Array contains only one element, which matches v"},
35         {'a', {}, -1, "Error / Invalid Input (character)"}, // Invalid input
36         {3.14, {}, -1, "Error / Invalid Input (floating-point)"}, // Invalid input
37         {"test", {}, -1, "Error / Invalid Input (string)"}, // Invalid input
38     };
39 }

```

Test Case: v found at the first index  
Expected: 0, Got: 0

Test Case: v found at the last index  
Expected: 4, Got: 4

Test Case: v is not in the array  
Expected: -1, Got: -1

Test Case: Searching in an empty array, returns -1  
Expected: -1, Got: -1

Test Case: Array contains only one element, which matches v  
Expected: 0, Got: 0

Test Case: Error / Invalid Input (character)  
Expected: Error / Invalid Input, Got: Invalid Input

Test Case: Error / Invalid Input (Floating-point)

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

```

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

```

}

**Answer:**

**Equivalence Class:**

Valid Inputs:

- Integer value v exists in the array multiple times.
- Integer value v exists in the array only once.
- Integer value v does not exist in the array.
- The array contains one element, which may or may not match v.
- The array is empty.

Invalid Inputs:

- Non-integer value (e.g., character, string, floating-point).
- Array containing non-integer types (e.g., floating-point numbers, characters).

**Test Cases:**

Test Case	v (value to search)	Array (a[])	Expected Result	Description
1	3	{1, 2, 3, 4, 5, 3, 3}	3	3 appears 3 times in the array
2	10	{10, 20, 30, 40, 50}	1	10 appears 1 time in the array
3	100	{10, 20, 30, 40, 50}	0	100 does not appear in the array
4	0	{}	0	Searching in an empty array, returns 0
5	'a'	{1,2,3}	Error / Invalid Input	Searching with a non-integer (char) value, should result in error
6	3.14	{1,2}	Error / Invalid	Searching with a

			Input	floating-point value, should result in error
7	"Test"	{1,2}	Error / Invalid Input	Searching with a string value, should result in error
8	3	{1.1, 2.2, 3.3}	Error / Invalid Input	Array contains floating-point values, should result in error
9	1	{'a', 'b', 'c'}	Error / Invalid Input	Array contains characters instead of integers, should result in error

**Equivalence Partitioning Test Cases:** 1,2,3,4,5,6,7,8,9

**Boundary Value Analysis Test Cases:** 4,5,6,7,8,9

**C++ Code:**

```
#include <iostream>
#include <vector>
#include <variant>
#include <string>

// Type alias for test case input, allowing int or other types
using TestInput = std::variant<int, char, double, std::string>;

// Function to count occurrences of a value in an array
int countItem(int v, const std::vector<int>& a) {
    int count = 0;
    for (int i = 0; i < a.size(); i++) {
        if (a[i] == v) {
            count++;
        }
    }
    return count;
}
```

```

void testCountItem() {
    struct TestCase {
        TestInput v; // The value to search, can be int or other types
        std::vector<int> a; // The array
        int expected; // Expected result or -1 for invalid inputs
        std::string description; // Description of the test case
        int testCaseNumber; // Test case number
    };

    std::vector<TestCase> testCases = {
        {3, {1, 2, 3, 4, 5, 3, 3}, 3, "3 appears 3 times in the array", 1},
        {10, {10, 20, 30, 40, 50}, 1, "10 appears 1 time in the array", 2},
        {100, {10, 20, 30, 40, 50}, 0, "100 does not appear in the array", 3},
        {0, {}, 0, "Searching in an empty array, returns 0", 4},
        {'a', {}, -1, "Error / Invalid Input (character)", 5}, // Invalid input
        {3.14, {}, -1, "Error / Invalid Input (floating-point)", 6}, // Invalid input
        {"Test", {}, -1, "Error / Invalid Input (string)", 7}, // Invalid input
        {1, {'a', 'b', 'c'}, -1, "Error / Invalid Input (array of chars)", 8}, // Invalid input
    };

    for (const auto& testCase : testCases) {
        std::cout << "Testcase " << testCase.testCaseNumber << ": " <<
testCase.description << "\n";

        // Check if the input is a valid integer before performing the count
        if (std::holds_alternative<int>(testCase.v)) {
            int result = countItem(std::get<int>(testCase.v), testCase.a);
            std::cout << "Expected: " << testCase.expected << ", Got: " << result << "\n\n";
        } else {
            // Handle invalid inputs
            std::cout << "Expected: Error / Invalid Input, Got: Invalid Input\n\n";
        }
    }
}

int main() {
    testCountItem();
    return 0;
}

```

```
}
```

## Output:

The expected output match the code output

```
20 void testCountItem() {
25     std::string description; // Description of the test case
26     int testcaseNumber; // Test case number
27 }
28
29 std::vector<TestCase> testCases = {
30     {3, {1, 2, 3, 4, 5, 3, 3}, 3, "3 appears 3 times in the array", 1},
31     {10, {10, 20, 30, 40, 50}, 1, "10 appears 1 time in the array", 2},
32     {100, {10, 20, 30, 40, 50}, 0, "100 does not appear in the array", 3},
33     {0, {}, 0, "Searching in an empty array, returns 0", 4},
34     {'a', {}, -1, "Error / Invalid Input (character)", 5}, // Invalid input
35     {3.14, {}, -1, "Error / Invalid Input (floating-point)", 6}, // Invalid input
36     {"Test", {}, -1, "Error / Invalid Input (string)", 7}, // Invalid input
37     {1, {'a', 'b', 'c'}, -1, "Error / Invalid Input (array of chars)", 8}, // Invalid input
38 };
--
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS COMMENTS

Test Case 1: 3 appears 3 times in the array  
Expected: 3, Got: 3

Test Case 2: 10 appears 1 time in the array  
Expected: 1, Got: 1

Test Case 3: 100 does not appear in the array  
Expected: 0, Got: 0

Test Case 4: Searching in an empty array, returns 0  
Expected: 0, Got: 0

Test Case 5: Error / Invalid Input (character)  
Expected: Error / Invalid Input, Got: Invalid Input

Test Case 6: Error / Invalid Input (floating-point)  
Expected: Error / Invalid Input, Got: Invalid Input

Test Case 7: Error / Invalid Input (string)  
Expected: Error / Invalid Input, Got: Invalid Input

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. Assumption: the elements in the array `a` are sorted in non-decreasing order.

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

```

}
return(-1);
}

```

**Answer:**

**Equivalence Class:**

Valid Inputs:

- Integer value v exists in the array (multiple occurrences).
- Integer value v exists in the array (only one occurrence).
- Integer value v does not exist in the array.
- The array is empty.
- The array contains duplicate values of v.

Invalid Inputs:

- Non-integer value (e.g., character, string, floating-point).
- Array containing non-integer types (e.g., floating-point numbers, characters).

**Test cases:**

Test Case	v (value to search)	Array (a[])	Expected Result	Description
1	3	{1,2,3,4,5}	2	3 found at index 2
2	0	{1,2,3,4,5}	-1	0 is not in the array
3	3	{}	-1	Searching in an empty array, returns -1
4	'a'	{1,2,3}	Error/ Invalid Input	Searching with a non-integer (char) value, should result in error
5	3.14	{1,2,3}	Error/ Invalid Input	Searching with a floating-point value, should result in error
6	"test"	{1,2,3}	Error/Invalid	Searching with a



			Input	string value, should result in error
7	3	{1,2,3.3}	Error/ Invalid Input	Array contains a floating-point number, should result in error
8	6	{1,6,6,6}	1	6 found at index 1, first occurrence is considered
9	1	{'a','b','c'}	Error/ Invalid Input	Array contains character, should result in error

**Equivalence Partitioning Test Cases:** 1,2,3,4,5,6,7,8,9

**Boundary Value Analysis Test Cases:** 2,3,4,5,6,7,8,9

**C++ Code:**

```
#include <iostream>
#include <vector>
#include <variant>
#include <string>

// Function to perform binary search
int binarySearch(int v, const std::vector<int>& a) {
    int lo = 0;
    int hi = a.size() - 1;

    while (lo <= hi) {
        int mid = (lo + hi) / 2;

        if (v == a[mid]) {
            return mid; // Return the index where v is found
        } else if (v < a[mid]) {
            hi = mid - 1; // Search in the left half
        } else {
            lo = mid + 1; // Search in the right half
        }
    }
}
```

```

    }
    return -1; // v not found
}

void testBinarySearch() {
    struct TestCase {
        std::variant<int, char, double, std::string> v; // The value to search
        std::vector<int> a; // The array
        int expected; // Expected result
        std::string description; // Description of the test case
    };

    std::vector<TestCase> testCases = {
        {3, {1, 2, 3, 4, 5}, 2, "3 found at index 2"},
        {0, {1, 2, 3, 4, 5}, -1, "0 is not in the array"},
        {3, {}, -1, "Searching in an empty array, returns -1"},
        {'a', {1, 2, 3}, -1, "Error / Invalid Input (character)"}, // Invalid input
        {3.14, {1, 2, 3}, -1, "Error / Invalid Input (floating-point)"}, // Invalid input
        {"test", {1, 2, 3}, -1, "Error / Invalid Input (string)"}, // Invalid input
        {4, {1, 2, 3, 3}, -1, "Error / Invalid Input (array of floats)"}, // Invalid input
        {6, {1, 6, 6, 6}, 1, "6 found at index 1, first occurrence is considered"},
        {1, {'a', 'b', 'c'}, -1, "Error / Invalid Input (array of chars)"}, // Invalid input
    };

    for (const auto& testCase : testCases) {
        // Check if the input is a valid integer before performing the search
        if (std::holds_alternative<int>(testCase.v)) {
            int result = binarySearch(std::get<int>(testCase.v), testCase.a);
            std::cout << "Test Case: " << testCase.description << "\n";
            std::cout << "Expected: " << testCase.expected << ", Got: " << result << "\n\n";
        } else {
            // Handle invalid inputs
            std::cout << "Test Case: " << testCase.description << "\n";
            std::cout << "Expected: Error / Invalid Input, Got: Invalid Input\n\n";
        }
    }
}

```

```
int main() {
    testBinarySearch();
    return 0;
}
```

## Output:

The Code Output matches with expected Output.

```
7  int binarySearch(int v, const std::vector<int>& a) {
8      int lo = 0;
9      int hi = a.size() - 1;
10
11     while (lo <= hi) {
12         int mid = (lo + hi) / 2;
13
14         if (v == a[mid]) {
15             return mid; // Return the index where v is found
16         } else if (v < a[mid]) {
17             hi = mid - 1; // Search in the left half
18         } else {
19             lo = mid + 1; // Search in the right half
20         }
21     }
22 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS COMMENTS

```
Test Case: 3 found at index 2
Expected: 2, Got: 2

Test Case: 0 is not in the array
Expected: -1, Got: -1

Test Case: Searching in an empty array, returns -1
Expected: -1, Got: -1

Test Case: Error / Invalid Input (character)
Expected: Error / Invalid Input, Got: Invalid Input

Test Case: Error / Invalid Input (floating-point)
Expected: Error / Invalid Input, Got: Invalid Input

Test Case: Error / Invalid Input (string)
Expected: Error / Invalid Input, Got: Invalid Input

Test Case: Error / Invalid Input (array of floats)
Expected: -1, Got: -1
```

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).

```
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 >= b+c || b >= a+c || c >= a+b)
        return(INVALID);
    if (a == b && b == c)
```

```

return(EQUILATERAL);
if (a == b || a == c || b == c)
return(ISOSCELES);
return(SCALENE);
}

```

**Answer:**

### Equivalence Class

Valid Inputs:

- All sides are equal (Equilateral).
- Two sides are equal (Isosceles).
- All sides are different (Scalene).
- Valid lengths that do not form a triangle (Invalid).

Invalid Inputs:

- At least one side is negative (Invalid).
- At least one side is zero (Invalid).
- Non-integer values (e.g., characters).

**Test Cases:**

Test Case	a	b	c	Expected Result	Description
1	3	3	3	EQUILATERAL (0)	All sides equal, triangle is equilateral
2	5	5	3	ISOSCELES (1)	Two sides equal, triangle is isosceles
3	4	5	6	SCALENE (2)	All sides different, triangle is scalene
4	2	2	5	INVALID (3)	Sides do not form a triangle
5	-1	1	1	INVALID (3)	Invalid length

					(negative), cannot form a triangle
6	5	5	0	INVALID (3)	Invalid length (zero), cannot form a triangle
7	'a'	1	2	INVALID(3)	Invalid length, cannot be character

**Equivalence Partitioning Test Cases:** 1,2,3,4,5,6,7

**Boundary Value Analysis Test Cases:** 4,5,6,7

**C++ Code:**

```
#include <iostream>
#include <vector>    // Include the vector header
#include <variant>
#include <string>

// Constants for triangle types
const int EQUILATERAL = 0;
const int ISOSCELES = 1;
const int SCALENE = 2;
const int INVALID = 3;

// Function to determine the type of triangle
int triangle(int a, int b, int c) {
    if (a <= 0 || b <= 0 || c <= 0 || (a >= b + c || b >= a + c || c >= a + b)) {
        return INVALID; // Invalid triangle condition
    }
    if (a == b && b == c) {
        return EQUILATERAL; // Equilateral triangle
    }
    if (a == b || a == c || b == c) {
        return ISOSCELES; // Isosceles triangle
    }
    return SCALENE; // Scalene triangle
}
```

```

}

void testTriangle() {
    struct TestCase {
        std::variant<int, char> a; // Side a
        std::variant<int, char> b; // Side b
        std::variant<int, char> c; // Side c
        int expected; // Expected result
        std::string description; // Description of the test case
    };

    std::vector<TestCase> testCases = {
        {3, 3, 3, EQUILATERAL, "All sides equal, triangle is equilateral"},
        {5, 5, 3, ISOSCELES, "Two sides equal, triangle is isosceles"},
        {4, 5, 6, SCALENE, "All sides different, triangle is scalene"},
        {2, 2, 5, INVALID, "Sides do not form a triangle"},
        {-1, 1, 1, INVALID, "Invalid length (negative), cannot form a triangle"},
        {5, 5, 0, INVALID, "Invalid length (zero), cannot form a triangle"},
        {'a', 1, 2, INVALID, "Invalid length, cannot be character"}
    };

    for (const auto& testCase : testCases) {
        // Check if the inputs are valid integers before performing the triangle check
        if (std::holds_alternative<int>(testCase.a) && std::holds_alternative<int>(testCase.b) &&
std::holds_alternative<int>(testCase.c)) {
            int result = triangle(std::get<int>(testCase.a), std::get<int>(testCase.b),
std::get<int>(testCase.c));
            std::cout << "Test Case: " << testCase.description << "\n";
            std::cout << "Expected: " << testCase.expected << ", Got: " << result << "\n\n";
        } else {
            // Handle invalid inputs
            std::cout << "Test Case: " << testCase.description << "\n";
            std::cout << "Expected: INVALID, Got: Invalid Input\n\n";
        }
    }
}

int main() {

```

```
testTriangle();
return 0;
}
```

## Output:

The code output matches with expected output.

```
1 #include <iostream>
2 #include <vector>    // Include the vector header
3 #include <variant>
4 #include <string>
5
6 // Constants for triangle types
7 const int EQUILATERAL = 0;
8 const int ISOSCELES = 1;
9 const int SCALENE = 2;
10 const int INVALID = 3;
11
12 // Function to determine the type of triangle
13 int triangle(int a, int b, int c) {
14     if (a <= 0 || b <= 0 || c <= 0 || (a >= b + c || b >= a + c || c >= a + b)) {
15         return INVALID; // Invalid triangle condition
16     }
17
18     if (a == b && b == c) return EQUILATERAL;
19     if (a == b || b == c || a == c) return ISOSCELES;
20     return SCALENE;
21 }
22
23 int main() {
24     // Test Case 1: All sides equal, triangle is equilateral
25     int a1 = 3, b1 = 3, c1 = 3;
26     int result1 = triangle(a1, b1, c1);
27     std::cout << "Test Case: All sides equal, triangle is equilateral\n";
28     std::cout << "Expected: 0, Got: " << result1 << "\n";
29
30     // Test Case 2: Two sides equal, triangle is isosceles
31     int a2 = 3, b2 = 3, c2 = 4;
32     int result2 = triangle(a2, b2, c2);
33     std::cout << "Test Case: Two sides equal, triangle is isosceles\n";
34     std::cout << "Expected: 1, Got: " << result2 << "\n";
35
36     // Test Case 3: All sides different, triangle is scalene
37     int a3 = 3, b3 = 4, c3 = 5;
38     int result3 = triangle(a3, b3, c3);
39     std::cout << "Test Case: All sides different, triangle is scalene\n";
40     std::cout << "Expected: 2, Got: " << result3 << "\n";
41
42     // Test Case 4: Sides do not form a triangle
43     int a4 = 3, b4 = 4, c4 = 8;
44     int result4 = triangle(a4, b4, c4);
45     std::cout << "Test Case: Sides do not form a triangle\n";
46     std::cout << "Expected: 3, Got: " << result4 << "\n";
47
48     // Test Case 5: Invalid length (negative), cannot form a triangle
49     int a5 = -3, b5 = 4, c5 = 5;
50     int result5 = triangle(a5, b5, c5);
51     std::cout << "Test Case: Invalid length (negative), cannot form a triangle\n";
52     std::cout << "Expected: 3, Got: " << result5 << "\n";
53
54     // Test Case 6: Invalid length (zero), cannot form a triangle
55     int a6 = 0, b6 = 4, c6 = 5;
56     int result6 = triangle(a6, b6, c6);
57     std::cout << "Test Case: Invalid length (zero), cannot form a triangle\n";
58     std::cout << "Expected: 3, Got: " << result6 << "\n";
59
60     // Test Case 7: Invalid length, cannot be character
61     int a7 = 'a', b7 = 4, c7 = 5;
62     int result7 = triangle(a7, b7, c7);
63     std::cout << "Test Case: Invalid length, cannot be character\n";
64     std::cout << "Expected: INVALID, Got: " << result7 << "\n";
65
66     return 0;
67 }
```

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).

```
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;  
}
```

**Answer:**

**Equivalence Class:**

Valid Inputs:

- s1 is a prefix of s2: Both strings have the same starting sequence.
- s1 is an empty string: An empty string is a prefix of any non-empty string.
- s2 is an empty string: If s1 is not empty, it cannot be a prefix of an empty string.
- s1 is not a prefix of s2: The strings do not share the same starting sequence

Invalid Inputs:

- Case-sensitive mismatch: Strings that are identical except for letter casing.

**Test Cases:**

Test Case	s1 (Prefix String)	s2 (Target String)	Expected Result	Description
1	"test"	"testcase"	true	s1 is prefix of s2
2	"hello"	"hello world"	true	s1 is prefix of s2
3	"abcd"	"abc"	false	s1 is not prefix of s2
4	"	"abc"	true	An empty string is prefix of any-non empty string
5	"abc"	"	false	s1 cannot be a prefix of an empty string
6	"bcd"	"abcd"	false	s1 is not prefix of s2
7	"abc"	"Abc"	false	Case-sensitive check, s1 does not match s2

**Equivalence Partitioning Test Cases:** 1,2,3,4,5,6,7



## Boundary Value Analysis Test Cases: 4,5,6,7

C++ Code:

```
#include <iostream>
#include <string>
#include <vector>

// Function to check if s1 is a prefix of s2
bool prefix(const std::string& s1, const std::string& s2) {
    if (s1.length() > s2.length()) {
        return false;
    }
    for (size_t i = 0; i < s1.length(); i++) {
        if (s1[i] != s2[i]) {
            return false;
        }
    }
    return true;
}

void testPrefix() {
    struct TestCase {
        std::string s1; // Prefix String
        std::string s2; // Target String
        bool expected; // Expected result
        std::string description; // Description of the test case
    };

    std::vector<TestCase> testCases = {
        {"test", "testcase", true, "s1 is prefix of s2"},
        {"hello", "hello world", true, "s1 is prefix of s2"},
        {"abcd", "abc", false, "s1 is not prefix of s2"},
        {"", "abc", true, "An empty string is prefix of any non-empty string"},
        {"abc", "", false, "s1 cannot be a prefix of an empty string"},
        {"bcd", "abcd", false, "s1 is not prefix of s2"},
    };
}
```

```

        {"abc", "Abc", false, "Case-sensitive check, s1 does not match s2"}
    };

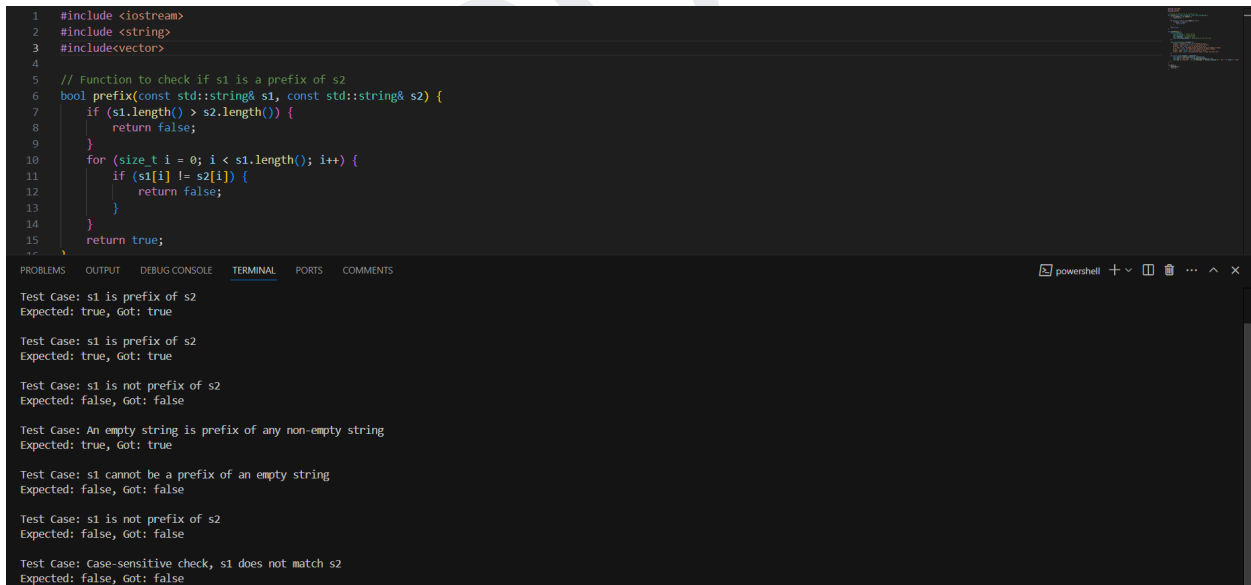
    for (const auto& testCase : testCases) {
        bool result = prefix(testCase.s1, testCase.s2);
        std::cout << "Test Case: " << testCase.description << "\n";
        std::cout << "Expected: " << std::boolalpha << testCase.expected << ", Got: " <<
result << "\n\n";
    }
}

int main() {
    testPrefix();
    return 0;
}

```

## Output:

The code Output matches with expected output.



```

1  #include <iostream>
2  #include <string>
3  #include <vector>
4
5  // Function to check if s1 is a prefix of s2
6  bool prefix(const std::string& s1, const std::string& s2) {
7      if (s1.length() > s2.length()) {
8          return false;
9      }
10     for (size_t i = 0; i < s1.length(); i++) {
11         if (s1[i] != s2[i]) {
12             return false;
13         }
14     }
15     return true;
16 }

```

Test Case: s1 is prefix of s2  
Expected: true, Got: true

Test Case: s1 is prefix of s2  
Expected: true, Got: true

Test Case: s1 is not prefix of s2  
Expected: false, Got: false

Test Case: An empty string is prefix of any non-empty string  
Expected: true, Got: true

Test Case: s1 cannot be a prefix of an empty string  
Expected: false, Got: false

Test Case: s1 is not prefix of s2  
Expected: false, Got: false

Test Case: Case-sensitive check, s1 does not match s2  
Expected: false, Got: 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) Identify the equivalence classes for the system

Valid Triangle Classes:

- Equilateral Triangle: All three sides are equal ( $A = B = C$ ).
- Isosceles Triangle: Exactly two sides are equal ( $A = B \neq C$ ,  $A = C \neq B$ , or  $B = C \neq A$ ).
- Scalene Triangle: All sides are different ( $A \neq B$ ,  $B \neq C$ ,  $A \neq C$ ).
- Right-Angled Triangle: Satisfies the Pythagorean theorem ( $A^2 + B^2 = C^2$ ).

Invalid Triangle Classes:

- Non-triangle: The sum of the lengths of any two sides must be greater than the length of the third side ( $A + B \leq C$ ,  $A + C \leq B$ , or  $B + C \leq A$ ).
- Non-positive Input: One or more sides are less than or equal to zero ( $A \leq 0$ ,  $B \leq 0$ ,  $C \leq 0$ ).

b) Identify test cases to cover the identified equivalence classes. Also, explicitly mention which test case would cover which equivalence class. (Hint: you must need to be ensure that the identified set of test cases cover all identified equivalence classes)

Test Case	A	B	C	Expected Output	Equivalence Class Covered
1	3	3	3	Equilateral Triangle	Equilateral Triangle
2	5	5	3	Isosceles Triangle	Isosceles Triangle
3	4	5	6	Scalene Triangle	Scalene Triangle
4	5	12	13	Right-Angled Triangle	Right-Angled Triangle
5	1	2	3	Not a Triangle	Non-Triangle

6	0	1	1	Invalid Input	Invalid Input
7	-1	2	2	Invalid Input	Non-Positive Input

c) For the boundary condition  $A + B > C$  case (scalene triangle), identify test cases to verify the Boundary.

Test Case	A	B	C	Expected Result	Description
1	4	5	6	Scalene Triangle	Valid scalene triangle
2	6	5	4	scalene triangle	Valid scalene triangle
3	3	4	8	Not Triangle	Fails the triangle inequality
4	-1	1	1	Invalid Input	Non-Positive Input
5	1	1	0	Invalid Input	Non-Positive Input

d) For the boundary condition  $A = C$  case (isosceles triangle), identify test cases to verify the Boundary.

Test Case	A	B	C	Expected Result	Description
1	3	3	5	Isosceles Triangle	Two Sides Equal
2	-1	1	1	Invalid Input	Non-Positive Input
3	1	1	0	Invalid Input	Non-Positive
4	3	4	8	Not Triangle	Fails the triangle inequality
5	1.1	1.1	2	Isosceles Triangle	Two Sides Equal

e) For the boundary condition  $A = B = C$  case (equilateral triangle), identify test cases to verify the boundary.

Test Case	A	B	C	Expected Result	Description
1	2	2	2	Equilateral Triangle	All sides equal
2	1.6	1.6	1.6	Equilateral Triangle	All sides equal
3	3	4	8	Not Triangle	Fails the triangle inequality
4	1	1	0	Invalid Input	Non-Positive

f) For the boundary condition  $A^2 + B^2 = C^2$  case (right-angle triangle), identify test cases to verify the boundary.

Test Case	A	B	C	Expected Result	Description
1	3	4	5	Right Angled Triangle	Fulfills Pythagorean theorem
2	1	1	0	Invalid Input	Non-Positive
3	3	4	8	Not Triangle	Fails the triangle inequality

g) For the non-triangle case, identify test cases to explore the boundary.

Test Case	A	B	C	Expected Result	Description
1	1	2	3	Not a Triangle	Fails triangle inequality
2	3	4	8	Not Triangle	Fails the triangle inequality

h) For non-positive input, identify test points.

Test Case	A	B	C	Expected Result	Description
1	-1	1	1	Invalid Input	Non-Positive Input
2	1	1	0	Invalid Input	Non-Positive