



# CUI Abbottabad

Department of Computer Science

## SOFTWARE TESTING

**Lecture 7 & 8**

**Whitebox Testing Technique  
Control Flow Testing**

# WHITEBOX TESTING

- ▶ Whitebox testing is also known as structural testing.
- ▶ In Whitebox testing, the software is viewed as a white box and test cases are determined from the implementation of the software.
- ▶ Whitebox testing techniques include control flow testing and data flow testing.

# WHITEBOX TESTING

## What do you verify in White Box Testing ?

White box testing involves the testing of the software code for the following:

- ☒ Internal security holes
- ☒ Broken or poorly structured paths in the source code
- ☒ The functionality of conditional loops
- ☒ Testing of each statement, condition and function
- ☒ Data Flow analysis :The flow of specific inputs through the code
- ☒ Dynamic Analysis : Memory leakages , wild pointers ....
- ☒ Mutation testing

# CONTROL FLOW TESTING

- ▶ Control flow testing uses the ***control structure of a program*** to develop the test cases for the program.
- ▶ The control structure of program is represented by ***control flow graph***.
- ▶ Control-flow testing techniques are based on carefully ***selecting a set of test paths*** through the program. The set of paths chosen is used to achieve a certain measure of testing thoroughness.
  - ▶ E.g., pick enough paths to assure that every source statement is executed as least once.

# CONTROL FLOW TESTING

For another set of input data, the unit may execute a different path.

1. if(a>b)
2. {a}
3. Else if (b>a)
4. {b}

The main idea in **control flow testing** is to appropriately **select a few paths** in a program unit and **observe whether or not the selected paths produce the expected outcome**. By executing a few paths in a program unit, the programmer tries to assess the behavior of the entire program unit.

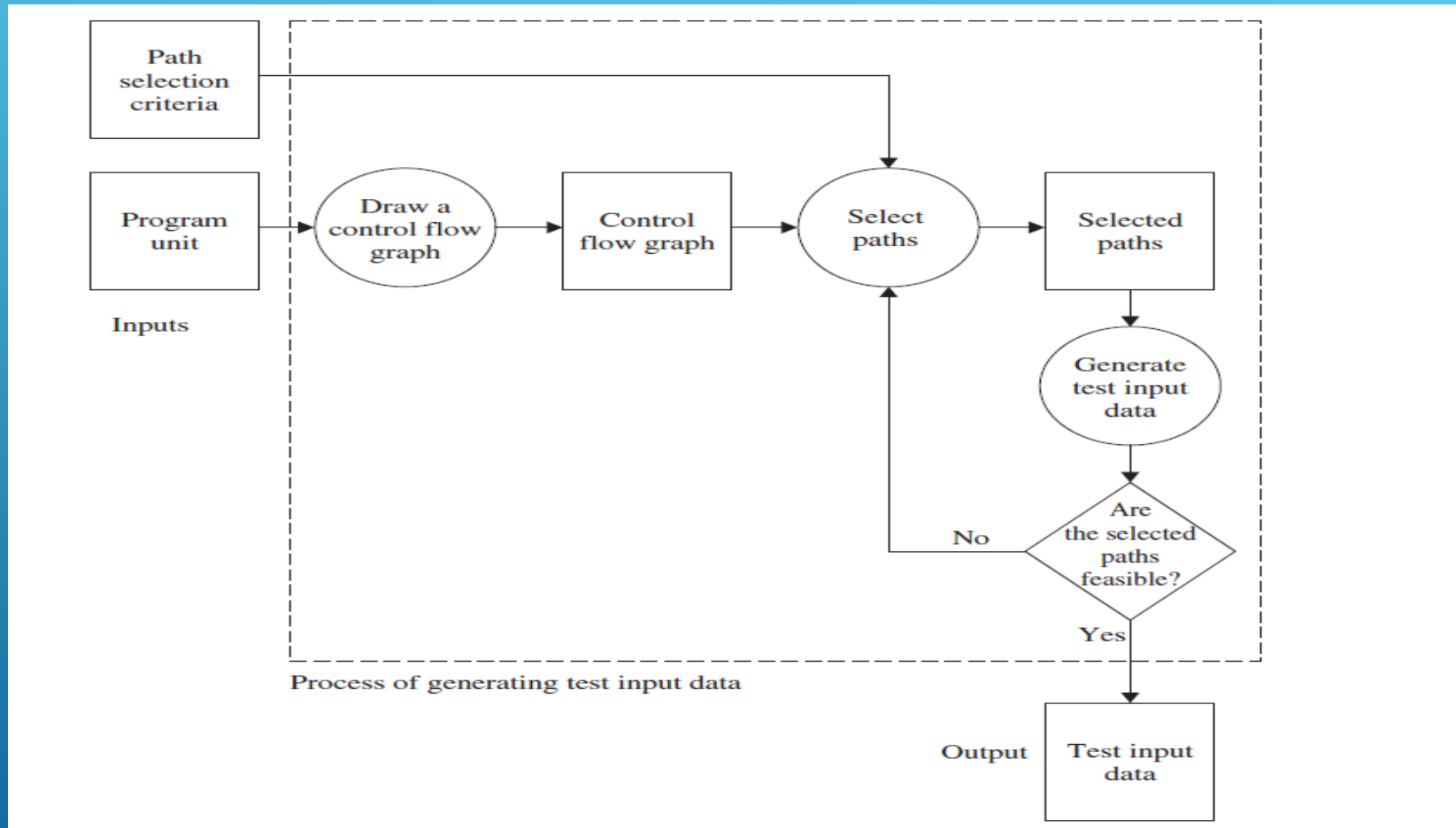
# PATH

- ▶ **Path:** A sequence of statement execution consisting as a sequence of computation and decision nodes that begins at an entry node and ends at an exit node.
- ▶ Structurally, a path is a sequence of statements in a program unit, whereas, semantically, it is an execution instance of the unit.
- ▶ We also specify whether control exits a decision node via its true or false branch while including it in a path.
- ▶ **Linearly Independent Path**
  - ▶ A path through the system is Linearly Independent from other paths only if it includes some segment or edge that is not covered in the other path.

# PATH SELECTION CRITERIA

- ▶ Select paths such that every statement is executed at least once.
- ▶ Select paths such that every conditional statement, for example, an if() statement, evaluates to *true* and *false* at least once on different occasions.
- ▶ A conditional statement may evaluate to true in one path and false in a second path.

# PROCESS OF GENERATING TEST INPUT DATA FOR CONTROL FLOW TESTING

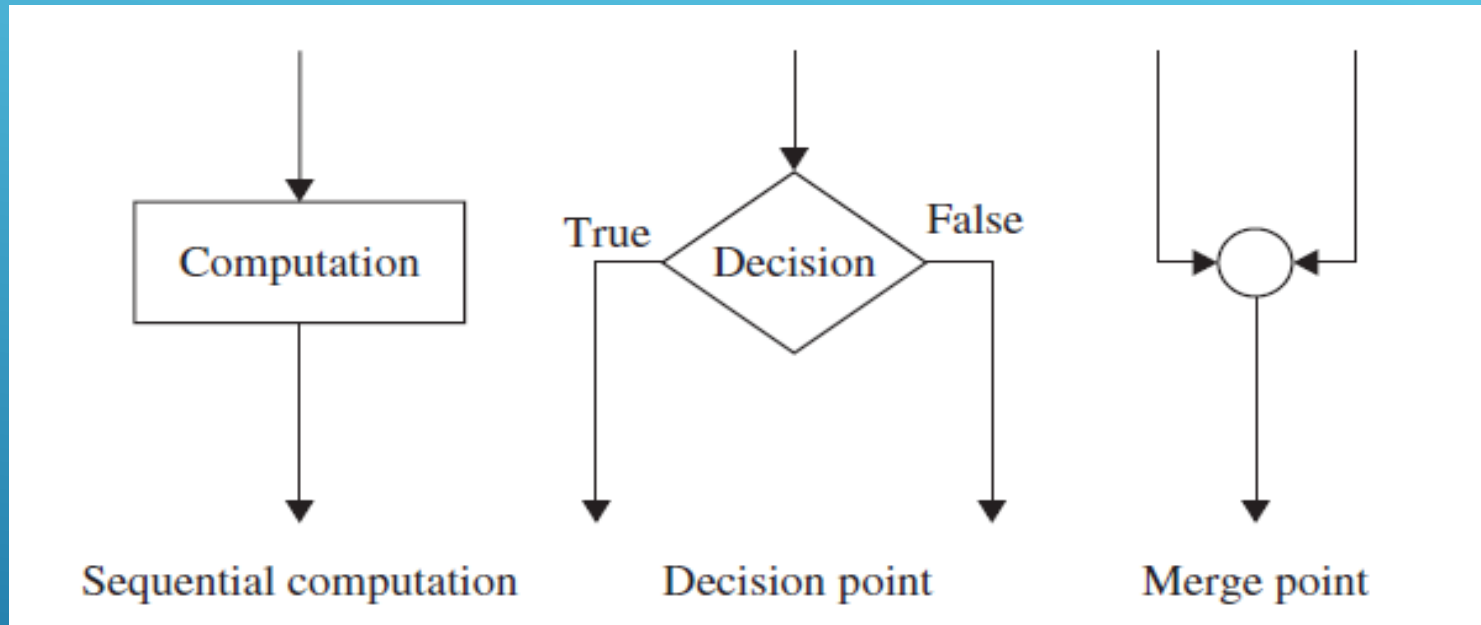




# CONTROL FLOW GRAPH (CFG)

- ▶ **Control Flow Graph** is formed from the node, edge, decision node, junction node to specify all possible execution path.
- ▶ The control flow graph  $G = (N, E)$  of a program consists of a set of nodes  $N$  and a set of edge  $E$ .
- ▶ Notations used for Control Flow Graph
  - ▶ **Node (N)** : Each node represents a set of program statements.
  - ▶ **Edge (E)** : An edge is represented as direction arrows in which the nodes are connected to other nodes. It is responsible for connecting the first node until the end node.
  - ▶ **Decision Node**: A decision is a program point at which the control can deviate. (e.g., if and case statements).
  - ▶ **Merge/Junction node**: A junction is a program point where the multiple control branches merge. (e.g., end if, end loop, goto label)

# NOTATIONS IN CFG

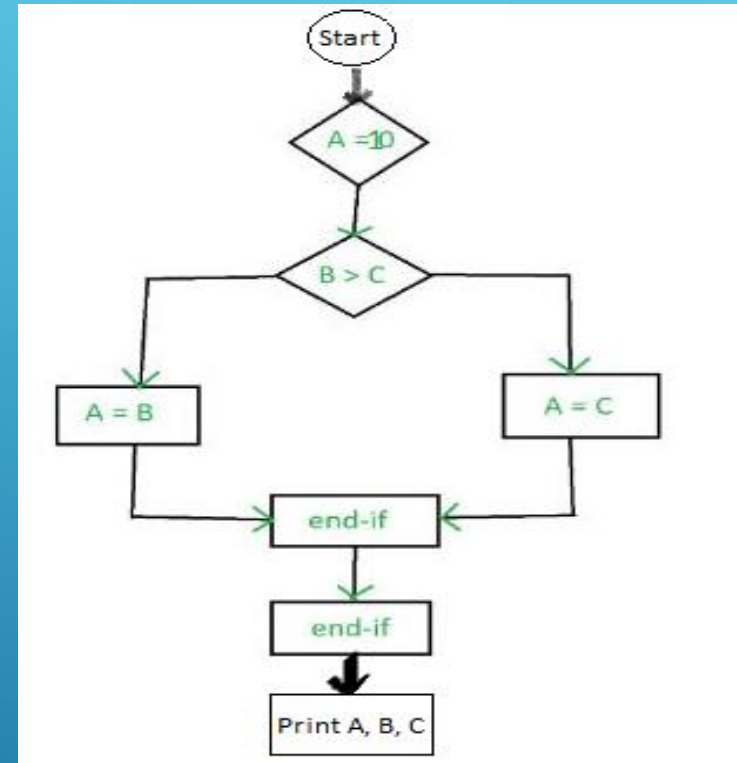


# CFG EXAMPLE

1. if  $A = 10$  then
2.   if  $B > C$
3.      $A = B$
4.   else  $A = C$
5.   endif
6.   endif
7. print  $A, B, C$

# CFG EXAMPLE

1. if  $A = 10$  then
2.   if  $B > C$
3.      $A = B$
4.   else  $A = C$
5.   endif
6. endif
7. print A, B, C

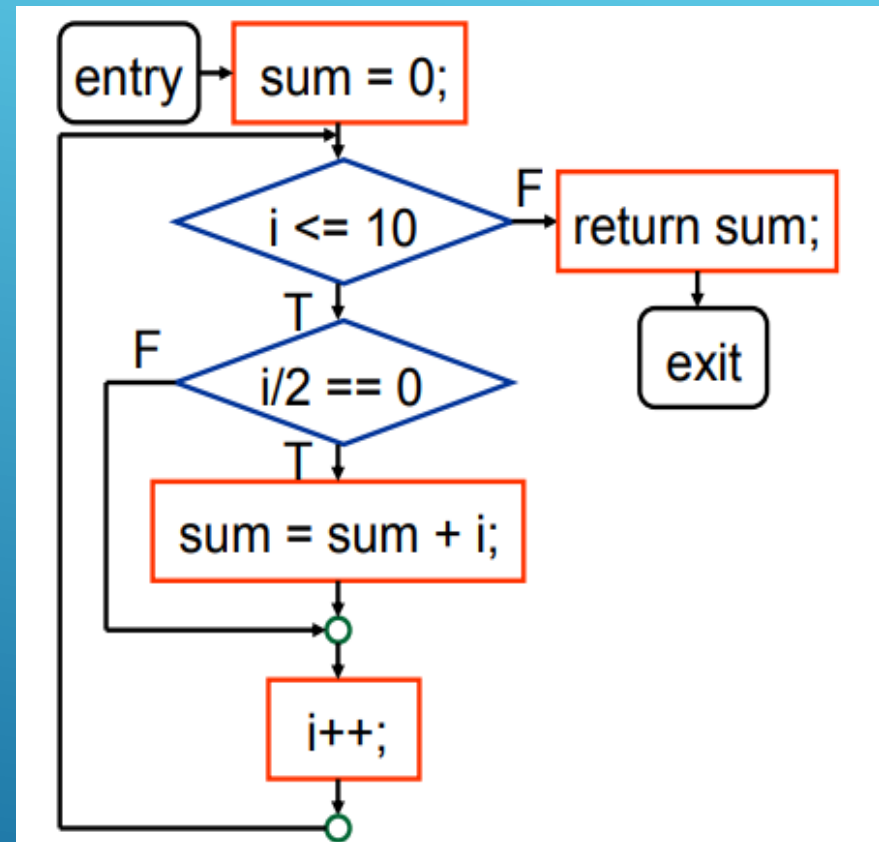


# CFG ANOTHER EXAMPLE


```
1. int evensum(int i)
2. {
3.   int sum = 0;
4.   while (i <= 10) {
5.     if (i/2 == 0)
6.       sum = sum + i;
7.     i++;
8.   }
9.   return sum;
10. }
```

# CFG ANOTHER EXAMPLE

```
1. int evensum(int i)
2. {
3.   int sum = 0;
4.   while (i <= 10) {
5.     if (i/2 == 0)
6.       sum = sum + i;
7.     i++;
8.   }
9.   return sum;
10. }
```



# TEST CASES

- ▶ A **test case** is a complete path from the entry node to the exit node of a control flow graph.
  - ▶ A **test coverage criterion** measures the extent to which a set of test cases covers a program.
    - ▶ Coverage refers to the extent to which a given testing activity has satisfied its objectives.
- 
- A series of white diagonal lines of varying lengths and thicknesses are positioned in the bottom right corner of the slide, creating a modern, abstract graphic element.

# WHITEBOX TEST TECHNIQUES

- ▶ All Path coverage
- ▶ Statement coverage
- ▶ Branch coverage
- ▶ Decision coverage
- ▶ Condition coverage or Predicate coverage
- ▶ Multiple condition coverage



# PATH COVERAGE

- ▶ Path coverage is concerned with linearly independent paths through the code.
- ▶ Every **complete path** in the program has been executed at least once.
- ▶ A loop usually has an **infinite** number of complete paths.

# WELL KNOWN PATHS' SELECTION CRITERIA

- ▶ Select *all* paths.
- ▶ Select paths to achieve complete *statement* coverage.
- ▶ Select paths to achieve complete *branch* coverage.
- ▶ Select paths to achieve *predicate* coverage.

# ALL PATHS COVERAGE CRITERIA

- ▶ All possible paths from input to output.
- ▶ Cover positive as well as negative scenario of methods/conditions.
- ▶ Any program includes, multiple entry and exit points. Testing each of these points is a challenging & time-consuming.
- ▶ To reduce the redundant tests and to achieve maximum test coverage, basis path testing is used.

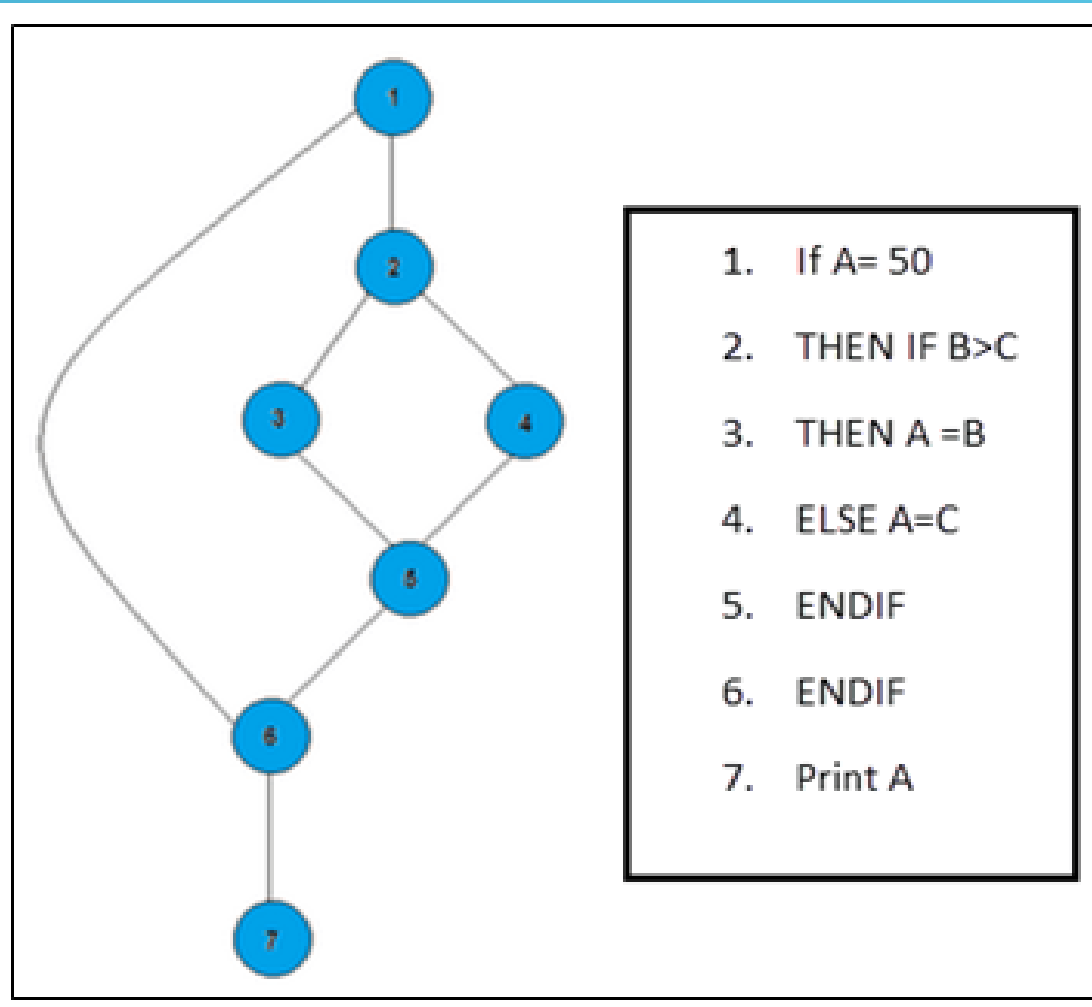
# BASIS PATH TESTING

- ▶ **Basis path testing** is defined as a path through the program from the start node until the end node that introduces at least one new set of processing statements or a new condition (i.e., new nodes)
- ▶ Must move along at least one edge that has not been traversed before by a previous path
- ▶ One can create a control flow graph and calculate its cyclomatic complexity, which is used to determine the number of independent paths.
- ▶ With the cyclomatic complexity, one can determine the minimum number of test cases needed for each independent path of the flow graph.

# STEPS FOR BASIS PATH TESTING

- ▶ Draw a control graph (to determine different program paths)
- ▶ Calculate Cyclomatic complexity (metrics to determine the number of independent paths)
- ▶ Find a basis set of paths
- ▶ Generate test cases to exercise each path

# BASIS PATH TESTING



- Path 1: 1,2,3,5,6, 7
- Path 2: 1,2,4,5,6, 7
- Path 3: 1, 6, 7

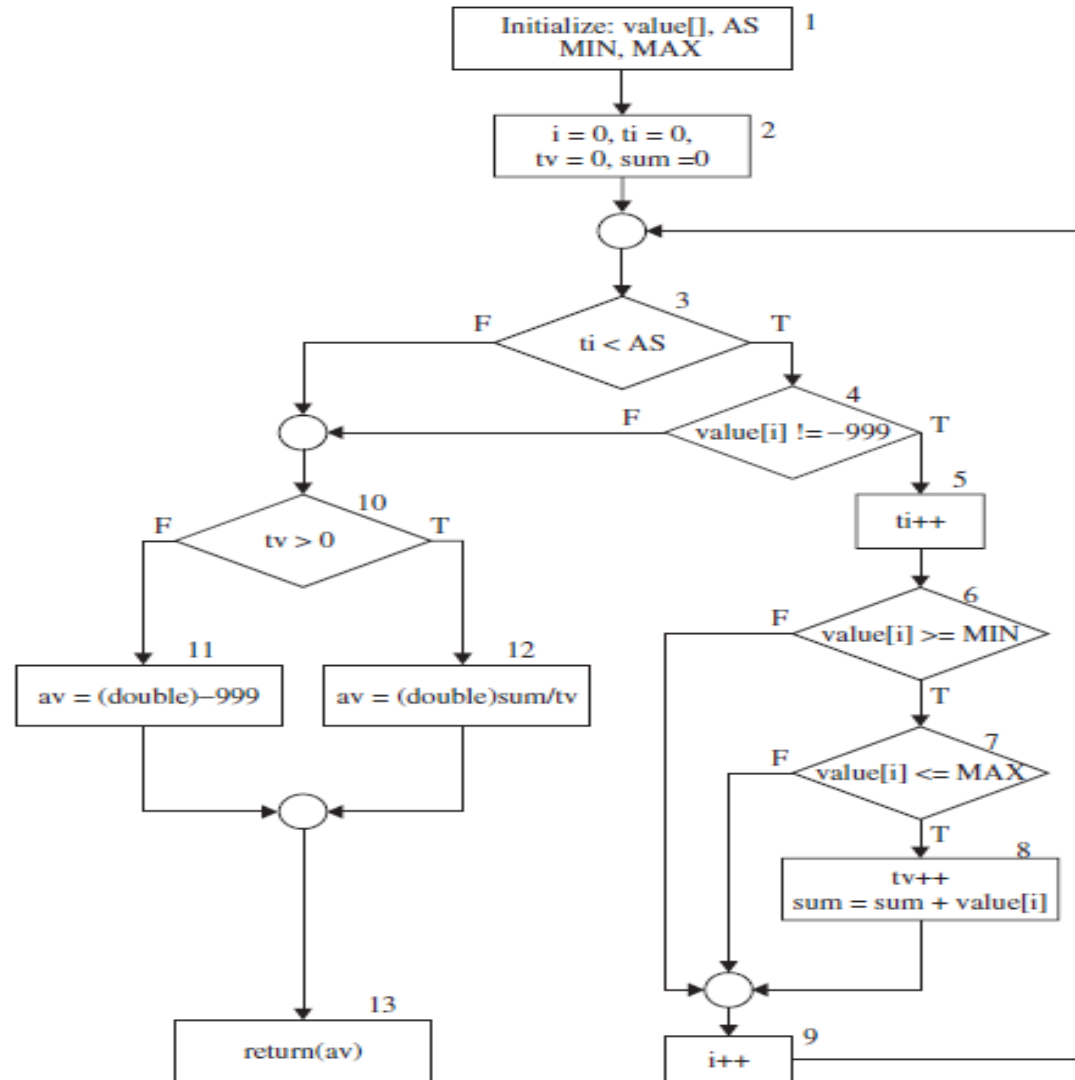
# PATH COVERAGE EXAMPLE

## FUNCTION TO COMPUTE AVERAGE OF SELECTED INTEGERS IN AN ARRAY

```
1. Value[]={2,3,4,8,7,-999,0,6,5,3}
2. public static double ReturnAverage(int value[],int AS, int MIN, int MAX)
3. int i, ti, tv, sum; double av;
4. i = 0; ti = 0; tv = 0; sum = 0;
5. while ( ti< AS && value[i] != -999) {
6.   ti++;// total index ... ti=ti+1
7.   if (value[i] >= MIN && value[i] <= MAX) {
8.     tv++; total value
9.     sum = sum + value[i];
10.  } i++; }
11. if (tv > 0)
12.  av = (double)sum/tv;
13. else av = (double) -999;
14. return (av);
15. }
```

# PATH COVERAGE EXAMPLE

FUNCTION TO COMPUTE AVERAGE OF SELECTED INTEGERS IN AN ARRAY





# FUNCTION TO COMPUTE AVERAGE OF SELECTED INTEGERS IN AN ARRAY

## EXAMPLES OF PATHS IN CFG

**Path 1** 1-2-3(F)-10(T)-12-13

**Path 2** 1-2-3(F)-10(F)-11-13

**Path 3** 1-2-3(T)-4(T)-5-6(T)-7(T)-8-9-3(F)-10(T)-12-13

**Path 4** 1-2-3(T)-4(T)-5-6(T)-7(T)-8-9-3(T)-4(T)-5-6(T)-7(T)-8-9-3(F)-10(T)-12-13

# STATEMENT COVERAGE / NODE COVERAGE

- ▶ **Statement coverage** refers to executing individual program statements and observing the outcome.
- ▶ 100% statement coverage has been achieved if all the statements have been executed at least once.
- ▶ Complete statement coverage is the weakest coverage criterion in program testing.
- ▶ Any test suite that achieves less than statement coverage for new software is considered to be unacceptable.

# STATEMENT COVERAGE METHOD

- ▶ Visiting one or more nodes representing the statement, more precisely, selecting a **feasible** entry–exit path that includes the corresponding nodes.
- ▶ A single entry–exit path includes many nodes, we need to select just a few paths to cover all the nodes of a CFG.
- ▶ Select short paths.
- ▶ Select paths of increasingly longer length. Unfold a loop several times if there is a need.
- ▶ Select arbitrarily long, “complex” paths.

# STATEMENT COVERAGE

$$\text{Statement coverage} = \frac{\text{Number of executed statements}}{\text{Total number of statements}} * 100$$

# STATEMENT COVERAGE EXAMPLE

```
1) print (int a, int b) {  
2) int sum = a+b;  
3) if (sum>0)  
4) print ("This is a positive result")  
5) else  
6) print ("This is negative result")  
7) }
```

# STATEMENT COVERAGE EXAMPLE SCENARIO 1:

Scenario 1:

If a = 5, b = 4

```
1) print (int a, int b) {  
2) int sum = a+b;  
3) if (sum>0)  
4) print ("This is a positive result")  
5) else  
6) print ("This is negative result")  
7) }
```

Here;

1.Total number of statements = 7

2.Number of executed statements = 5

Statement coverage =  $5/7 * 100$

Statement coverage =  $500/7$

Statement coverage = 71%

# STATEMENT COVERAGE EXAMPLE SCENARIO 2:

Scenario 2:

If a = -2, b = -5

```
1) print (int a, int b) {  
2) int sum = a+b;  
3) if (sum>0)  
4) print ("This is a positive result")  
5) else  
6) print ("This is negative result")  
7) }
```

Here;

1.Total number of statements = 7

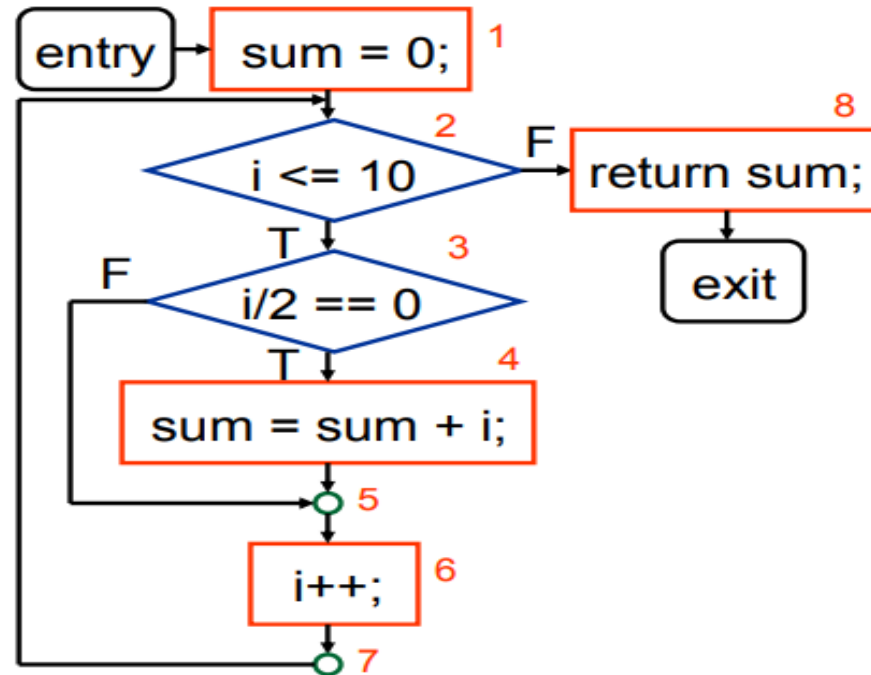
2.Number of executed statements = ?

Statement coverage = ?

# STATEMENT COVERAGE

Every **statement** in the program has been executed at least once.

1 → 2 → 3 → 4 →  
5 → 6 → 7 → 2 → 8



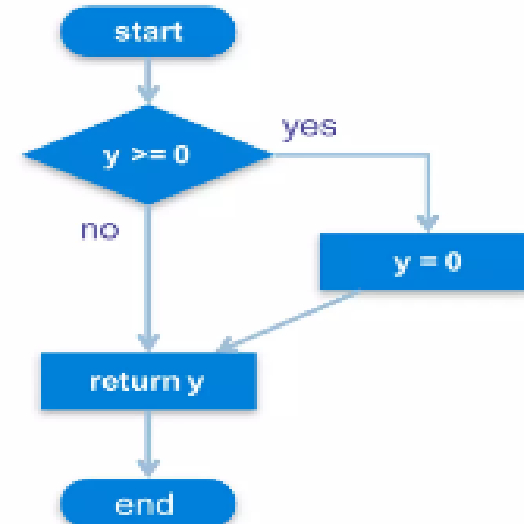


# STATEMENT COVERAGE

Execute each statement at least once

```
// Return absolute value of input
int abs(int y) {
    if ( y >= 0)
        y = 0;
    return y;
}
```

Test case	Input	Expected	Actual
1	0	0	0



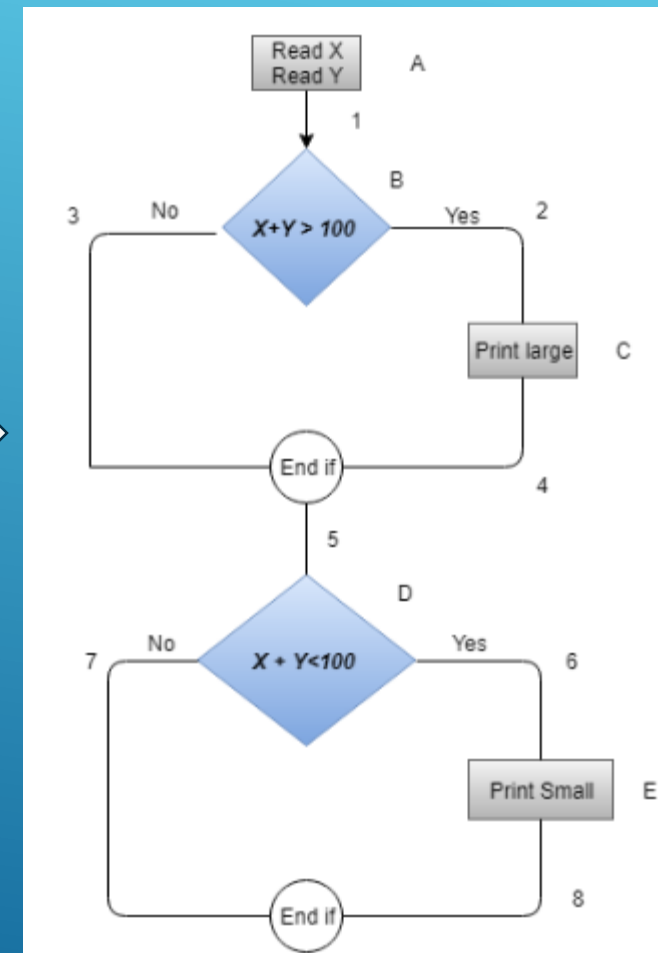
**100% statement coverage!**

# BRANCH COVERAGE /EDGE COVERAGE

- ▶ Complete branch coverage means selecting a number of paths such that every branch/edge is included in at least one path.
- ▶ A branch is an outgoing edge from a node.
- ▶ All the rectangle nodes have at most one outgoing branch (edge).
- ▶ The exit node of a CFG does not have an outgoing branch.
- ▶ All the diamond nodes have two outgoing branches.

# BRANCH COVERAGE EXAMPLE

```
Read X
Read Y
IF X+Y > 100 THEN
  Print "Large"
ENDIF
If X + Y < 100 THEN
  Print "Small"
ENDIF
```



# BRANCH COVERAGE EXAMPLE

**Path 1** - A1-B2-C4-D6-E8

**Path 2** - A1-B3-5-D7

Branch Coverage (BC) = Number of paths  
=2

Case	Covered Branches	Path	Branch coverage
Yes	1, 2, 4, 5, 6, 8	A1-B2-C4-D6-E8	2
No	3,7	A1-B3-5-D7	

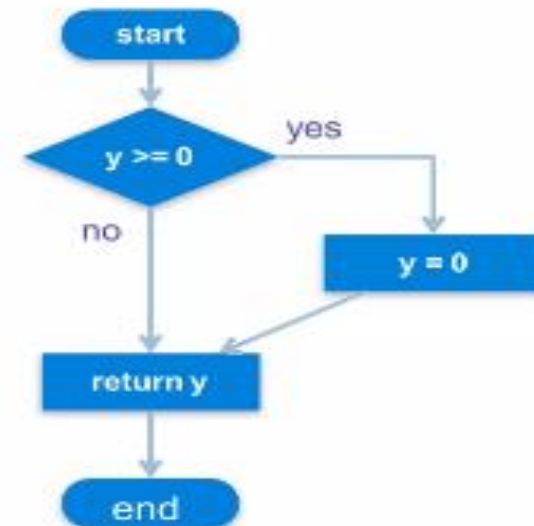
# BRANCH COVERAGE EXAMPLE

Execute each edge in the CFG at least once

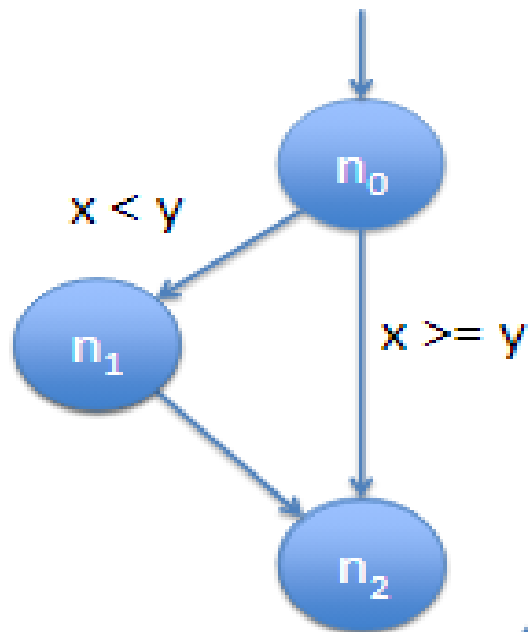
```
// Return absolute value of input
int abs(int y) {
    if (y >= 0)
        y = 0;
    return y;
}
```

Test case	Input	Expected	Actual
1	0	0	0
2	-5	5	-5

**100% branch coverage!**



## Statement vs Branch Coverage



Abstract test  
cases

$$p_1 = n_0, n_1, n_2$$

$$p_2 = n_0, n_2$$

$TS_1 = \{p_1\}$  satisfies  
Statement/node coverage

$TS_2 = \{p_1, p_2\}$  satisfies  
Branch/edge coverage

# DECISION COVERAGE

- Decision coverage measures whether each possible outcome of the decision statement has been executed at least once during testing.
- A decision statement is a statement in the code that involves a choice between two or more possible outcomes. E.g., an if-else statement
- For instance, if there is an if-else statement in the code, decision coverage requires that both the if and else branches are executed at least once during testing.

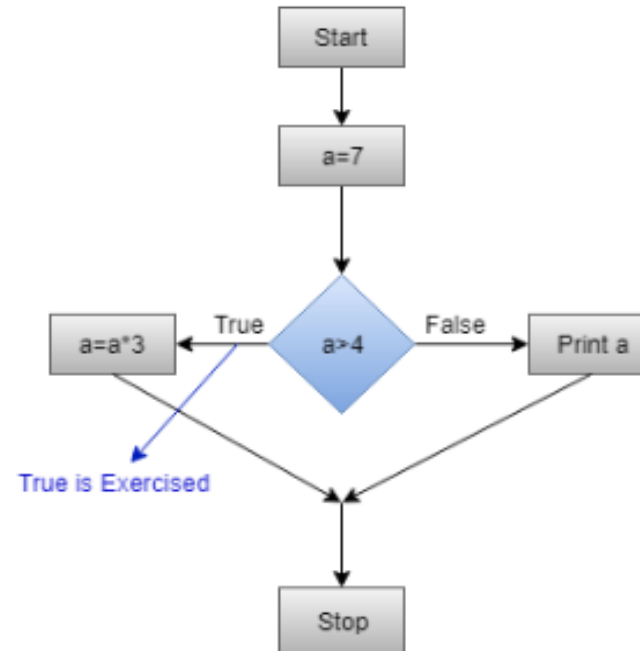
$$\text{Decision Coverage} = \frac{\text{Number of Decision Outcomes Exercised}}{\text{Total Number of Decision Outcomes}} * 100$$

# DECISION COVERAGE

```
Test (int a)
{
  If(a>4)
  a=a*3
  Print (a)
}
```



Control flow graph when the value of a is 7.



Decision Coverage =  $\frac{1}{2} \times 100$  (Only "True" is exercised)

=  $100/2$

= 50

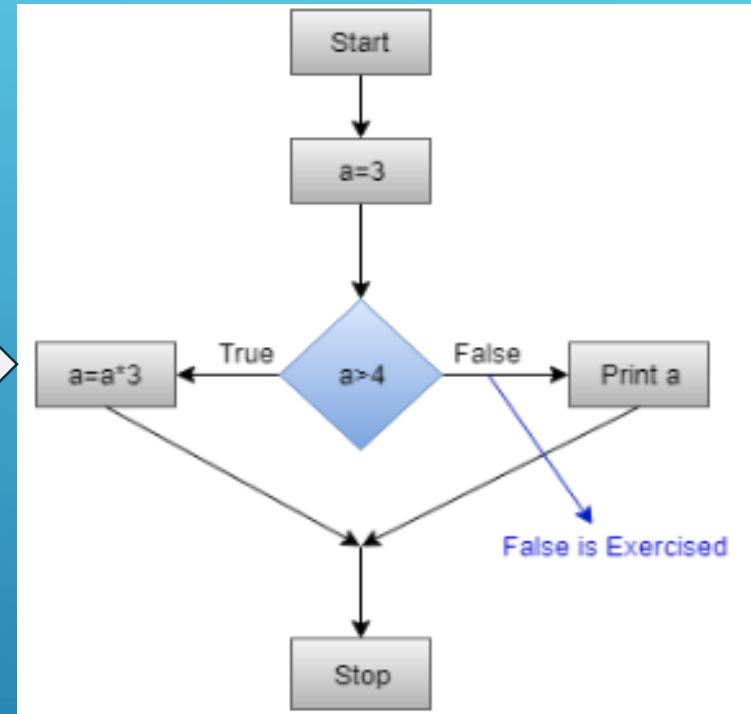
Decision Coverage is 50%



# DECISION COVERAGE

a=3

```
Test (int a)
{
  If(a>4)
  a=a*3
  Print (a)
}
```



=  $\frac{1}{2} \times 100$  (Only "False" is exercised) <br>  
=  $100/2$   
= 50  
Decision Coverage = 50%

# DECISION COVERAGE

Test Case	Value of A	Output	Decision Coverage
1	3	3	50%
2	7	21	50%

# CYCLOMATIC COMPLEXITY

- ▶ Complexity is a software metric that give the quantitative measure of logical complexity of the program.
- ▶ The Cyclomatic complexity defines the number of independent paths in the basis set of the program that provides the upper bound for the number of tests that must be conducted to ensure that all the statements have been executed at least once.

# METHODS TO COMPUTE CYCLOMATIC COMPLEXITY

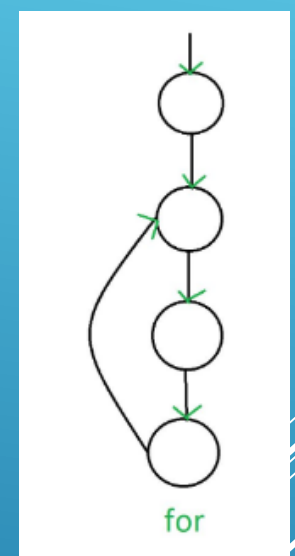
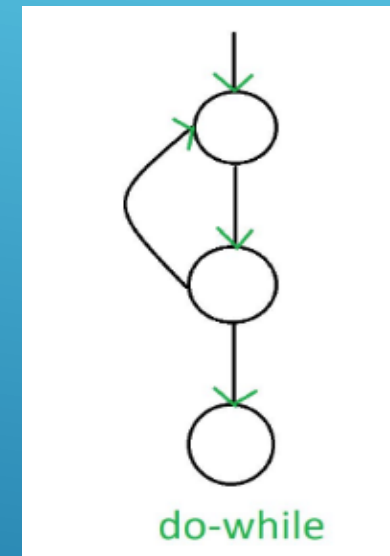
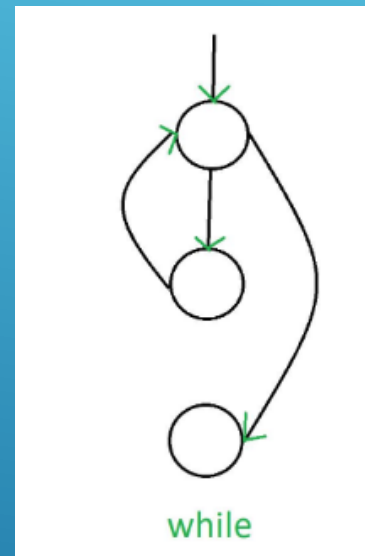
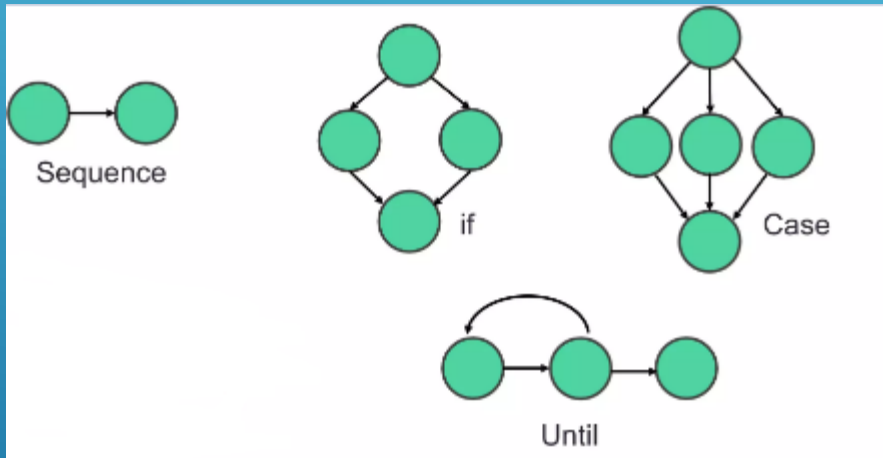
- ▶ **Method 1:** Total number of regions in the flow graph is a Cyclomatic complexity.
- ▶ **Method 2:** The Cyclomatic complexity,  $V(G)$  for a flow graph  $G$  can be defined as  $V(G) = E - N + 2$

Where:  $E$  is total number of edges and  $N$  is the total number of nodes in the flow graph.

- ▶ **Method 3:** The Cyclomatic complexity  $V(G)$  for a flow graph  $G$  can be defined as  $V(G) = P + 1$

Where:  $P$  is the total number of predicate nodes contained in the flow  $G$ .

# CONTROL FLOW GRAPH IS REPRESENTED DIFFERENTLY FOR CONDITIONAL AND LOOPS STATEMENTS



# TEST ORACLE

- ▶ To verify the execution is correct, we need to compare the actual outcome with the expected outcome.
- ▶ Test oracle is a tool that can return the expected outcome for a given input vector.
- ▶ An executable specification of a program can be used as a test oracle for that program.

# REFERENCES

## Book:

The Art of Software Testing, Second Edition, Glenford J. Myers

Chapter 4