1. Conside any software application you have to develop for your customer. Apply an appropropriate test case writing technique for numerical input.
=>
Step 1: Identify Equivalence Classes

Start by identifying the different equivalence classes for numerical input. These classes represent groups of input values that should be treated in the same way by software. For example, if developing software for a calculator application, the following equivalence classes for numerical input:

Positive integers (e.g., 1, 10, 100)
Negative integers (e.g., -1, -10, -100)
Zero (0)
Positive decimal numbers (e.g., 0.1, 1.23, 10.5)
Negative decimal numbers (e.g., -0.1, -1.23, -10.5)
Boundary values (e.g., the smallest and largest valid input values)
Invalid input (e.g., non-numeric characters, empty input)

Step 2: Write Test Cases for Each Equivalence Class
Once identified the equivalence classes, write test cases for each class. For example:

Test Case 1: Positive Integer

Input: 5
Expected Output: 5
Test Case 2: Negative Integer

Input: -3
Expected Output: -3
Test Case 3: Zero

Input: 0
Expected Output: 0
Test Case 4: Positive Decimal Number

Input: 3.14
Expected Output: 3.14
Test Case 5: Negative Decimal Number

Input: -2.5
Expected Output: -2.5
Test Case 6: Boundary Value (Smallest Valid Input)

Input: 1
Expected Output: 1
Test Case 7: Boundary Value (Largest Valid Input)

Input: 9999999
Expected Output: 9999999
Test Case 8: Invalid Input (Non-Numeric Characters)

Input: "abc"
Expected Output: Error message (e.g., "Invalid input")
Test Case 9: Invalid Input (Empty Input)

Input: ""
Expected Output: Error message (e.g., "Input is empty")

Step 3: Execute and Verify Test Cases

Execute each test case with the specified input values and verify that the actual output matches the expected output. If they match, the software passes the test case. If they don't match, there may be a defect that needs to be addressed.

2. If you have any complex decision making situation in your application, which technique you can apply for writing test cases? Give example.

When dealing with complex decision-making situations in a software application, one effective technique for writing test cases is the Decision Table Testing technique. Decision tables are a structured way to represent different combinations of conditions and their corresponding actions or outcomes. Here's how you can apply Decision Table Testing with an example:

Step 1: Identify Conditions and Actions

Identify the various conditions and actions within your complex decision-making situation. Conditions are factors that influence the outcome, while actions are the results of those conditions.

Example:

Suppose you are developing a flight booking system, and you have a complex decision-making situation for determining ticket prices. Conditions could include factors like:

- Passenger age (Adult, Child, Senior)
- Flight class (Economy, Business, First Class)
- Booking type (One-way, Round-trip)
- Membership status (Frequent Flyer, Standard)
- Departure date (Weekday, Weekend)

Actions could include determining the ticket price for each combination of conditions.

Step 2: Create the Decision Table

Create a decision table that includes all possible combinations of conditions and the corresponding actions or outcomes. Use 'X' to indicate which actions apply to each combination of conditions.

Here's a simplified example of what the decision table might look like:

Passenger Age	Flight Class	Booking Type	Membership Status	Departure Date	Ticket Price
Adult	Economy	One-way	Standard	Weekday	$200
Child	Business	Round-trip	Frequent Flyer	Weekend	$350
Senior	First Class	One-way	Standard	Weekday	$500

Step 3: Write Test Cases

For each unique combination of conditions in the decision table, write a test case. The test case should specify the input conditions and the expected output (i.e., the ticket price in this example).

Example Test Case:

Test Case 1:
- Passenger Age: Adult

- Flight Class: Economy
- Booking Type: One-way
- Membership Status: Standard
- Departure Date: Weekday
- Expected Ticket Price: $200

Step 4: Execute and Verify Test Cases

Execute each test case by providing the specified input conditions to your software and verify that the actual output matches the expected output. If they match, the software passes the test case. If they don't match, it indicates a potential issue with the decision-making logic in your application.

By using Decision Table Testing, you can systematically cover various combinations of conditions and ensure that your software handles complex decision-making situations accurately and consistently.

3. Write a pseudocode or draw a flowchart for your any functinoal requirement of your system and find different types of coverage using three test cases.

=>

Let's consider a functional requirement for a simple online shopping cart system where we want to implement the "Add Item to Cart" functionality. Here's a pseudocode for this requirement:

```
Function: AddItemToCart(item, quantity)
  Input: item (Product)
      quantity (Integer)
  Output: Success or Failure

  if item is not available in the catalog:
    return Failure (Item not found)

  if quantity <= 0:
    return Failure (Invalid quantity)

  if item is already in the cart:
    increase quantity of item in the cart
  else:
    add item to the cart with the specified quantity

  return Success (Item added to cart)
```

Now, let's identify different types of coverage and create three test cases:

Test Case 1: Basic Positive Test Case
Input:
Item: "Product A"
Quantity: 2
Expected Output:
Success (Item added to cart)

Test Case 2: Item Not Found Test Case
Input:
Item: "Product Z" (An item not available in the catalog)
Quantity: 1

Expected Output:
Failure (Item not found)

Test Case 3: Invalid Quantity Test Case
Input:
Item: "Product B"
Quantity: -5 (Invalid quantity)
Expected Output:
Failure (Invalid quantity)

Now, let's identify different types of coverage for these test cases:

Statement Coverage: This measures whether each line of code has been executed at least once. To achieve statement coverage, we need to ensure that every line in the pseudocode has been executed by at least one test case.

Branch Coverage: This measures whether each branch or decision point in the code has been tested. In our pseudocode, decision points include conditions like "item is not available in the catalog" and "quantity <= 0." We need to ensure that each of these conditions has been both true and false at least once.

Path Coverage: This is the most comprehensive coverage level, which ensures that every possible path through the code has been tested. In our pseudocode, the possible paths include success (item added to cart), failure (item not found), and failure (invalid quantity). We need to cover all these paths.

With the provided three test cases, we have achieved statement coverage for all lines of code. We have also achieved branch coverage for both the "item is not available in the catalog" and "quantity <= 0" conditions. However, path coverage requires additional test cases to cover all possible paths. To achieve full path coverage, we would need to create additional test cases to cover scenarios like adding an item that's already in the cart and other edge cases.