- **Boundary Value Analysis (BVA):**

- Boundary Value Analysis is a software testing technique used to identify defects or issues at the boundaries or limits of input ranges.
- It focuses on testing values that are on the edges or just outside the valid input domain.
- BVA is particularly effective at uncovering off-by-one errors, boundary-related problems, and issues related to data validation.

1. **Minimum Boundary:** Test with the smallest valid input values. For example, if an input field requires a number between 1 and 100, you would test it with the value 1.

2. **Minimum-1 Boundary:** Test with values just below the minimum boundary. In the same example, you would test with the value 0.

3. **Maximum Boundary:** Test with the largest valid input values. Continuing the example, you would test with the value 100.

4. **Maximum+1 Boundary:** Test with values just above the maximum boundary. For the given range, you would test with 101.

5. **Middle Value:** Test with values in the middle of the valid input range. For the range 1 to 100, this would be a value like 50.

6. **On-Boundary Values:** Test with values exactly on the boundaries. In the example, you would test with 1, 100, 0, and 101.

7. **Just Inside the Boundaries:** Test with values just inside the boundaries. For the minimum, it would be 2, and for the maximum, it would be 99.

8. **Just Outside the Boundaries:** Test with values just outside the boundaries. For the minimum, it would be -1, and for the maximum, it would be 101.

- **Boundary Interactions:** Test scenarios where multiple inputs interact at the boundaries. For example, if you have two input fields with ranges, test combinations of boundary values for both fields.

- **Null or Empty Values:** Test how the system handles null or empty values, especially if these are allowed within the input range.

- **Special or Invalid Values:** Test with values that are explicitly prohibited or considered invalid, such as characters in a numeric field or negative values in a field that should only accept positive integers.

- **Data Type Conversion:** Test the behaviour of the system when different data types are used at the boundaries. For example, if an input expects an integer, test with a floating-point number just outside the range.

- **Performance at Boundaries:** In some cases, you may want to test how the system performs or behaves when it's operating at or near its capacity or boundary limits.

- **Negative Testing:** Consider testing extreme boundary cases where inputs are well beyond the expected boundaries. This helps ensure the system doesn't behave unpredictably or crash when faced with unexpected inputs.

➢ Decision Table Testing:

- Decision Table Testing is a software testing technique that is particularly useful for testing complex business rules or logic where multiple conditions and inputs result in various outcomes.
- Decision tables are a systematic and structured way to represent these rules and generate test cases to ensure thorough coverage.

- **Components of a Decision Table:**

1. **Conditions (Inputs):** These are the factors or variables that affect the outcome of a decision. Conditions can be binary (true or false) or have multiple possible values.

2. **Actions (Outputs):** These are the possible results or actions based on the combinations of conditions. Each action corresponds to a specific set of condition values.

3. **Rules:** Rules are combinations of conditions that determine which action should be taken. Each rule represents a specific scenario or condition combination.

- **Steps to Create and Use a Decision Table:**

1. **Identify Conditions and Actions:** First, you need to identify all the relevant conditions (inputs) and actions (outputs) that the system or process will consider.

2. **Create the Table:** Construct a table with rows representing all possible combinations of condition values and columns for each condition and action.

3. **Define Rules:** Populate the decision table with rules that specify which action(s) should be taken for each combination of condition values. A "P" (positive) or "Y" (yes) is often used to indicate when a condition is met, and "N" (negative) or "N" (no) is used to indicate when it is not met.

4. **Test Case Generation:** From the decision table, you can generate test cases by selecting various combinations of conditions and corresponding expected actions. Each test case represents a unique scenario.

5. **Execute Tests:** Execute the test cases derived from the decision table against the software system or process being tested.

6. **Evaluate Results:** Compare the actual outcomes with the expected outcomes specified in the decision table.

- **Advantages of Decision Table Testing:**

1. **Coverage:** Decision table testing ensures comprehensive coverage of various condition combinations, helping to uncover hidden defects.

2. **Clarity:** Decision tables provide a clear and structured way to document complex decision logic, making it easier to understand and maintain.

3. **Reusability:** Test cases generated from decision tables can be reused as regression tests when changes are made to the software.

4. **Efficiency:** It is efficient for testing situations with a large number of possible inputs and outcomes.

- **Challenges and Considerations:**

1. **Complexity:** Decision tables can become complex when there are many conditions and actions, leading to a large number of rules and test cases.

2. **Maintenance:** As the system evolves, decision tables may need to be updated to reflect changes in business rules or logic.

3. **Validation:** It's important to validate that the conditions and actions in the decision table accurately represent the real-world rules and logic.

4. **Tooling:** There are specialized tools available for creating and managing decision tables, which can be helpful for larger and more complex systems.

➢ Use Case Testing:

- Use Case Testing is a software testing technique that focuses on validating the functionality of a software system from the perspective of an end user.
- It is particularly useful for testing how well a system meets the requirements and expectations of its intended users by examining how they interact with it in real-world scenarios.

1. Identify Use Cases:

- Start by identifying and documenting the various use cases of the software.
- A use case represents a specific interaction or scenario in which a user interacts with the system to achieve a goal or perform a specific task.

2. Understand User Scenarios:

- Thoroughly understand the typical scenarios in which users will interact with the system.
- This involves understanding the goals, actions, and expected outcomes for each use case.

3. Create Test Scenarios:

- For each use case, create test scenarios that cover different aspects of the user's interaction with the system.
- **Preconditions:** The conditions or state of the system before the use case starts.
- **User Actions:** The specific steps the user takes to interact with the system.
- **Expected Results:** The expected outcomes or behaviour of the system after the user completes the use case.

4. Design Test Cases:

- Based on the test scenarios, design test cases that specify the exact inputs, actions, and expected outcomes for each scenario.
- Test cases should be clear, concise, and unambiguous.

5. Execute Test Cases:

- Execute the test cases by following the specified steps and recording the actual results. During execution, pay attention to any discrepancies between the expected and actual outcomes.

6. **Report Defects:**

- If any discrepancies or defects are found during testing, document them in a defect tracking system.
- Include information such as the steps to reproduce the issue, the severity, and any supporting evidence.

7. **Regression Testing:**

- After defects are fixed, perform regression testing to ensure that the changes do not introduce new issues in the use cases that were previously tested and passed.

8. **Traceability:**

- Ensure that each test case can be traced back to the specific use case it is testing. This traceability helps in requirements validation and coverage analysis.

9. **Boundary Testing:**

- Consider edge cases and boundary conditions within the use cases. Test scenarios where users input values at the extreme ends of the allowed ranges.

10. **Performance and Load Testing:**

- For use cases involving heavy user interactions or data processing, consider performance and load testing to ensure the system can handle the expected load and respond within acceptable time frames.

11. **Usability Testing:**

- In addition to functional testing, consider conducting usability testing to evaluate the overall user experience, including user interface design and user-friendliness.

12. **Acceptance Testing:**

- Use case testing often aligns with user acceptance testing (UAT), where actual end users validate that the software meets their requirements and expectations.

➢ Linear Code Sequence and Jump (LCSAJ) Testing:

- Linear Code Sequence and Jump (LCSAJ) testing, also known as Linear Code Sequence and Jump Testing, is a structural white-box testing technique used primarily in software testing.
- LCSAJ testing focuses on the coverage of program paths through the source code.
- It aims to ensure that all linear code sequences (also called program paths or code paths) are executed and that jump statements (e.g., branches and loops) are adequately exercised.
- This technique helps identify and verify the correctness of different control flow paths within a program.

1. **Linear Code Sequence (LCS):** An LCS is a continuous sequence of executable statements in a program without any branching or loop structures. It represents a straight-line path through the code, from the starting point to the endpoint. LCSs can be as short as one statement or span multiple statements.

2. **Jump Statements:** Jump statements are program constructs that cause a change in the control flow. These include conditional branches (if statements), loops (for, while, do-while), and unconditional branches (goto or break statements).

3. **LCSAJ Coverage:** LCSAJ coverage measures the coverage of both linear code sequences and jumps. To achieve full LCSAJ coverage, you need to ensure that every LCS is executed at least once, and all possible branches and loops are exercised under various conditions.

4. **Test Case Design:** Designing test cases for LCSAJ coverage involves creating test inputs and scenarios that lead to the execution of all linear code sequences and exercise various paths through conditional branches and loops.

5. **Coverage Metrics:** Metrics such as LCSAJ coverage percentage are used to quantify the coverage achieved by a set of test cases. A high LCSAJ coverage percentage indicates that most program paths have been tested.

6. **Benefits:** LCSAJ testing helps in identifying logic errors, missing or incomplete code, and corner cases that might be missed with traditional testing methods. It is particularly useful for achieving high coverage in safety-critical software where thorough testing is essential.

7. **Challenges:** Achieving full LCSAJ coverage can be challenging, especially in complex software with numerous conditional statements and loops. It may require a substantial number of test cases.

8. **Automation:** Automated testing tools and code coverage analysers can assist in identifying which LCSs and jumps have been covered during testing.