Unit – 4
Software Coding and Testing

Thanki Kunal

Lecturer, Government Polytechnic-Porbandar

4.1 Coding and Code Review

Coding

- The input to the coding phase is the design document.
- During coding phase:
  - modules identified in the design document are coded according to the module specifications.
  - Unit test modules
- At the end of the design phase we have:
  - module structure (e.g. structure chart) of the system
  - module specifications:
    - data structures and algorithms for each module.
- Objective of coding phase:
  - transform design into code
  - unit test the code.

Coding Standards

- Good software development organizations require their programmers to adhere to some standard style of coding: called ***coding standards***.
- Many software development organizations: formulate their own coding standards that suits them most
- Advantages of using Coding Standards:
  - it gives a uniform appearance to the codes written by different engineers,
  - it enhances code understanding,
  - encourages good programming practices.

Coding Standards and Guidelines

- A coding standard sets out standard ways of doing several things:
  - Header information,
  - the way variables are named,
  - code is properly indented
  - maximum number of source lines allowed per function, etc.
  - Use of proper comment

Representative Coding Standards

- Contents of headers for different modules:
  - The headers of different modules should be standard for an organization.
  - The exact format for header information is usually specified.

- Header data:
  - Name of the module,
  - date on which the module was created,
  - author's name,
  - modification history,
  - synopsis of the module,
  - different functions supported, along with their input/output parameters,
  - global variables accessed/modified by the module.

Representative Coding Standards

- Rules for limiting the use of globals:
  - what types of data can be declared global and what can not.
- Naming conventions for
  - global variables,
  - local variables, and
  - constant identifiers.
- Error return conventions and exception handling mechanisms.
  - the way error and exception conditions are handled should be standard within an organization.
  - For example, when different functions encounter error conditions
  - should either return a 0 or 1 consistently.

Representative Coding Guidelines

- Do not use too clever and difficult to understand coding style:
  - Code should be easy to understand.
- Avoid obscure side effects:
  - one that is not obvious from a casual examination of the code.
  - makes later maintenance difficult.
  - The side effects of a function call include:
    - modification of parameters passed by reference,
    - modification of global variables,
    - I/O operations.
  - For example,
    - if a global variable is changed obscurely in a called module, it becomes difficult for anybody trying to understand the code.

Representative Coding Guidelines

- Do not use an identifier (variable name) for multiple purposes:
  - Programmers often use the same identifier for multiple purposes.
  - For example, some programmers use a temporary loop variable also for storing the final result.

    ```
    for(i=1;i<100;i++)
                  {… ..}
                  i=2*p*q;
                                         return(i);
    ```
  - There are several things wrong with this approach, hence should be avoided.
  - Each variable should be given a name indicating its purpose:
    - This is not possible if an identifier is used for multiple purposes.

Representative Coding Guidelines

- Code should be well-documented.
    - on the average there must be at least one comment line for every three source lines.
    - The length of any function should not exceed 10 source lines.
- Avoid Lengthy functions:
    - usually very difficult to understand
    - probably do too many different things.
- Do not use goto statements.
  - Use of go to statements:
    - make a program unstructured
    - make it very difficult to understand.

Code Review

- After a module has been coded,
  - ***code inspection*** and ***code walk through*** are carried out
  - ensures that coding standards are followed
  - helps detect as many errors as possible before testing.

- Detect as many errors as possible during inspection and walkthrough:
  - detected errors require less effort for correction
  - much higher effort needed if errors were to be detected during integration or system testing.

Code Review

- Reviewer checks for :
  - Potential flaws of code
  - Consistency with overall program design
  - Quality of comment
  - General rules of coding standard

- Two types of code review
  - Code walkthrough
  - Code inspection

Code Walk Through

- An informal code analysis technique.
  - author presents the code to peers for feedback.
  - It is more focused on learning, discussion, and identifying potential improvements rather than detecting defects.
- **Key Features**
  - **Less Formal**: No strict roles or predefined checklists.
  - **Interactive:** The author explains the code, and reviewers provide feedback.
  - **Focus on Learning**: Helps team members understand the codebase and discuss potential improvements.
  - **Minimal Preparation**: No extensive analysis before the meeting is required.
- The main objectives of the code walk through is to discover the algorithmic and logical errors in the code.

Code Walk Through

- **Process of Code Walkthrough**
  - **Preparation (Optional)** – The author may provide backg round on the code.
  - **Presentation** – The author walks the team through the code.
  - **Discussion & Feedback** – Peers provide feedback on lo gic, efficiency, and clarity.
  - **Revision** – The author incorporates the suggested changes.

Code Walk Through

- **Advantages of Code Walkthrough**

  ✔ Encourages collaboration and knowledge sharing.
  ✔ Helps new developers understand the project.
  ✔ Provides quick feedback with minimal overhead.

- **Disadvantages of Code Walkthrough**

  ✖ May miss defects due to its informal nature.
  ✖ Lacks structured defect tracking and metrics.

Code Inspection

- In contrast to code walk through,
  - Code Inspection is a formal, rigorous review process where a team thoroughly examines the code against predefined standards
  - Code inspection aims mainly at discovery of commonly made errors.
- During code inspection:
  - the code is examined for the presence of certain kinds of errors,
  - in contrast to the hand simulation of code execution done in code walk through. Standards are checked

Code Inspection

- For instance, consider:
  - classical error of writing a procedure that modifies a formal parameter
  - while the calling routine calls the procedure with a constant actual parameter.
- It is more likely that such an error will be discovered:
  - by looking for this kind of mistakes in the code,
  - rather than by simply hand simulating execution of the procedure.

Code Inspection

- **Key Features**
  - **Highly Structured**: Follows a predefined process with specific roles.
  - **Formal Process**: Uses checklists, predefined review criteria, and detailed documentation.
  - **Focus on Defects**: The primary goal is to find issues, not to discuss alternative solutions.
  - **Requires Preparation**: Reviewers must analyze the code before the inspection meeting.
  - **Metrics-Based**: Collects data on defects and the review process for future improvements.

Code Inspection

- **Roles in Code Inspection**
  - **Moderator (Facilitator)** – Organizes the review process and ensures adherence to guidelines.
  - **Author (Developer)** – Writes the code and provides necessary documentation.
  - **Reviewer (Inspector)** – Checks for defects based on c oding standards and checklists.
  - **Recorder (Scribe)** – Documents the issues found d uring the inspection.
  - **Manager (Optional)** – Oversees the review process a nd ensures compliance.

Code Inspection

- **Process of Code Inspection**
  - **Planning** – Moderator schedules the review and selects the team.
  - **Preparation** – Reviewers analyze the code and use checklists to identify defects.
  - **Inspection Meeting** – The team discusses identified issues and logs defects.
  - **Rework** – The author corrects the identified defects.
  - **Follow-up** – The moderator verifies wheth er the fixes are implemented correctly.

Code Inspection

- **Advantages of Code Inspection**
  - ✔ Detects a high number of defects early.
  - ✔ Improves software quality and maintainability.
  - ✔ Enhances compliance with coding standards.
  - ✔ Facilitates knowledge sharing among team members.
- **Disadvantages of Code Inspection**
  - ✖ Time-consuming due to its structured nature.
  - ✖ Requires significant preparation and resources.

Code Inspection

- Common Errors:
  - Use of uninitialized variables.
  - Non terminating loops.
  - Array indices out of bounds.
  - Incompatible assignments.
  - Improper storage allocation and deallocation.
  - Actual and formal parameter mismatch in procedure calls.
  - Jumps into loops.
  - Use of incorrect logical operators
    - or incorrect precedence among operators.
  - Improper modification of loop variables.
  - Comparison of equality of floating point values, etc.
- Also during code inspection,
  - adherence to coding standards is checked.

4.2 Software Documentation

Software Documentation

- When developing a software product we develop various kinds of documents :

  In addition to executable files and the source code:

  - users' manual,
  - software requirements specification (SRS) document,
  - design document, test document,
  - installation manual, etc.

- All these documents are a vital part of good software development practice.

Software Documentation

- Good documents enhance understandability and maintainability of a software product.

- Different types of software documents can be classified into:
  - Internal documentation:
    - documentation provided in the source code itself.
  - External documentation (supporting documents):
    - documentation other than those present in the source code.

Internal Documentation

- Internal documentation provided through:
  - use of meaningful variable names,
  - code indentation,
  - code structuring,
  - use of enumerated types and constant identifiers,
  - use of user-defined data types, etc.
  - module headers and comments

Internal Documentation

- Good software development organizations:
  - ensure good internal documentation
  - through coding standards and coding guidelines.
- Example of unhelpful documentation:
  - a = 10;   /* a made 10 */
- Careful experimentation suggests:
  - meaningful variable names is the most useful internal documentation.

External Documentation

- Examples of External Documentation:
  - Users' manual,
  - Software requirements specification document,
  - Design document,
  - Test documents,
  - Installation instructions, etc.

External Documentation

- A systematic software development style ensures:
  - all external documents are produced in an orderly fashion.
- An important feature of good documentation is ***consistency***.
  - Unless all documents are consistent with each other,
  - a lot of confusion is created for somebody trying to understand the product.

External Documentation

- All the documents for a product should be up-to-date:
  - Even a few out-of-date documents can create severe confusion.
- Readability is an important attribute of textual documents.
  - Readability determines understandability
  - hence determines maintainability.

4.3 Testing

Testing

- The aim of testing is to identify all defects existing in a software product.
- However, even after satisfactorily carrying out the testing phase it is not possible to guarantee that the software is error free.
  - As input data domain of the most software products is very large
  - It is not possible to test software exhaustively with respect to each value that input data can take
- Though testing does expose many defects existing in the software.

Testing

- Testing a program:
  - Subjecting the program to a set of test inputs (or test cases) and observing if the program behaves as expected.
  - If the program fails to behave as expected then the conditions under which failure occurs are noted for later **debugging** and **correction**.
- Common terms associated with testing:
  - A ***failure*** is manifestation of ***error*** (or ***defect*** or ***bug***). But mere presence of an error may not necessarily lead to a failure
  - A ***test case*** is the triplet (I, S, O). I- Data input to the system, S- State of the system at which the data is input, O- Expected Output
  - A ***test suite*** is the set of all test cases with which a given software product is to be tested.

Verification vs Validation

- Verification:
  - Process of determining whether the output of one phase of software development conforms to that of its previous phase
  - Concerned with phase containment of errors.
- Validation:
  - Process of determining whether a fully developed system conforms to its requirements specification.
  - Aimed at final product to be error free.

Verification vs Validation

Verification – "Are We Building the Product Right? "

Goals of Verification

- Ensure that the software design meets the specified requirements.
- Identify inconsistencies, errors, and missing requirements before development.
- Prevent defects rather than detect them later.

Verification in the Software Development Lifecycle SDLC

- **Requirement Phase**: Ensuring the requirement specifications are correct and complete.
- **Design Phase**: Checking if system design matches requirements.
- **Implementation Phase**: Reviewing source code for errors before execution.

Verification vs Validation

Validation – "Are We Building the Right Product?"

Validation ensures that the developed software meets the actual needs of end-users.

Goals of Validation

- Ensure that the final product meets user expectations and intended use.
- Detect functional and non-functional defects after software development.
- Validate the system in real-world scenarios before release.

Validation in the Software Development Lifecycle (SDLC)

- **Development Phase**: Unit testing is performed after coding.
- **Integration Phase**: Ensures different modules function together.
- **System Testing Phase**: Confirms the complete system meets requirements.
- **Deployment Phase**: User Acceptance Testing (UAT) ensures end-user satisfaction.

Real-World Examples of V&V

Example 1: Online Banking System

- **Verification**: Ensuring that the software design includes all required security measures (e.g., encryption, authentication).
- **Validation**: Running security tests to confirm that a hacker cannot access accounts.

Example 2: E-Commerce Website

- **Verification**: Reviewing the database schema to ensure correct relationships between products, orders, and customers.
- **Validation**: Running functional tests to confirm users can successfully add items to the cart and complete a purchase.

Differences Between Verification and Validation

Aspect	Verification	Validation
Definition	Ensures the software meets design requirements.	Ensures the software meets user expectations.
Focus	Process correctness (building the software right).	Product correctness (building the right software).
Methods	Static (reviews, walkthroughs, inspections).	Dynamic (testing, execution).
Performed By	Developers, QA teams, system architects.	Testers, end-users, quality assurance teams.
When Done?	Before software execution.	After software execution.
Example	Reviewing a database schema for correctness.	Running a test case to verify the system processes transactions correctly.

Design of Test cases

- Exhaustive testing of any system is impractical as domain of input data values is large or infinite.
- There for we must design ***optimal test suite*** of reasonable size to uncover as many errors existing in system as possible.
- Many test cases do not detect any additional defects that are already detected by other test cases in test suite. E.g.

```
If (x>y)
        max = x;
      else
        max = x;
```

- In above code segment, test suite {(x=3,y=2), (x=2,y=3)} can detect the error, where as test suite {(x=3,y=2), (x=4,y=3), (x=5,y=1)} does not detect error.

Design of Test cases

- There are two approaches to systematically designing test cases:
  - **Black-Box approach**
    - Without knowledge of internal structure
    - Designed using functional specification of the software – functional testing.
  - **White-Box approach**
    - With knowledge of internal structure
    - Designed after examine structure of code – structural testing

Testing in the large vs Testing in the small

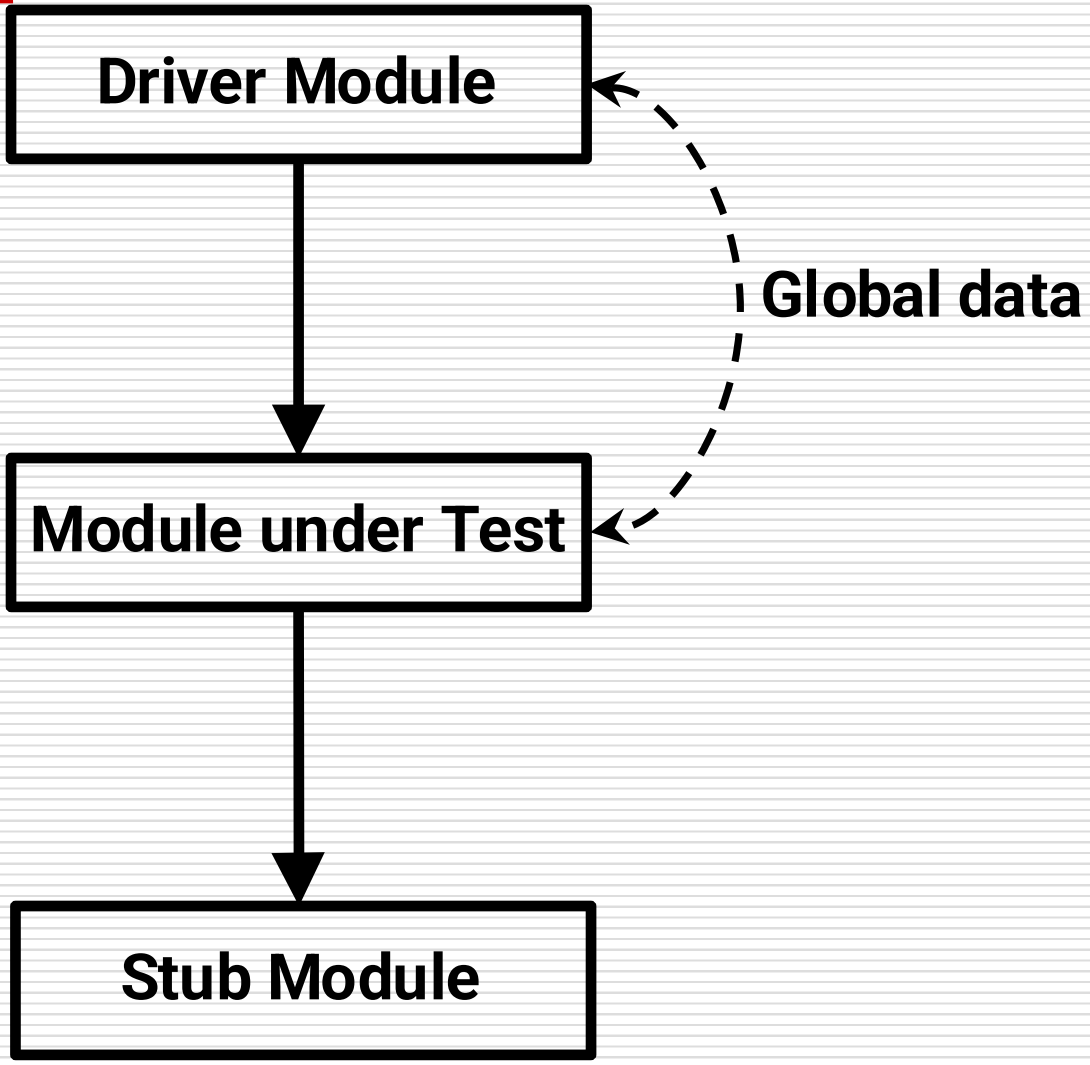
- Software product goes through three levels of testing:
  - ***Unit testing***
    - Individual component/unit/module is tested independently
  - ***Integration testing***
    - Components are integrated and tested at each level of integration
  - ***System testing***
    - Fully integrated system is tested
- Testing in the small: Unit testing
- Testing in the large: Integration testing, system testing.

Unit Testing

- It is a good idea to test modules in isolation before they are integrated. it makes debugging easier.
- If an error is detected when several modules are being tested together,
  - it would be difficult to determine which module has the error.
- Another reason:
  - the modules with which this module needs to interface may not be ready.
- **Benefits:**
  - **Early Bug Detection:**
    Identifies issues at an early stage, reducing the cost and effort required for fixes.
  - **Improved Code Quality:**
    Encourages developers to write modular and maintainable code.
  - **Facilitates Refactoring:**
    Provides confidence that changes or optimizations do not introduce new bugs.

Unit Testing

- **Driver** and **Stub** modules are used to provide necessary environment to perform unit testing.
- Driver module:
  - Contain non local data structures access by the module under test.
  - Code to call the different functions of the module under test.
- Stub module:
  - Stub procedure is dummy procedure that has the same I/O procedure that has the same I/O parameters as the given procedure but has a highly simplified behavior.



Integration Testing

- During each integration step, a number of modules are added to the partially integrated system and the system is tested.
- Once all modules have been integrated and tested, system testing can start.
- **Benefits:**
  - **Detects Interface Issues:**
    Identifies problems in the interactions between integrated components.
  - **Ensures Module Compatibility:**
    Verifies that different modules or services work together as intended.
  - **Reduces Risk of System Failures:**
    Addresses integration issues before the system is deployed, reducing the likelihood of failures in production.

System Testing

- System testing assesses the complete and integrated software system to verify that it meets the specified requirements. It evaluates both functional and non-functional aspects of the system
- Types of System Testing

1.Functional Testing:

- **Objective:** Verify that the system's functionalities align with specified requirements.
- **Approach:** Input data into the system and compare the output against expected results.

2.Performance Testing:

- **Objective:** Assess the system's responsiveness and stability under different conditions.
- **Sub-types:**
  - **Load Testing:** Evaluate system behavior under expected user loads.
  - **Stress Testing:** Determine the system's robustness by testing beyond normal operational capacity.
  - **Scalability Testing:** Analyze the system's ability to scale up or down based on demand.

System Testing

3. **Usability Testing:**
   - **Objective:** Ensure the system is user-friendly and intuitive.
   - **Approach:** Gather feedback from real users to identify areas of improvement in the user interface and experience.
4. **Security Testing:**
   - **Objective:** Identify vulnerabilities and ensure data protection mechanisms are effective.
   - **Approach:** Simulate attacks to uncover potential security loopholes.
5. **Compatibility Testing:**
   - **Objective:** Verify that the system operates correctly across various devices, browsers, and operating systems.
   - **Approach:** Test the application in different environments to ensure consistent behavior.
6. **Regression Testing:**
   - **Objective:** Ensure that new code changes do not adversely affect existing functionalities.
   - **Approach:** Re-run previously conducted tests and compare results.
7. **Recovery Testing:**
   - **Objective:** Assess the system's ability to recover from failures.
   - **Approach:** Intentionally cause failures and observe the system's recovery process.
8. **Installation Testing:**
   - **Objective:** Ensure the system installs and uninstalls correctly.
   - **Approach:** Test the installation process on various platforms and configurations.

Validation Testing

- Tested whether software is acceptable or not
  - Acceptance testing
  - Alpha testing: by customer at development site
  - Beta testing: On client side testing

Black Box Testing

- In the black-box testing, test cases are designed from an examination of the input/output values only and no knowledge of design, or code is required. The
- Following are the two main approaches to designing black box test cases.
  - Equivalence class partitioning
  - Boundary value analysis
  - Decision table testing

Equivalence Class Partitioning

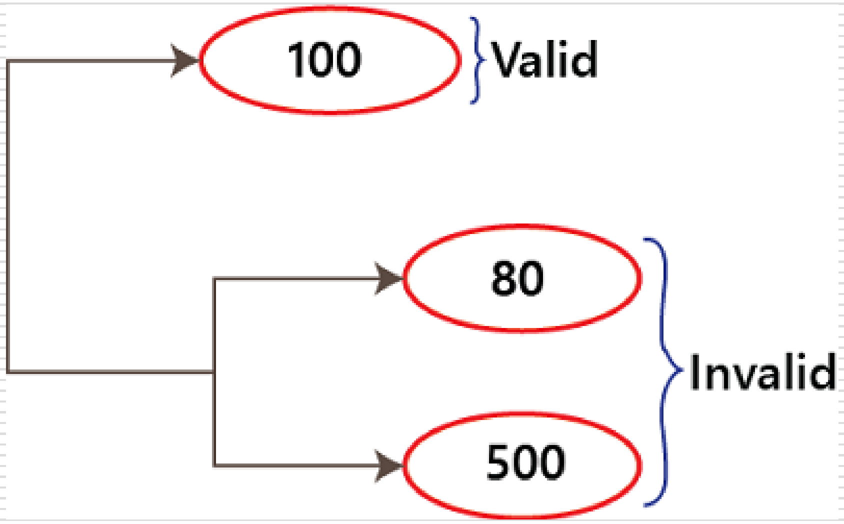
- The domain of input values to a program is partitioned into a set of equivalence classes. This partitioning is done such that the behavior of the program is similar for every input data belonging to the same equivalence class.
- The main idea behind defining the equivalence classes is that testing the code with any one value belonging to an equivalence class is as good as testing the software with any other value belonging to that equivalence class.

Equivalence Class Partitioning

- **General guidelines** for designing the equivalence classes:
  - If the input data values to a system can be specified by **a range of values**, then **one valid and two invalid equivalence classes** should be defined.
  - If the input data assumes values from a **set of discrete members** of some domain, then **one equivalence class for valid input values and another equivalence class for invalid input** values should be defined.
  - If the input data is **Boolean**, **one valid and one invalid class** should be defined

- **According to pressman:**

  For example, the Amount of test field accepts a Range (100-400) of values:



- 1 valid 2 invalid value for test case
- {100,0,401}-Valid test case
- {0,401,402}- not valid test case

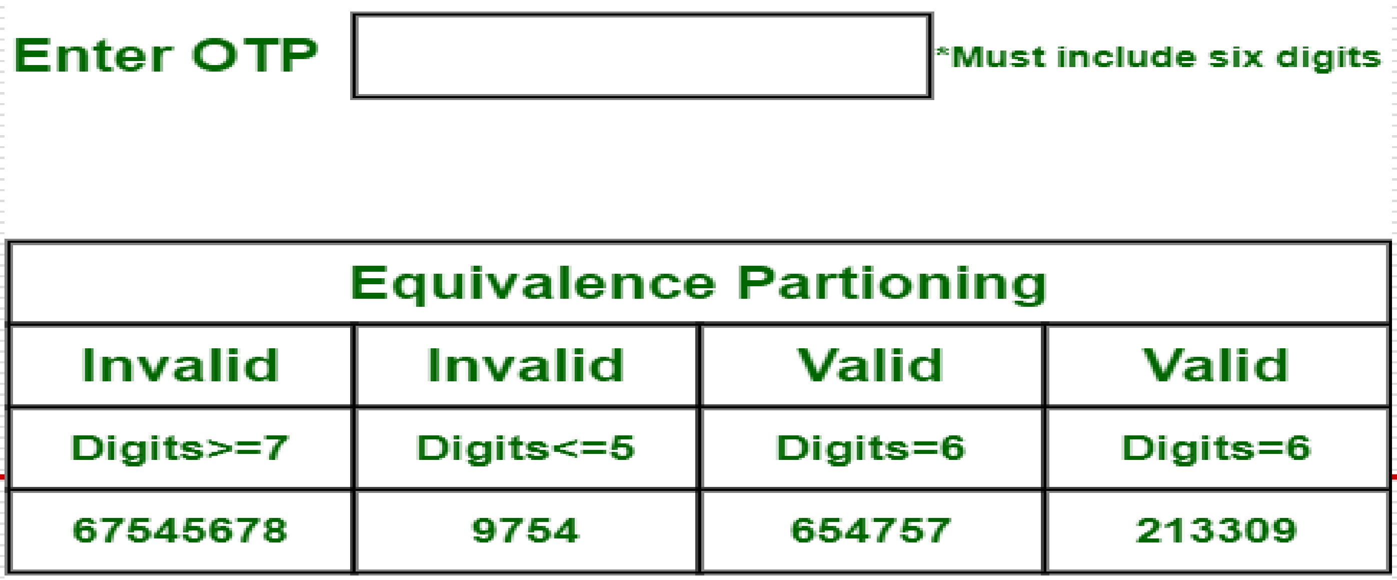
Equivalence Class Partitioning

- For a software that computes the square root of an input integer which can assume values in the range of 0 to 5000, there are three equivalence classes: The set of negative integers, the set of integers in the range of 0 and 5000, and the integers larger than 5000. Therefore, the test cases must include representatives for each of the three equivalence classes and a possible

- Test input set can be: {-5, 500, 6000}.

Equivalence Class Partitioning

- Let us consider an example of software application. There is function of software application that accepts only particular number of digits, not even greater or less than that particular number. Consider an OTP number that contains only 6 digit number, greater and even less than six digits will not be accepted, and the application will redirect customer or user to error page. If password entered by user is less or more than six characters, that equivalence partitioning method will show an invalid OTP. If password entered is exactly six characters, then equivalence partitioning method will show valid OTP.

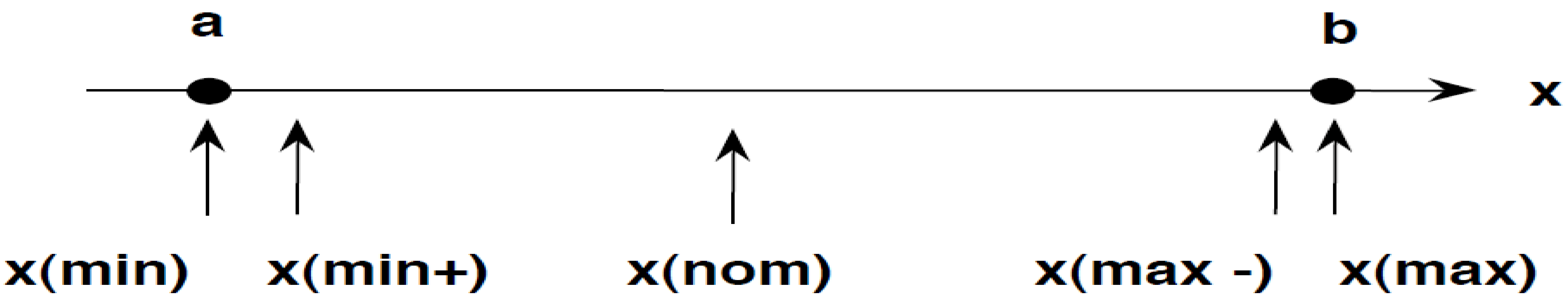


Enter OTP *Must include six digits

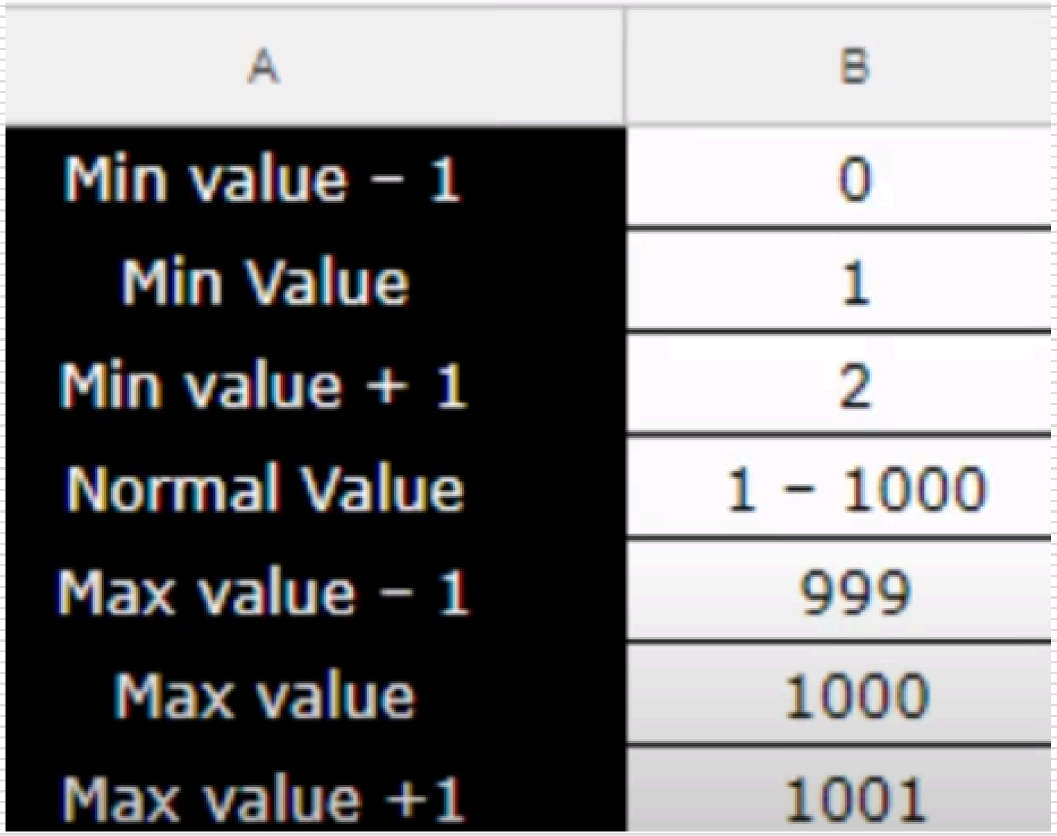
Equivalence Partioning			
Invalid	Invalid	Valid	Valid
Digits>=7	Digits<=5	Digits=6	Digits=6
67545678	9754	654757	213309

Boundary Value Analysis

- A type of programming error frequently occurs at the boundaries of different equivalence classes of inputs.
- The reason behind such errors might purely be due to psychological factors. Programmers often fail to see the special processing required by the input values that lie at the boundary of the different equivalence classes.
- For example, programmers may improperly use < instead of <=, or conversely <= for <. Boundary value analysis leads to selection of test cases at the boundaries of the different equivalence classes.
- ***Example:*** For a function that computes the square root of integer values in the range of 0 and 5000, the test cases must include the following values: {0, -1,5000,5001}.



- The basic idea in normal boundary value testing is to select input variable values at their:
  - Minimum
  - Just above the minimum
  - A nominal value
  - Just below the maximum
  - Maximum



A	B
Min value – 1	0
Min Value	1
Min value + 1	2
Normal Value	1 – 1000
Max value – 1	999
Max value	1000
Max value +1	1001

- Example for value in range of 1-1000

Advantages

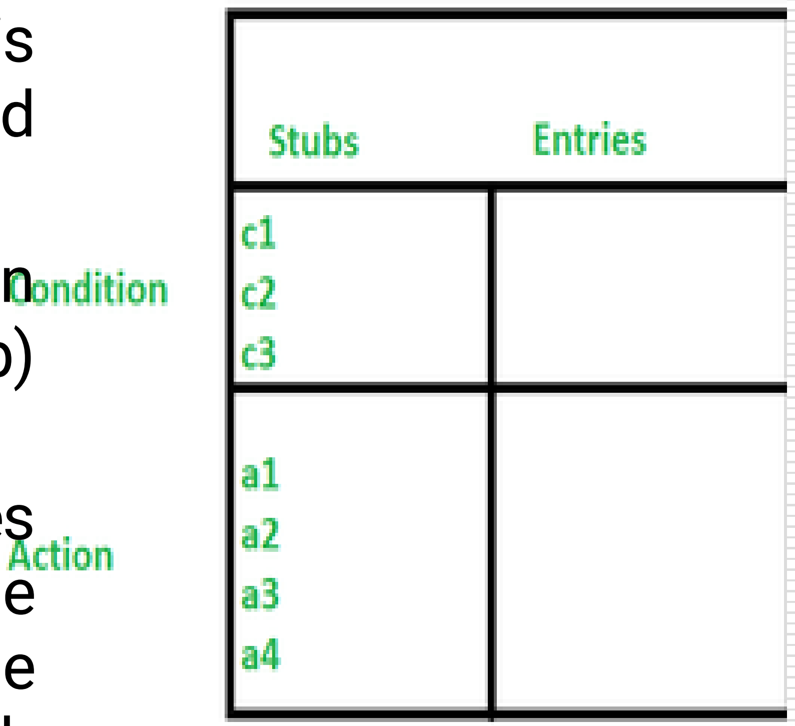
- Efficient for large system
- Code access is not required.
- Testing is balanced and non prejudiced
- Non technical tester can test-No functional knowledge needed

Disadvantages

- Test cases are challenging to design without having clear functional specification
- High probability of repeating test cases
- Inefficient testing, due to the fact that the tester only has limited knowledge about an application.
- Blind coverage, since the tester cannot target specific code segments or error prone areas.

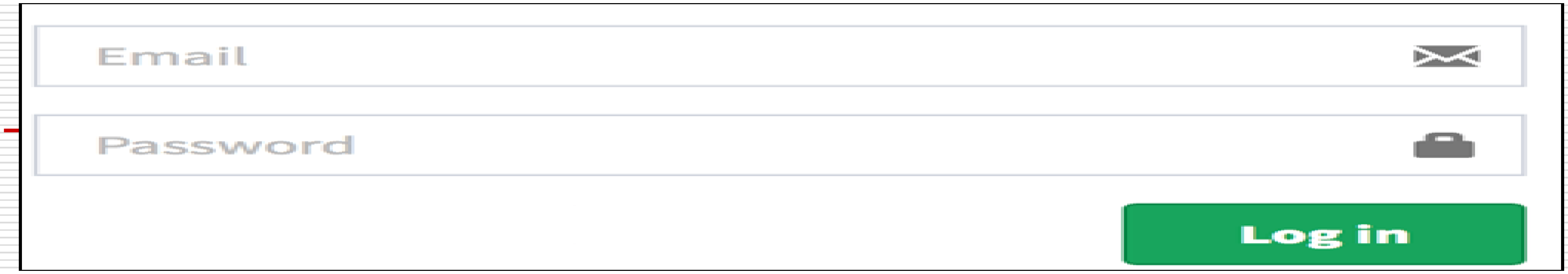
Decision Table Based Testing

- **Condition Stubs :** The conditions are listed in this first upper left part of the decision table that is used to determine a particular action or set of actions.
- **Action Stubs :**All the possible actions are given in the first lower left portion (i.e, below condition stub) of the decision table.
- **Condition Entries** : In the condition entry, the values are inputted in the upper right portion of the decision table. In the condition entries part of the table, there are multiple rows and columns which are known as Rule.



- **Action Entries :** In the action entry, every entry has some associated action or set of actions in the lower right portion of the decision table and these values are called outputs.

Example 1: How to make Decision Base Table for Login Screen



Conditions	Rule 1	Rule 2	Rule 3	Rule 4
Username (T/F)	F	T	F	T
Password (T/F)	F	F	T	T
Output (E/H)	E	E	E	H

Legend:

- **T** – Correct username/password
- **F** – Wrong username/password
- **E** – Error message is displayed
- **H** – Home screen is displayed

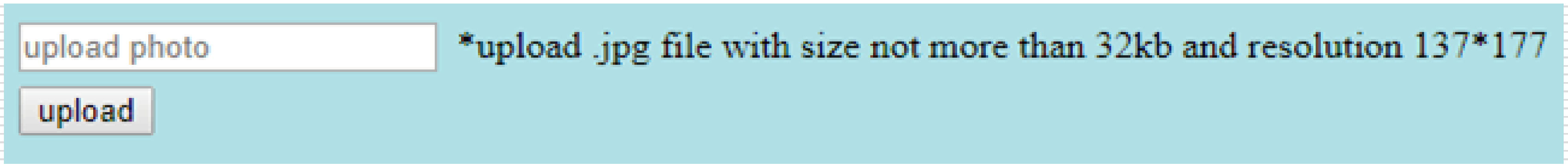
Interpretation:

- **Case 1** – Username and password both were wrong. The user is shown an error message.
- **Case 2** – Username was correct, but the password was wrong. The user is shown an error message.
- **Case 3** – Username was wrong, but the password was correct. The user is shown an error message.
- **Case 4** – Username and password both were correct, and the user navigated to the homepage

Example 2: How to make Decision Table for Upload Screen

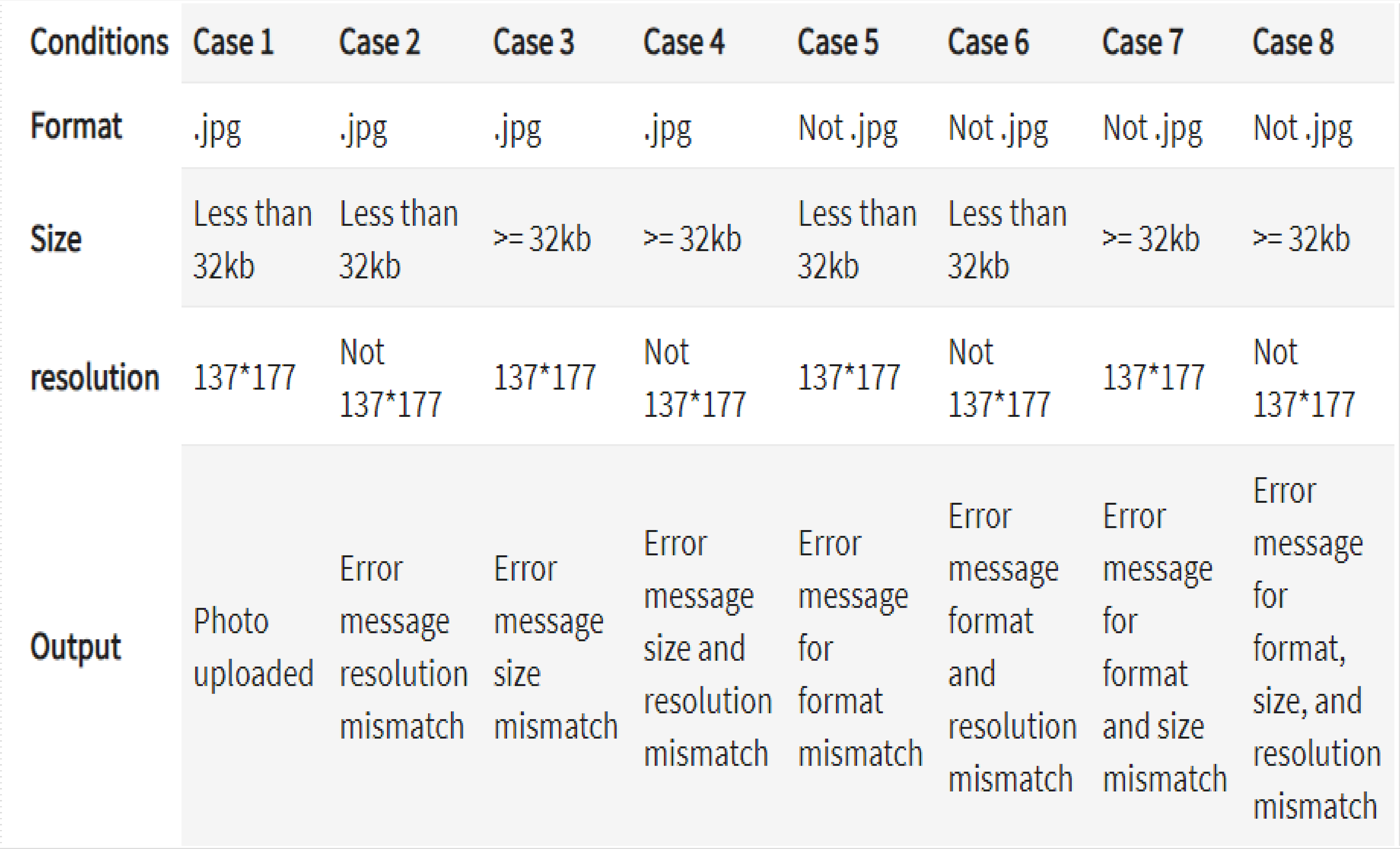
certain conditions like –

- You can upload only ‘.jpg’ format image
- file size less than 32kb
- resolution 137*177.



*upload .jpg file with size not more than 32kb and resolution 137*177

Descision Table:



Conditions	Case 1	Case 2	Case 3	Case 4	Case 5	Case 6	Case 7	Case 8
Format	.jpg	.jpg	.jpg	.jpg	Not .jpg	Not .jpg	Not .jpg	Not .jpg
Size	Less than 32kb	Less than 32kb	>= 32kb	>= 32kb	Less than 32kb	Less than 32kb	>= 32kb	>= 32kb
resolution	137*177	Not 137*177	137*177	Not 137*177	137*177	Not 137*177	137*177	Not 137*177
Output	Photo uploaded	Error message resolution mismatch	Error message size mismatch	Error message size and resolution mismatch	Error message for format mismatch	Error message format and resolution mismatch	Error message for format and size mismatch	Error message for format, size, and resolution mismatch

1. Upload a photo with format ‘.jpg’, size less than 32kb and resolution 137*177 and click on upload. Expected result is Photo should upload successfully
2. Upload a photo with format ‘.jp g’, size less than 32kb and res olution not 137*177 and click on upload. Expected result is Error message resolution mismatch should be displayed
3. Upload a photo with format ‘.jpg’, size more than 32kb and resolution 137*177 and click on upload. Expected result is Error message size mismatch should be displayed
4. Upload a photo with format ‘.jpg’, size more than equal to 32kb and resolution not 137*177 and click on upload. Expected result is Error message size and resolution mismatch should be displayed

5. Upload a photo with format other than ‘.jpg’, size less than 3 2kb and resolution 137*177 and click on upload. Expected result is Error message for format mismatch should be displayed
6. Upload a photo with format other than ‘.jpg’, size less than 3 2kb and resolution not 137*177 and click on upload. Expected result is Error message format and resolution mismatch should be displayed
7. Upload a photo with format other than ‘.jpg’, size more than 3 2kb and resolution 137*177 and click on upload. Expected result is Error message for format and size mismatch should be displayed
8. Upload a photo with format other than ‘.jpg’, size more than 3 2kb and resolution not 137*177 and click on upload. Expected result is Error message for format, size and resolution mismatch should be displayed

Advantages of Decision Table Testing

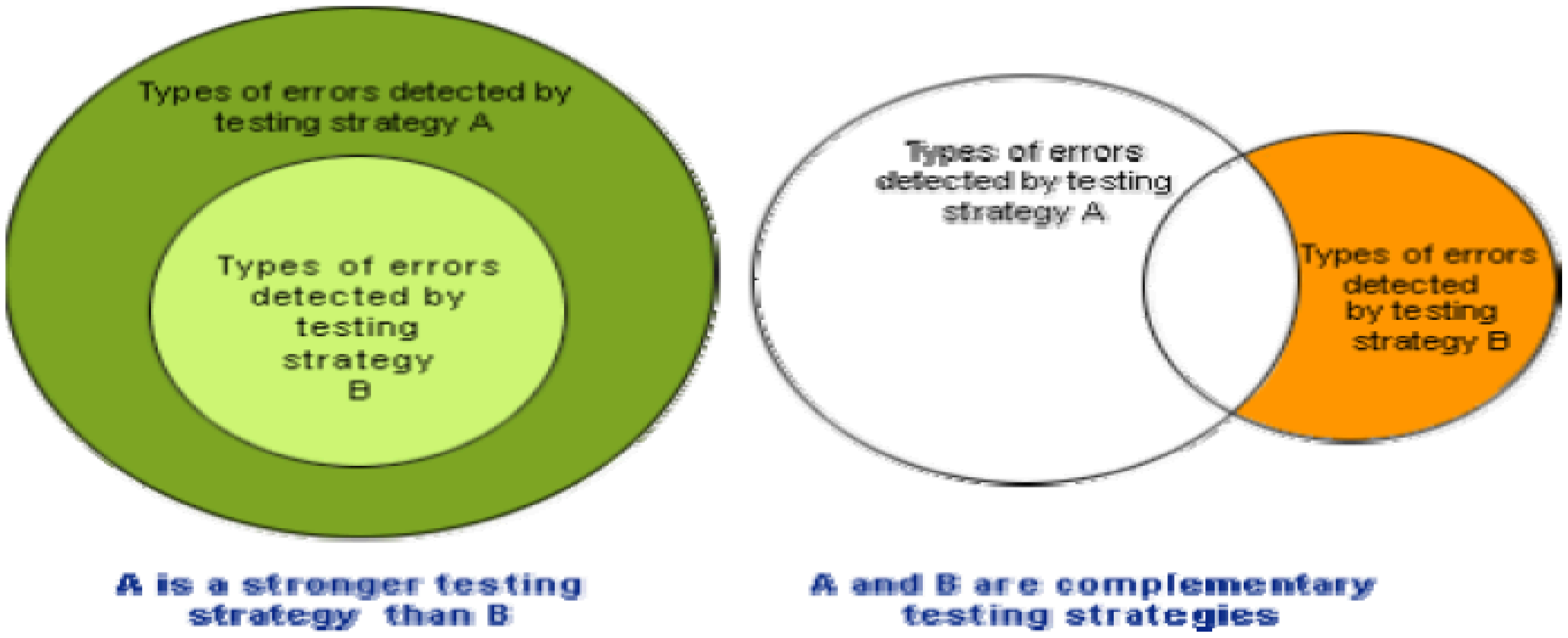
- When the system behavior is different for different inputs and not the same for a range of inputs, both equivalent partitioning, and boundary value analysis won' t help, but a decision table can be used.
- The representation is simple so that it can be easily interpreted and is used for development and business as well.
- This table will help to make effective combinations and can ensure better coverage for testing
- Any complex business conditions can be easily turned into decision tables
- In a case we are going for 100% coverage typically when the input combinations are low, this technique can ensure the coverage.

Disadvantages of Decision Table Testing

The main disadvantage is that when the number of inputs increases the table will become more complex

White Box Testing

- Concerned with testing implementation.
- Need to know the code to investigate internal logic and structure of code
- Known as glassbox, transparent testing, structural testing etc.
- Stronger vs complementary Test strategies


Types of errors detected by testing strategy A
Types of errors detected by testing strategy B
A is a stronger testing strategy than B
Types of errors detected by testing strategy A
Types of errors detected by testing strategy B
A and B are complementary testing strategies

Statement coverage

- Statement coverage strategy aims to design test cases so that every statement in a program is executed at least once.
- Executing some statement once and observing that it behaves properly for that input value is no guarantee that it will behave correctly for all input values.

```
1. printsum( int a, int b)
2. int res=a+b;
3. if (res>0)
4. printf( “positive” );
5. else
6. Printf( “Negative” );
7. }
```

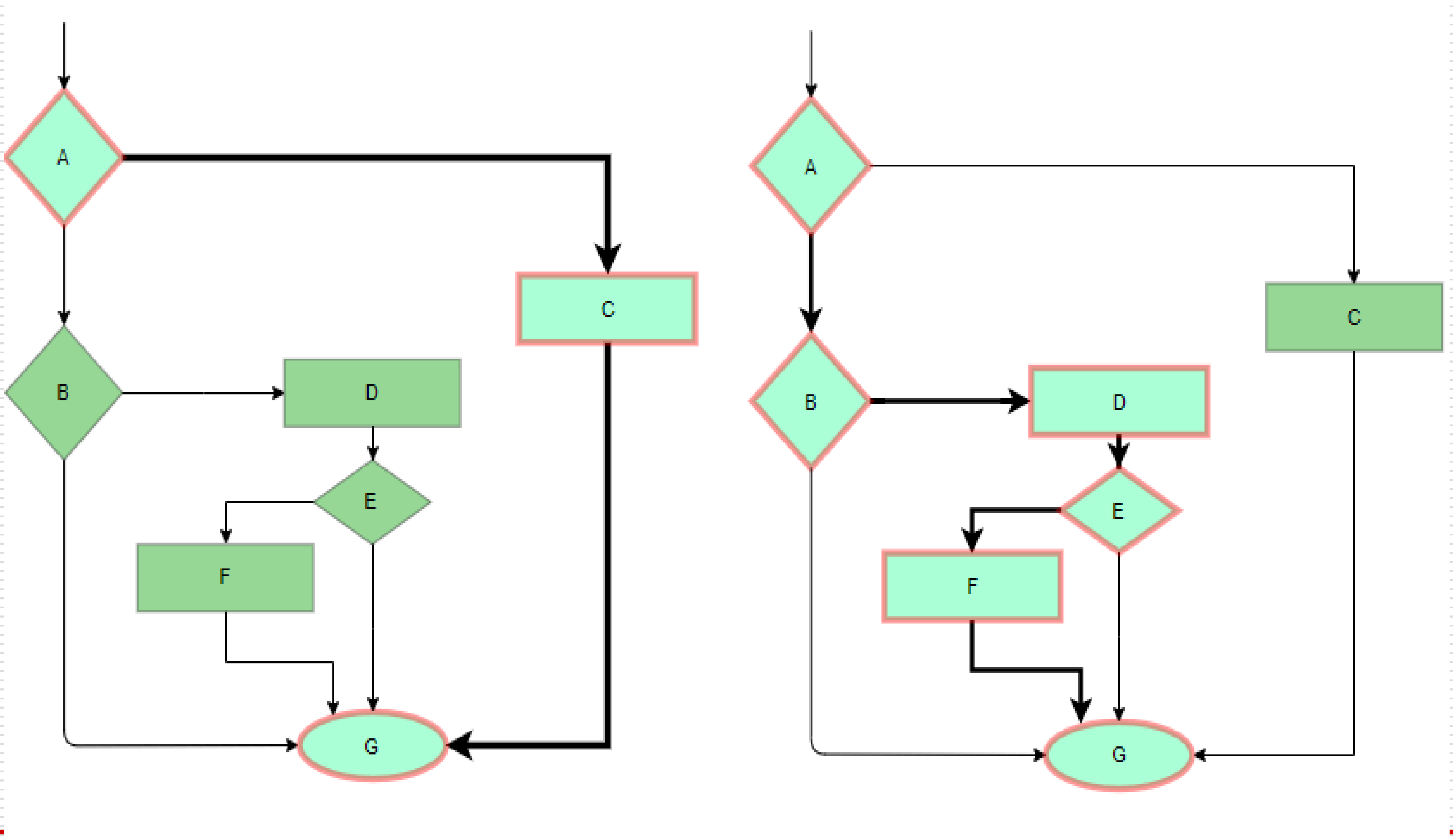
TC#1 a=2,b=3
TC#2 a=-1, b=-2

$$\text{Coverage} = \frac{\text{no. Of executed stmt}}{\text{Total no. Of stmt}}$$

Target: 80-100%

Statement coverage

- Statement coverage helps uncover unused statements, unused branches, missing statement that are referenced by part of the code, and dead code left over from previous versions.



Branch coverage

- Test cases are designed to make each branch condition to assume true and false values in turn.
- Branch testing is also known as edge testing
- It is obvious that branch testing guarantees statement coverage and thus is a stronger testing strategy compared to the statement coverage-based testing.

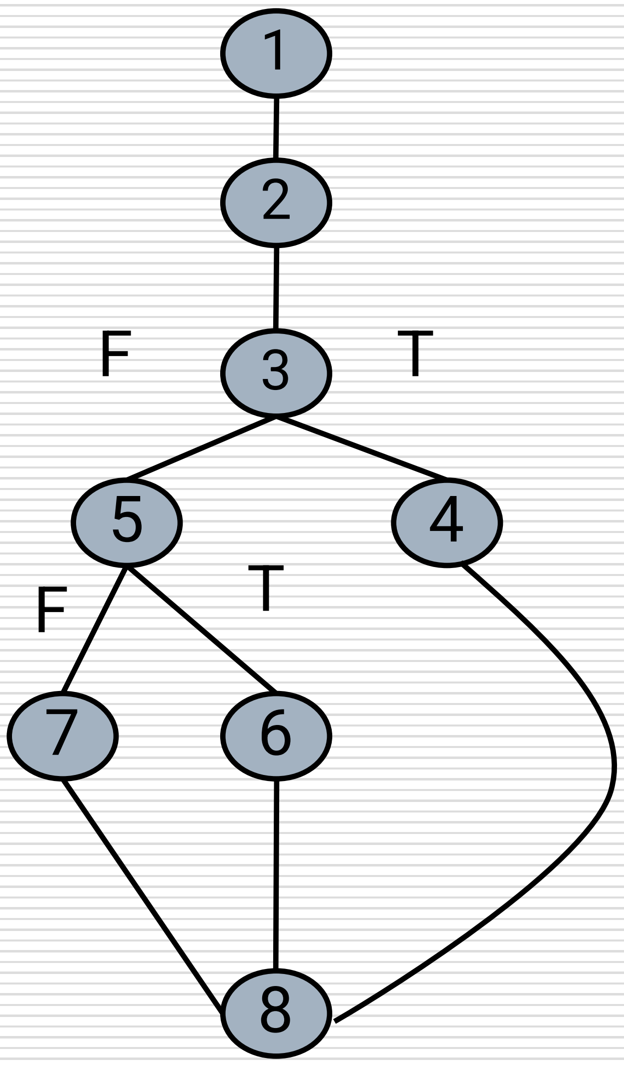
```
1. printsum( int a, int b)
2. int res=a+b;
3. if (res>0)
4. printf( "positive" );
5. Else if(res<0)
6. Printf( "Negative" );
7. Else do nothing }
```

TC#1 a=2,b=3
TC#2 a=-1, b=-2
TC#3 a=0, b=0

$$\text{Coverage} = \frac{\text{no. Of executed branch}}{\text{Total no. Of branch}}$$

Target: 80-100%

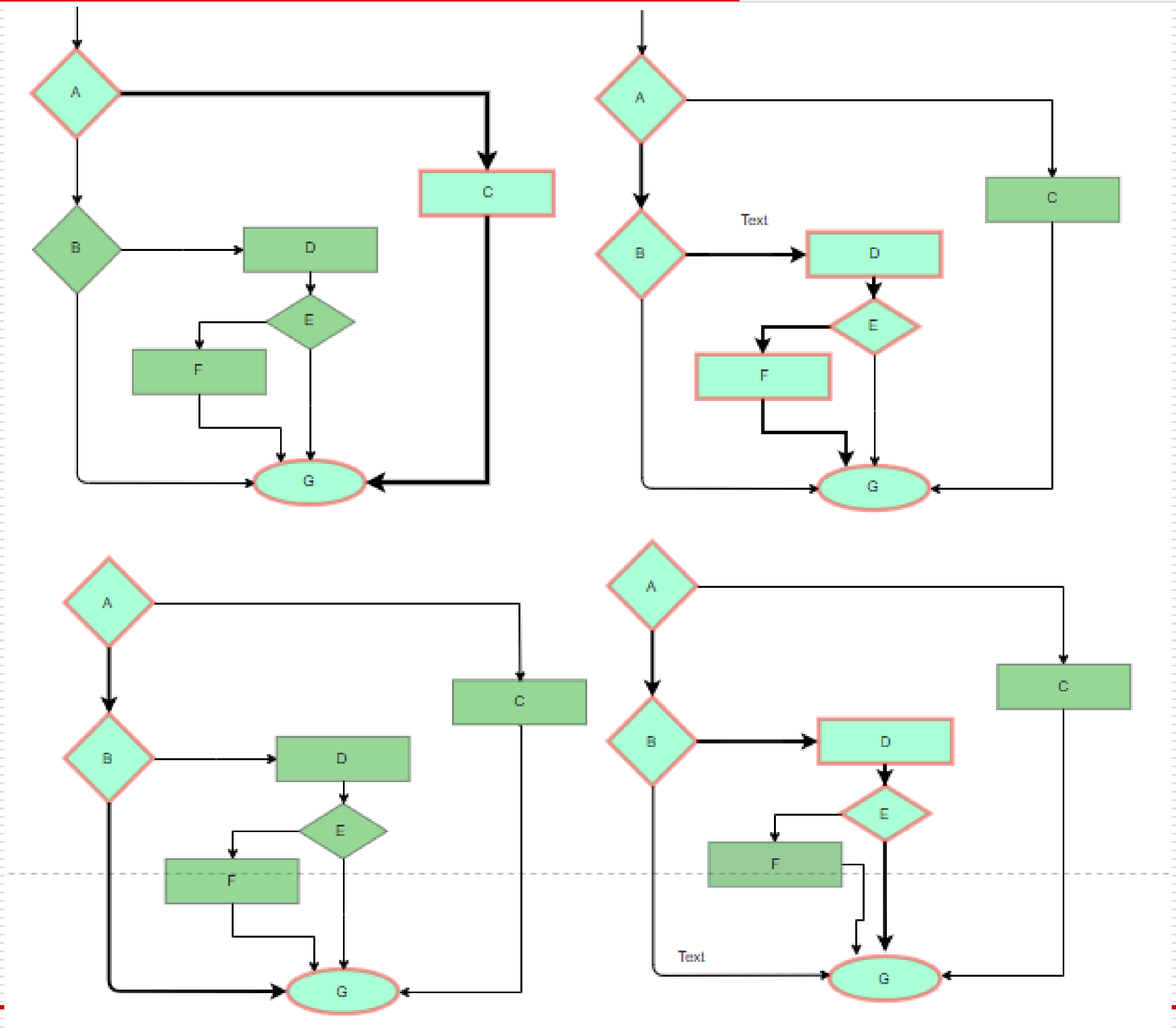
Branch coverage



TC#1
a=2,b=3
TC#2
a=-1, b=-2
TC#3
a=0, b=0

1. printsum(int a, int b)
2. int res=a+b;
3. if (res>0)
4. printf(“positive”);
5. Else if(res<0)
6. Printf(“Negative”);
7. Else do nothing
8. }

Branch coverage



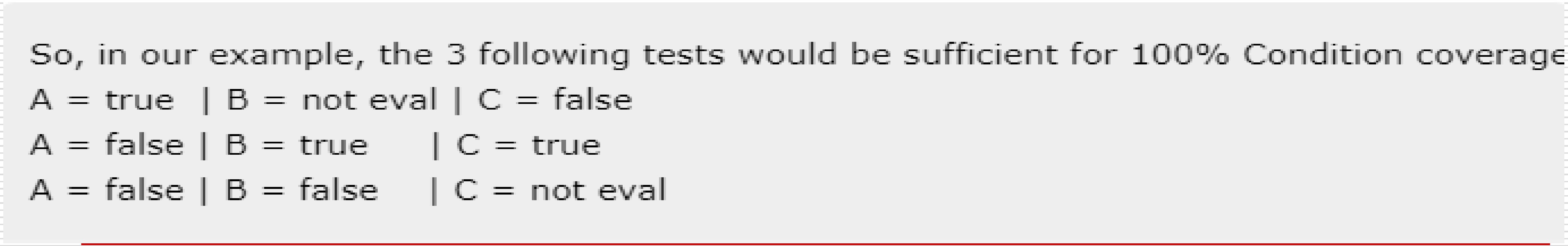
Condition coverage

- test cases are designed to make each component of a composite conditional expression to assume both true and false values.
- For example, in the conditional expression ((c1.and.c2).or.c3), the components c1,c2 and c3 are each made to assume both true and false values.
- Branch testing is probably the simplest condition testing strategy where only the compound conditions appearing in the different branch statements are made to assume the true and false values.
- Thus, condition testing is a stronger testing strategy than branch testing and branch testing is stronger testing strategy than the statement coverage-based testing.

Condition coverage

- For a composite conditional expression of n components, for condition coverage, 2^n test cases are required. Thus, for condition coverage, the number of test cases increases exponentially with the number of component conditions.
- Condition coverage-based testing technique is practical only if n (the number of conditions) is small.

```
if ((A || B) && C)
{ << Few Statements >> }
else
{ << Few Statements >> }
```



So, in our example, the 3 following tests would be sufficient for 100% Condition coverage

```
A = true  | B = not eval | C = false
A = false | B = true     | C = true
A = false | B = false    | C = not eval
```

Path coverage

- The path coverage-based testing strategy requires us to design test cases such that all linearly independent paths in the program are executed at least once.
- A linearly independent path can be defined in terms of the control flow graph (CFG) of a program.
- **Steps:**
  - Make the corresponding control flow graph
  - Calculate the cyclomatic complexity
  - Find the independent paths
  - Design test cases corresponding to each independent path

Control Flow Graph

- A control flow graph describes the sequence in which the different instructions of a program get executed. In other words, a control flow graph describes how the control flows through the program.
- How to draw:
  - all the statements of a program must be numbered first
  - The different numbered statements serve as nodes of the control flow graph
  - An edge from one node to another node exists if the execution of the statement representing the first node can result in the transfer of control to the other node.

Control Flow Graph

Sequence

1.a=5;

2.b=a*2-1;

$$\begin{tikzcd}
1 \arrow[d] \\
2
\end{tikzcd}$$

Selection

1.If(a>b)

2.   c=3;

3.else

4.c=5;

5.c=c*c;

$$\begin{tikzcd}
 & 1 \arrow[dl] \arrow[dr] & \\
2 \arrow[dr] & & 3,4 \arrow[dl] \\
 & 5 &
\end{tikzcd}$$

Iteration

1.while(a>b) {

2.   b=b - 1;

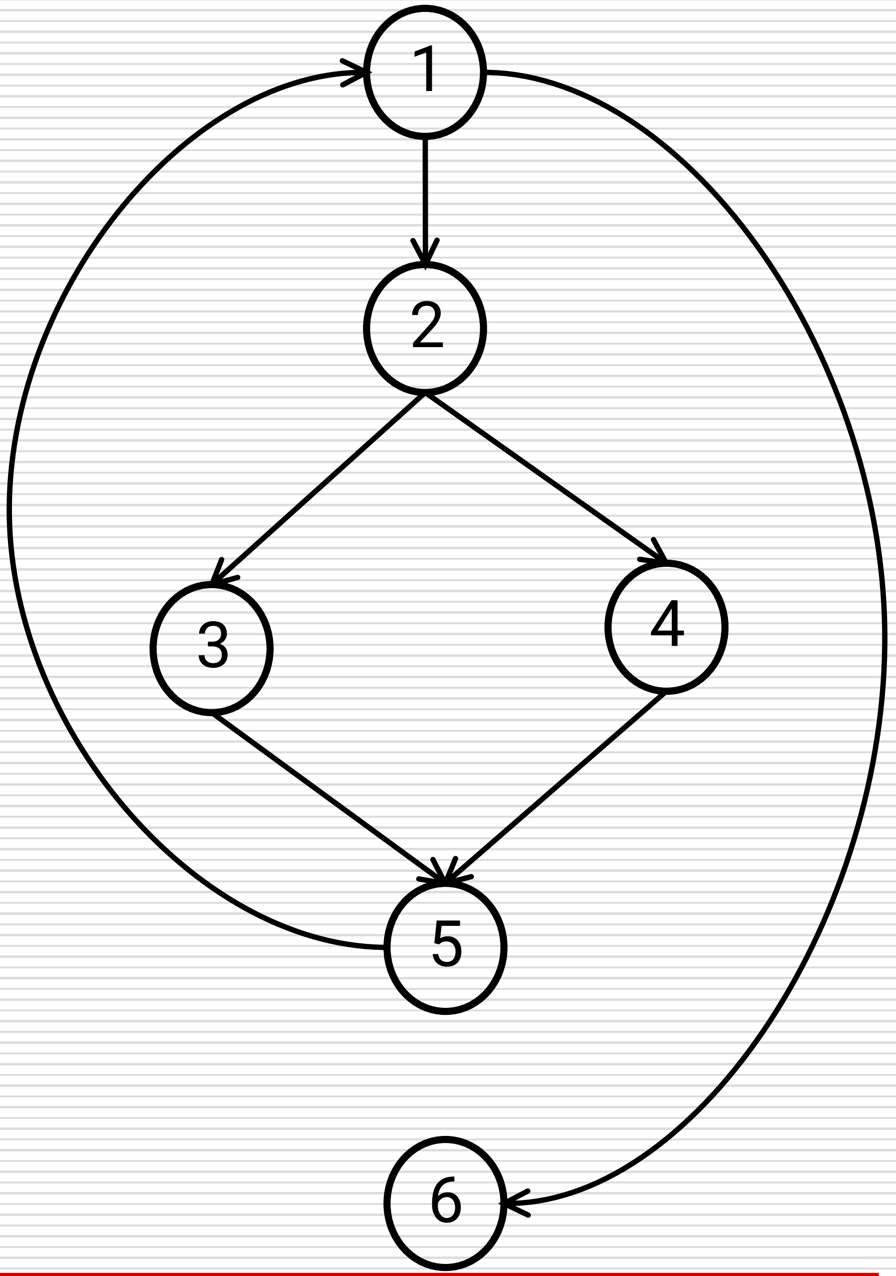
3.   b=b * a;}

4.c=a*b;

$$\begin{tikzcd}
1 \arrow[d] \arrow[ddd, bend left] \\
2 \arrow[d] \\
3 \arrow[uu, bend left] \\
4
\end{tikzcd}$$

Control Flow Graph

```
int compute_gcd(int x, int y)  {
1while(x != y)  {
2       if(x > y)
3           x = x - y;
4       else   y = y - x;
5 }
6  return x;
```



$$\begin{tikzcd}
 & 1 \arrow[d] \arrow[ddd, bend left=60] & \\
 & 2 \arrow[dl] \arrow[dr] & \\
3 \arrow[dr] & & 4 \arrow[dl] \\
 & 5 \arrow[uuu, bend left=60] & \\
 & 6 &
\end{tikzcd}$$

Path

- A path through a program is a node and edge sequence from the starting node to a terminal node of the control flow graph of a program.
- There can be more than one terminal node in a program.
- E.g. 123516, 124516, 1235124516 etc.
- Writing test cases to cover all the paths of a typical program is impractical.
- For this reason, the path-coverage testing does not require coverage of all paths but only coverage of linearly independent paths.

Linearly Independent Paths

- A linearly independent path is any path through the program that introduces at least one new edge that is not included in any other linearly independent paths.
- If a path has one new node compared to all other linearly independent paths, then the path is also linearly independent.
- This is because, any path having a new node automatically implies that it has a new edge.
- Thus, a path that is subpath of another path is not considered to be a linearly independent path.

Cyclomatic Complexity Metric

- It is not easy to determine the number of independent paths of the program.
- McCabe’ s cyclomatic complexity d efines an upper bound for the number of linearly independent paths through a program.
- The McCabe’ s cyclomatic complexity metric provides a practical way of determining the maximum number of linearly independent paths in a program.
- There are three different ways to compute the cyclomatic complexity. The answers computed by the three methods are guaranteed to agree.

Cyclomatic Complexity Metric

- **Method – 1:**

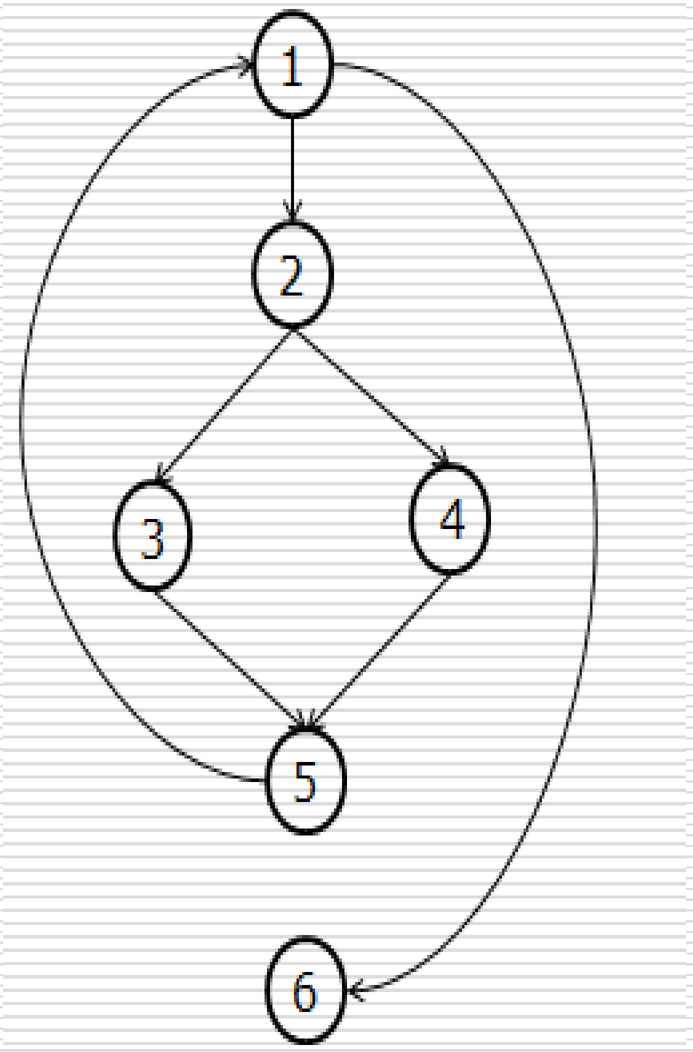
Cyclomatic Complexity V(G)

V(G) = E – N + 2

Where N is the number of nodes of the control flow graph and E is the number of edges in the control flow graph.

- For the CFG of example E=7 and N=6. Therefore, the

cyclomatic complexity = 7-6+2 = 3.



Cyclomatic Complexity Metric

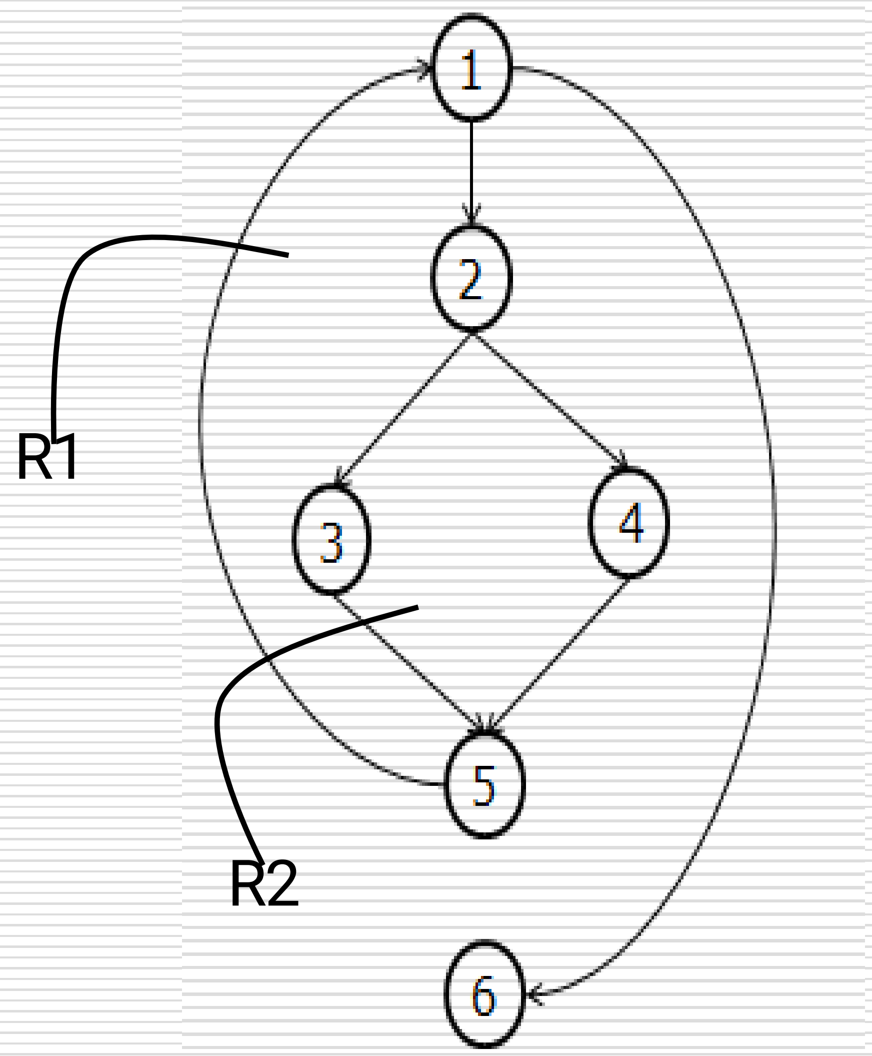
Method – 2:

Cyclomatic Complexity V(G)

V(G) = Total number of bounded areas + 1

- Any region enclosed by nodes and edges can be called as a bounded area.
- For the CFG of example bounded areas are 2. Therefore, the

cyclomatic complexity = 2+1 = 3.



Cyclomatic Complexity Metric

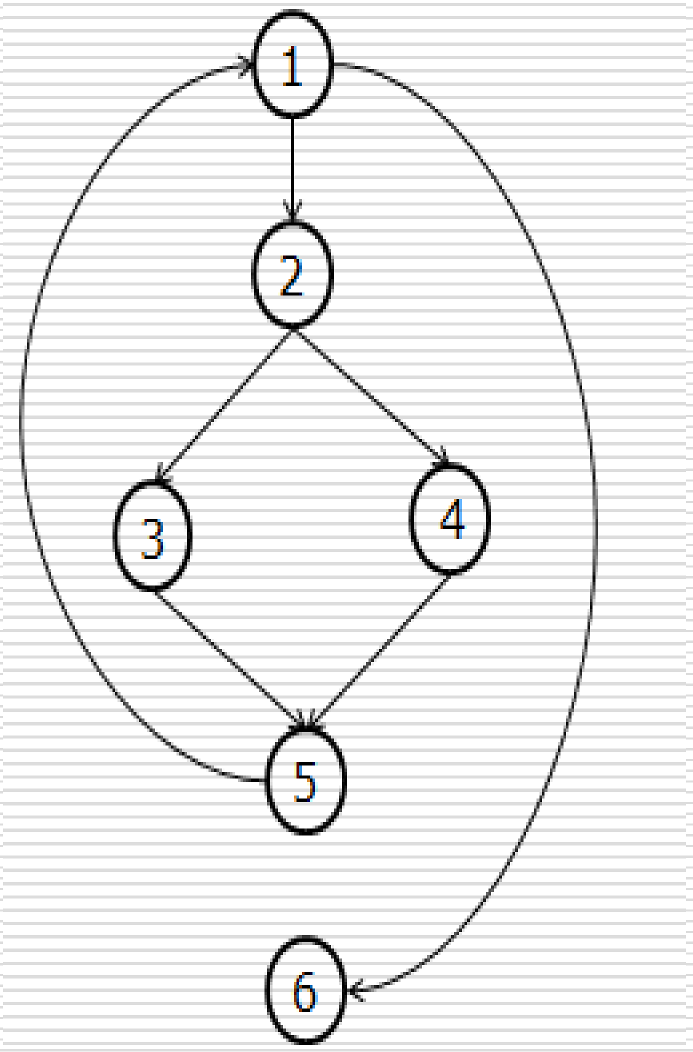
- **Method – 3:**

Cyclomatic Complexity V(G)

V(G) = Total number of decision statements + 1

- The cyclomatic complexity of a program can also be easily computed by computing the number of decision statements of the program.
- For the CFG of example decision statements are 2. Therefore, the

cyclomatic complexity = 2+1 = 3.


$$\begin{tikzcd}
 & 1 \arrow[d] \arrow[ddddr, bend left] & \\
 & 2 \arrow[dl] \arrow[dr] & \\
3 \arrow[dr] & & 4 \arrow[dl] \\
 & 5 \arrow[uuu, bend left=60] & \\
 & & 6
\end{tikzcd}$$

Preparing test cases

1. Draw the control flow graph
2. Determine V(G)
3. Determine the basis of linearly independent paths
4. Prepare the test cases that will force execution of each path in the basis set.

Another example

- V(G) = E-N+2

  10-8+2 =4

  OR

  V(G)= Bounded region +1=

  3+1=4

  OR

  V(G) = total no. of decision stmt+

  3+1=4

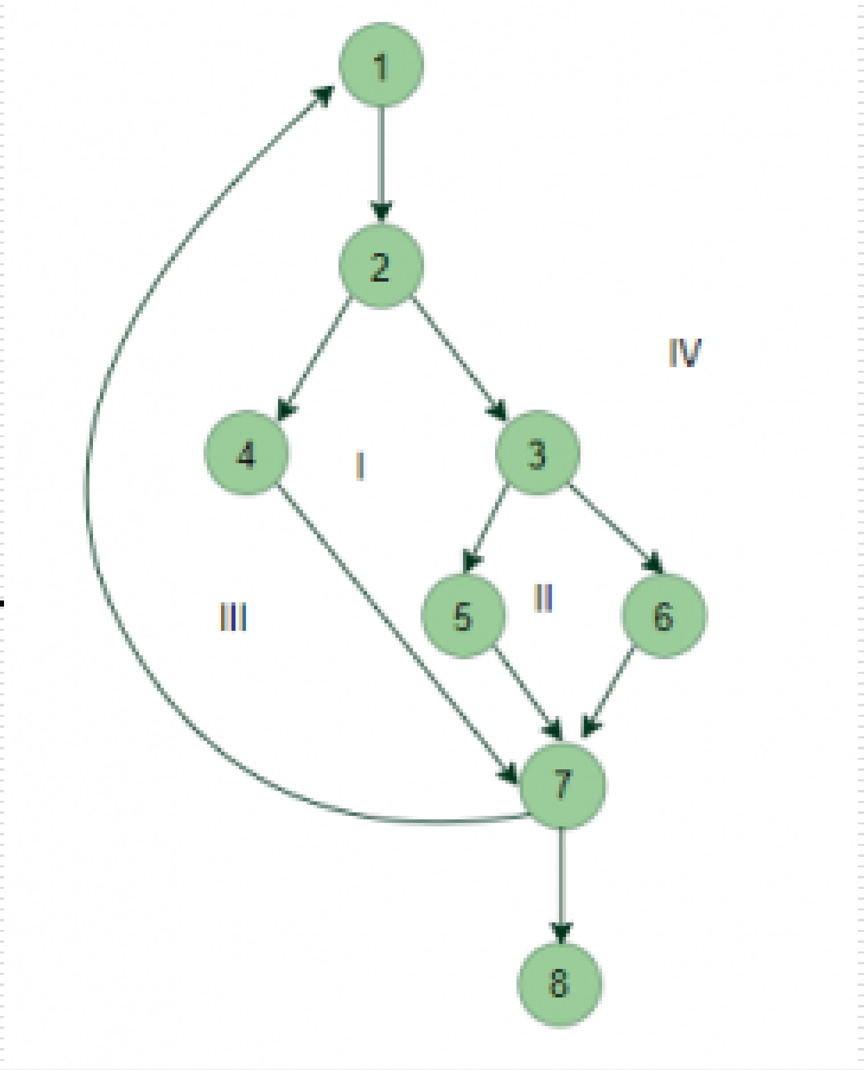
No of independent paths = 4

#P1: 1 – 2 – 4 – 7 – 8

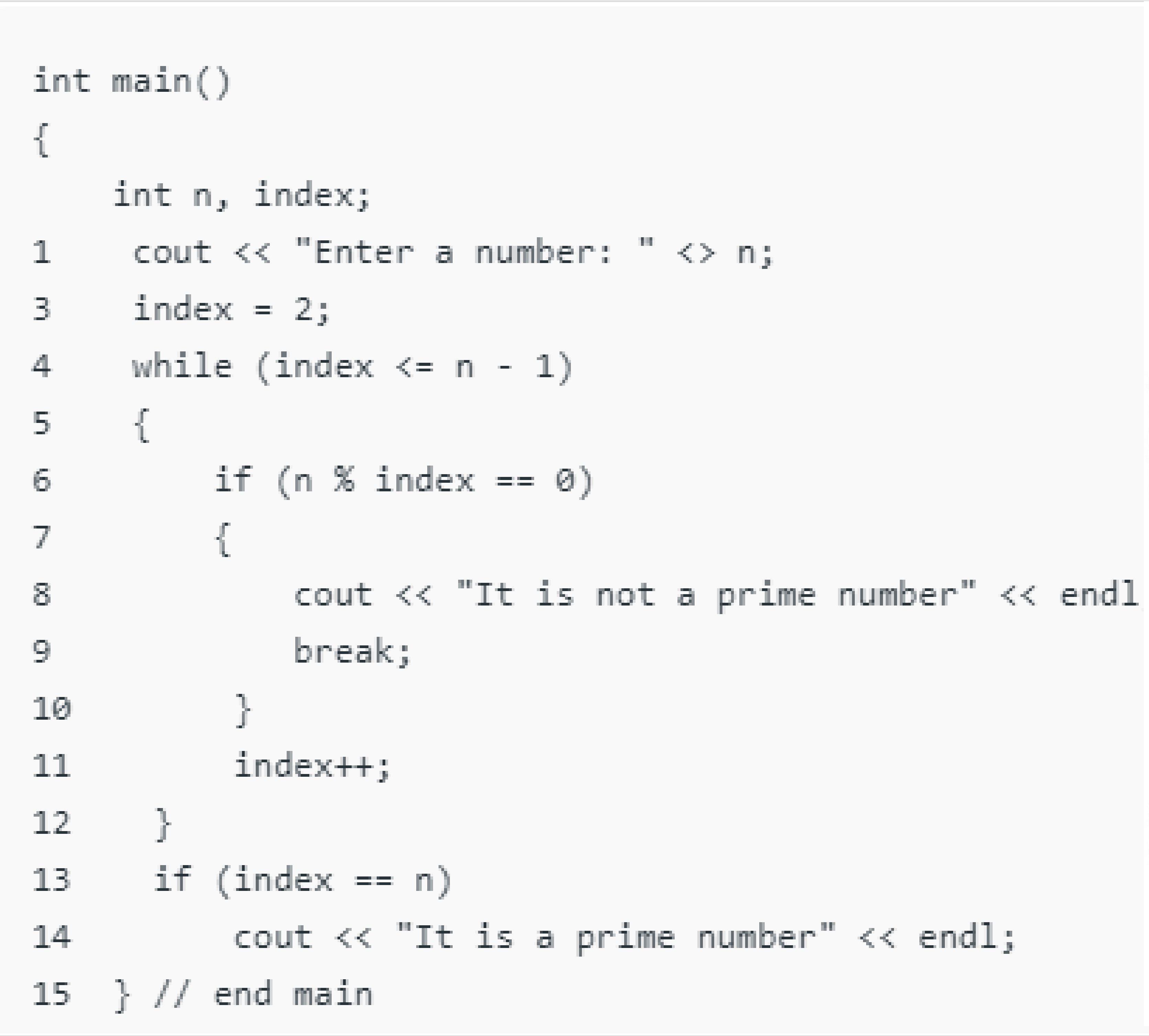
#P2: 1 – 2 – 3 – 5 – 7 – 8

#P3: 1 – 2 – 3 – 6 – 7 – 8

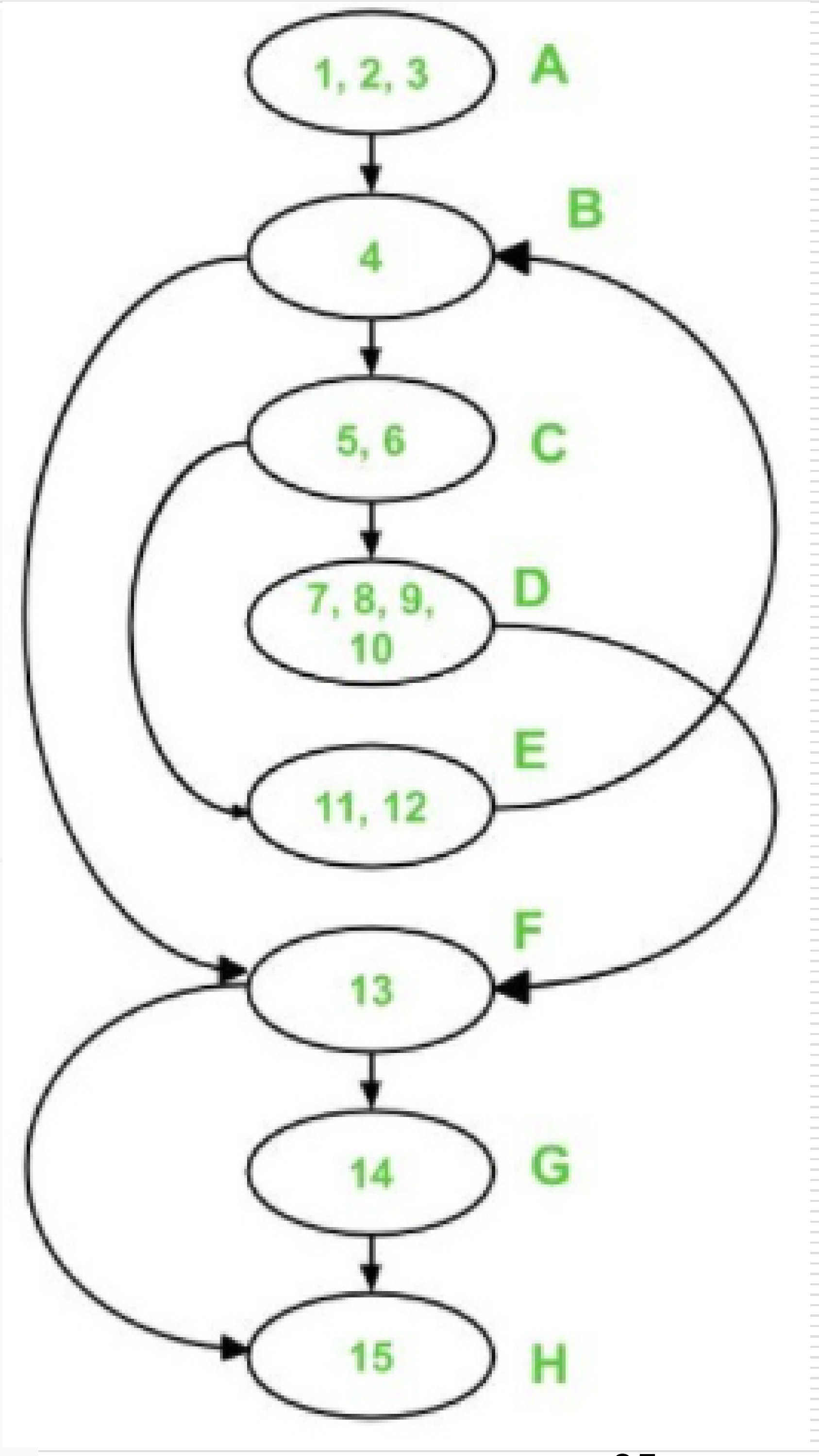
#P4: 1 – 2 – 4 – 7 – 1 – . . . – 7 – 8



Finding prime number



```
int main()
{
    int n, index;
1    cout << "Enter a number: " <> n;
3    index = 2;
4    while (index <= n - 1)
5    {
6        if (n % index == 0)
7        {
8            cout << "It is not a prime number" << endl
9            break;
10        }
11        index++;
12     }
13     if (index == n)
14         cout << "It is a prime number" << endl;
15   } // end main
```



Cyclomatic complexity

V(G)=E-N+2

=10-8+2

=4

V(G)=Decision stmt+1

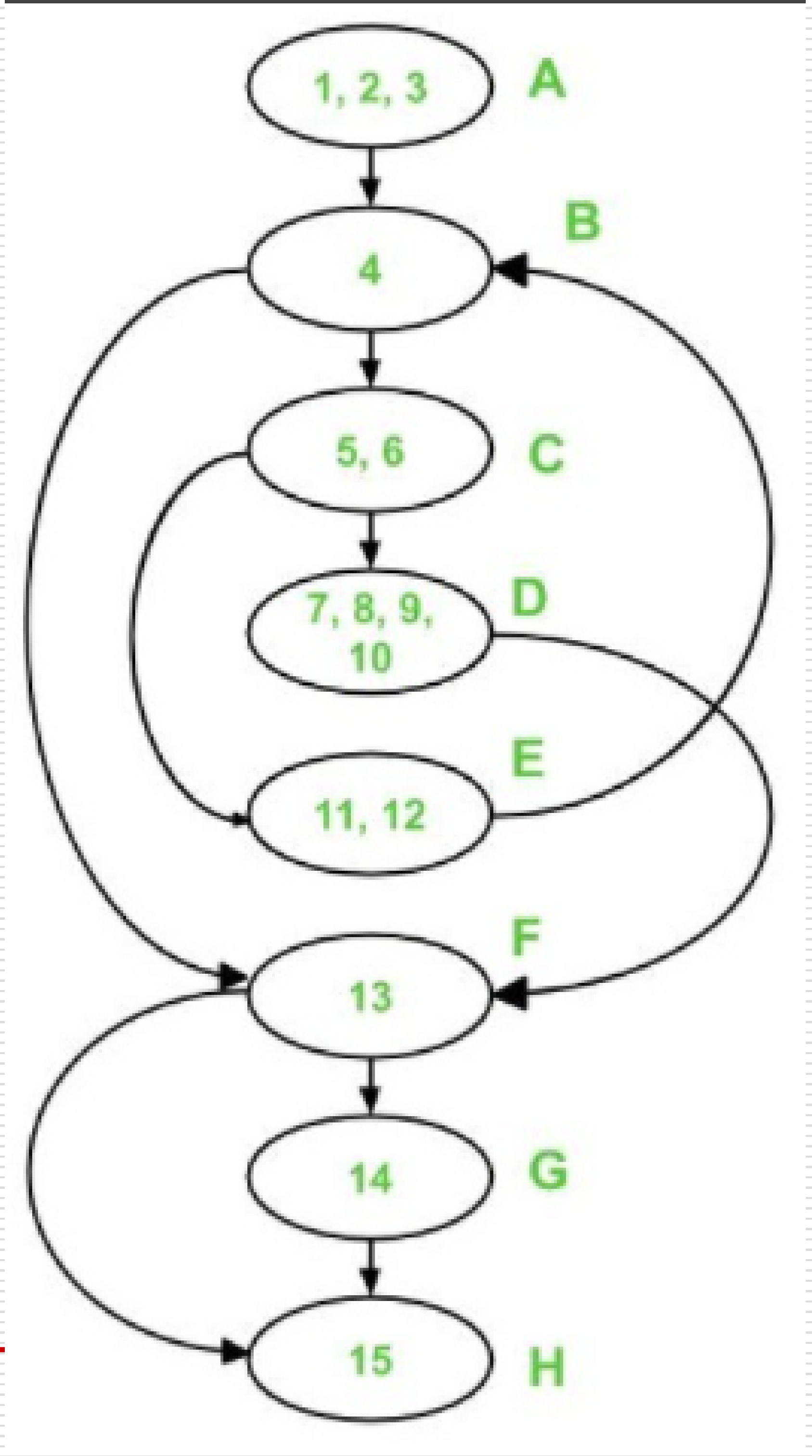
=3+1(B,C,F)

=4

V(G)= bounded region +1

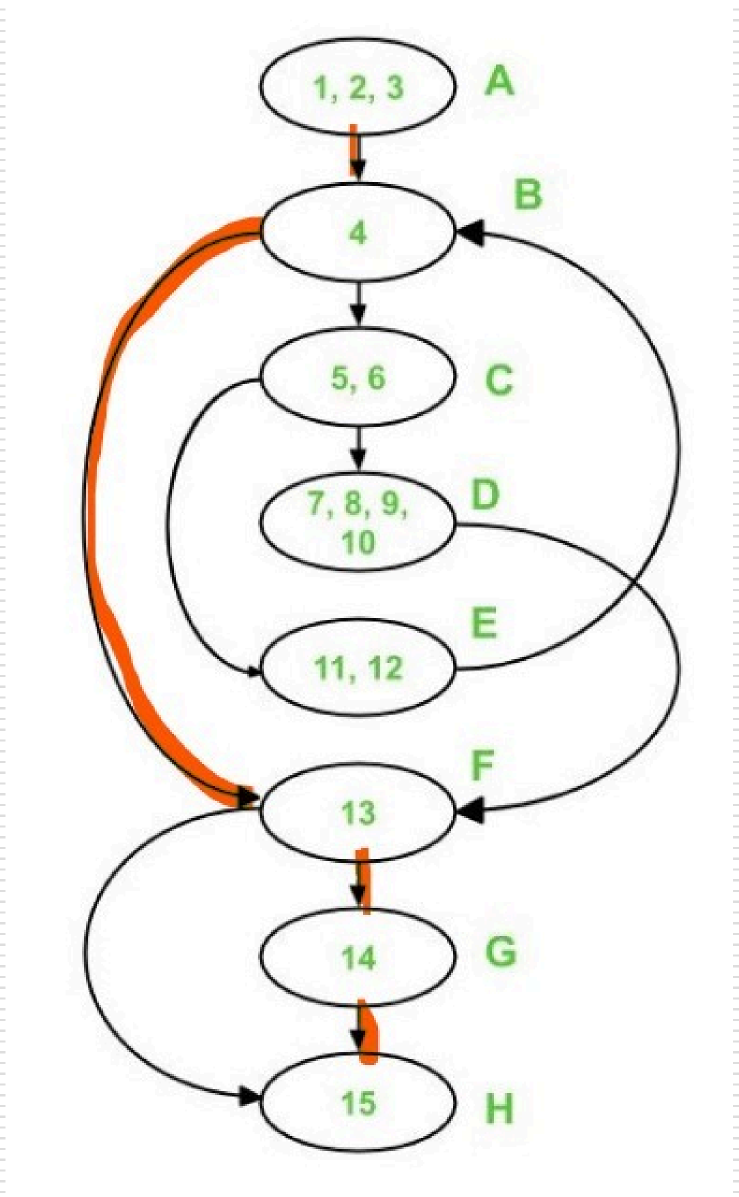
=3+1

=4



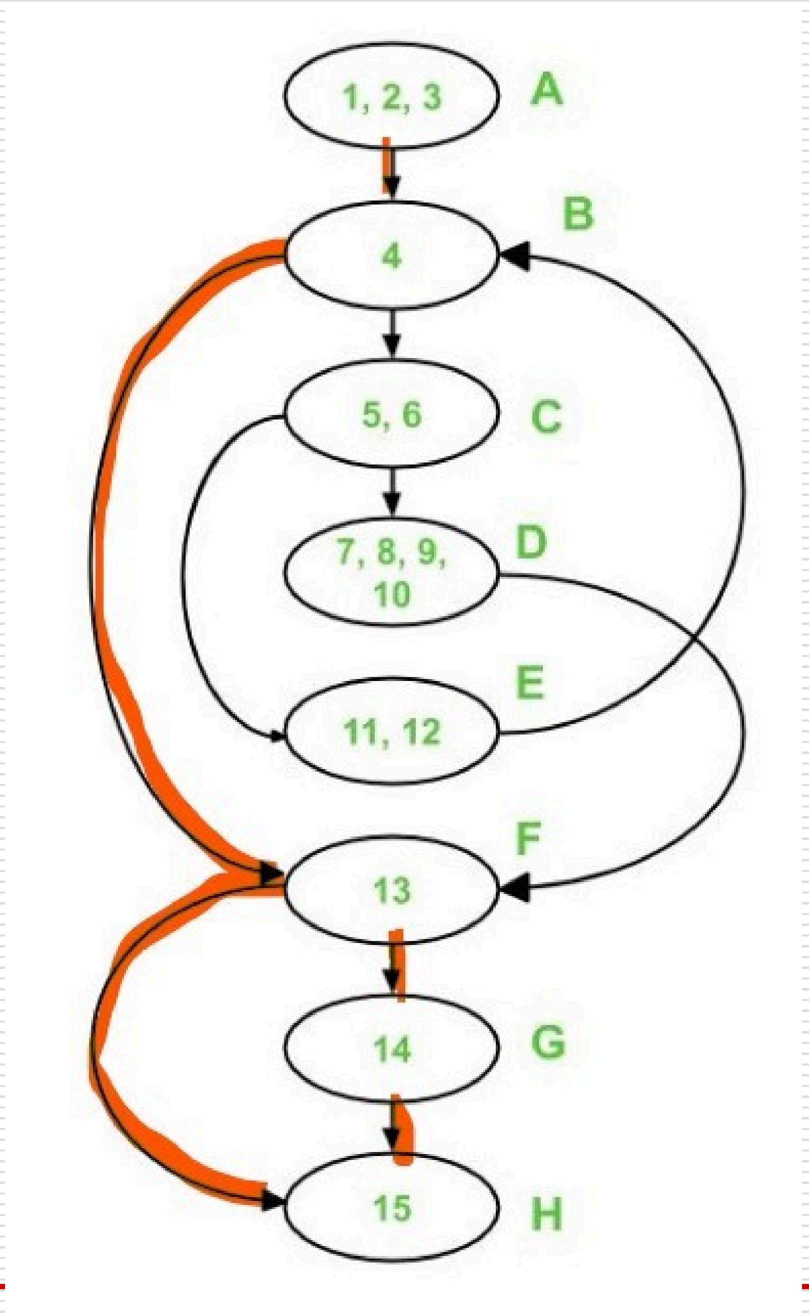
Independent paths:

PATH 1 cover:A-B-F-G-H



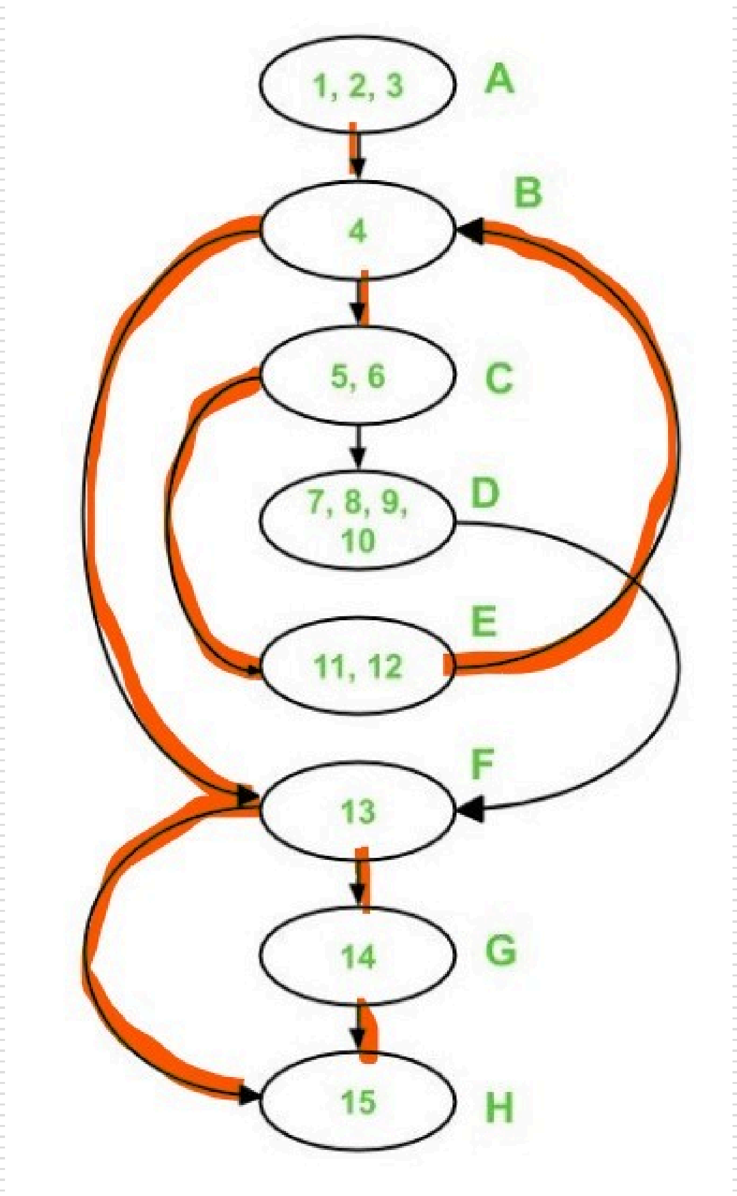
Independent paths:

PATH 1, 2 cover :A-B-F-H



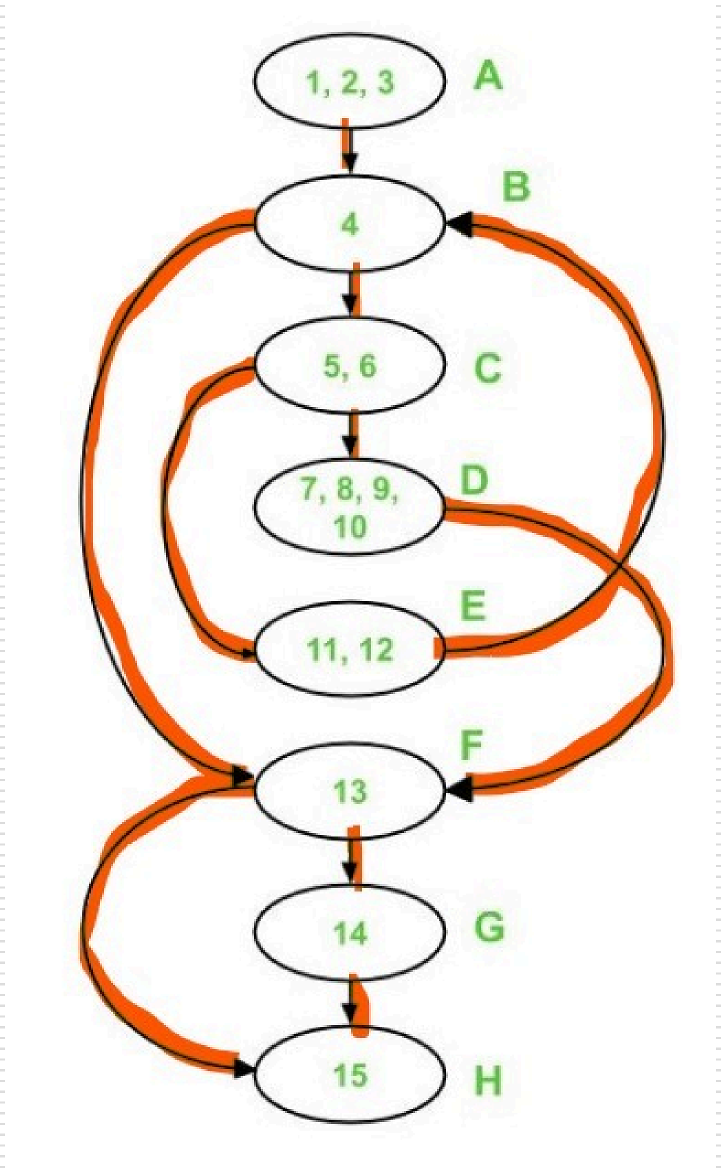
Independent paths:

PATH 1,2,3 cover:A - B - C - E - B - F - G - H

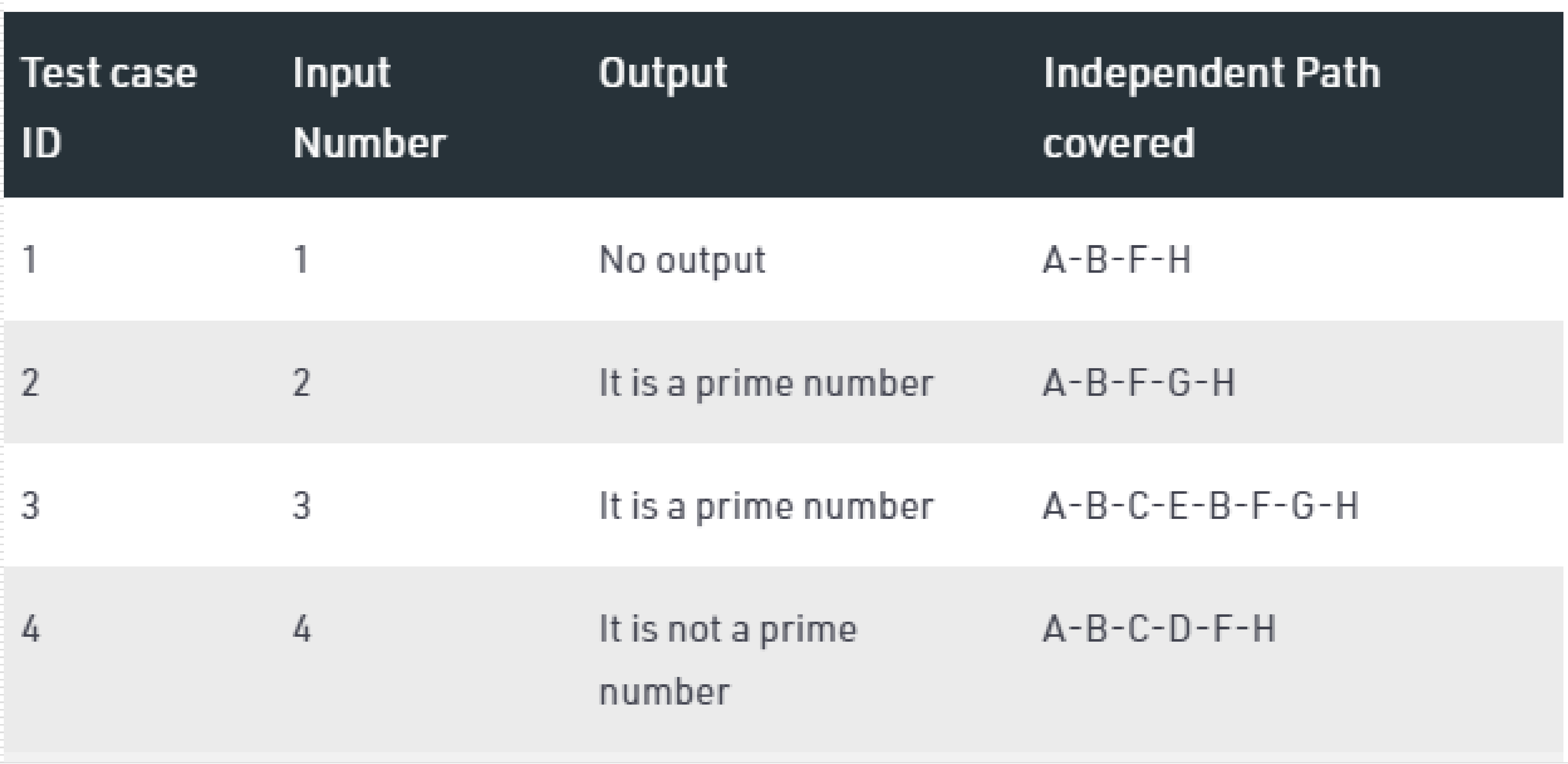


Independent paths:

Path 4 over : A - B - C - D - F - H

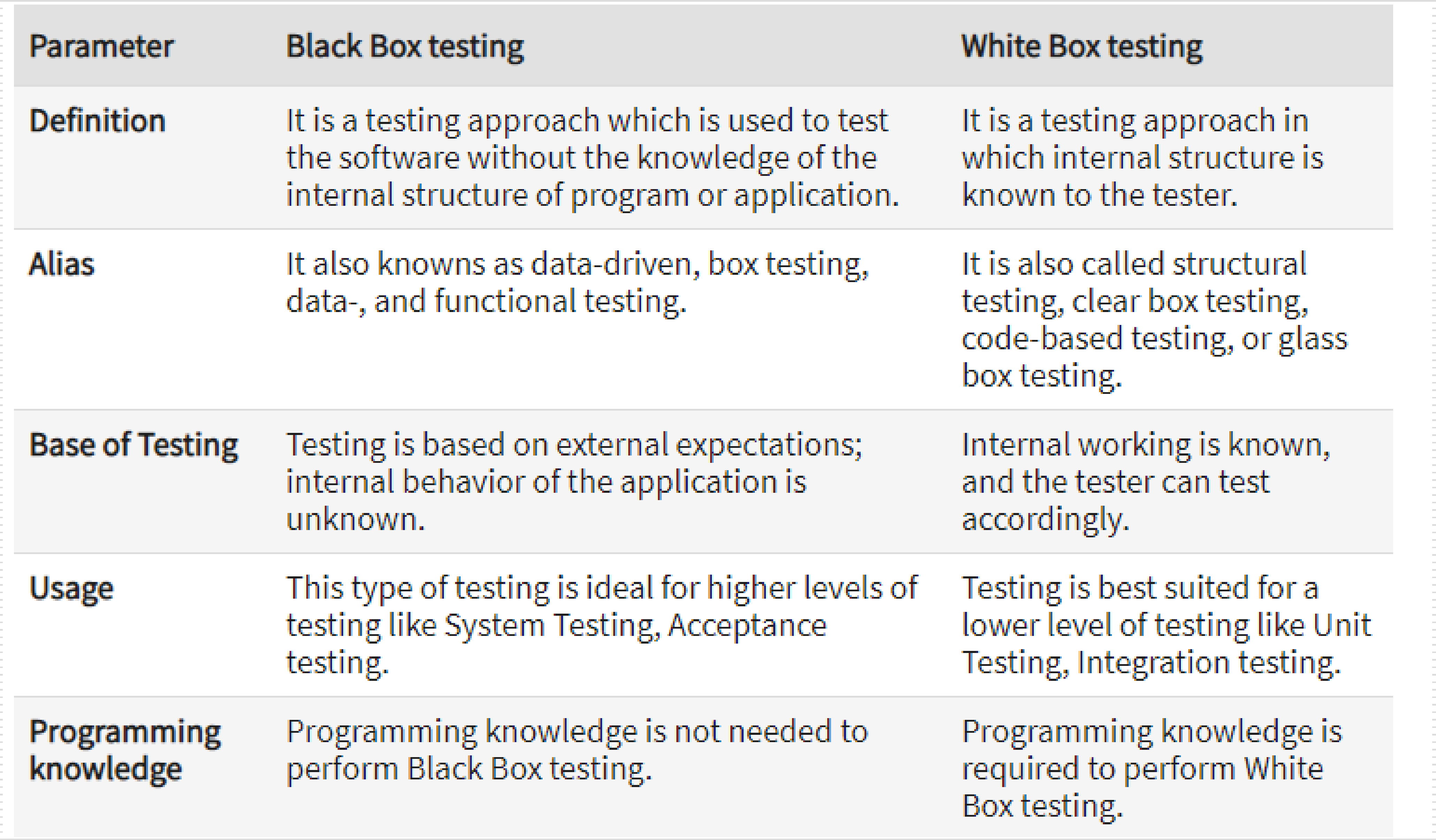


Test cases



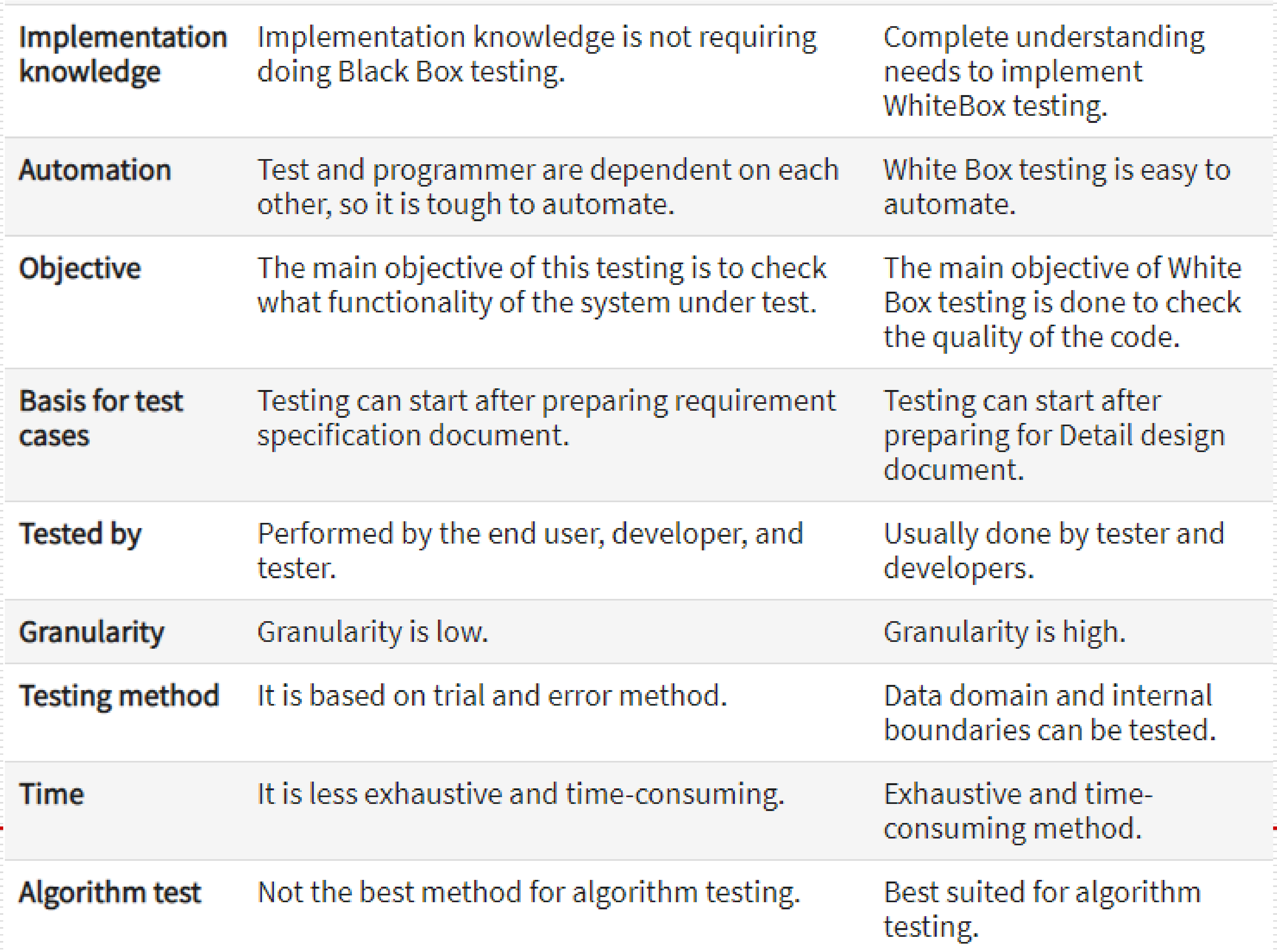
Test case ID	Input Number	Output	Independent Path covered
1	1	No output	A-B-F-H
2	2	It is a prime number	A-B-F-G-H
3	3	It is a prime number	A-B-C-E-B-F-G-H
4	4	It is not a prime number	A-B-C-D-F-H

Difference between Black Box testing and White Box testing



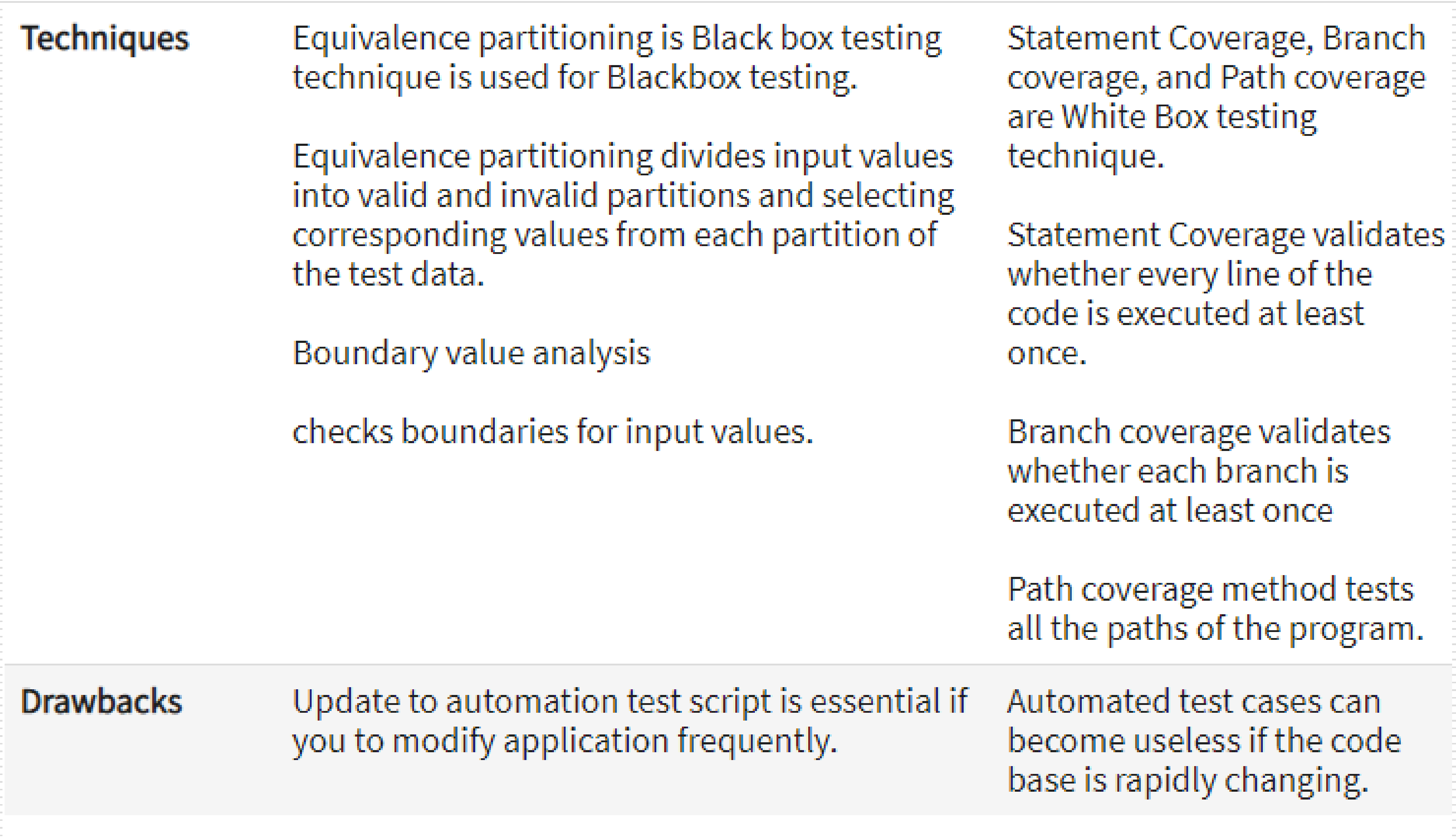
Parameter	Black Box testing	White Box testing
Definition	It is a testing approach which is used to test the software without the knowledge of the internal structure of program or application.	It is a testing approach in which internal structure is known to the tester.
Alias	It also knowns as data-driven, box testing, data-, and functional testing.	It is also called structural testing, clear box testing, code-based testing, or glass box testing.
Base of Testing	Testing is based on external expectations; internal behavior of the application is unknown.	Internal working is known, and the tester can test accordingly.
Usage	This type of testing is ideal for higher levels of testing like System Testing, Acceptance testing.	Testing is best suited for a lower level of testing like Unit Testing, Integration testing.
Programming knowledge	Programming knowledge is not needed to perform Black Box testing.	Programming knowledge is required to perform White Box testing.

Difference between Black Box testing and White Box testing



Implementation knowledge	Implementation knowledge is not requiring doing Black Box testing.	Complete understanding needs to implement WhiteBox testing.
Automation	Test and programmer are dependent on each other, so it is tough to automate.	White Box testing is easy to automate.
Objective	The main objective of this testing is to check what functionality of the system under test.	The main objective of White Box testing is done to check the quality of the code.
Basis for test cases	Testing can start after preparing requirement specification document.	Testing can start after preparing for Detail design document.
Tested by	Performed by the end user, developer, and tester.	Usually done by tester and developers.
Granularity	Granularity is low.	Granularity is high.
Testing method	It is based on trial and error method.	Data domain and internal boundaries can be tested.
Time	It is less exhaustive and time-consuming.	Exhaustive and time-consuming method.
Algorithm test	Not the best method for algorithm testing.	Best suited for algorithm testing.

Difference between Black Box testing and White Box testing



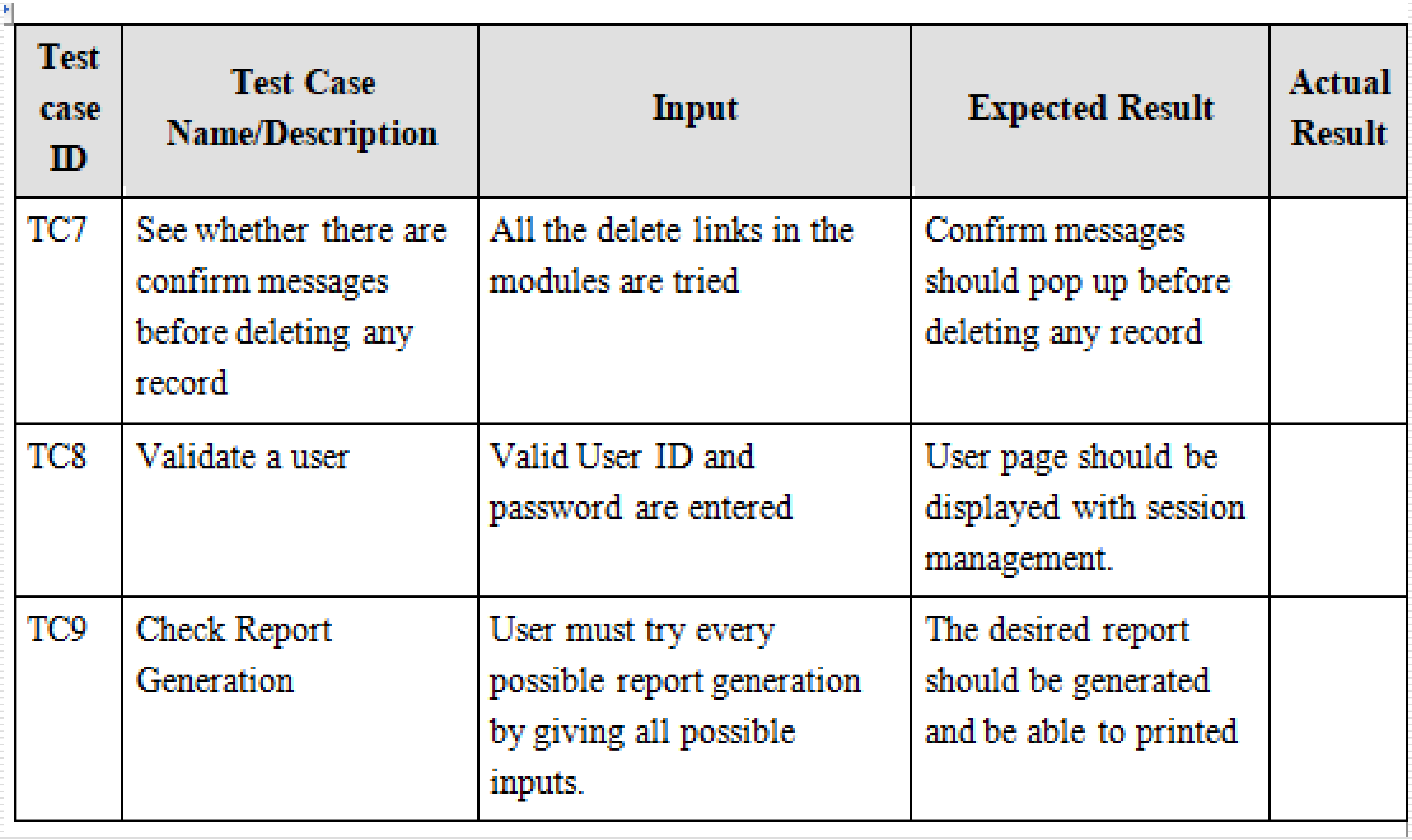
Techniques	Equivalence partitioning is Black box testing technique is used for Blackbox testing. Equivalence partitioning divides input values into valid and invalid partitions and selecting corresponding values from each partition of the test data. Boundary value analysis checks boundaries for input values.	Statement Coverage, Branch coverage, and Path coverage are White Box testing technique. Statement Coverage validates whether every line of the code is executed at least once. Branch coverage validates whether each branch is executed at least once Path coverage method tests all the paths of the program.
Drawbacks	Update to automation test script is essential if you to modify application frequently.	Automated test cases can become useless if the code base is rapidly changing.

Test Case and its Design

Test Case Design – The test case generally consists of the following details:

- Test Case ID
- Test Case Name
- Test Case Description
- Possible valid and invalid inputs
- Related Conditions
- Expected Results or Output
- Actual Result
- Test has Passed/Failed
- Remarks

Test case ID	Test Case Name/Description	Input	Expected Result	Actual Result
TC1	Check all links to other pages.	Click on all the links	The user should be redirected to the desired page.	
TC2	Check client side validations by attempting invalid inputs	Form fields are left blank, characters are entered for integer fields and integer entered for character fields	It should prompt user for valid inputs	
TC3	Check Server side validations	Form fields are left blank, characters are entered for integer fields and integer entered for character fields	It should notify user about action taken	
TC4	Check that the output is correct for each given input	All valid inputs are entered and the output action is triggered	Output must be correct with respect to Input	
TC5	Try out Session time out of a user	User signs out then home page URL is typed or back button is pressed	User should be asked for re-login	
TC6	Check out the application in other browsers also	Application tested in Internet Explorer and Mozilla Firefox	It should display similar results	



Test case ID	Test Case Name/Description	Input	Expected Result	Actual Result
TC7	See whether there are confirm messages before deleting any record	All the delete links in the modules are tried	Confirm messages should pop up before deleting any record	
TC8	Validate a user	Valid User ID and password are entered	User page should be displayed with session management.	
TC9	Check Report Generation	User must try every possible report generation by giving all possible inputs.	The desired report should be generated and be able to printed	

References

- *"Software Engineering – A Practitioner' s Approach" – Roger S. Pressman*
- *"Software Engineering" – Ian Sommerville*
- *"Fundamentals of Software Engineering" – Rajib Mall*