LAB_8_202201457

Kalp Shah

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?

A1.

1. Equivalence Classes for Input Parameters

A. Month (1 to 12)

- **Valid Classes**:
  - Valid month: {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}
- **Invalid Classes**:
  - Below valid range: {0, -1, -5, -100}
  - Above valid range: {13, 14, 20, 100}

B. Day (1 to 31)

- **Valid Classes**:
  - Valid day for each month:
    - January: {1 to 31}
    - February: {1 to 28} (or 29 for leap years)
    - March: {1 to 31}
    - April: {1 to 30}
    - May: {1 to 31}
    - June: {1 to 30}
    - July: {1 to 31}
    - August: {1 to 31}
    - September: {1 to 30}
    - October: {1 to 31}
    - November: {1 to 30}
    - December: {1 to 31}
- **Invalid Classes**:
  - Day below valid range: {0, -1, -5, -31}
  - Day above valid range (for month with fewer days):

- February: {29, 30, 31} (for non-leap years)
- April, June, September, November: {31}

C. Year (1900 to 2015)

- **Valid Classes**:
  - Valid year: {1900, 1901, ..., 2015}
- **Invalid Classes**:
  - Below valid range: {1899, 1800, 0, -100}
  - Above valid range: {2016, 2020, 2500}

2. Test Case Design

Based on these equivalence classes, we can now define test cases. Each test case will consist of a triplet of (day, month, year) and the expected output (previous date or invalid date).

Test Case Scenarios

1. **Valid Date Cases**
   - **(1, 1, 1900)**: Expected Output: (31, 12, 1899) (last day of previous year)
   - **(1, 3, 2000)**: Expected Output: (29, 2, 2000) (Leap year case)
   - **(1, 5, 2015)**: Expected Output: (30, 4, 2015)
   - **(1, 12, 2015)**: Expected Output: (30, 11, 2015)
   - **(29, 2, 2012)**: Expected Output: (28, 2, 2012) (Valid leap year)
2. **Invalid Date Cases**
   - **(0, 1, 1900)**: Expected Output: "Invalid Date" (day out of range)
   - **(32, 1, 2000)**: Expected Output: "Invalid Date" (day out of range for January)
   - **(1, 13, 2000)**: Expected Output: "Invalid Date" (month out of range)
   - **(29, 2, 2013)**: Expected Output: "Invalid Date" (non-leap year case)
   - **(31, 4, 2015)**: Expected Output: "Invalid Date" (April has 30 days)
   - **(1, 1, 1899)**: Expected Output: "Invalid Date" (year out of range)
   - **(1, 1, 2016)**: Expected Output: "Invalid Date" (year out of range)
3. **Edge Cases**
   - **(1, 2, 1900)**: Expected Output: (31, 1, 1900) (First day of February)
   - **(29, 2, 2000)**: Expected Output: (28, 2, 2000) (Leap year transition)
   - **(1, 3, 2015)**: Expected Output: (28, 2, 2015) (Non-leap year transition)

est Cases for Equivalence Partitioning

Test Case ID	Input (day, month, year)	Expected Outcome

TC1	**(1, 1, 1900)**	**(31, 12, 1899)**
TC2	**(1, 2, 2000)**	**(31, 1, 2000)**
TC3	**(1, 5, 2015)**	**(30, 4, 2015)**
TC4	**(1, 3, 2012)**	**(29, 2, 2012)**
TC5	**(29, 2, 2013)**	**"Invalid Date"**
TC6	**(32, 1, 2000)**	**"Invalid Date"**
TC7	**(31, 4, 2015)**	**"Invalid Date"**
TC8	**(1, 13, 2000)**	**"Invalid Date"**
TC9	**(1, 1, 1899)**	**"Invalid Date"**
TC10	**(1, 1, 2016)**	**"Invalid Date"**

Q2

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);
}
```

A1

Test Case ID	Tester Action and Input Data	Expected Outcome	Description
EP-1	`linearSearch(3, [1, 2, 3, 4])`	`2`	This test case verifies a valid input scenario where the value `3` is present in the array. The function should return the index of the first occurrence, which is `2`.
EP-2	`linearSearch(5, [1, 2, 3, 4])`	`-1`	This case checks the behavior when searching for a value that does not exist in the array. The expected outcome is `-1`, indicating that the search value is absent.
EP-3	`linearSearch(2, [1, 2, 2, 3])`	`1`	Here, we are testing for multiple occurrences of the same value. The first occurrence of `2` is at index `1`, so the function should return this index.

EP-4	`linearSearch(1, [])`	`-1`	This test examines the edge case of an empty array. Since there are no elements to search through, the expected result is `-1`, indicating the absence of the value.
BVA-1	`linearSearch(1, [1, 2, 3, 4])`	`0`	This boundary case tests for the value at the start of the array. The function should return `0`, which is the index of the first element that matches the search value.
BVA-2	`linearSearch(4, [1, 2, 3, 4])`	`3`	Here, we check for a value at the end of the array. The function should correctly return `3`, the index of the last element that matches the search value.
BVA-3	`linearSearch(0, [1, 2, 3, 4])`	`-1`	This test explores a value that is just below the smallest element in the array. The expected outcome is `-1`, confirming that `0` is not present in the array.
BVA-4	`linearSearch(5, [1, 2, 3, 4])`	`-1`	This case examines a value just above the largest element. The expected result is `-1`, indicating that `5` is not found in the array.

Detailed Explanation of Each Test Case

1. **Equivalence Partitioning (EP)**:
   - **EP-1** checks for a valid case where the search value exists in the array, ensuring the function correctly identifies the index of the first occurrence.
   - **EP-2** involves a negative test case where the search value is not present, which should prompt the function to return `-1`.
   - **EP-3** tests a scenario with multiple occurrences of the search value, confirming that the function returns the index of the first occurrence.
   - **EP-4** examines how the function handles an empty array, which is crucial for robustness in scenarios where no data is provided.
2. **Boundary Value Analysis (BVA)**:
   - **BVA-1** checks the boundary condition where the search value is the first element in the array, validating that the function can access the starting index correctly.
   - **BVA-2** looks at the scenario where the search value is the last element, ensuring the function can accurately return the correct index.
   - **BVA-3** tests the edge condition just below the smallest value in the array, which helps to confirm that the function correctly handles values outside the array's bounds.

- **BVA-4** examines the case where the search value is just above the largest element, testing the function's ability to return `-1` for non-existent values.

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);
}
```

A2.

Equivalence Partitioning (EP)

Test Case ID	Tester Action and Input Data	Expected Outcome	Description
EP-1	`countItem(2, [1, 2, 2, 3, 4])`	`2`	This test case checks a valid input where the value `2` appears twice in the array. The function should return `2`, the total count of occurrences.
EP-2	`countItem(5, [1, 2, 2, 3, 4])`	`0`	This case tests for a value that does not exist in the array. The expected outcome is `0`, indicating that `5` is absent.
EP-3	`countItem(2, [1, 2, 2, 2, 3])`	`3`	This test checks for multiple occurrences of the same value. The function should return `3`, reflecting the total count of `2`s in the array.
EP-4	`countItem(1, [])`	`0`	This test examines the case of an empty array. Since there are no elements to count, the expected result is `0`, indicating that the value is absent.

Boundary Value Analysis (BVA)

Test Case ID	Tester Action and Input Data	Expected Outcome	Description
BVA-1	`countItem(1, [1, 2, 3, 4])`	`1`	This boundary case tests for the value at the beginning of the array. The function should return `1`, reflecting that `1` appears once.
BVA-2	`countItem(4, [1, 2, 3, 4])`	`1`	Here, we check for the value at the end of the array. The function should return `1`, indicating that `4` appears once.
BVA-3	`countItem(0, [1, 2, 3, 4])`	`0`	This test explores a value just below the smallest element. The expected outcome is `0`, confirming that `0` is not present in the array.
BVA-4	`countItem(5, [1, 2, 3, 4])`	`0`	This case examines a value just above the largest element. The expected result is `0`, indicating that `5` is not found in the array.

Detailed Explanation of Each Test Case

Equivalence Partitioning (EP)

1. **EP-1**: This test checks a scenario where the search value exists multiple times in the array. The function should correctly return the count of `2`, which appears twice.
2. **EP-2**: This negative test case evaluates how the function behaves when the search value is not present in the array. It confirms that the function returns `0`, indicating no occurrences.
3. **EP-3**: This test examines a case with more than two occurrences of the search value. The function should count and return `3`, as there are three `2`s in the array.
4. **EP-4**: This test evaluates the behavior of the function when provided with an empty array. The expected output is `0`, demonstrating that the function handles cases with no data correctly.

Boundary Value Analysis (BVA)

1. **BVA-1**: This test checks the situation where the value being counted is at the beginning of the array. The function should return `1`, reflecting that `1` appears once.
2. **BVA-2**: Similar to the first boundary case, this one tests the value at the end of the array. The expected output is `1`, as the value `4` is present once.

3. **BVA-3**: This test examines the case of a value just below the smallest element in the array. The expected result is `0`, ensuring that the function does not mistakenly count values that are outside the range of the array.
4. **BVA-4**: This case investigates the function's response to a value just above the largest element. The expected output is `0`, confirming that the function accurately identifies that `5` is not present.

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);
}
```

A3.

Equivalence Partitioning (EP)

Test Case ID	Tester Action and Input Data	Expected Outcome	Description
EP-1	`binarySearch(3, [1, 2, 3, 4, 5])`	`2`	This test case verifies a valid input where the value `3` is present in the array. The function should return `2`, the index of the first occurrence.
EP-2	`binarySearch(6, [1, 2, 3, 4, 5])`	`-1`	This case tests for a value that does not exist in the array. The expected outcome is `-1`, indicating that `6` is absent.

EP-3	`binarySearch(1, [1, 2, 3, 4, 5])`	`0`	Here, we test for the lowest value in the sorted array. The expected output is `0`, the index of the first occurrence of `1`.
EP-4	`binarySearch(5, [1, 2, 3, 4, 5])`	`4`	This case checks for the highest value in the sorted array. The expected output is `4`, the index of the first occurrence of `5`.
EP-5	`binarySearch(2, [1, 2, 2, 3, 4])`	`1`	This test checks for a duplicate value in the array. The function should return `1`, the index of the first occurrence of `2`.
EP-6	`binarySearch(7, [])`	`-1`	This test examines the behavior of the function with an empty array. The expected result is `-1`, indicating that there are no elements to search.

Boundary Value Analysis (BVA)

Test Case ID	Tester Action and Input Data	Expected Outcome	Description
BVA-1	`binarySearch(1, [1, 2, 3, 4, 5])`	`0`	This boundary case tests for the value at the beginning of the array. The function should return `0`, reflecting that `1` appears at the first index.
BVA-2	`binarySearch(5, [1, 2, 3, 4, 5])`	`4`	Here, we check for the value at the end of the array. The expected output is `4`, indicating that `5` appears at the last index.
BVA-3	`binarySearch(0, [1, 2, 3, 4, 5])`	`-1`	This test explores a value just below the smallest element. The expected outcome is `-1`, confirming that `0` is not present in the array.
BVA-4	`binarySearch(6, [1, 2, 3, 4, 5])`	`-1`	This case examines a value just above the largest element. The expected result is `-1`, indicating that `6` is not found in the array.

Detailed Explanation of Each Test Case

Equivalence Partitioning (EP)

1. **EP-1**: This test checks a valid case where the search value `3` is present in the ordered array. The expected outcome is `2`, which is the index of `3`.
2. **EP-2**: This negative test evaluates how the function behaves when the search value `6` is not present in the array. It should return `-1` to indicate that the value is absent.
3. **EP-3**: This case tests the lowest value present in the array, which is `1`. The function should return `0`, confirming that the first element matches the search value.
4. **EP-4**: This test checks for the highest value present in the array, which is `5`. The expected outcome is `4`, indicating that the function correctly identifies the last element.
5. **EP-5**: This test examines a case with a duplicate value in the array. The function should return `1`, the index of the first occurrence of `2`.
6. **EP-6**: This test assesses the function's response when provided with an empty array. The expected output is `-1`, demonstrating that the function handles cases with no data correctly.

Boundary Value Analysis (BVA)

1. **BVA-1**: This boundary case tests the search for the first element of the array. The function should return `0`, confirming that it can find the value at the beginning correctly.
2. **BVA-2**: This case evaluates the search for the last element of the array. The expected outcome is `4`, as `5` is the last value.
3. **BVA-3**: This test examines a value just below the smallest element in the array. The expected result is `-1`, ensuring that the function does not mistakenly find values that are outside the bounds.
4. **BVA-4**: This case investigates the function's behavior when searching for a value just above the largest element. The expected output is `-1`, confirming that the function accurately identifies that `6` is not present.

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);
}
```

A4,

Equivalence Partitioning (EP)

Test Case ID	Tester Action and Input Data	Expected Outcome	Description
EP-1	`triangle(3, 3, 3)`	`0`	This test case checks for an equilateral triangle. All sides are equal, so the function should return `EQUILATERAL` (0).
EP-2	`triangle(3, 3, 4)`	`1`	This case tests for an isosceles triangle. Two sides are equal, and one is different, so the expected outcome is `ISOSCELES` (1).
EP-3	`triangle(3, 4, 5)`	`2`	Here, we test for a scalene triangle where all sides are different. The function should return `SCALENE` (2).

EP-4	`triangle(1, 2, 3)`	3	This test examines an invalid triangle scenario where the sum of two sides equals the third. The expected result is `INVALID` (3).
EP-5	`triangle(5, 5, 10)`	3	This case tests another invalid triangle where one side is equal to the sum of the other two. The function should return `INVALID` (3).
EP-6	`triangle(-1, 2, 3)`	3	This test checks for invalid lengths (negative side). The expected outcome is `INVALID` (3).

Boundary Value Analysis (BVA)

Test Case ID	Tester Action and Input Data	Expected Outcome	Description
BVA-1	`triangle(0, 0, 0)`	3	This boundary case tests for all sides equal to zero, which is invalid. The expected outcome is `INVALID` (3).
BVA-2	`triangle(1, 1, 2)`	3	Here, we check for the minimum values that result in an invalid triangle. The expected outcome is `INVALID` (3).
BVA-3	`triangle(2, 2, 3)`	1	This case tests for an isosceles triangle with minimal valid lengths. The expected outcome is `ISOSCELES` (1).
BVA-4	`triangle(2, 3, 4)`	2	This case examines a scalene triangle with the smallest non-equal lengths. The expected outcome is `SCALENE` (2).

Detailed Explanation of Each Test Case

Equivalence Partitioning (EP)

1. **EP-1**: This test verifies the case of an equilateral triangle where all sides are equal (3, 3, 3). The function should return `EQUILATERAL` (0).
2. **EP-2**: This case checks for an isosceles triangle (3, 3, 4). The expected output is `ISOSCELES` (1) since two sides are equal.
3. **EP-3**: This test examines a scalene triangle (3, 4, 5), where all sides are different. The function should return `SCALENE` (2).

4. **EP-4**: This test evaluates an invalid triangle where the sum of two sides equals the third side (1, 2, 3). The expected result is `INVALID` (3).
5. **EP-5**: This case checks for another invalid triangle scenario (5, 5, 10) where one side equals the sum of the other two. The function should return `INVALID` (3).
6. **EP-6**: This test examines invalid lengths where one side is negative (-1, 2, 3). The expected output is `INVALID` (3), as negative lengths are not valid.

Boundary Value Analysis (BVA)

1. **BVA-1**: This boundary test checks for a triangle with all sides equal to zero (0, 0, 0), which is invalid. The expected outcome is `INVALID` (3).
2. **BVA-2**: This test checks for lengths that are at their minimal valid values but still result in an invalid triangle (1, 1, 2). The expected result is `INVALID` (3).
3. **BVA-3**: This case examines a valid isosceles triangle with the smallest valid lengths (2, 2, 3). The expected outcome is `ISOSCELES` (1).
4. **BVA-4**: This test checks for the smallest scalene triangle (2, 3, 4). The expected result is `SCALENE` (2), confirming that the function can handle minimal valid lengths.

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;
}
```

A5.

Equivalence Partitioning (EP)

Test Case ID	Tester Action and Input Data	Expected Outcome	Description
EP-1	`prefix("abc", "abcdef")`	`true`	This test case checks a valid prefix scenario where `s1` ("abc") is a prefix of `s2` ("abcdef"). The expected outcome is `true`.
EP-2	`prefix("abc", "ab")`	`false`	This case tests when `s1` is longer than `s2`, making it impossible for `s1` to be a prefix. The expected outcome is `false`.
EP-3	`prefix("abc", "abcd")`	`true`	This test checks for a valid prefix where `s1` ("abc") is exactly the start of `s2` ("abcd"). The function should return `true`.

EP-4	`prefix("abc", "xyz")`	`false`	This test checks when `s1` is not a prefix of `s2`. Here, `s1` ("abc") and `s2` ("xyz") do not match at any position. The expected outcome is `false`.
EP-5	`prefix("", "anything")`	`true`	This case checks when `s1` is an empty string, which is a valid prefix for any string `s2`. The expected outcome is `true`.
EP-6	`prefix("anything", "")`	`false`	This test examines when `s1` is longer than `s2`, which is empty. The expected outcome is `false`, as a non-empty string cannot be a prefix of an empty string.

Boundary Value Analysis (BVA)

Test Case ID	Tester Action and Input Data	Expected Outcome	Description
BVA-1	`prefix("", "")`	`true`	This boundary case checks when both strings are empty. An empty string is considered a prefix of another empty string, so the expected outcome is `true`.
BVA-2	`prefix("a", "a")`	`true`	This tests for the case where both strings are equal and single-character. The expected outcome is `true`, as they are the same.
BVA-3	`prefix("a", "ab")`	`true`	This case checks for a single-character prefix where `s1` is the first character of `s2`. The expected outcome is `true`.
BVA-4	`prefix("b", "a")`	`false`	This case examines a single-character mismatch between the strings. The expected outcome is `false`.

Detailed Explanation of Each Test Case

Equivalence Partitioning (EP)

1. **EP-1**: This test checks a valid prefix where `s1` ("abc") is indeed a prefix of `s2` ("abcdef"). The expected outcome is `true`.

2. **EP-2**: This case verifies that when `s1` is longer than `s2`, such as `s1` ("abc") and `s2` ("ab"), the function correctly returns `false`.
3. **EP-3**: This test examines a situation where `s1` is a valid prefix of `s2`. Here, `s1` ("abc") is a prefix of `s2` ("abcd"), and the expected outcome is `true`.
4. **EP-4**: This case tests when `s1` ("abc") does not match `s2` ("xyz") at any position, confirming that the function returns `false`.
5. **EP-5**: This case checks that an empty string `s1` is considered a prefix of any non-empty string `s2`, yielding an expected outcome of `true`.
6. **EP-6**: This test ensures that when `s1` is longer than `s2` (non-empty vs. empty), the function correctly returns `false`.

Boundary Value Analysis (BVA)

1. **BVA-1**: This boundary case checks when both strings are empty (`""`). Since an empty string is a prefix of another empty string, the expected outcome is `true`.
2. **BVA-2**: This test checks for two identical single-character strings. The expected outcome is `true`, as `s1` is equal to `s2`.
3. **BVA-3**: This case verifies that a single-character string (`s1` = "a") is correctly recognized as a prefix of a two-character string (`s2` = "ab"). The expected outcome is `true`.
4. **BVA-4**: This test examines the case where a single-character string (`s1` = "b") does not match the first character of `s2` (`s2` = "a"). The expected outcome is `false`.

By executing these comprehensive test cases, you can thoroughly assess the functionality and robustness of the `prefix` function, ensuring it behaves correctly across various scenarios, including normal conditions, edge cases, and situations with empty inputs.

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
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)
c) For the boundary condition A + B > C case (scalene triangle), identify test cases to verify the boundary.
d) For the boundary condition A = C case (isosceles triangle), identify test cases to verify the boundary.
e) For the boundary condition A = B = C case (equilateral triangle), identify test cases to verify the boundary.
f) For the boundary condition A2 + B2 = C2 case (right-angle triangle), identify test cases to verify
the boundary.
g) For the non-triangle case, identify test cases to explore the boundary.
h) For non-positive input, identify test points.

A6.

A) Identify the Equivalence Classes

The equivalence classes for the triangle classification program can be categorized as follows:

Equivalence Class	Description
Valid Equilateral Triangle	All three sides are equal (A = B = C).
Valid Isosceles Triangle	Two sides are equal (A = B ≠ C or A = C ≠ B or B = C ≠ A).
Valid Scalene Triangle	All three sides are different (A ≠ B, B ≠ C, A ≠ C).
Valid Right-Angled Triangle	Satisfies the condition $A^2 + B^2 = C^2$, where C is the longest side.

Invalid Triangle	Any case that does not satisfy the triangle inequality (A + B ≤ C, A + C ≤ B, or B + C ≤ A).
Non-Triangle	Any case where one or more sides are zero or negative (A ≤ 0, B ≤ 0, or C ≤ 0).

B) Identify Test Cases to Cover the Identified Equivalence Classes

Test Case ID	Input Data (A, B, C)	Expected Outcome	Covered Equivalence Class
TC-1	`3, 3, 3`	`Equilateral Triangle`	Valid Equilateral Triangle
TC-2	`5, 5, 3`	`Isosceles Triangle`	Valid Isosceles Triangle
TC-3	`3, 4, 5`	`Scalene Triangle`	Valid Scalene Triangle
TC-4	`3, 4, 6`	`Scalene Triangle`	Valid Scalene Triangle
TC-5	`5, 12, 13`	`Right-Angled Triangle`	Valid Right-Angled Triangle
TC-6	`1, 2, 3`	`Invalid Triangle`	Invalid Triangle
TC-7	`0, 4, 5`	`Non-Triangle`	Non-Triangle
TC-8	`-1, 2, 3`	`Non-Triangle`	Non-Triangle

C) Test Cases for the Boundary Condition A + B > C (Scalene Triangle)

Test Case ID	Input Data (A, B, C)	Expected Outcome	Description
BC-1	`3, 4, 5`	`Scalene Triangle`	All sides are different, satisfying A + B > C.
BC-2	`5, 6, 10`	`Invalid Triangle`	Just fails the condition A + B > C (5 + 6 = 11, 11 > 10).
BC-3	`3, 4, 6`	`Scalene Triangle`	Satisfies A + B > C (3 + 4 = 7 > 6).

D) Test Cases for the Boundary Condition A = C (Isosceles Triangle)

Test Case ID	Input Data (A, B, C)	Expected Outcome	Description
BC-4	`5, 5, 3`	`Isosceles Triangle`	Two sides are equal (A = B).
BC-5	`4, 5, 4`	`Isosceles Triangle`	Another valid isosceles triangle.
BC-6	`2, 3, 2`	`Isosceles Triangle`	Testing with smaller values, satisfying A = C.

E) Test Cases for the Boundary Condition A = B = C (Equilateral Triangle)

Test Case ID	Input Data (A, B, C)	Expected Outcome	Description
BC-7	`3, 3, 3`	`Equilateral Triangle`	All sides are equal.
BC-8	`1, 1, 1`	`Equilateral Triangle`	Smallest valid equilateral triangle.
BC-9	`10, 10, 10`	`Equilateral Triangle`	Larger valid equilateral triangle.

F) Test Cases for the Boundary Condition $A^2 + B^2 = C^2$ (Right-Angled Triangle)

Test Case ID	Input Data (A, B, C)	Expected Outcome	Description
BC-10	`3, 4, 5`	`Right-Angled Triangle`	Classic Pythagorean triple.
BC-11	`5, 12, 13`	`Right-Angled Triangle`	Another Pythagorean triple.
BC-12	`8, 15, 17`	`Right-Angled Triangle`	Larger Pythagorean triple.

G) Test Cases for Non-Triangle Cases

Test Case ID	Input Data (A, B, C)	Expected Outcome	Description
NT-1	`1, 2, 3`	`Non-Triangle`	A + B = C, which does not form a triangle.
NT-2	`1, 10, 12`	`Invalid Triangle`	Fails triangle inequality A + B < C.
NT-3	`2, 2, 5`	`Non-Triangle`	Fails triangle inequality A + B < C.

H) Test Cases for Non-Positive Input

Test Case ID	Input Data (A, B, C)	Expected Outcome	Description
NP-1	`0, 4, 5`	`Non-Triangle`	Zero length should not form a triangle.
NP-2	`-1, 2, 3`	`Non-Triangle`	Negative length should not form a triangle.
NP-3	`1, -2, 3`	`Non-Triangle`	Negative length should not form a triangle.
NP-4	`0, 0, 0`	`Non-Triangle`	All sides zero should not form a triangle.