
بسم الله الرحمن الرحيم

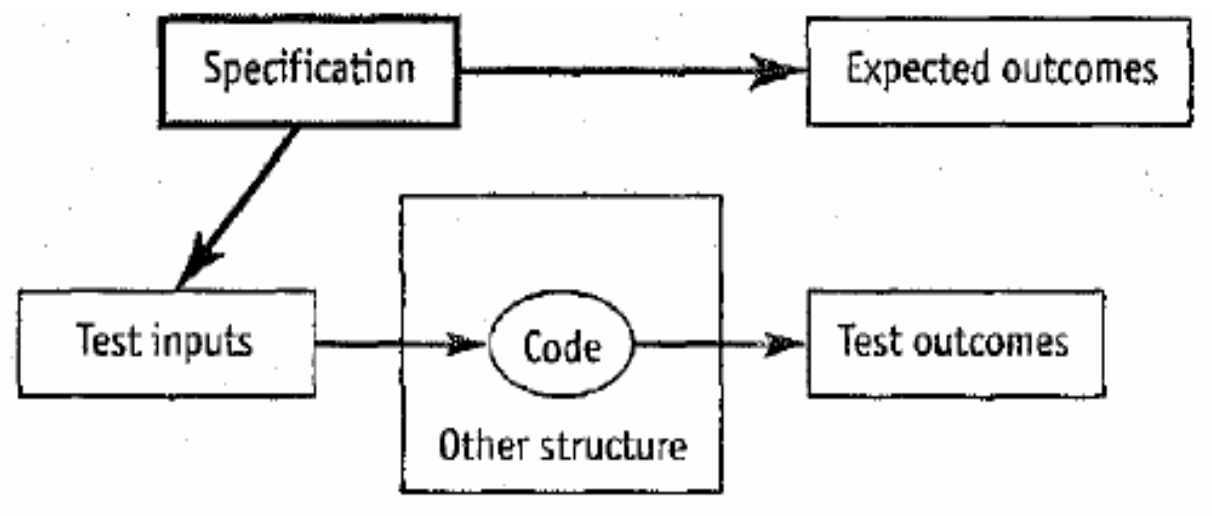
Test Case Design Through *White Box*

White Box Testing

- White box testing is also know as Glass Testing or Open Box testing. It is detailed examination of internal structure and logic of code.

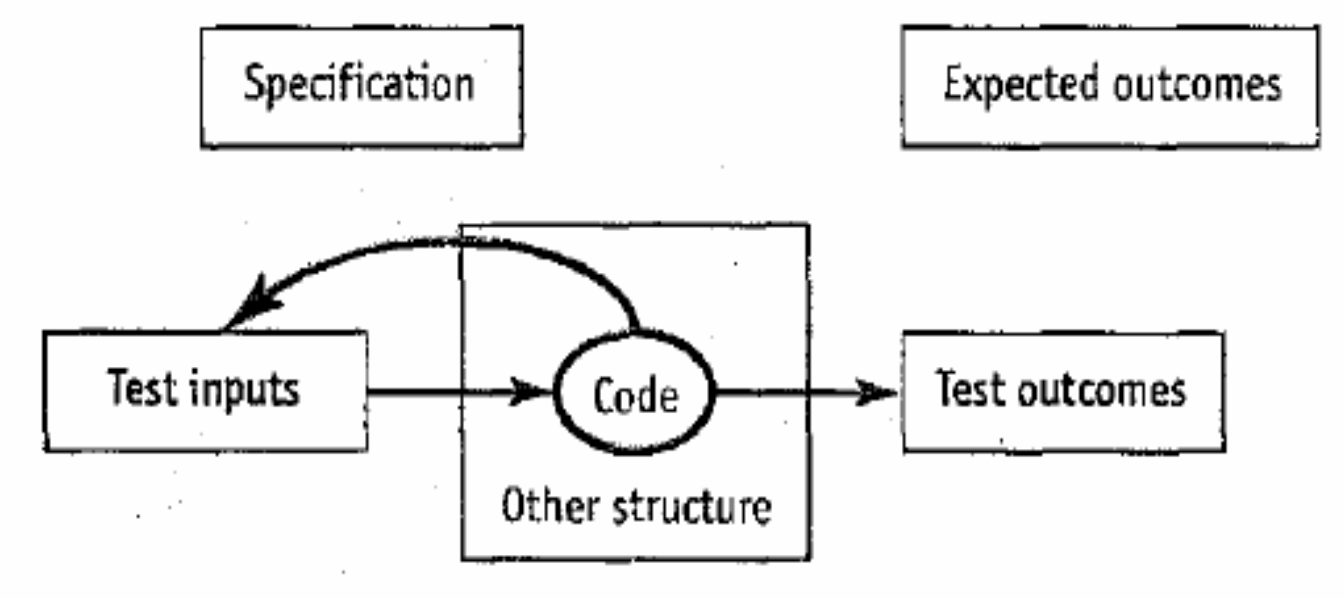
- This examination is demonstrated through test cases creation by looking at the code to detect any potential failure scenarios.

Test Case Design



Black Box Testing

Test Case Design based upon Source Code



White Box Testing

Challenges in Test Case Designing based upon Source Code

- This approach generates test inputs, but a test also requires expected outcomes to compare against.

- Code-based test case design cannot tell whether the outcomes produced by the software are the correct values, it is the specification for the code that tells you what it should do.

- So this approach is incomplete, since it cannot generate expected outcomes.

Challenges in Test Case Designing based upon Source Code

- Another problem with this approach is that it only tests code that exists and cannot identify code that is missing.
- It is also testing that 'the code does what the code does.'
- This is generally not the most useful kind of testing, since we normally want to test that ‘the code does what it should do.’
  - Software works as coded' (not 'as specified').

White Box Testing Test Case Basic Templae

Test ID	Input	Output	Pass/Fail

White Box Testing

- White box testing utilizes the logical flow through a program to propose test cases.
- Logical flow means the way in which certain parts of a program may be executed as we run the program.
- Logical flow of a program can be represented by a Control Flow Graph (CFG).
- Most common white box testing methods are:
  - Statement Coverage
  - Decision/Branch Coverage
  - Condition Coverage
  - Path Coverage

Control Flow Graph
(CFG)

Control Flow Graph (CFG)

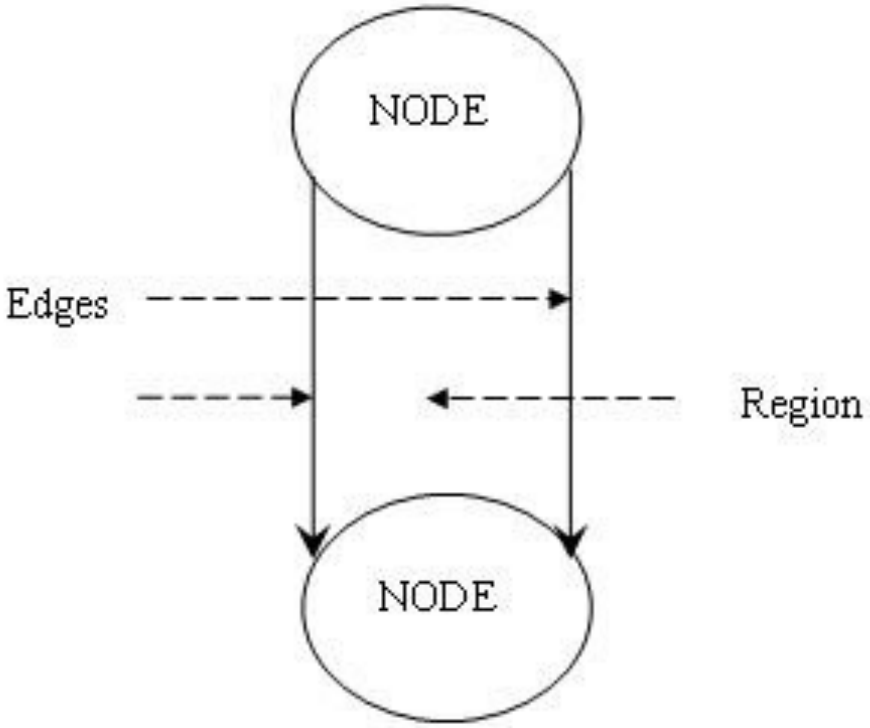
- The entire structure design and code of the software have to be studied for this type of testing.
- This method is implemented with the intention to test logic of the code so that the required results or functionalities can be achieved.
- Its applicability is mostly to relatively small programs or segments of larger programs, thus it is mostly used for unit testing.
- It catches 50% of all the bugs caught during unit testing. That amounts to 33% of all the bugs caught in the program.

Control Flow Graph

```
9  if ( $Number == 3 ) {
10
11 echo "Display Something, if the Number is 3";
12
13 } else {
14
15                ,,,,,
16
17 }
```

Failure

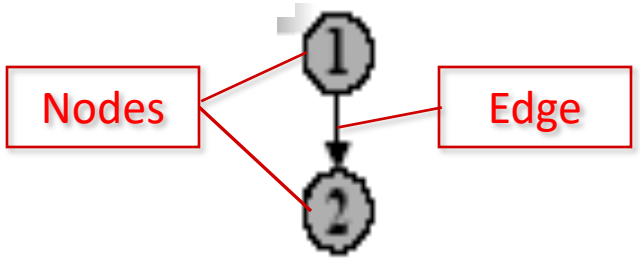
Control Flow Graph



Control Flow Graph (Example 1)

How to Draw a Control Flow Graph

- Number all the statements of a program



Sequence

1. a = 5;
2. b = a * b – 4;

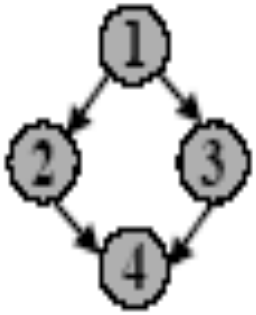
Numbered statements:

- Represent nodes of the 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

Control Flow Graph (Example 2)



Selection

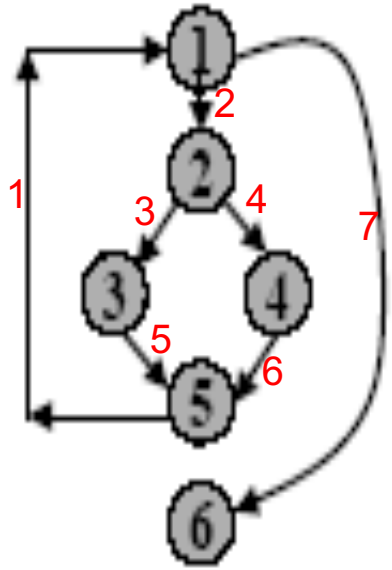
1. If (a > b)
2. c = 3;
3. Else c = 5;
4. c = c*c;

How many edges = 4

How many nodes = 4

Control Flow Graph (Example 3)

Flow Graph




```
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;
```

How many edges = 7

How many nodes = 6

How Much Percentage of Coverage ?

- **Statement Coverage=** $\frac{\text{Number of executed statements}}{\text{Total number of statements}} \times 100$
- **Decision Coverage=** $\frac{\text{Number of decision outcomes exercised}}{\text{Total number of decision outcomes}} \times 100$
- **Branch Coverage=** $\frac{\text{Number of executed branches}}{\text{Total number of branches}} \times 100$
- **Condition Coverage=** $\frac{\text{Number of executed operands}}{\text{Total number of operands}} \times 100$

Example
(CFG)

CFG For The Following Pseudocode Code

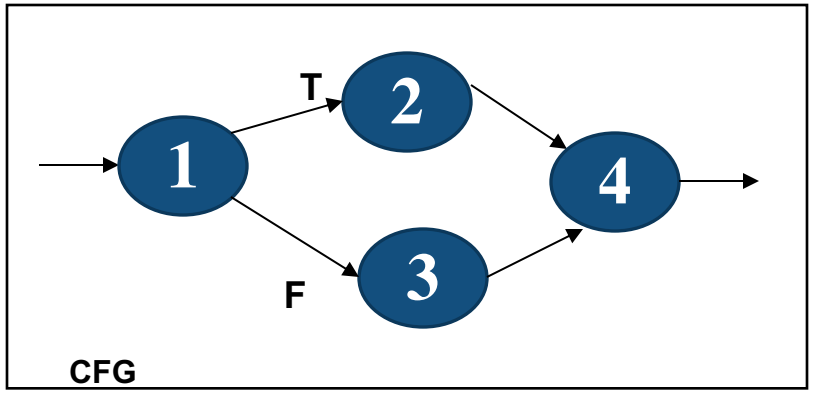
```
if X > 0 then
   Statement1;
else
   Statement2;
```

```
while X < 10 {
   Statement1;
   X++; }
```

Possible Solution

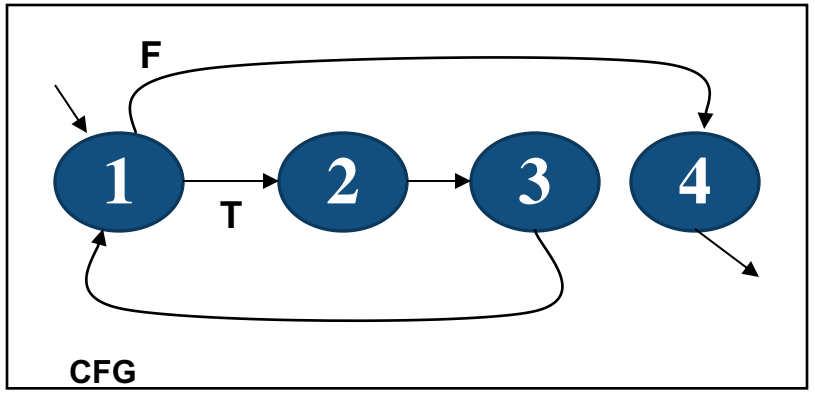
```
if X > 0 then
   Statement1;
else
   Statement2;
end
```

1
2
3
4




```
while X < 10 {
   Statement1;
   X++; }
end
```

1
2
3
4



We Can Also Use Flow Chart To

Understand The Logic of Code

Statement Coverage

Statement Coverage

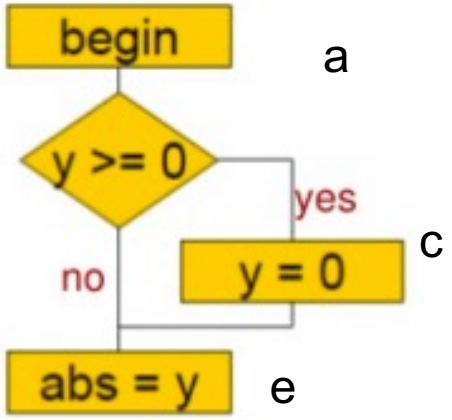
- The goal is to design test cases so that every statement of the program is executed at least once

- Code that has not been executed during testing is more likely to contain errors
  - Often this is the “low-probability” code

Statement Coverage

- Exercise all statements at least once

```
1. Begin
2. if (y >= 0)
3.       then y = 0;
4. abs = y;
5. end;
```



Test Case ?

Input: y= ?

Test ID	Input	Output	Pass/Fail
WBT 2(SC)	Y=o	Path (ace)	Pass

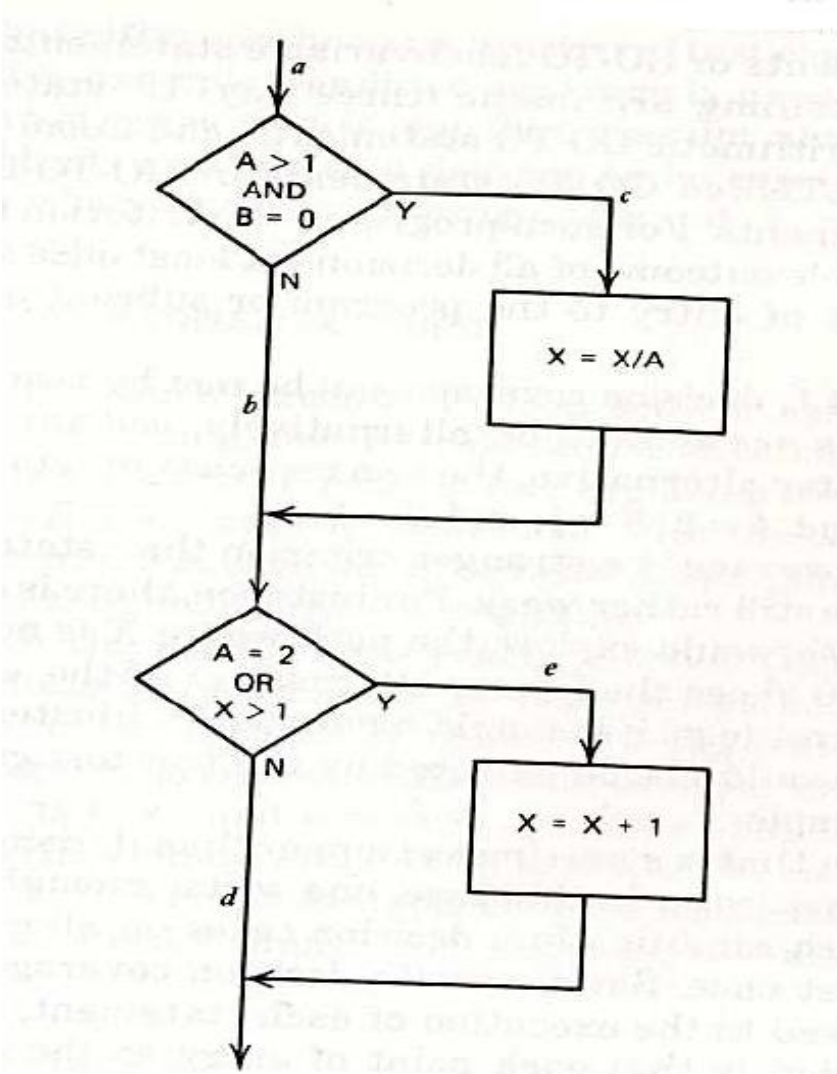
100% statement coverage

Statement Coverage

Test case:

A=2 and B=0 (ace)

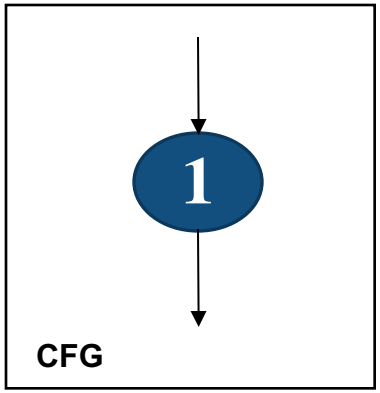
Test ID	Input	Output	Pass/Fail
WBT 1 (SC)	A=2 and B=0	Path (ace)	Pass



Multiple Statements

```
Statement1;
Statement2;
Statement3;
Statement4;
```

Can be represented as **one** node as there is no branch.



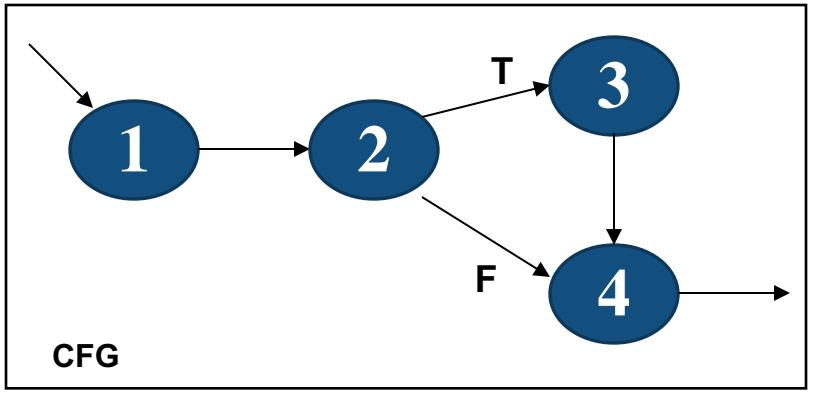

```
Statement1;
Statement2;

if X < 10 then
   Statement3;

Statement4;
```

1

2

3

4



Branch Coverage

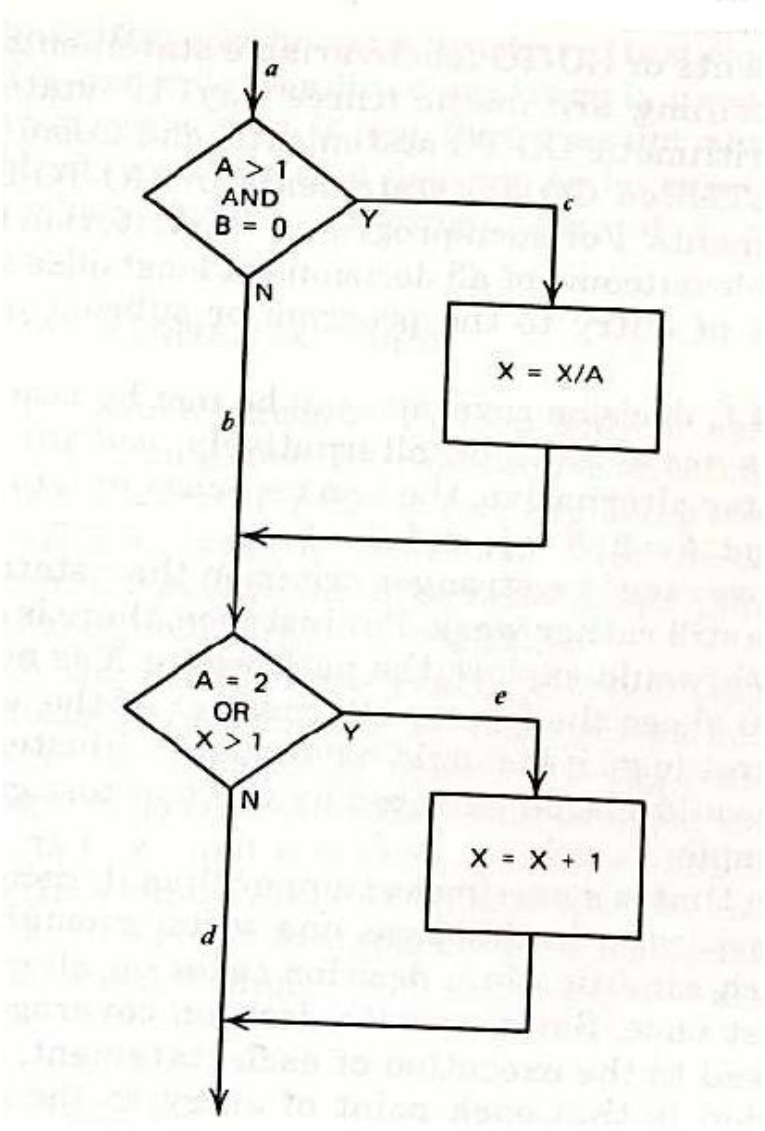
Branch Coverage/Decision Coverage

- In many codes if statement run and else does not runs, in branch coverage we run both if and if else
- The goal is to design test cases to ensure that each branch is executed once
- Each decision is evaluated to *true* and *false,* at least once traversed
- Generally satisfies statement coverage
- Branch coverage has problems with multiple conditions within a decision (&&, ||)

Branch Coverage/Decision Coverage

- **Each decision has a true and a false outcome at least once**

Test ID	Input	Output	Pass/Fail
WBT 3 (DC)	A=2 and B=0	Path (ace)	Pass
WBT 4(DC)	A=1 and X=1	Path (abd)	Pass



Branch Coverage

```
/* x is valid if 0 ≤ x ≤ 100 */
int check (int x ){
        if( x ≥ 0 && x ≤ 100 )
                return TRUE;
        else return FALSE;
        }
```

Test ID	Input	Output	Pass/Fail
WBT 5 (DC)	X=5	Run True	Pass
WBT 6(DC)	X=-5	Run False	Pass

Branch Coverage

```
/* finding the maximum of two integers x and y
*/
int Max_check(int x, y, max) {
If (x>y) max = x;
else max = y;
  }
```

Test ID	Input	Output	Pass/Fail
WBT 7(DC)	X=3 Y=2	Max=4	Pass
WBT 8(DC)	X=2 Y=4	Else max=y	Pass

Condition Coverage

Condition Coverage

- The goal is to design test cases to ensure that each condition in a decision takes on all possible outcomes at least once.
- Ensures that every condition within a decision is covered
- Consider the conditional expression:

  ((c1 and c2)or c3)

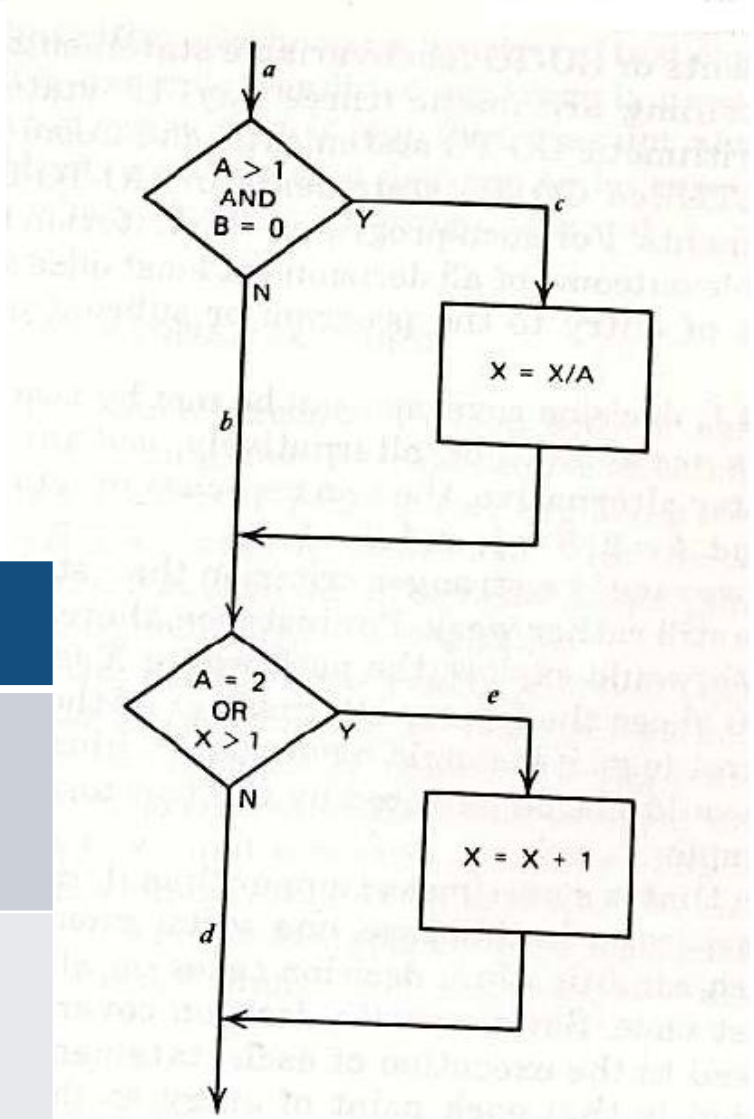
- Each of c1, c2, and c3 are exercised at least once,

i.e. given true and false values require 2^3 test cases.

Condition Coverage

- Each condition in a decision takes on all possible outcomes at least once
- Conditions: A>1, B=0, A=2, X>1

Test ID	Input	Output	Pass/Fail
WBT 8 (CC)	A=5 B=0 X=4 (for true)	ace	Pass
WBT 9(CC)	A=1 B=1 X=1 (for false)	abd	Pass



Condition Coverage

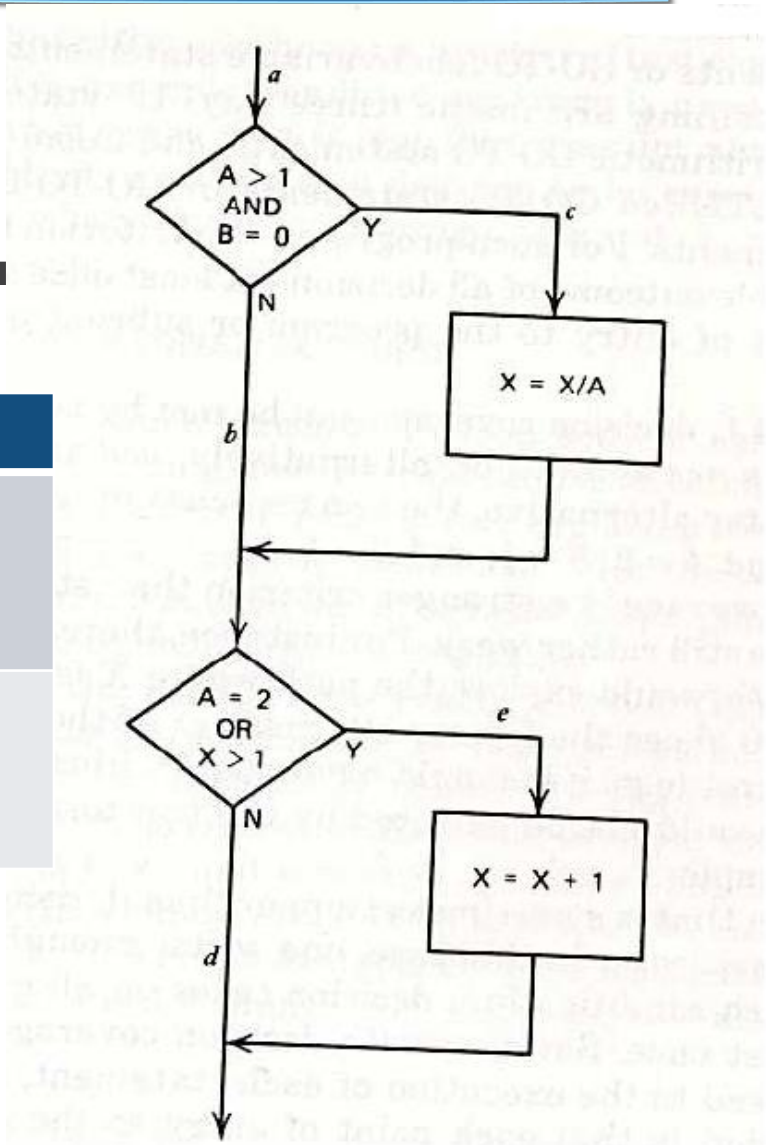
- **Each condition in a decision takes possible outcomes at least once, and**
- **Each decision takes on all possible ou at least once**

Test ID	Input	Output	Pass/Fail
WBT 10(CC)	A=2 B=0 X=4 (for true)	ace	Pass
WBT 9(CC)	A=1 B=1 X=1 (for false)	abd	Pass

- **What about these?**

A=1, B=0, and X=3 (abe)

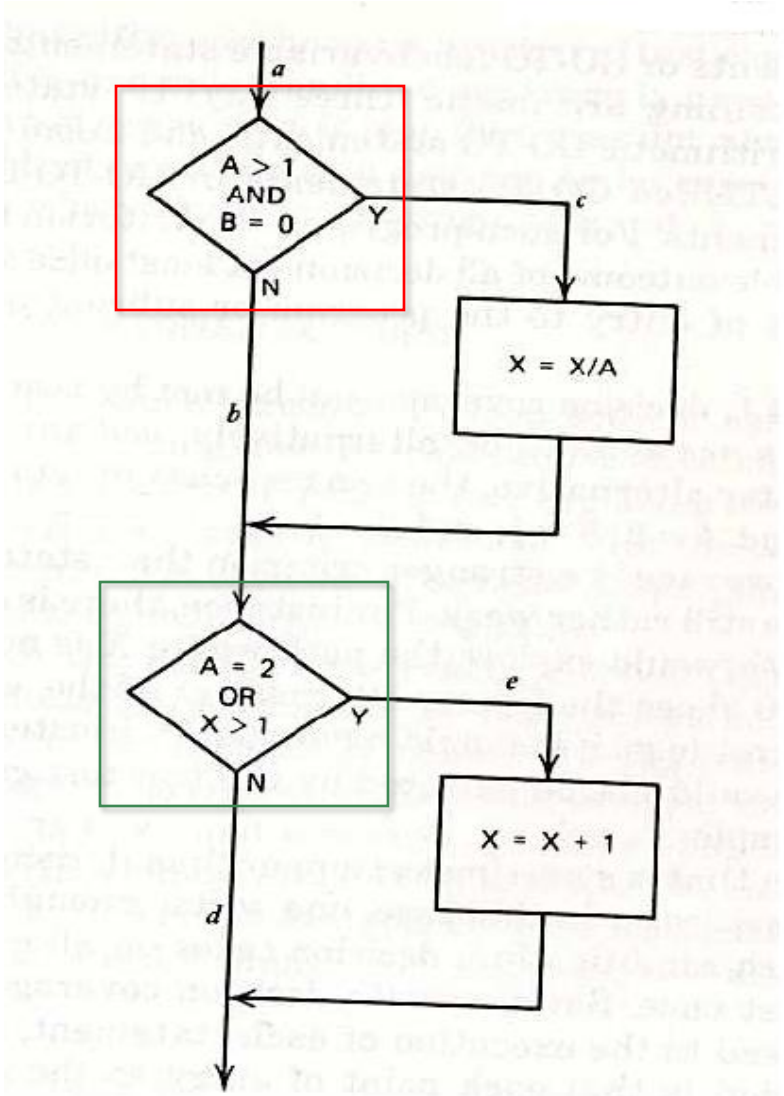
A=2, B=1, and X=1 (abe)



Multiple Condition Coverage

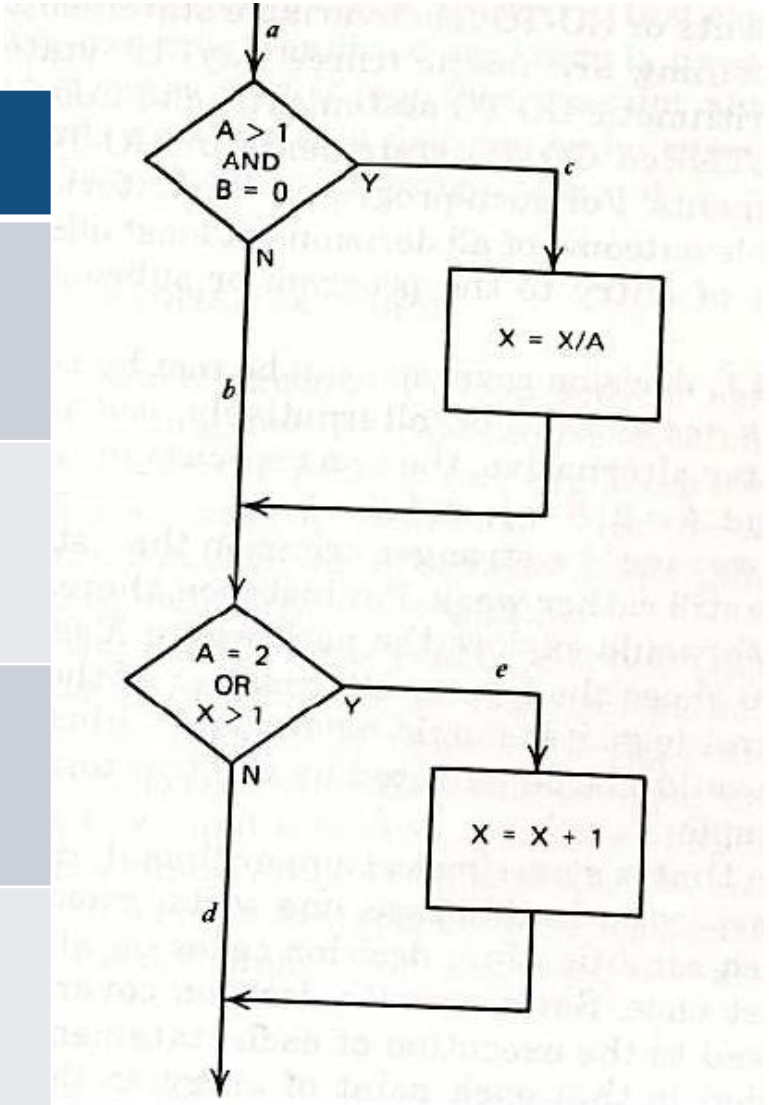
- **Exercise all possible combinations of condition outcomes in each decision**
- **Conditions:**

A > 1, B = 0	A = 2, X > 1
A > 1, B <> 0	A = 2, X <= 1
A <= 1, B = 0	A <> 2, X > 1
A <=1, B <> 0	A <> 2, X <= 1



Multiple Condition Coverage

Test ID	Input	Output	Pass/Fail
WBT 11 (CC)	A=2 B=0 X=4	ace	Pass
WBT 12(CC)	A=2 B=1 X=1	abe	Pass
WBT 13(CC)	A=1 B=0 X=2	abe	Pass
WBT 14(CC)	A=1 B=1 X=1	abd	Pass



Path Coverage

Path Testing

- Path testing is an approach of testing where you ensure that every path through a program has been executed at least once.
- We must execute these paths at least once in order to test the program thoroughly.
- We design test cases according to the identified paths.

Path Testing Approach

❑Steps

1. Draw a control flow graph.
2. Determine the cyclomatic complexity.
3. Find a set of paths.
4. Generate test cases for each path.

Cyclomatic Complexity

- Invented by Thomas McCabe (1974) to measure the complexity of a program's conditional logic
- Provides a quantitative measure of the logical complexity of a program
- Provides an upper bound for the number of tests that must be conducted to ensure all statements have been executed at least once

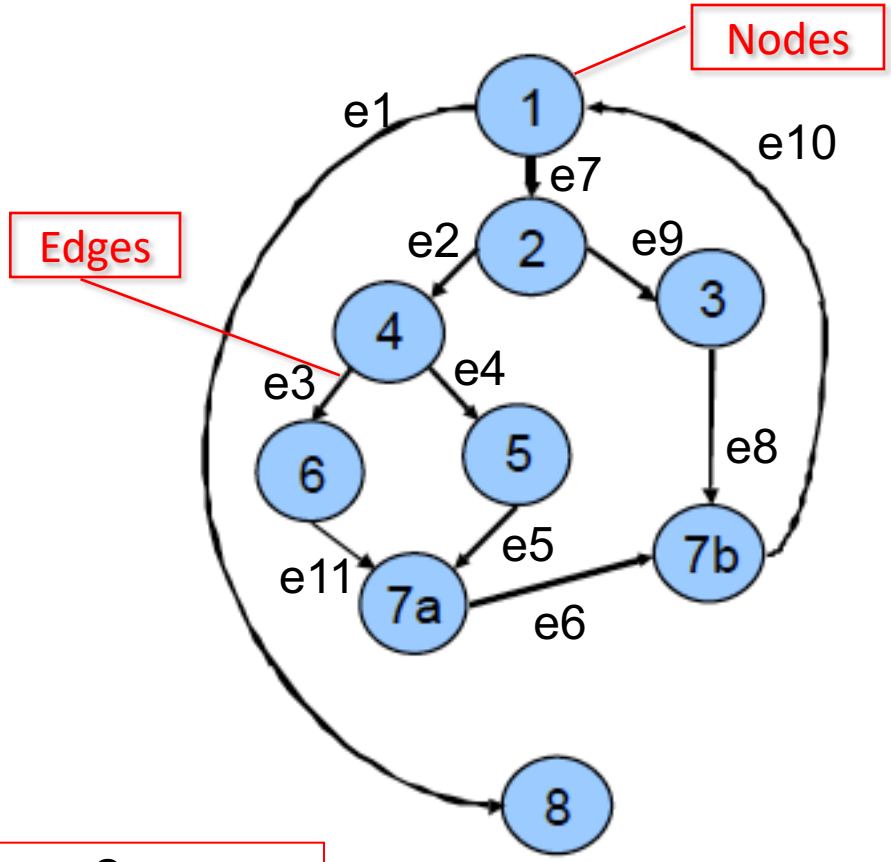
Cyclomatic Complexity

- **Cyclomatic Complexity**
  - V(G) = E – N + 2,
    - V refers to the cyclomatic number in graph
    - where E is the number of edges
    - and N is the number of nodes in graph G

- ***Though the McCabe's metric does not directly identify the linearly independent paths, but it informs approximately how many paths to look for***.

Example # 01

```
1.      do while records remain
            read record;
2.          if record field 1 = 0
3.              then process record;
                    store in buffer;
                    increment counter;
4.          elsif record field 2 = 0
5.              then reset record;
6.              else process record;
                    store in file;
7a.         endif;
        endif;
7b.     enddo;
8.  end;
```



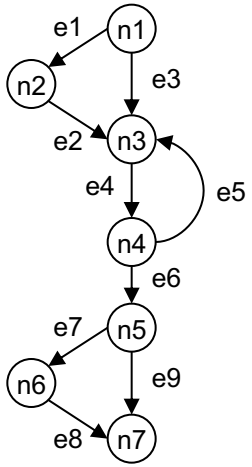
$$V(G) = e - n + 2$$

$$= 11 - 9 + 2$$

$$= 4$$

Example # 02

```
if expression1
then
     statement2
end if

do
     statement3
  while expr4
end do

if expression5
then
     statement6
end if
statement7
```



Paths:

P1 = e1, e2, e4, e6, e7, e8

P2 = e1, e2, e4, e5, e4, e6, e7, e8

P3 = e3, e4, e6, e7, e8, e10

P4 = e6, e7, e8, e10, e3, e4

P5 = e1, e2, e4, e6, e9, e10

P6 = e4, e5

P7 = e3, e4, e6, e9, e10

P8 = e1, e2, e4, e5, e4, e6, e9, e10

$$V(G) = e - n + 2$$

$$= 9 - 7 + 2$$

$$= 4$$

How Complex Should Code Be?

Complexity Number	Meaning
1-10	Structured and well written code High Testability Cost and Effort is less
10-20	Complex Code Medium Testability Cost and effort is Medium
20-40	Very complex Code Low Testability Cost and Effort are high
>40	Not at all testable Very high Cost and Effort

Higher the **Cyclomatic complexity** number, the more complex the code

Advantages of White Box Testing

- As tester has knowledge of the source code it is easy to find errors.
- **White box testing helps us to identify memory leaks.**
  - When we allocate memory using malloc( ) in C, we should explicitly release that memory also.
  - If this is not done then over time, there would be no memory available for allocating memory on requests.
- **Performance analysis**
  - Code coverage tests can identify the areas of a code that are executed most frequently.
  - Extra efforts can be then made to check these sections of code.

Advantages of White Box Testing

- **White box testing is useful in identifying bottlenecks in resource usage.**
  - For example, if particular resource like RAM or ROM or even network is perceived as a bottleneck then code can help identify where the bottlenecks are and point towards possible solutions.
- **White box testing can help identify security holes in dynamically generated code.**
  - For example in case of Java, some intermediate code may also be generated.
  - Testing this intermediate code requires code knowledge.
  - White box testing only does this.

Advantages of White Box Testing

- Due to testers knowledge of code , maximum coverage is attained during test scenario.

- Testers can then see if the program diverges from its intended goal.

Disadvantages of White Box Testing

- As skilled tester are performing tests, costs is increased.
- As it is very difficult to look into every corner of the code, some code will go unchecked.
- White box testing does not account for errors caused by omission.