

# Software Testing



# Organization of this Lecture



⌘ Introduction to Testing.

⌘ White-box testing:

- ☐ statement coverage

- ☐ path coverage


- ☐ branch testing

- ☐ condition coverage

- ☐ Cyclomatic complexity

⌘ Summary

# How do you test a system?

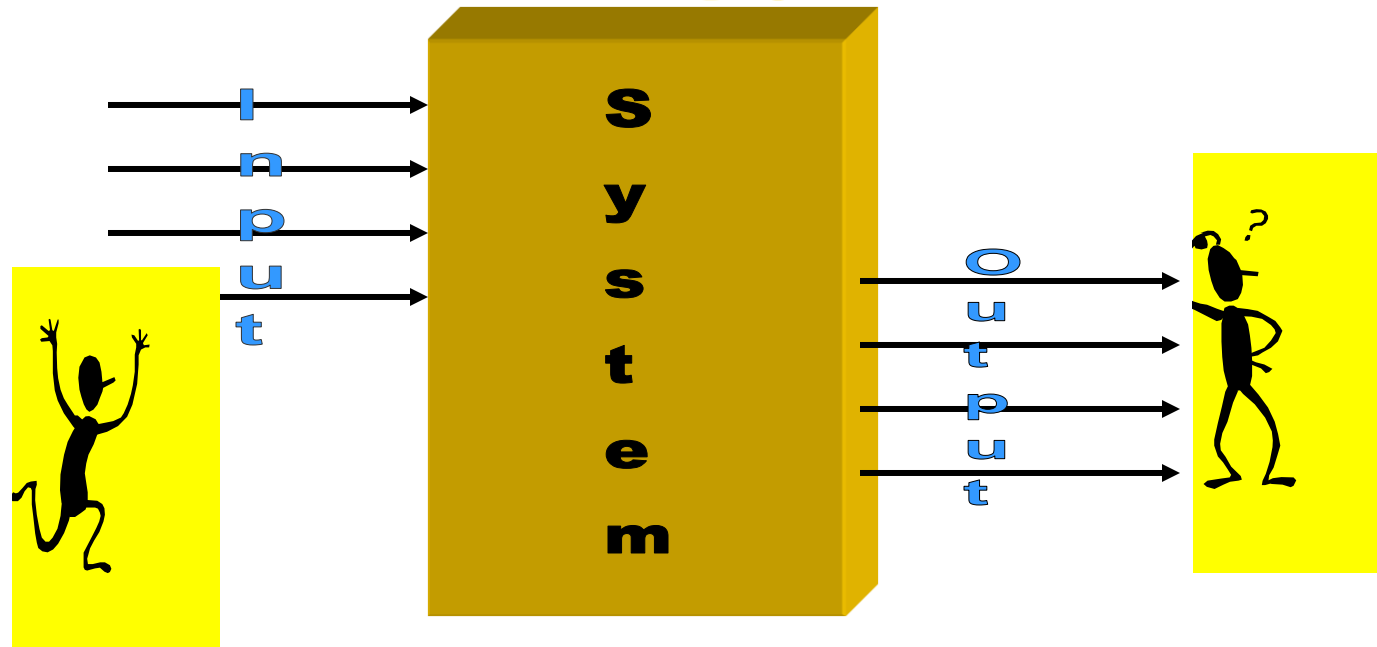


⌘ Input test data to the system.

⌘ Observe the output:

☑ Check if the system behaved as expected.

# How do you test a system?



# How do you test a system?

⌘ If the program does not behave as expected:

☐ note the conditions under which it failed.

☐ later debug and correct.

# Errors and Failures

⌘ A failure is a event of an error (aka defect or bug).

☐ mere presence of an error may not lead to a failure.

# Test cases and Test suite

⌘ Test a software using a set of carefully designed test cases:

☑ the set of all test cases is called the test suite

⌘ A test case is a triplet  $[I, S, O]$ :

☑ I is the data to be input to the system,

☑ S is the state of the system at which the data is input,

☑ O is the expected output from the system.

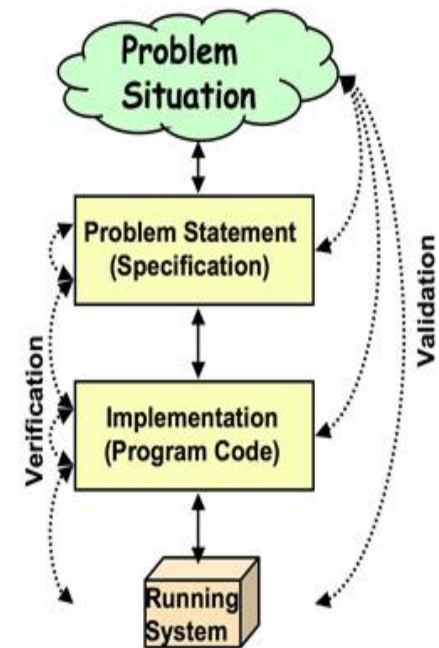
# Verification versus Validation

⌘ Verification is the process of determining:

☑ whether output of one phase of development conforms to its previous phase.

⌘ Validation is the process of determining

☑ whether a fully developed system conforms to its SRS document.





# Verification versus Validation



⌘ Aim of Verification:

- ☑ phase containment of errors
- ☑ are we doing right?

⌘ Aim of validation:

- ☑ final product is error free.
- ☑ have we done right?

# Design of Test Cases

- ⌘ Exhaustive testing of any non-trivial system is impractical:
  - ☐ input data domain is extremely large.
- ⌘ Design an **optimal test suite**:
  - ☐ of reasonable size
  - ☐ to uncover as many errors as possible.

# Design of Test Cases

⌘ If test cases are selected randomly:

- ☒ many test cases do not contribute to the significance of the test suite,
- ☒ do not detect errors not already detected by other test cases in the suite.

⌘ The number of test cases in a randomly selected test suite:

- ☒ not an indication of the effectiveness of the testing.

# Design of Test Cases

⌘ Testing a system using a large number of randomly selected test cases:

⏏ does not mean that many errors in the system will be uncovered.

⌘ Consider an example:

⏏ finding the maximum of two integers  $x$  and  $y$ .

⌘ If  $(x > y)$   $\max = y$ ;  
    else  $\max = x$ ;

⌘ The code has a simple error:

⌘ test suite  $\{(x=3, y=2); (x=2, y=3)\}$  can detect the error,

⌘ a larger test suite  $\{(x=3, y=2); (x=4, y=3); (x=5, y=1)\}$  does not detect the error.

# Testing in the large vs. testing in the small

- ⌘ Software products are normally tested first at the individual component (or unit) level. This is referred to as testing in the small.
- ⌘ After testing all the components individually, the components are slowly integrated and tested at each level of integration (integration testing). Finally, the fully integrated system is tested (called system testing) or testing at large.

## Unit testing

- ⌘ Unit testing is undertaken after a module has been coded and successfully reviewed. Unit testing (or module testing) is the testing of different units (or modules) of a system in isolation.

# Design of Test Cases



- ⌘ Systematic approaches are required to design an optimal test suite:
  - ☐ each test case in the suite should detect different errors.
- ⌘ Two main approaches to design test cases:
  - ☐ Black-box approach
  - ☐ White-box (or glass-box) approach

# Black-box Testing

⌘ Test cases are designed using only functional specification of the software:

☑ without any knowledge of the internal structure of the software.

⌘ For this reason, black-box testing is also known as functional testing.

# White-box Testing

⌘ Designing white-box test cases:

☑ requires knowledge about the internal structure of software.

☑ white-box testing is also called structural testing.



# Black-box Testing



⌘ Two main approaches to design black box test cases:

- ☑ Equivalence class partitioning

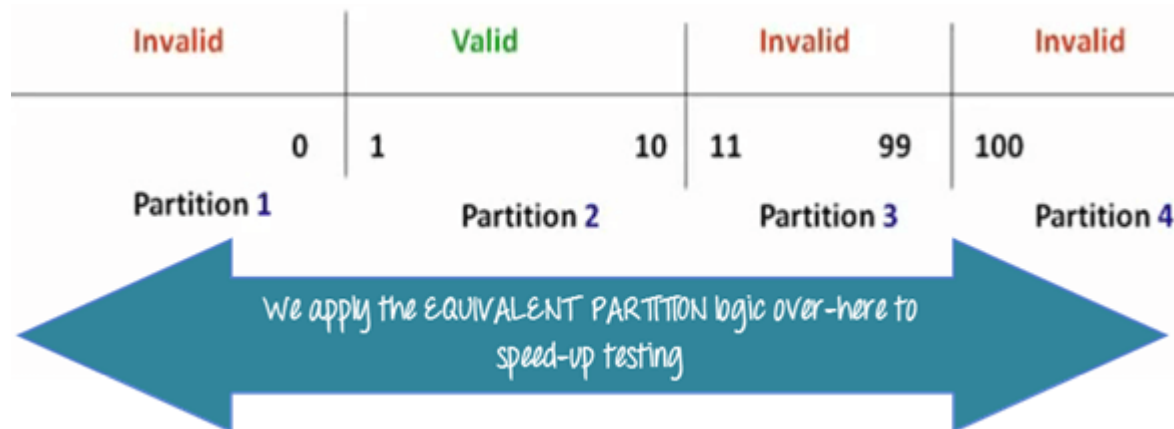
- ☑ Boundary value analysis

# Equivalence Class Partitioning

- ⌘ It is a software testing technique that divides the input test data of the application under test into each partition at least once of equivalent data from which test cases can be derived.
- ⌘ An advantage of this approach is it reduces the time required for performing testing of a software due to less number of test cases.
- ⌘ Example: Assume that the application accepts an integer in the range 100 to 999
  - ☒ Valid Equivalence Class partition: 100 to 999 inclusive.
  - ☒ Non-valid Equivalence Class partitions: less than 100, more than 999, decimal numbers and alphabets/non-numeric characters.

## ⌘ Example

- ⌘ Let's consider the behavior of Order Pizza Text Box Below
- ⌘ Pizza values 1 to 10 is considered valid. A success message is shown.
- ⌘ While value 11 to 99 are considered invalid for order and an error message will appear, **"Only 10 Pizza can be ordered"**
- ⌘ **Here is the test condition**
  - ☒ Any Number greater than 10 entered in the Order Pizza field(let say 11) is considered invalid.
  - ☒ Any Number less than 1 that is 0 or below, then it is considered invalid.
  - ☒ Numbers 1 to 10 are considered valid
  - ☒ Any 3 Digit Number say -100 is invalid.



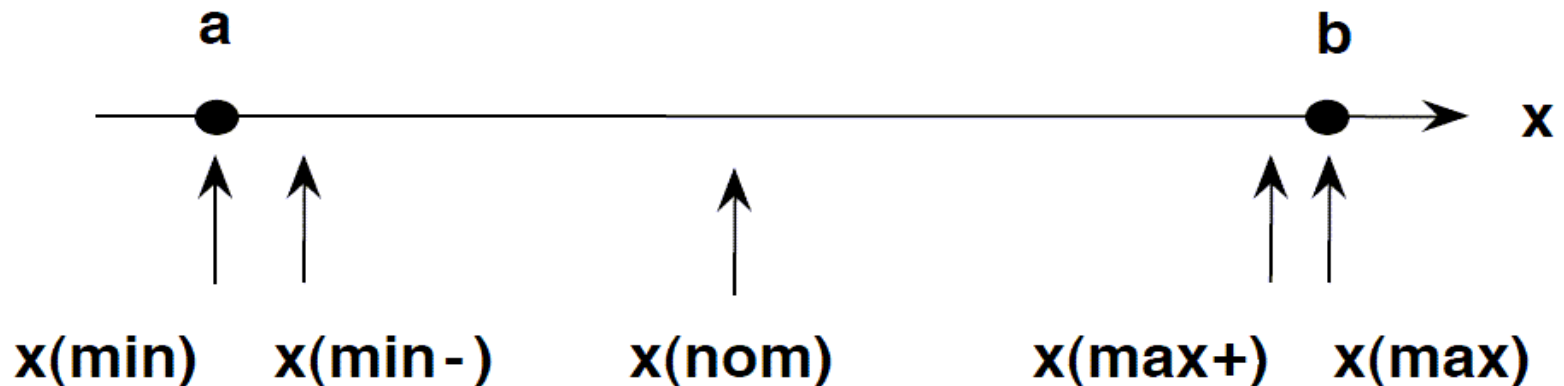
# Boundary value analysis

⌘ Boundary testing is the process of testing between extreme ends or boundaries between partitions of the input values.

⌘ So these extreme ends like Start- End, Lower- Upper, Maximum-Minimum, Just Inside-Just Outside values are called boundary values and the testing is called "**boundary testing**".

⌘ The basic idea in normal boundary value testing is to select input variable values at their:

1. Minimum
2. Just above the minimum
3. A nominal value
4. Just below the maximum
5. Maximum



# Boundary value analysis

- ⌘ Boundary value analysis is a type of black box or specification based testing technique in which tests are performed using the boundary values.
- ⌘ Example: An exam has a pass boundary at 50 percent, merit at 75 percent and distinction at 85 percent.
  - ☒ The Valid Boundary values for this scenario will be as follows:
    - ☒ 49, 50 - for pass
    - ☒ 74, 75 - for merit
    - ☒ 84, 85 - for distinction
- ⌘ Boundary values are validated against both the valid boundaries and invalid boundaries.
- ⌘ The Invalid Boundary Cases for the above example can be given as follows:
  - ☒ 0 - for lower limit boundary value
  - ☒ 101 - for upper limit boundary value

S.NO.	Boundary value analysis	Equivalence partitioning
1.	It is a technique where we identify the <b>errors at the boundaries of input data</b> to discover those errors in the input center.	It is a technique where <b>the input data is divided into partitions of valid and invalid values.</b>
2.	Boundary values are those that contain <b>the upper and lower limit</b> of a variable.	In this, the inputs to the software or the application are separated into groups expected to show similar behavior.
3.	Boundary value analysis is testing the boundaries between partitions.	It allows us to divide a set of test conditions into a partition that should be considered the same.
4.	It will help decrease testing time due to a lesser number of test cases from infinite to finite.	The Equivalence partitioning will reduce the number of test cases to a finite list of testable test cases covering maximum possibilities.
5.	The Boundary Value Analysis is often called a part of the Stress and Negative Testing.	The Equivalence partitioning can be suitable for all the software testing levels such as unit, integration, system.
6.	Sometimes the boundary value analysis is also known as Range Checking.	Equivalence partitioning is also known as Equivalence class partitioning.

# Condition coverage

- ⌘ In this structural testing, 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.
- ⌘ For a composite conditional expression of  $n$  components, for condition coverage,  $2^n$  test cases are required.
- ⌘ The number of test cases increases exponentially with the number of component conditions. It is practical only if  $n$  (the number of conditions) is small.

# White-Box Testing



⌘ There exist several popular white-box testing methodologies:

- ☑ Statement coverage
- ☑ branch coverage
- ☑ path coverage
- ☑ condition coverage
- ☑ mutation testing
- ☑ data flow-based testing



# Statement Coverage

- ⌘ In programming language, **statement is the line of code or instruction for the computer to understand** and act accordingly.
- ⌘ A statement becomes an executable statement when it gets compiled and converted into the object code and performs the action when the program is in running mode.
- ⌘ Hence “Statement Coverage”, as the name suggests, is the method of validating that each and every line of code is executed at least once.

# Statement Coverage

## ⌘ Statement coverage methodology:

- ⌘ design test cases so that

- ⌘ every statement in a program is executed at least once.

## ⌘ The principal idea:

- ⌘ unless a statement is executed,

- ⌘ we have no way of knowing if an error exists in that statement.

# Statement coverage criterion

⌘ Based on the observation:

- ⊡ an error in a program can not be discovered:
  - ⊗ unless the part of the program containing the error is executed.

⌘ Observing that a statement behaves properly for one input value:

- ⊡ no guarantee that it will behave correctly for all input values.

# Example

```
⌘ int f1(int x, int y){  
⌘ 1 while (x != y){  
⌘ 2   if (x>y) then  
⌘ 3       x=x-y;  
⌘ 4   else y=y-x;  
⌘ 5 }  
⌘ 6 return x; }
```

Euclid's GCD Algorithm

By choosing the test set

$\{(x=3, y=3), (x=4, y=3), (x=3, y=4)\}$

all statements are executed at least once.

# Branch Coverage

⏏ Branch Coverage is a testing method which when executed ensures that each branch from each decision point is executed. (e.g., if statements, loops)

⏏ So in Branch coverage (also called Decision coverage), we validate that each branch is executed at least once.

⏏ In case of a “IF statement”, there will be two test conditions:

⏏ One to validate the **true** branch and

⏏ Other to validate the **false** branch

# Branch testing

- ⌘ Branch testing is the simplest condition testing strategy:

- ☑ compound conditions appearing in different branch statements

- ☒ are given true and false values.

- ⌘ Condition testing

- ☑ stronger testing than branch testing:

- ⌘ Branch testing

- ☑ stronger than statement coverage testing.

# Branch Coverage

- ⌘ Test cases are designed such that:
  - ⏏ different branch conditions
  - ⏏ given true and false values in turn.
- ⌘ Branch testing guarantees statement coverage:
  - ⏏ a stronger testing compared to the statement coverage-based testing.

# Stronger testing

- ⌘ Test cases are a superset of a weaker testing:
  - ⏏ discovers at least as many errors as a weaker testing
  - ⏏ contains at least as many significant test cases as a weaker test.



# Condition Coverage

⌘ Test cases are designed such that:

☐ each component of a composite conditional expression

☒ given both true and false values.

⌘ Consider the conditional expression

☐  $((c1.and.c2).or.c3)$ :

⌘ Each of  $c1$ ,  $c2$ , and  $c3$  are exercised at least once,

☐ i.e. given true and false values.

# Path Coverage



⌘ Design test cases such that:

☑ all linearly independent paths in the program are executed at least once.

# Path coverage

- ⌘ Path coverage tests all the paths of the program. This is a comprehensive technique which ensures that all the paths of the program are traversed at least once.
- ⌘ Path Coverage is even more powerful than Branch coverage. This technique is useful for testing the complex programs.

# Example

```
1    INPUT A & B
2    C = A + B
3    IF C>100
4    PRINT "IT'S DONE"
```

⌘ For **Statement Coverage** – we would need only one test case to check all the lines of code.

⌘ If consider *TestCase\_01 to be (A=40 and B=70)*, then all the lines of code will be executed

⌘ Is that sufficient?

⌘ What if I consider my Test case as A=33 and B=45?

- ⌘ Because Statement coverage will only cover the true side, for the pseudo code, only one test case would NOT be sufficient to test it.
- ⌘ As a tester, we have to consider the negative cases as well.
- ⌘ Hence for maximum coverage, we need to consider “**Branch Coverage**”, which will evaluate the “FALSE” conditions.

# So now the pseudo code becomes:

```
1  INPUT A & B
2  C = A + B
3  IF C>100
4  PRINT "IT'S DONE"
5  ELSE
6  PRINT "IT'S PENDING"
```

⌘ So for Branch coverage, we would require two test cases to complete testing of this pseudo code.

☒ **TestCase\_01:** A=33, B=45

☒ **TestCase\_02:** A=25, B=80

⌘ With this, each and every line of code is executed at least once.

☒ **Branch Coverage ensures more coverage than Statement coverage**

☒ **100% Branch coverage itself means 100% statement coverage,**

⌘ Path coverage is used to test the complex code snippets, which basically involves loop statements or combination of loops and decision statements.

```
1  INPUT A & B
2  C = A + B
3  IF C>100
4  PRINT "IT'S DONE"
5  END IF
6  IF A>50
7  PRINT "IT'S PENDING"
8  END IF
```

⌘ ensure maximum coverage, which require 4 test cases.

⌘ there are 2 decision statements, so for each decision statement we would need to branches to test. One for true and other for false condition.

⌘ So for 2 decision statements, require 2 test cases to test the true side and 2 test cases to test the false side, which makes total of 4 test cases.

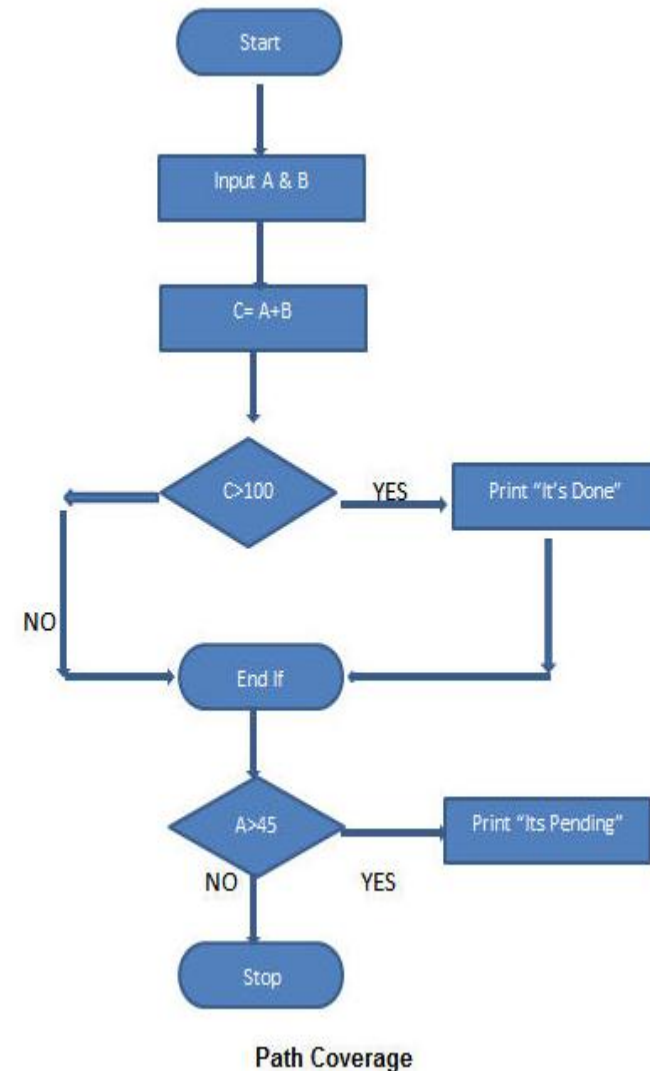
⌘ So, In order to have the full coverage, following test cases are:

⌘ **TestCase\_01:** A=50, B=60

⌘ **TestCase\_02:** A=55, B=40

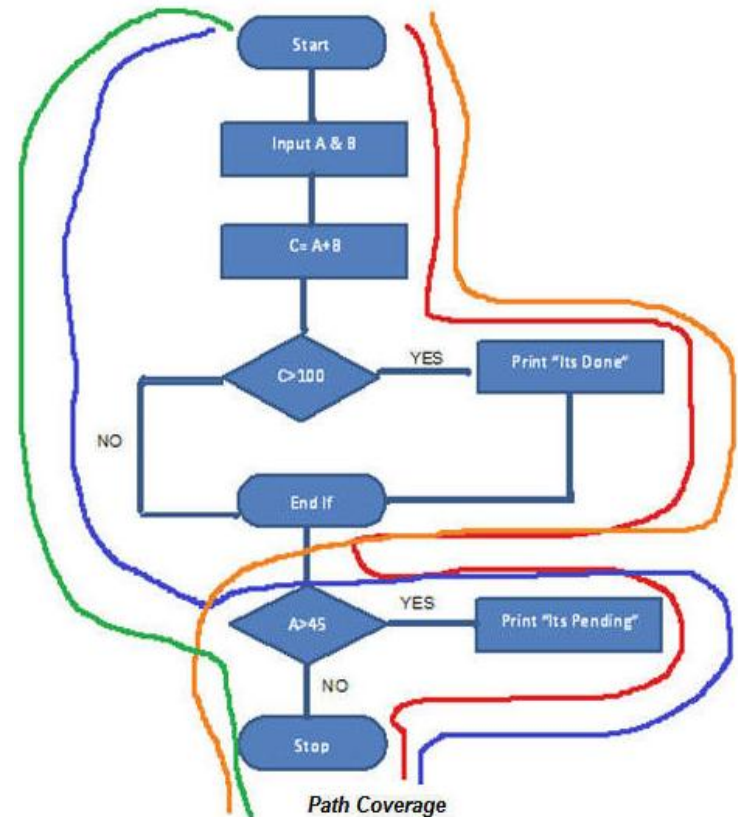
⌘ **TestCase\_03:** A=40, B=65

⌘ **TestCase\_04:** A=30, B=30





- ⌘ Red Line – TestCase\_01 = (A=50, B=60)
- ⌘ Blue Line = TestCase\_02 = (A=55, B=40)
- ⌘ Orange Line = TestCase\_03 = (A=40, B=65)
- ⌘ Green Line = TestCase\_04 = (A=30, B=30)



# White-Box Testing



⌘ There exist several popular white-box testing methodologies:

- ☑ Statement coverage
- ☑ branch coverage
- ☑ path coverage
- ☑ condition coverage
- ☑ mutation testing
- ☑ data flow-based testing

# Control flow graph (CFG)

- ⌘ A control flow graph (CFG) describes:
  - ☐ the sequence in which different instructions of a program get executed.
  - ☐ the way control flows through the program.

# How to draw Control flow graph?

- ☒ Number all the statements of a program.

- ☒ Numbered statements:

  - ☒ represent nodes of the control flow graph

- ⌘ An edge from one node to another node exists:

  - ☒ if execution of the statement representing the first node

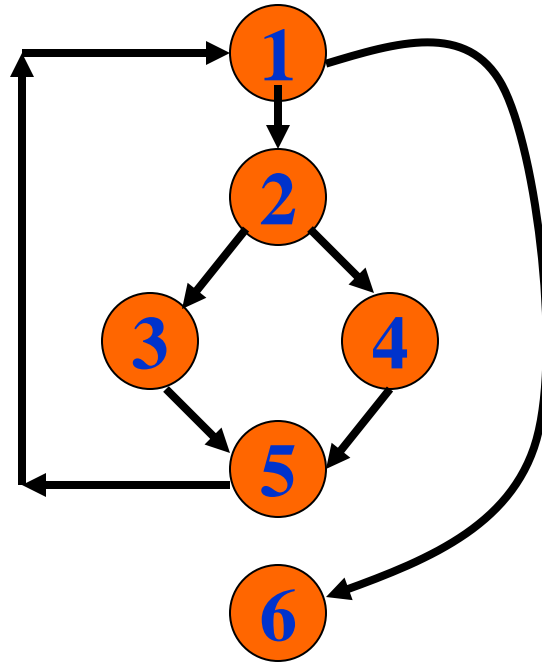
  - ☒ can result in transfer of control to the other node.

# Example



```
⌘ int f1(int x,int y){  
⌘1  while (x != y){  
⌘2    if (x>y) then  
⌘3      x=x-y;  
⌘4    else y=y-x;  
⌘5  }  
⌘6 return x;      }
```

# Example Control Flow Graph

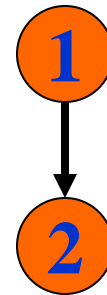


# How to draw Control flow graph?

⌘ Sequence:

▣1  $a=5;$

▣2  $b=a*b-1;$



# How to draw Control flow graph?

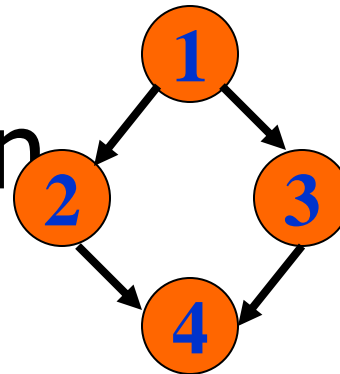
⌘ Selection:

☐ 1 if( $a > b$ ) then

☐ 2  $c = 3;$

☐ 3 else  $c = 5;$

☐ 4  $c = c * c;$

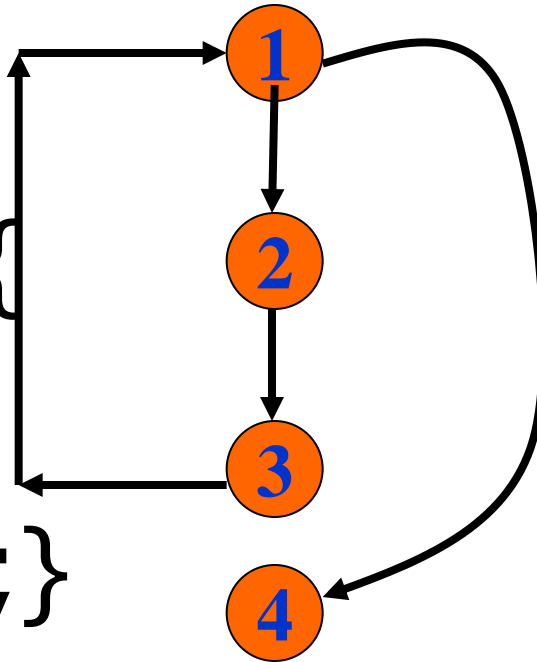




# How to draw Control flow graph?

⌘ Iteration:

```
⌘ 1 while(a>b){  
⌘ 2     b=b*a;  
⌘ 3     b=b-1;}  
⌘ 4 c=b+d;
```



# Path

⌘ A path through a program:

☐ a node and edge sequence from the starting node to a terminal node of the control flow graph.

☐ There may be several terminal nodes for program.

# Independent path

⌘ Any path through the program:

☑ introducing at least one new node:

☒ that is not included in any other independent paths.

⌘ It is straight forward:

☑ to identify linearly independent paths of simple programs.

⌘ For complicated programs:

☑ it is not so easy to determine the number of independent paths.

# McCabe's cyclomatic metric

⌘ An upper bound:

⌘ for the number of linearly independent paths of a program

⌘ Provides a practical way of determining:

⌘ the maximum number of linearly independent paths in a program.

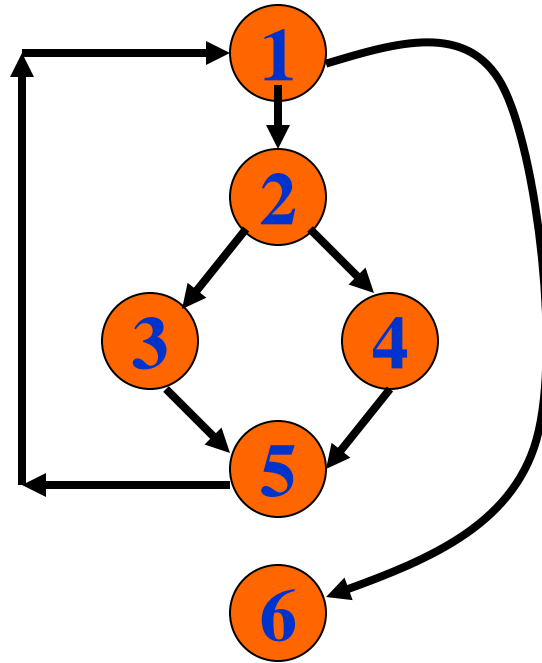
⌘ Given a control flow graph  $G$ , cyclomatic complexity  $V(G)$ :

⌘  $V(G) = E - N + 2$

⌘  $N$  is the number of nodes in  $G$

⌘  $E$  is the number of edges in  $G$

# Example Control Flow Graph



**Cyclomatic complexity =  $7 - 6 + 2 = 3$ .**

# Cyclomatic complexity

⌘ Another way of computing cyclomatic complexity:

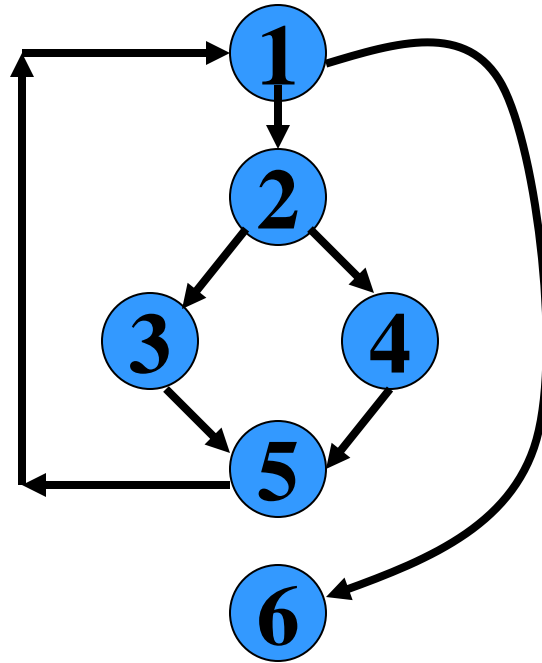
- ☒ inspect control flow graph

- ☒ determine number of bounded areas in the graph

- ☒ **bounded areas:** Any region enclosed by a nodes and edge sequence.

⌘  $V(G) = \text{Total number of bounded areas} + 1$

# Example Control Flow Graph



# Example



⌘ From a visual examination of the CFG:

☐ the number of bounded areas is 2.

☐ cyclomatic complexity =  $2+1=3$ .



# Cyclomatic complexity

⌘ McCabe's metric provides:

✕ a quantitative measure of testing difficulty and the ultimate reliability

⌘ Intuitively,

✕ number of bounded areas increases with the number of decision nodes and loops.

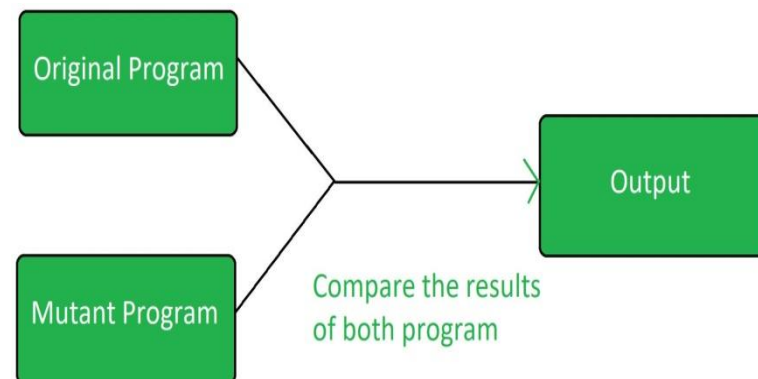
⌘ Knowing the number of test cases required:

✕ does not make it any easier to derive the test cases,

✕ only gives an indication of the minimum number of test cases required.

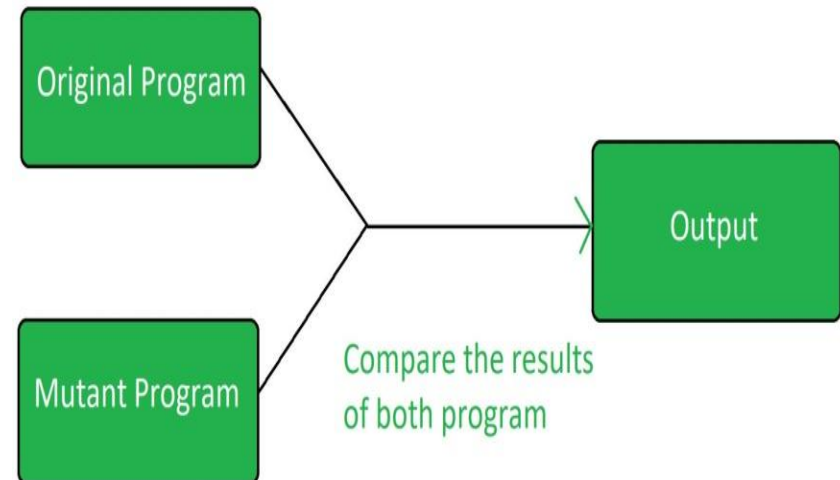
# Mutation testing

- ⌘ The software is first tested by using an initial test suite built up from the different white box testing strategies.
- ⌘ After the initial testing is complete, mutation testing is taken up.
- ⌘ The goal of **Mutation Testing** is ensuring the quality of test cases in terms of robustness that it should fail the mutated source code.
- ⌘ **Mutation Testing** is a type of Software Testing that is performed to design new software tests and also evaluate the quality of already existing software tests.
- ⌘ Mutation testing is related to modification a program in small ways. It focuses to help the tester develop effective tests or locate weaknesses in the test data used for the program.



# Mutation testing

- ⌘ Idea behind mutation testing is to make few arbitrary changes to a program at a time.
  - ☒ Each time the program is changed, it is called as a **mutated program** and the change effected is called as a **mutant**.
  - ☒ A mutated program is tested against the full test suite of the program.
  - ☒ If there exists at least one test case in the test suite for which a mutant gives an **incorrect result**, then the mutant is said to be dead.



# Mutation testing

## ⌘ Types of Mutation Testing:

Mutation testing is basically of 3 types:

### 1. Value Mutations:

In this type of testing the values are changed to detect errors in the program.

Basically a **small value is changed to a larger value or a larger value is changed to a smaller value**. In this testing basically constants are changed. **Example:**

#### 📁 Initial Code:

```
int mod = 1000000007;  
int a = 12345678;  
int b = 98765432;  
int c = (a + b) % mod;
```

#### Changed Code:

```
int mod = 1007;  
int a = 12345678;  
int b = 98765432;  
int c = (a + b) % mod;
```

# Mutation testing

## ⌘ Types of Mutation Testing:

### 2. Decision Mutations:

In decisions **mutations are logical or arithmetic operators** are changed to detect errors in the program. **Example:**

#### ⌘ Initial Code:

```
if(a < b)
c = 10;
else c = 20;
```

#### Changed Code:

```
if(a > b)
c = 10;
else c = 20;
```

### 3. Statement Mutations:

In statement mutations a **statement is deleted or it is replaced by some other statement.** **Example:**

⌘ **Initial Code:** `if(a < b)     c = 10; else c = 20;`

⌘ **Changed Code:** `if(a < b) d = 10; else d = 20;`

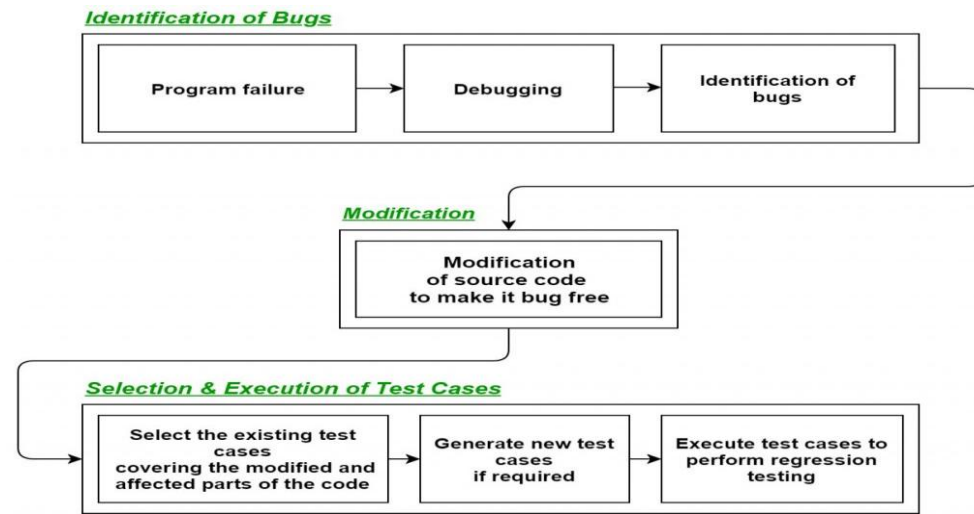
#### ⌘ Advantages of Mutation Testing:

- ☒ It brings a good level of error detection in the program.
- ☒ It discovers ambiguities in the source code.

⌘ **Disadvantage** is that it is computationally very expensive, since a large number of possible mutants can be generated.

# Regression Testing

- ⌘ **Regression Testing** is the process of testing the modified parts of the code and the parts that might get affected due to the modifications to ensure that no new errors have been introduced in the software after the modifications have been made.
- ⌘ Regression means return of something and in the software field, it refers to the return of a bug.
- ⌘ **When to do regression testing?**
  - ☒ When a new functionality is added to the system and the code has been modified to absorb and integrate that functionality with the existing code.
  - ☒ When some defect has been identified in the software and the code is debugged to fix it.
  - ☒ When the code is modified to optimize its working.



## ⌘ Advantages of Regression Testing:

- ☒ It ensures that no new bugs has been introduced after adding new functionalities to the system.
- ☒ As most of the test cases used in Regression Testing are selected from the existing test suite and we already know their expected outputs. Hence, it can be easily automated by the automated tools.
- ☒ It helps to maintain the quality of the source code.

## ⌘ Disadvantages of Regression Testing:

- ☒ It can be time and resource consuming if automated tools are not used.
- ☒ It is required even after very small changes in the code.