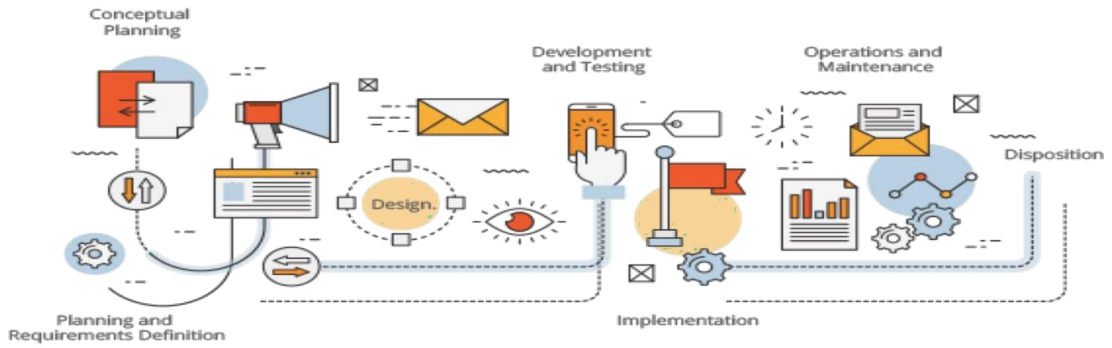
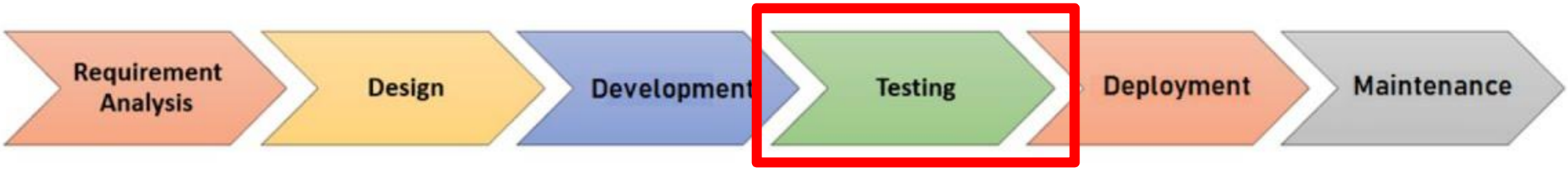
Software Engineering

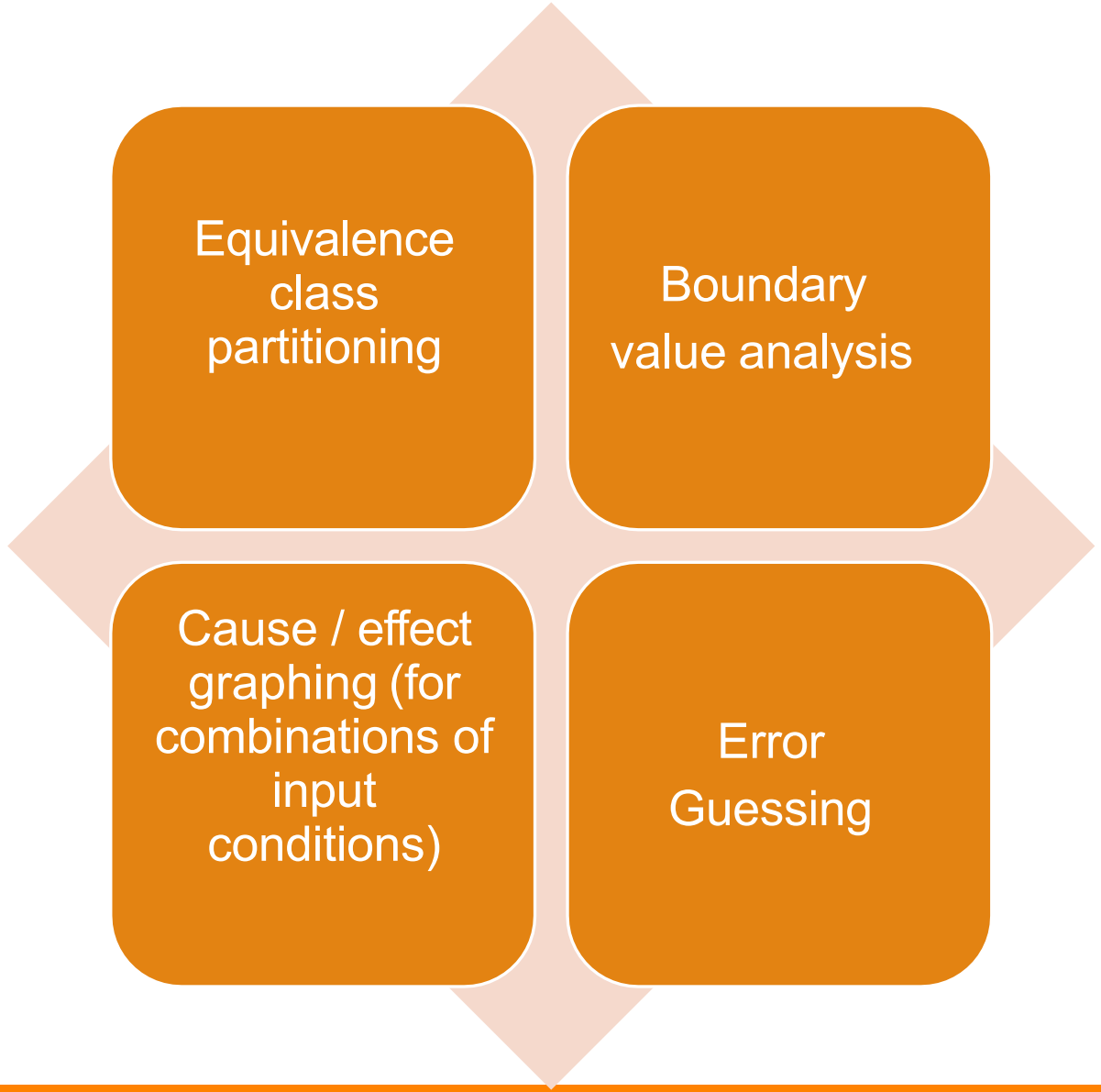
Testing



Software Development Life Cycle (SDLC)



How to Design Test Cases for blackbox testing?



Error-Guessing

- Some people design the test cases by intuition and experience.
- The basic idea is to enumerate a list of possible errors and then write test cases based on the list.

Black Box Testing: Limitations

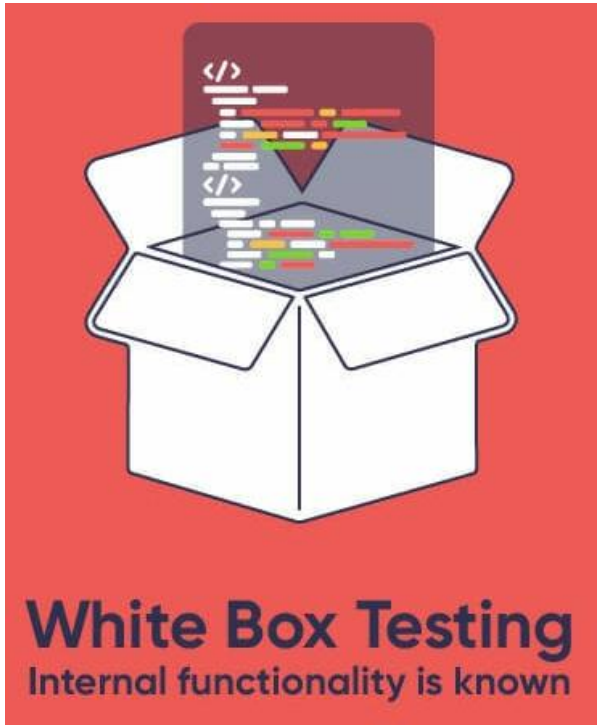
- **Requirements may not be complete and accurate**
  - May not describe all behaviors of the system
- **Requirements may not have sufficient details for testing**
  - Design decisions may be left to the implementation
- **The system may have unintended functionality**

- **Conclusion: supplement Black Box (Functional) Testing with White Box (Structural) Testing**

Black box Testing vs White box Testing

- **White box testing** can uncover structural problems, hidden errors and problems with specific components.

- **Black box testing** checks that the system as a whole is working as expected.



Working

1. In this, the developer will test every line of the code of the program.
2. The developers perform the White-box testing
3. Then send the application or the software to the testing team,
4. Team will perform the black box testing
5. Verify the application along with the requirements and identify the bugs and sends it to the developer.

White box Testing

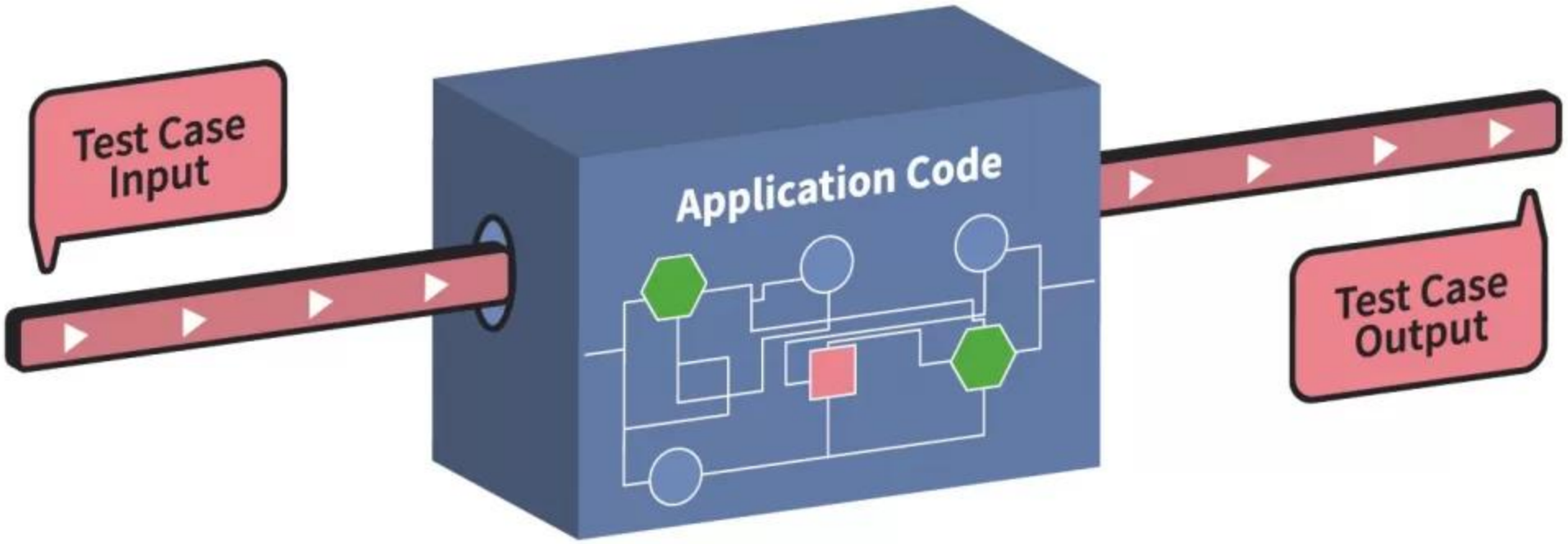
- Analyze the internal structures the used data structures, internal design, code structure, and the working of the software.
- It is also called glass box testing or clear box testing or structural testing or transparent testing or open box testing.
- The tester has access to the source code and uses this knowledge to design test cases that can verify the correctness of the software at the code level.
- The tester creates test cases to examine the code paths and logic flows to ensure they meet the specified requirements.
- White box testing is an essential part of automated build processes in a modern Continuous Integration/Continuous Delivery (CI/CD) development pipeline.

White Box Testing

Unit Testing

Integration Testing

System Testing



⚙ How White Box Testing Works: Step-by-Step

1. Understand the Source Code

Testers (often developers) **study the actual code**, algorithms, and logic. This might include:

- Reading the functions and their flow
- Checking how variables are used
- Understanding conditional logic (`if` , `switch` , etc.)

2. Create a Control Flow Graph (CFG)

As we discussed earlier:

- Nodes represent chunks of code
- Edges show flow of control This helps visualize **all possible paths** through the code.

3. **Design Test Cases to Cover Code Paths**

The goal is to create test cases that **cover as much of the code as possible**. Techniques include:

✅ a. **Statement Coverage**

- Test every line of code **at least once**

✅ b. **Branch Coverage**

- Test every **decision point** (like `if`, `else`, `case`) for **true and false**

✅ c. **Path Coverage**

- Test **every possible path** through the program (most thorough, but complex)

✅ d. **Condition Coverage**

- Test all **logical conditions** in decisions independently

4. **Run Tests and Analyze Results**

Run your test cases:

- Use tools like **JUnit (Java)**, **Pytest (Python)**, **NUnit (.NET)**, or others
- Check for logic errors, dead code, unhandled conditions, or unexpected outputs

5. **Optimize and Fix**

White box testing might reveal:

- Unreachable code
- Infinite loops
- Security issues (like hardcoded secrets)
- Logical errors not caught in black box testing

Relationship between statement coverage and branch coverage

1. **Branch coverage inherently includes statement coverage:**
   - If you've achieved 100% branch coverage, you've also achieved 100% statement coverage. This is because to test every branch (every outcome of a decision point), you must necessarily execute all the statements within those branches.
   - In essence, branch coverage is a more robust form of testing.

2. **However, statement coverage does not guarantee branch coverage:**
   - It's possible to execute every line of code (achieving 100% statement coverage) without testing all possible outcomes of decision points.
   - Branch coverage is then a check that all the possible outcomes of the conditions have been checked.

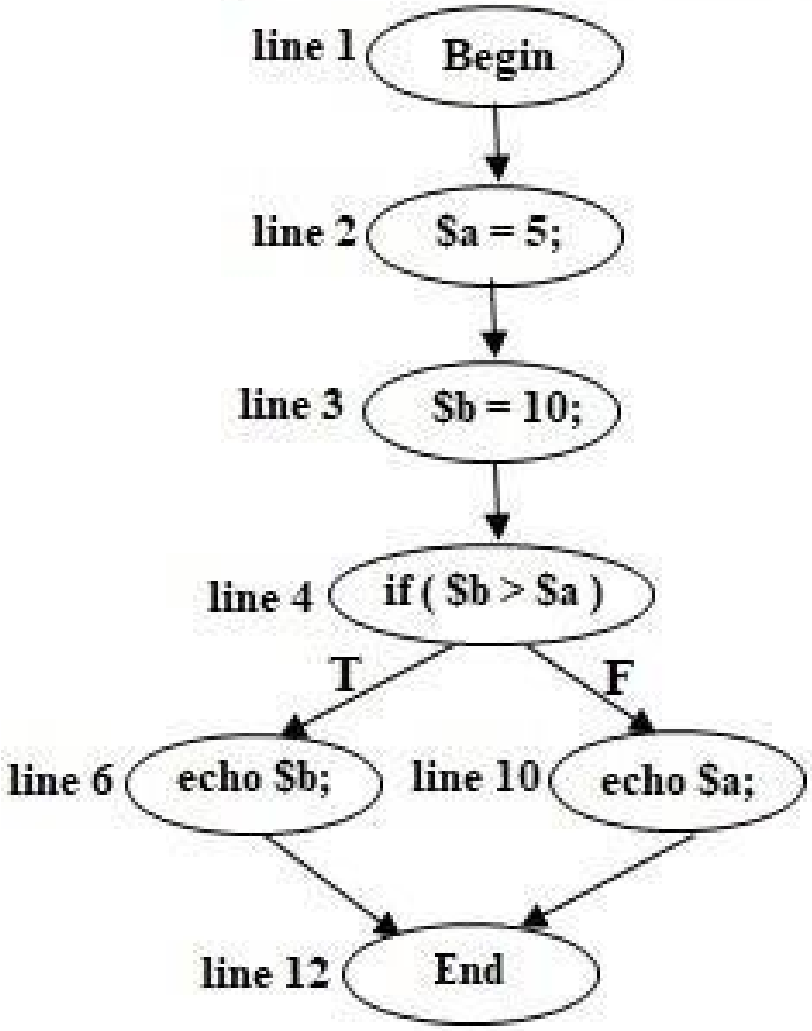
PHP Source Code :

```
1    <?php
2    $a = 5;
3    $b = 10;
4    if($b > $a)
5    {
6        echo $b;
7    }
8    else
9    {
10       echo $a;
11   }
12   ?>
```

Total Paths:

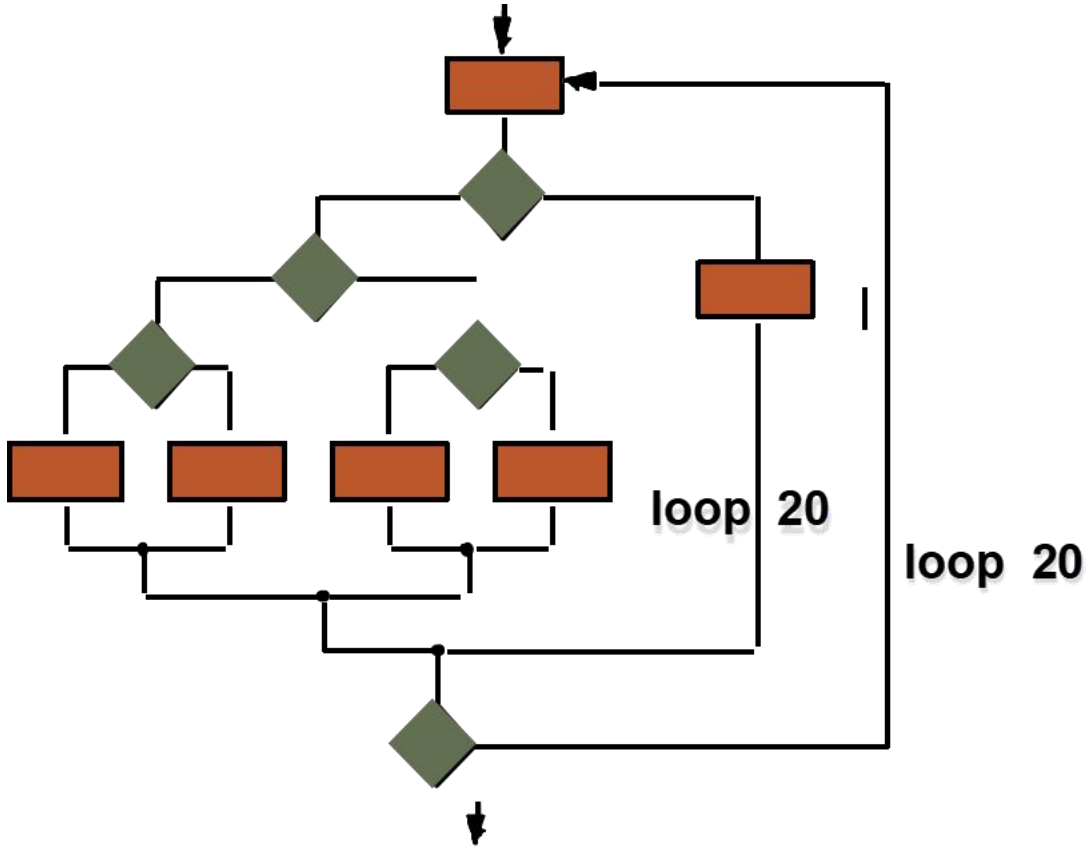
1) 1-2-3-4-6-12
2) 1-2-3-4-10-12

Control Flow Graph (CFG):



Exhaustive Testing (The Problem of Path Explosion)

The exponential growth of possible execution paths in programs with loops and branches.



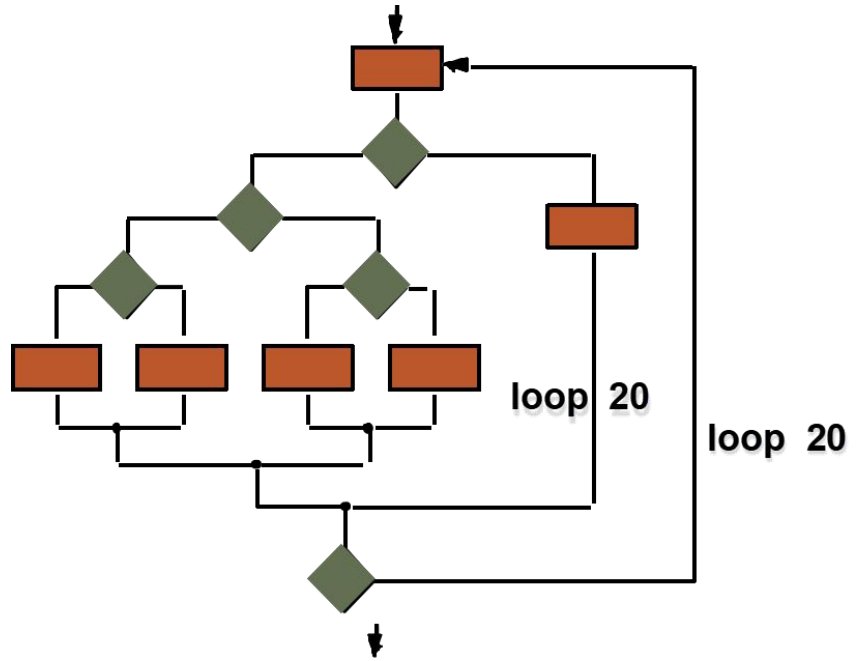
There are 10^{14} possible paths ! If we execute one test per millisecond, it would take 3,170 years to test this program!!

```
for i in range(10):
  for j in range(10):
    if i == j:
      print("Diagonal")
    else:
      print("Off-diagonal")
```

Here, we have two nested loops, each with a decision inside. The total number of paths becomes very large: 10 (outer loop) * 10 (inner loop) * 2 (if-else) for each combination = 200 paths.

What's Happening in the Diagram?

- The program contains multiple decision points (diamonds) and loops.
- Each decision (like if, switch) adds new branches.
- Loops (like for or while) multiply those paths even more because they can be executed 0, 1, or many times.
- If each of those 20 loops is just binary (run or not run), you already get 2^20 = 1,048,576 combinations!
- When nested or stacked, it can balloon into 10^{14} possible paths



The Implication for Testing

This "path explosion" makes exhaustive testing (testing every single possible path) impossible for all but the simplest programs. For a moderately complex program, the number of paths can easily reach into the trillions or even higher.

Strategies to Address It:

- **Code Coverage**: Measure how much of the code is tested (statement, branch, condition coverage), rather than testing every path.
- **Test Prioritization:** Focus on the most important or high-risk areas of the code, rather than trying to test everything. Techniques include risk-based testing, equivalence partitioning, and boundary value analysis.
- **Control Flow Graphs (CFGs):** Visualize program flow to understand its structure and identify critical paths. Metrics like cyclomatic complexity help assess code complexity.
- **Static and Dynamic Analysis:** Analyze code without or during execution to find potential issues and optimize testing efforts.
- **Formal Methods:** Use mathematical techniques for rigorous software development and verification.
- **Agile and Continuous Testing:** Test early and often throughout the development process, automating tests to enable frequent regression testing.

White-Box Testing

§ White box testing, sometimes called glass-box and structural testing.

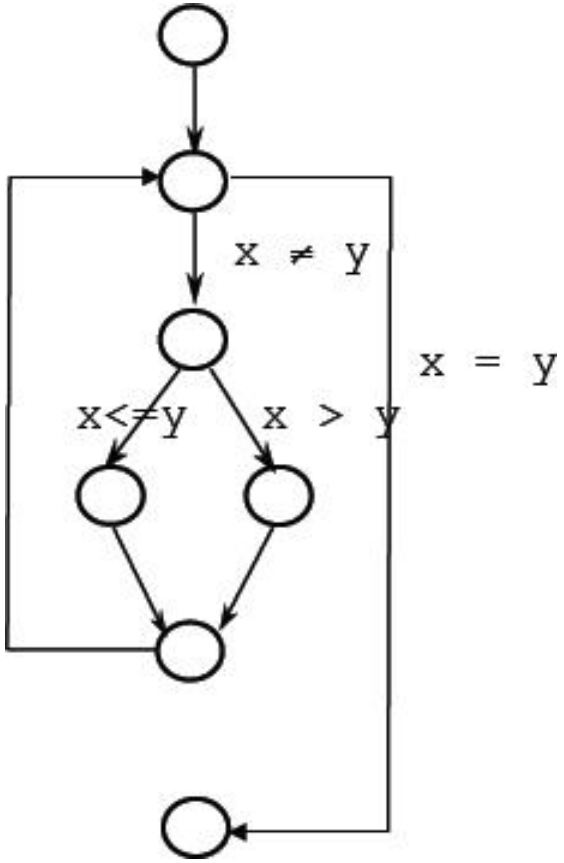
§ Various aspects like Statement Coverage Criteria, Edge coverage, Condition Coverage, Path Coverage are defined mathematically and test set is designed accordingly.

§ There are no algorithms for generating white box test data. However the checklist might help:

- Has every line of code been executed at least once by test data.
- Have all default paths been traversed at least once.
- Have all significant combinations of multiple conditions been identified and
  - Ø Have all logical decisions exercised on their true and false sides.
- Have all loops executed at their boundaries and within their operational bounds.
- Have internal data structures validated▪

Control flow graph (CFG)

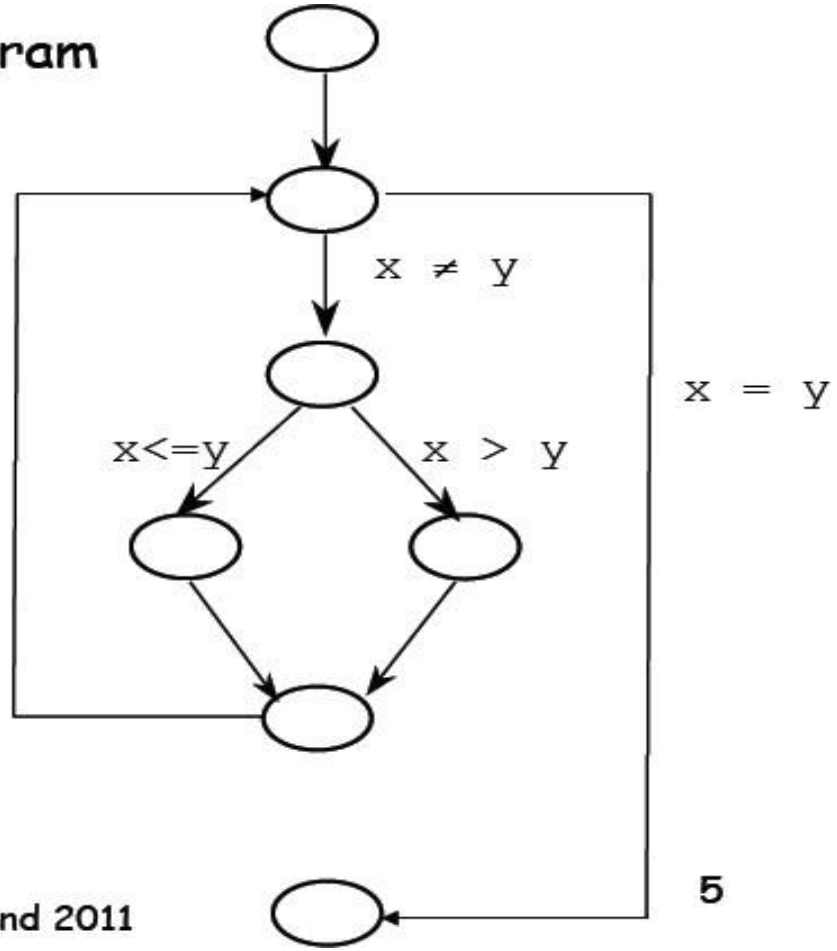
- Gives an abstract representation of the code
- Directed Graph
- Nodes are blocks of sequential statements
- Edges are transfers of control
- May be labeled with predicate representing the condition of control transfer



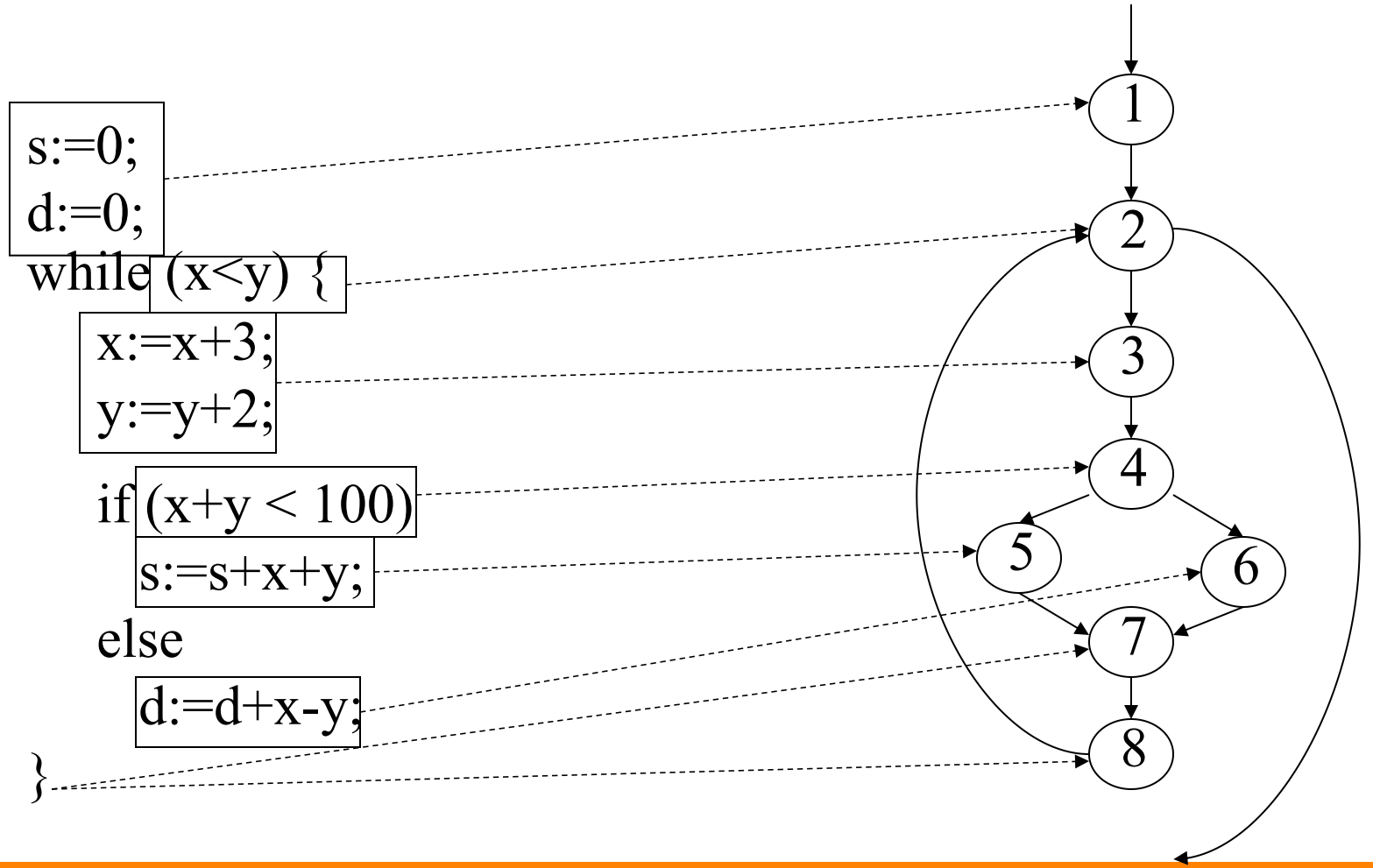
Control Flow Graph - Example

Greatest common divisor (GCD) program

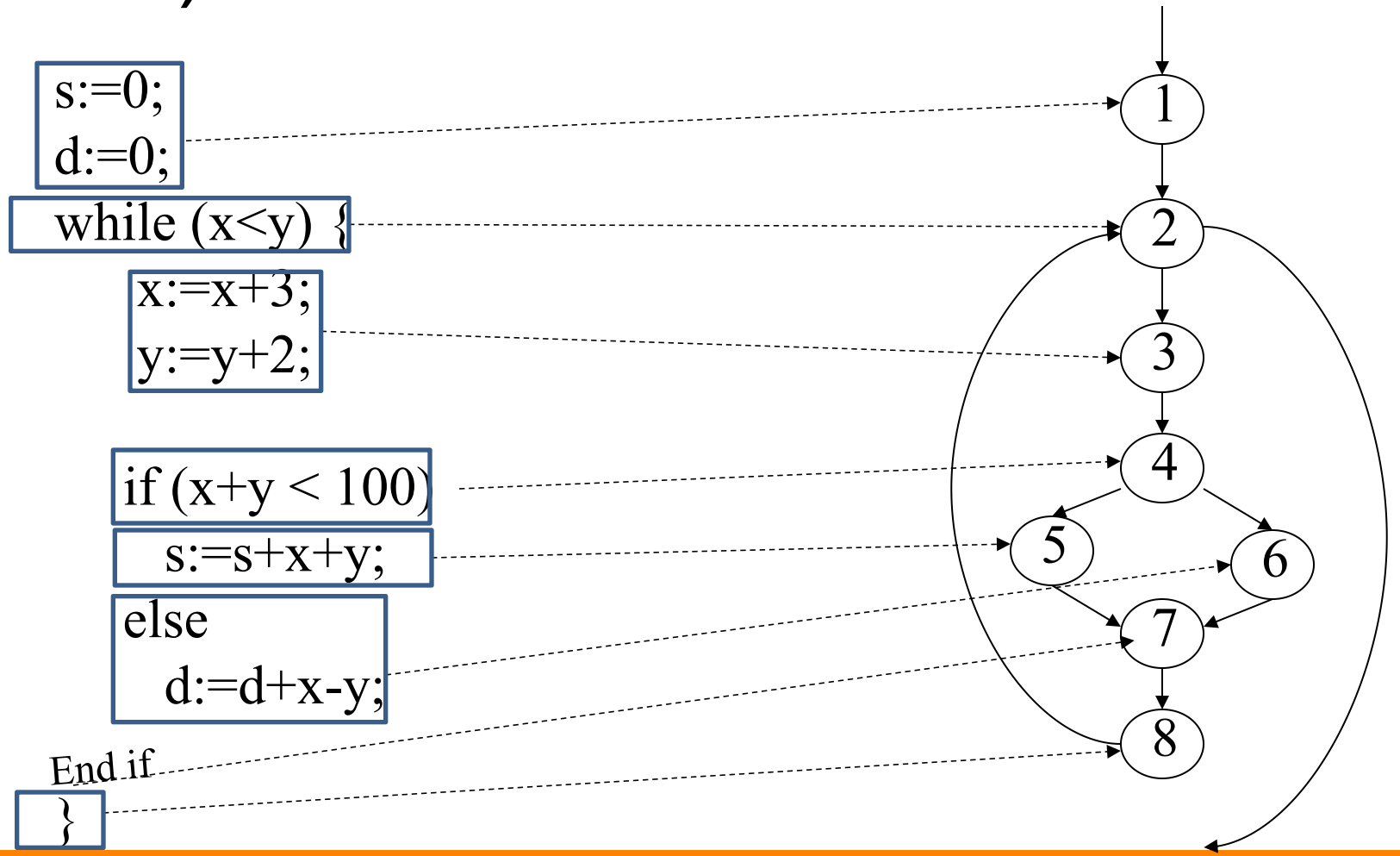
```
read(x);
read(y);
while x ≠ y loop
   if x>y then
     x := x - y;
   else
     y := y - x;
   end if;
end loop;
gcd := x;
```

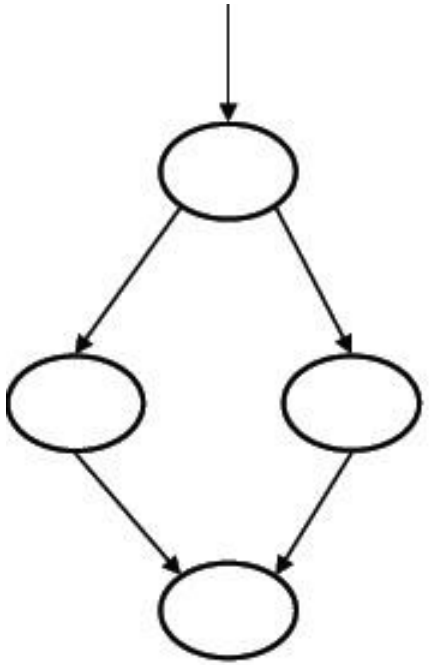


Example of a Control Flow Graph (CFG)

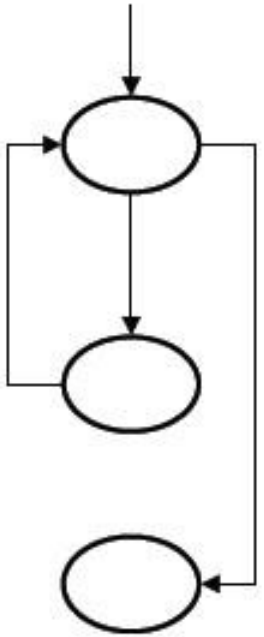


Example of a Control Flow Graph (CFG)

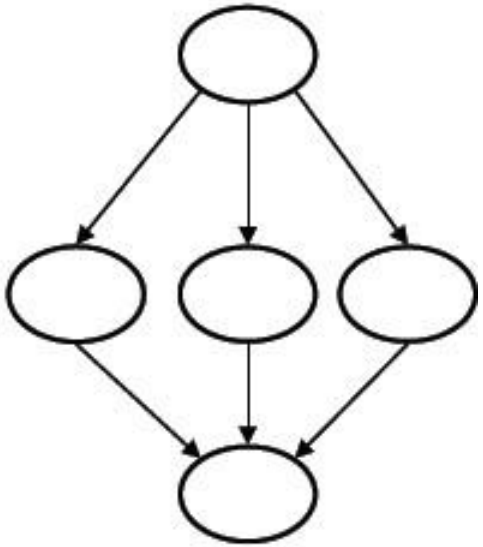




If-Then-Else



While loop



Switch

Example-CFG

A = 10

IF B >

C

THEN A = B

ELSE A =

C ENDIF

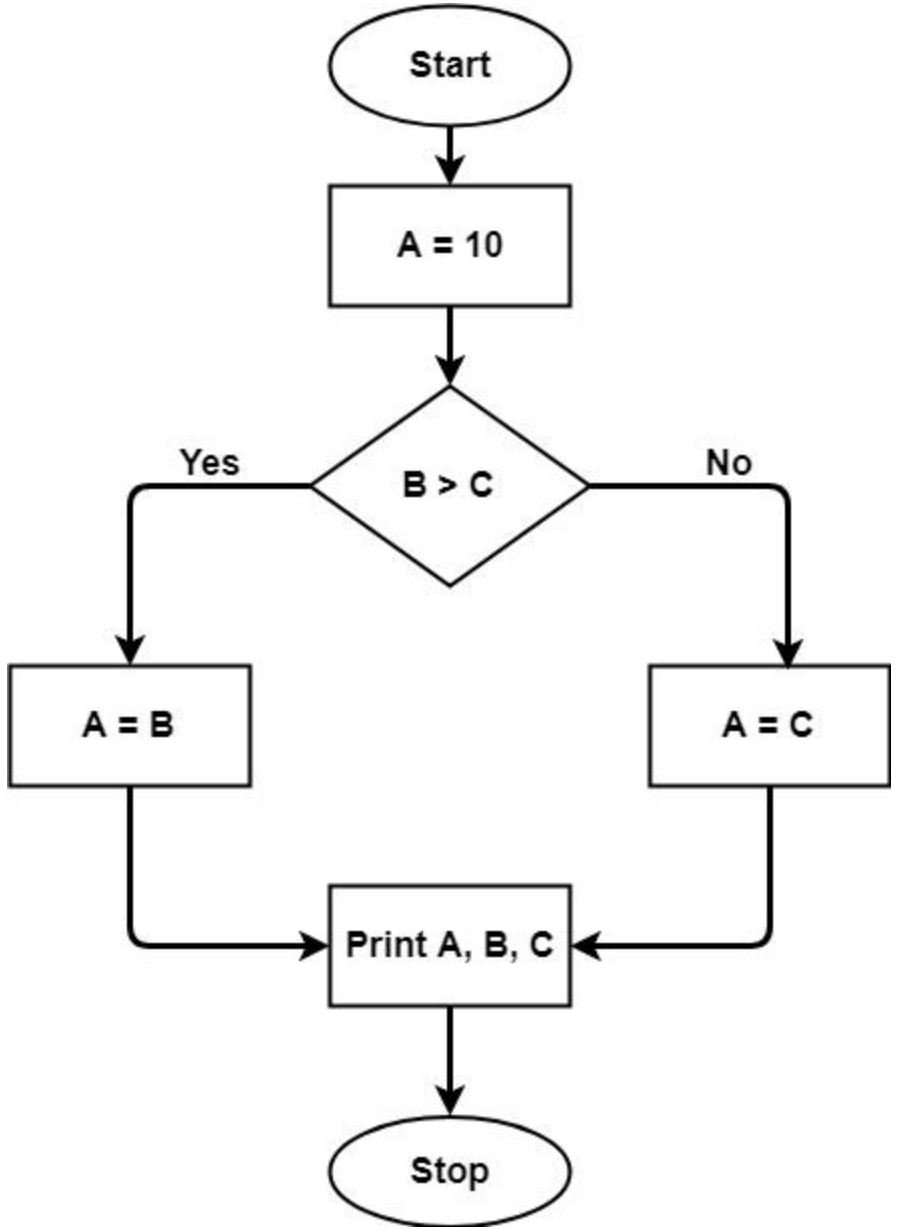
Print A

Print

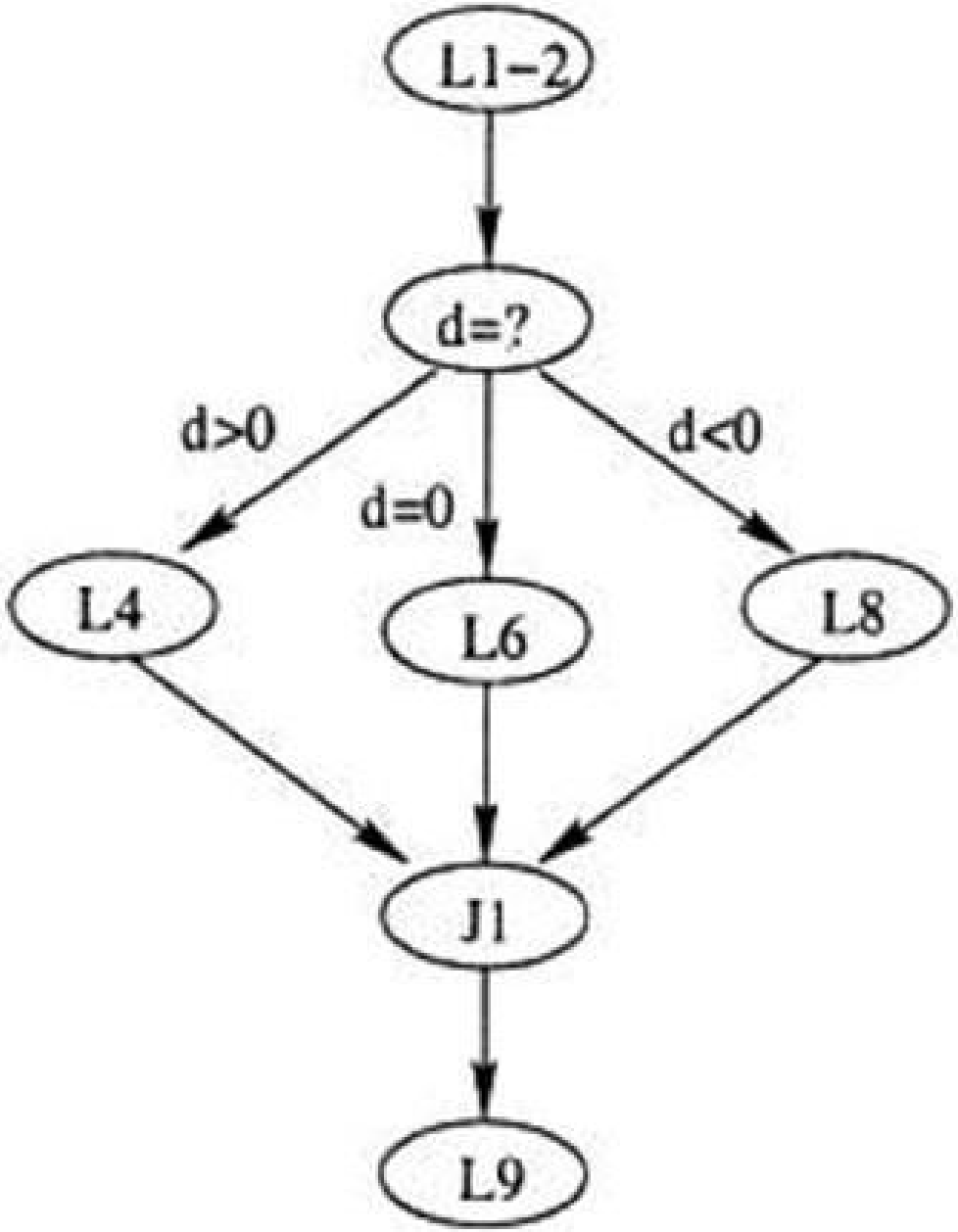
B

Print

C

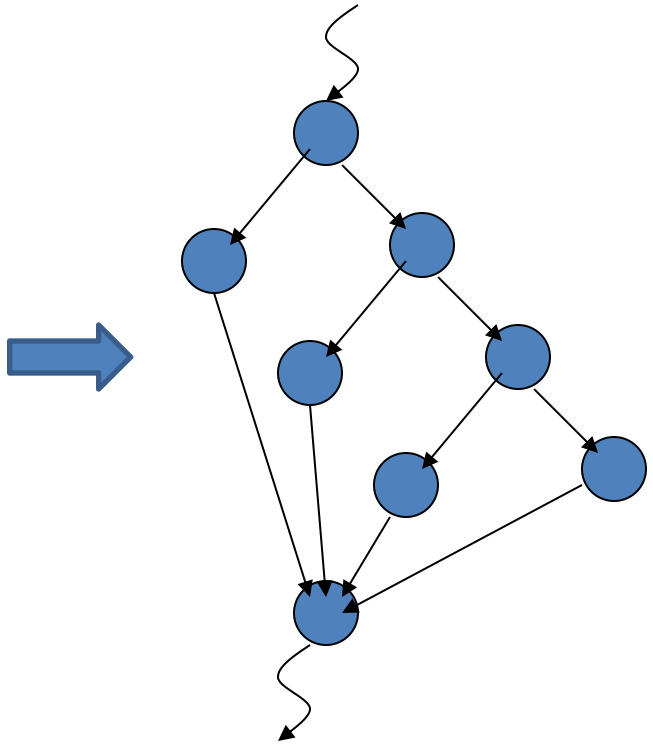


```
L1:     input(a, b, c);
L2:     d ← b * b - 4 * a * c;
L3:     if ( d > 0 ) then
L4:         r ← 2
L5:     else_if ( d = 0 ) then
L6:         r ← 1
L7:     else_if ( d < 0 ) then
L8:         r ← 0;
L9:     output(r);
```



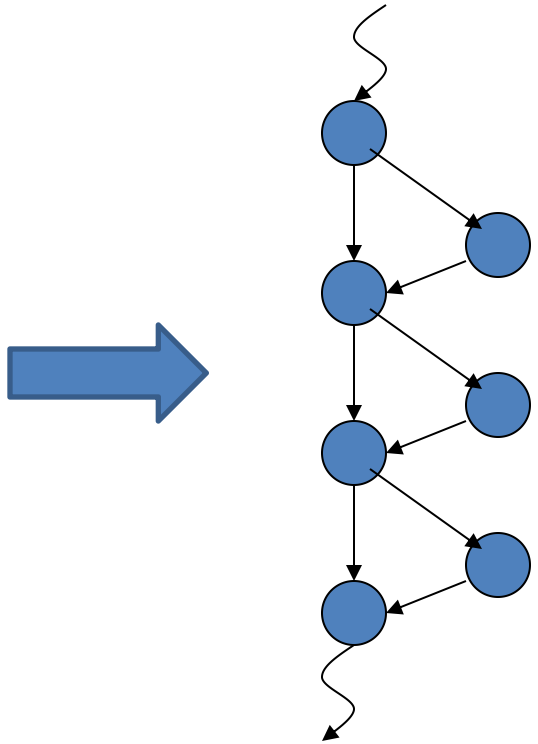
Example

```
if a then s1
else if b then s2
        else if c then s3
                else s4
                end_if_then_else
        end_if_then_else
end_if_then_else
```



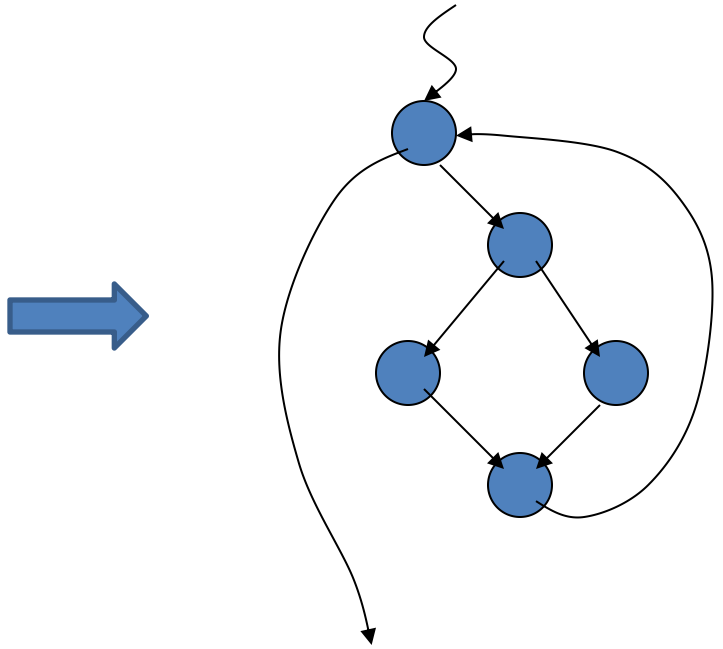
Example

```
if a then
            s1
end_if_then
if b then
            s2
end_if_then
if c then
            s3
end_if_then
```



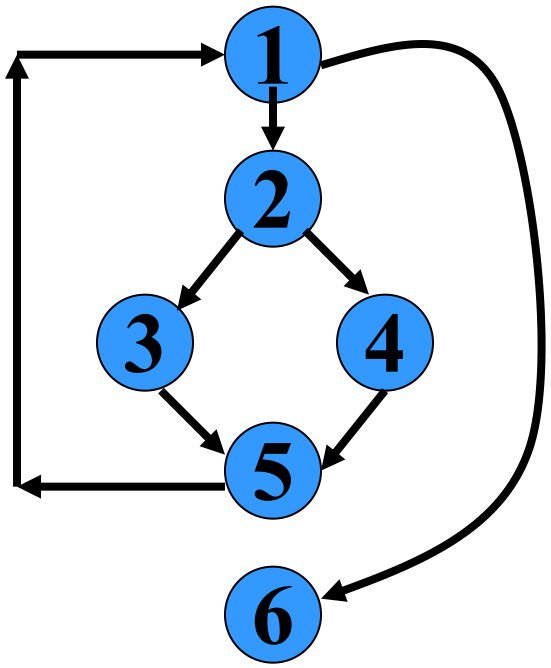
Example 3

```
while a do
    if b then s1
    else s2
    end_if_then_el
    se
end_while
```



Example 4

```
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;          }
```



Control Flow Coverage

Depending on the (adequacy) criteria to cover (traverse) CFGs, we can have different types of control flow coverage:

a) **Statement/Node Coverage**
b) **Edge Coverage**
c) **Condition Coverage**
d) **Path Coverage**

(1) Statement/Node Coverage

1) Hypothesis: Faults cannot be discovered if the statements containing them are not executed
2) Statement coverage criteria: Equivalent to covering all nodes in CFG
3) Executing a statement is a weak guarantee of correctness, but easy to achieve
4) Several inputs may execute the same statements
5) An important question in practice is:
   - how can we minimize (the number of) test cases so we can achieve a given statement coverage ratio?

What is Statement Coverage?

It is one type of white box testing technique that ensures that all the executable statements of the source code are executed at least once. It covers all the paths, lines, and statements of a source code.

$$\text{Statement Coverage} = \frac{\text{Number of executed statements}}{\text{Total number of statements}} \times 100$$

```
Prints (int a, int b) {
    int result = a+ b;
    If (result> 0)
        Print ("Positive", result)
    Else
        Print ("Negative", result)
    }
```

$$\text{Statement Coverage} = \frac{\text{Number of executed statements}}{\text{Total number of statements}} \times 100$$

Number of executed statements = 5,
Total number of statements = 7
Statement Coverage: 5/7 = 71%

If A = 3, B = 9

```
1  Prints (int a, int b) {
2  int result = a+ b;
3  If (result> 0)
4      Print ("Positive", result)
5  Else
6      Print ("Negative", result)
7  }
```

- The statements marked in yellow color are those which are executed as per the scenario

If A = -3, B = -9

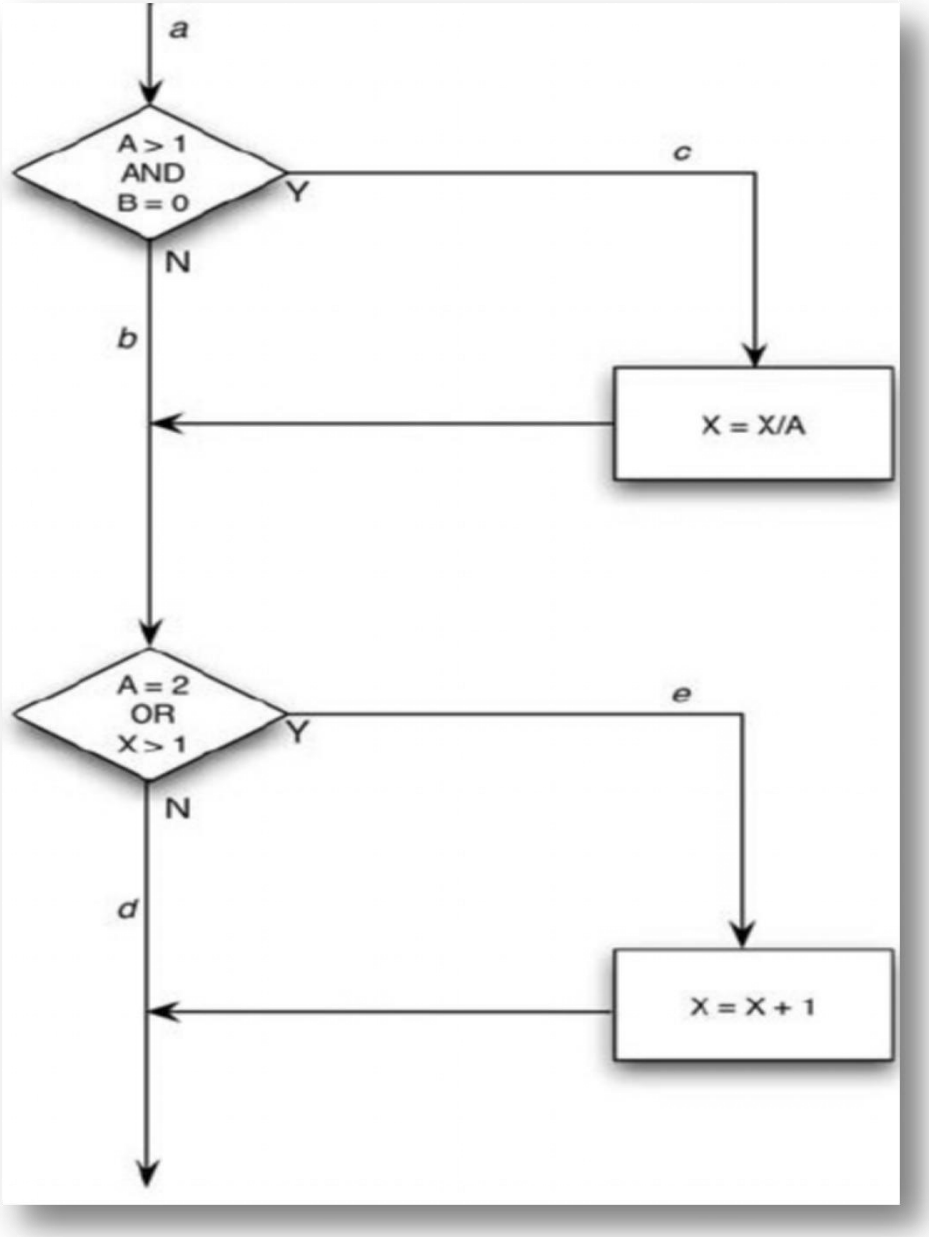
```
1  Prints (int a, int b) {
2  int result = a+ b;
3  If (result> 0)
4      Print ("Positive", result)
5  Else
6      Print ("Negative", result)
7  }
```

Number of executed statements = 6
Total number of statements = 7

Statement Coverage: 6/7 = 85%

Statement coverage

```
public void foo(int a, int b, int x) {
    if (a>1 && b==0)  {
        x=x/a;
    }
    if (a==2 || x>1)  {
        x=x+1;
    }
```



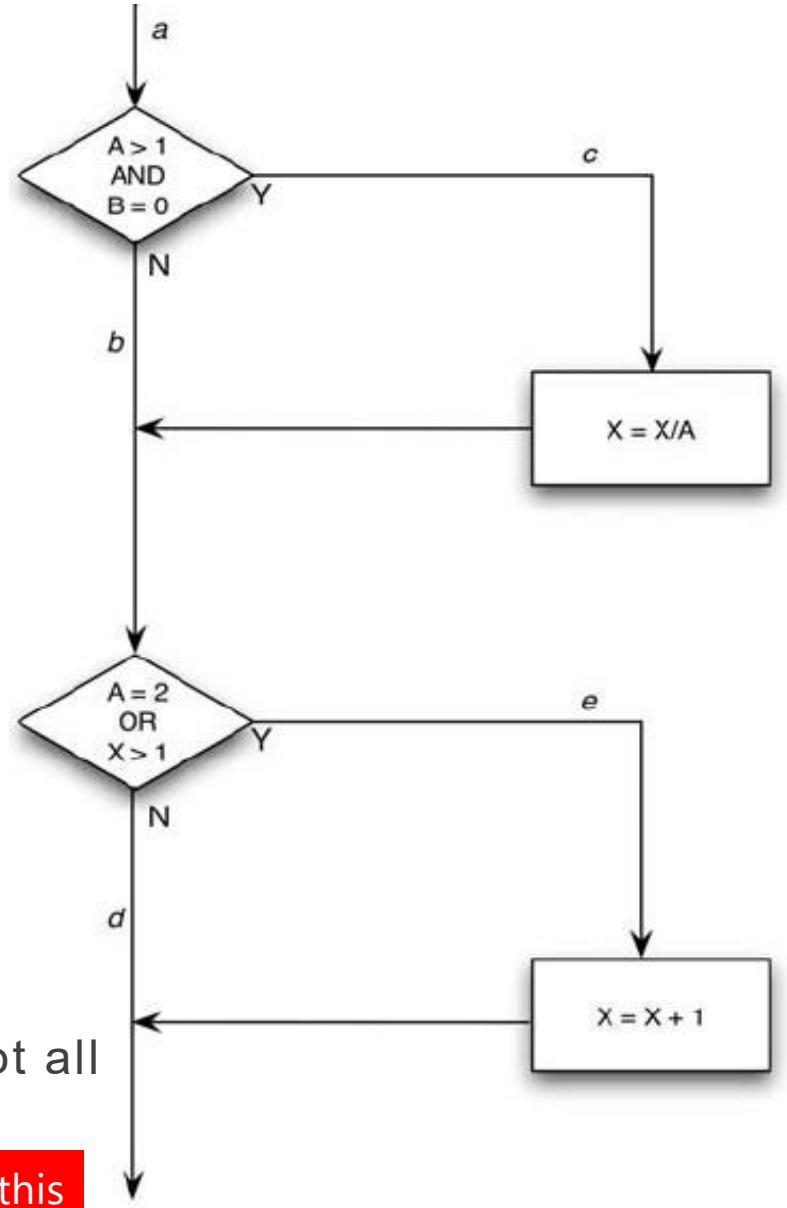
Execute every statement in the program at least once

For path: {a,c,e}

- (A, B, X) := (2, 0, 3) or
- (A, B, X) := (2, 0, any X)

Statement coverage

```
1  public void foo(int a, int b, int x) {
2      if (a>1 && b==0)   {
3          x=x/a;
4      }
5      if (a==2 || x>1)   {
6          x=x+1;
7      }
```



Problems:

- Line 2: if the requirement was || rather than &&
- Line 5: no difference if x > 1 condition was correct or even missing
- Didn't test the path {a, b, d} in which x is unchanged
- Conclusion: Same test (a=2, b=0) tests all statements but not all conditions.

- A single test case (a = 2, b = 0) can achieve 100% statement coverage for this code.
- However, this test case does not test all conditions and doesn't cover all possible execution paths.

Problem with statement coverage

Achieving statement coverage, which ensures every line of code is executed during testing, doesn't guarantee that all faults or bugs in the code will be detected. This is because it doesn't cover all possible execution paths, doesn't assess the quality of logic within each statement, and may miss edge cases or exceptional conditions. Therefore, while statement coverage is a useful metric, it's insufficient as a sole indicator of test thoroughness.

Statement coverage covers:

- Dead code.
- Unused statements.
- Unused branches.
- Missing statements.

Drawback of Statement Coverage:

- Cannot check the false condition.
- Different input values to check all the conditions.
- More than one test case may be required to cover all the paths with a coverage of 100%.

2. Decision Coverage

- Decision coverage reports the true or false outcomes of each Boolean expression.
- In this coverage, expressions can sometimes get complicated.
- Therefore, it is very hard to achieve 100% coverage

$$Decision\ Coverage = \frac{Number\ of\ Decision\ Outcomes\ Excercised}{Total\ Number\ of\ Decision\ Outcomes}$$

```
def check_value(x):
  if x > 10:
    print("x is greater than 10")  # Branch 1
  else:
    print("x is not greater than 10")  # Branch 2
```

Decisions: The condition x > 10

Branches:

- Branch 1: When x > 10 is true.
- Branch 2: When x > 10 is false.

Test Cases for 100% Decision Coverage:

check_value(15) (Tests the true branch)
check_value(5)   (Tests the false branch)

```
def check_grade(score):
  if score >= 90:
    print("A")  # Branch 1
  elif score >= 80:
    print("B")  # Branch 2
  else:
    print("C")  # Branch 3
```

Decisions:

- score >= 90
- score >= 80 (This decision is only evaluated if the first is false)

Branches:

- Branch 1: When score >= 90 is true.
- Branch 2: When score >= 90 is false AND score >= 80 is true.
- Branch 3: When score >= 90 is false AND score >= 80 is false.

Test Cases for 100% Decision Coverage:

•check_grade(95) (Tests Branch 1)
•check_grade(85) (Tests Branch 2)
•check_grade(75) (Tests Branch 3)

```
def process_input(a, b):
  if a > 0 and b < 10:
    print("Valid input")  # Branch 1
  else:
    print("Invalid input")  # Branch 2
```

Decision: a > 0 and b < 10

Branches:

- Branch 1: When the entire condition a > 0 and b < 10 is true.
- Branch 2: When the entire condition a > 0 and b < 10 is false.

Decision coverage

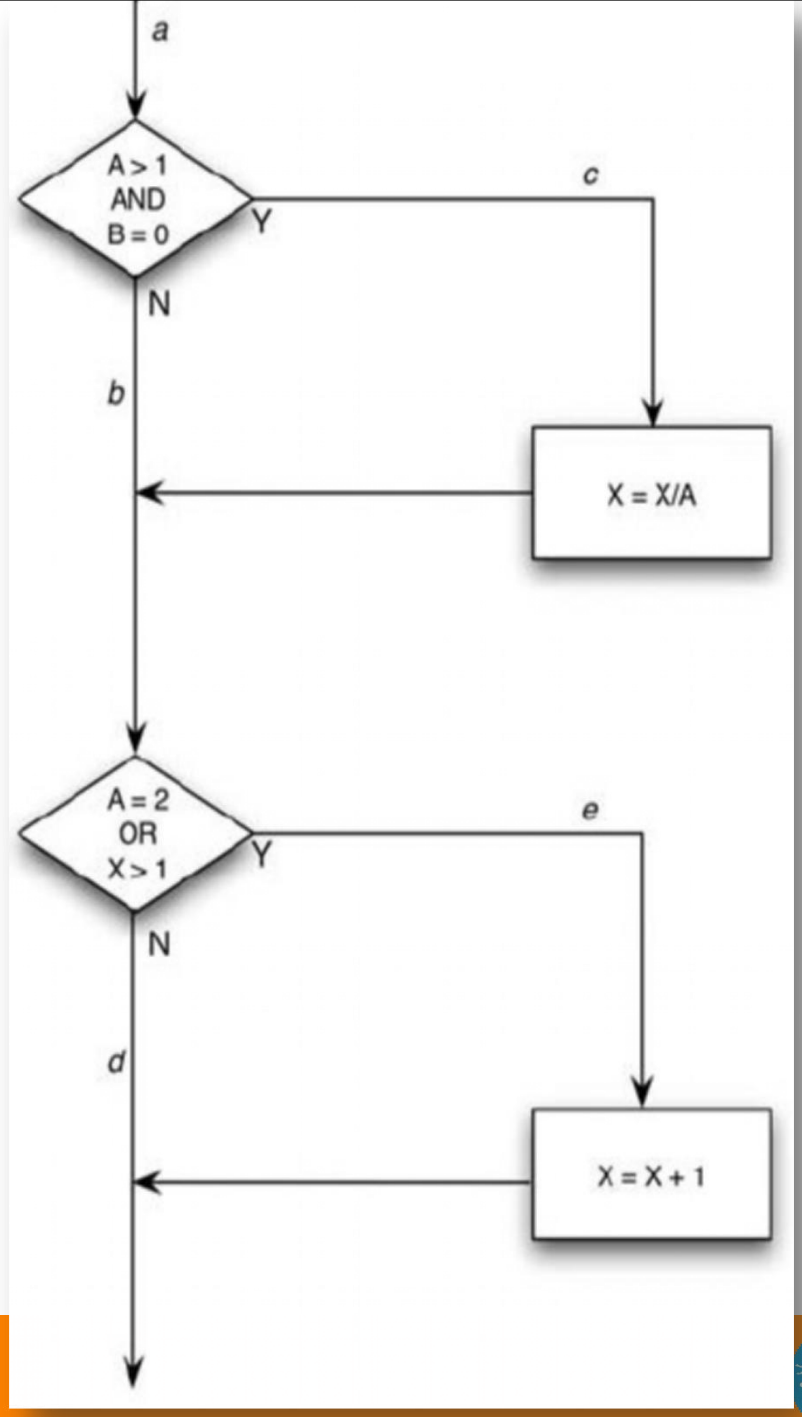
- You must write enough test cases that each decision has a T and a F outcome at least once.

  ace & abd or acd & abe

  - **(2, 0, 4) & (1, 1, 1) {TT & FF}**
  - **(3, 0, 3) & (2, 1, 1) {TF & FT}**

Problem

- If the condition x > 1 had a fault, e.g., x < 1, the mistake would not be detected



$$\text{Decision coverage} = \frac{\text{Number of decision outcomes exercised}}{\text{Total number of decision outcomes}} \times 100\%$$

```
Demo(int a) {
    if (a > 5)
        a = a * 3;
    Print
}
```

```
1  Demo(int a) {
2      If (a> 5)
3          a=a*3
4      Print (a)
5  }
```

- In the first scenario (a = 2), the if condition is false. So, in that specific scenario, only one outcome (the 'false' outcome) is exercised.
- In the second scenario (a = 6), the if condition is true. So, in that specific scenario, only one outcome (the 'true' outcome) is exercised.

- Value of a is 2
- Value of a is 6

In both cases 50% coverage

Explanation

- However, the provided test cases do not cover the case where a is exactly 5, which means that the conditional statement is neither true nor false. Hence, the total coverage achieved is not 50%.

- In fact, based on the provided test cases, only one branch of the code is covered (a > 5 evaluating to true). The other branch (a > 5 evaluating to false) is not covered.

- To **achieve 100% coverage**, test cases covering all possible branches of the code should be provided. Specifically, **including a test case where a = 5** would ensure coverage of both branches of the conditional statement.

Branch Coverage Testing

- **Branch Coverage** is a white box testing method in which every outcome from a code module(statement or loop) is tested. The purpose of branch coverage is to ensure that each decision condition from every branch is executed at least once.

- Decision coverage measures the coverage of conditional branches; branch coverage measures the coverage of both conditional and unconditional branches.

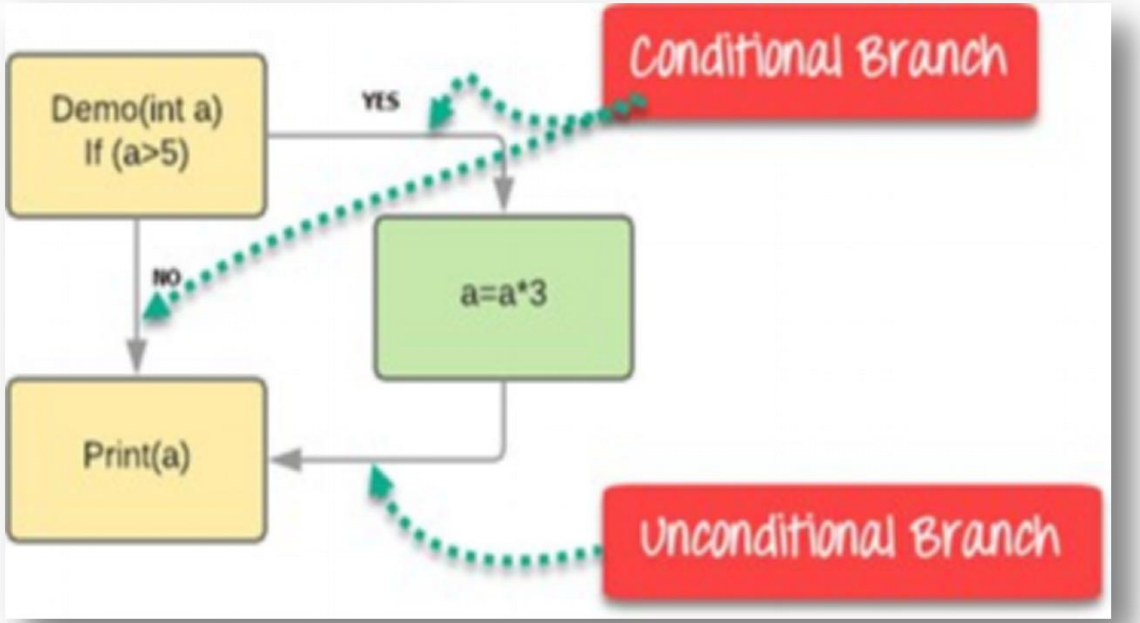
- The formula to calculate Branch Coverage:

$$Branch\ Coverage = \frac{Number\ of\ Executed\ Branches}{Total\ Number\ of\ Branches}$$

$$Branch\ Coverage = \frac{Number\ of\ Executed\ Branches}{Total\ Number\ of\ Branches}$$

```
Demo(int a) {
    if (a > 5)
        a = a * 3;
    Print(a);
}
```

Branch Coverage will consider unconditional branch as well



Test Case	Value of A	Output	Decision Coverage	Branch Coverage
1	2	2	50%	**33%**
2	6	18	50%	**67%**

```python
if x > 0 and y > 0:
    print("Both positive")
```

- **Branch Coverage**:
  - Tests that cover both the `True` and `False` **branches** of this `if` statement.
  - Example test cases:
    - `x = 1, y = 1` → `True` branch
    - `x = -1, y = 1` → `False` branch

- **Decision Coverage**:
  - Also needs to test both `True` and `False` **results of the whole condition** (`x > 0 and y > 0`)
  - But does **not** require checking **all combinations** of `x > 0` and `y > 0`.

So if you tested:

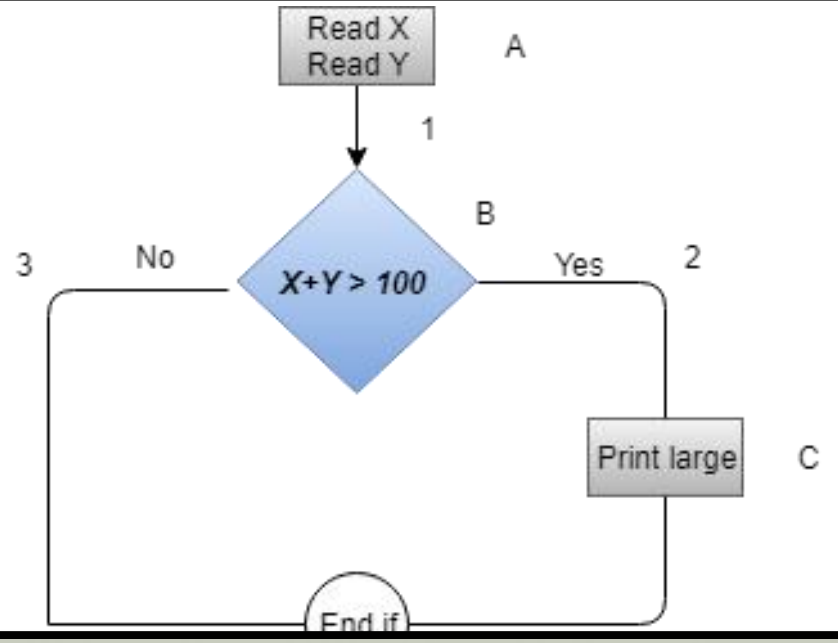
- `x = 1, y = 1` (True)
- `x = -1, y = 1` (False)

You're done with decision coverage , but for full branch coverage, you'd ideally also check:

•x = 1, y = -1 (another path where one subcondition fails)

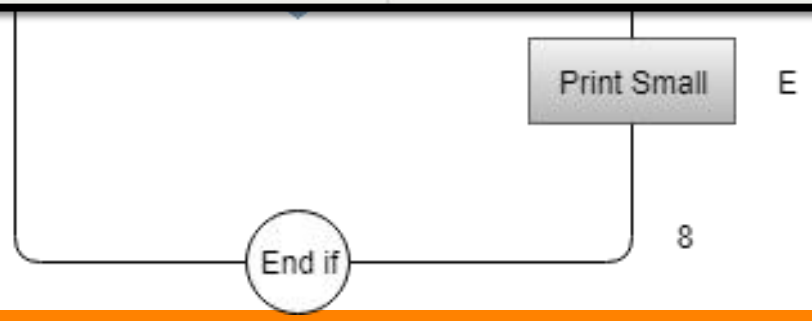
Example:

```
1  Read A
2  Read B
3  IF A+B > 10 THEN
4    Print "A+B is Large"
5  ENDIF
6  If A
7    Pri
8  ENDIF
```

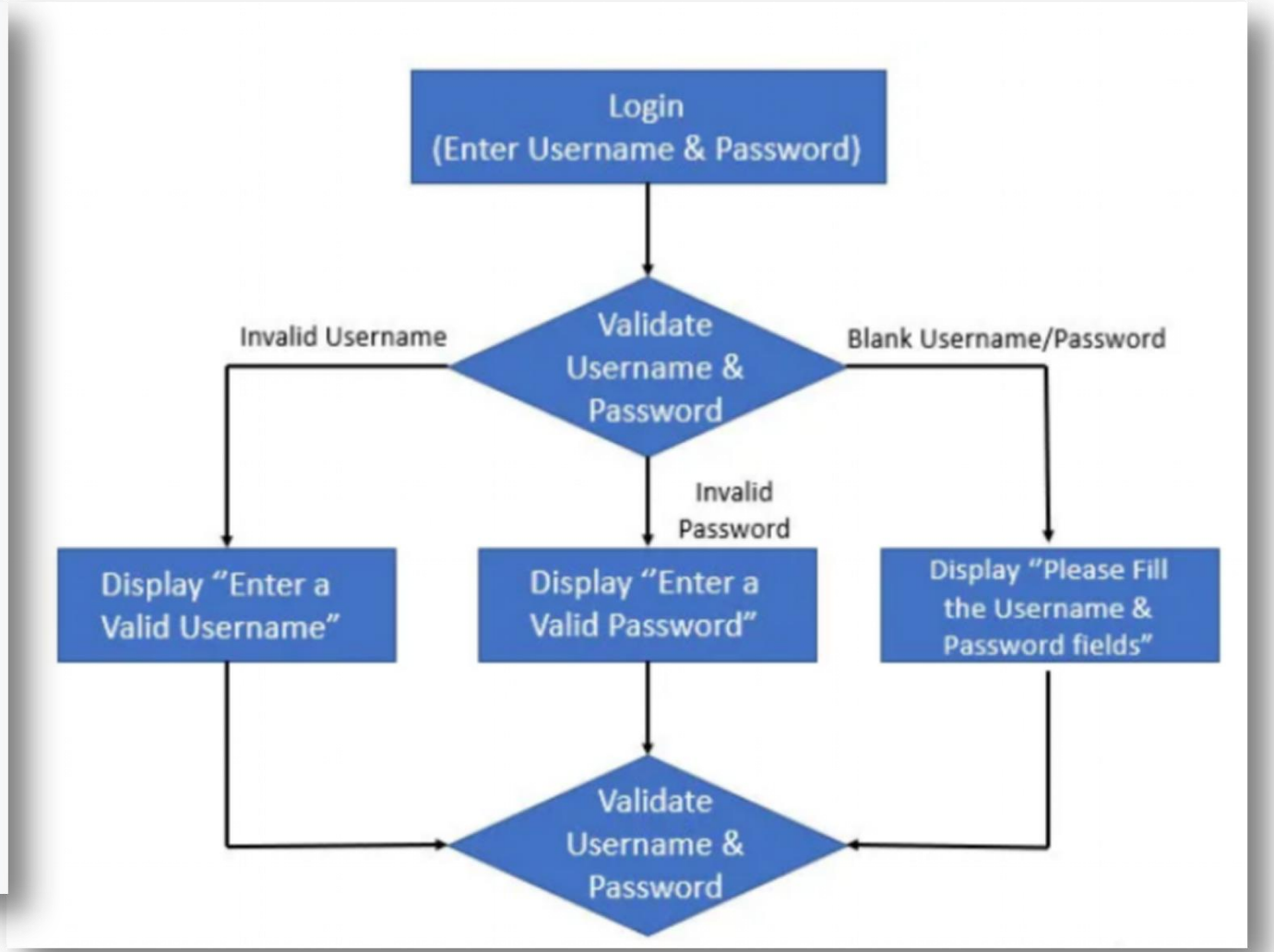


Path 1 - A1-B2-C4-D6-E8
Path 2 - A1-B3-5-D7

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	



```
READ Username
READ Password
IF Count (Username) < 8
PRINT “Enter a Valid Username”
ENDIF
IF Count (Password) < 5
PRINT “Enter a Valid Password”
ENDIF
IF Count(Username & Password) < 1
PRINT “Please Fill the Username & Password Fields”
ENDIF
ELSE
PRINT “Login Successfully”
```



$$\text{Branch Coverage (\%)} = \frac{\text{(Number of Functional Flow Implemented \& Tested)}}{\text{(Total Number of Success \& Fail Conditional Flow)}} * 100$$

Branch Coverage (%) = 3/3 * 100, which results in 100% coverage.

```python
def is_even_or_odd(x):
    if x % 2 == 0:
        return "Even"
    else:
        return "Odd"
```

Test case 1:

Input: x = 4
Expected Output: "Even“

Test case 2:

Input: x = 3
Expected Output: "Odd"

Test case 1 covers the branch where x % 2 == 0 evaluates to true (even numbers), and test case 2 covers the branch where it evaluates to false (odd numbers).

“focusing on branch coverage and ensuring that all branches of the code are tested, we inherently improve decision coverage as well, ensuring a more thorough evaluation of the code's behavior.”

Branch coverage advantages

- Allows you to validate-all the branches in the code
- Helps you to ensure that no branch leads to any abnormality of the program's operation
- Branch coverage method removes issues which happen because of statement coverage testing
- Allows you to find those areas which are not tested by other testing Methods
- It allows you to find a quantitative measure of code coverage Branch coverage ignores branches inside the Boolean expressions

Problem with Branch Coverage

- Branch coverage typically focuses on the execution paths within the code and does not directly consider branches within Boolean expressions or complex conditions

```
def is_even_or_odd(x):
    if x % 2 == 0 and x > 0:
        return "Even and Positive"
    elif x % 2 == 0 or x < 0:
        return "Even or Negative"
    else:
        return "Odd"
```

Even though there are additional branches within the Boolean expressions (x % 2 == 0 and x > 0 and x % 2 == 0 or x < 0), branch coverage will still only focus on the high-level branches related to the if, elif, and else blocks.

```python
def check_value(x):
    if x > 0 or x % 2 == 0:
        print("Pass")
    else:
        print("Fail")
```

✅ Decision Coverage:

- Decision coverage checks **if each decision (like the `if` condition)** has evaluated to **both `true` and `false`**.

Test Cases:

1. `x = 1` → `x > 0` is `True` → enters `if` → prints "Pass"
2. `x = -3` → `x > 0` is `False`, `x % 2 == 0` is `False` → `if` is `False` → prints "Fail"

✅ **Both outcomes of the `if` statement (True and False)** are exercised → **100% Decision Coverage**

✅ Branch Coverage:

- Branch coverage checks **all individual parts of compound conditions** — like in `x > 0 or x % 2 == 0`, each **sub-condition** is a branch.

To achieve 100% **Branch Coverage**, you must:

1. Evaluate `x > 0` as **True**
2. Evaluate `x > 0` as **False**
3. Evaluate `x % 2 == 0` as **True**
4. Evaluate `x % 2 == 0` as **False**

Test Cases:

1. `x = 1` → `x > 0` is `True` ✅
2. `x = -4` → `x > 0` is `False`, `x % 2 == 0` is `True` ✅
3. `x = -3` → both conditions are `False` ✅

✅ Now **all branches of each condition are covered → 100% Branch Coverage**

Condition Coverage

- **Condition Coverage** or expression coverage is a testing method used to test and evaluate the variables or sub-expressions in the conditional statement.

```
1   IF (x < y) AND (a>b) THEN
```

- The goal of condition coverage is to check individual outcomes for each logical condition. Condition coverage offers better sensitivity to the control flow than decision coverage.
- In this coverage, expressions with logical operands are only considered.
- For example, if an expression has Boolean operations like AND, OR, XOR, which indicates total possibilities.
- Condition coverage does not give a guarantee about full decision coverage.

$$Condition\ Coverage = \frac{Number\ of\ Executed\ Operands}{Total\ Number\ of\ Operands}$$

What is Condition Coverage Testing?

- Condition coverage is also known as Predicate Coverage in which each one of the Boolean expression have been evaluated to both TRUE and FALSE

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

- To ensure complete Condition coverage criteria for the above example, A, B and C should be evaluated at least once against "true" and "false".

```
read a, b, c;
if(a==0 || b==0)
{
   print 1;
}
else
{
   if(c==0 && d==0)
   {
     print 2;
   }
}
```

- With **a=0, b=0, c=0, d=0** It will only check condition a==0 because of II operator (1 condition checked out of 4) [25% coverage]
- With **a=1, b=0, c=0, d=0** It will also check condition b==0 (2 condition checked out of 4) [50% coverage]
- With **a=1, b=1, c=0, d=0** (4 condition checked out of 4) [100% coverage]

100% all the conditions are checked

Problem with Condition Coverage

Note that 100% condition coverage does not guarantee 100% decision coverage. For example, “if (A || B) {do something} else {do something else}” is tested with [0 1], [1 0], then A and B will both have been evaluated to 0 and 1, but the else branch will not be taken because neither test leaves both A and B false.

```
if (A || B) {
    // do something
} else {
    // do something else
}
```

- A and B both took on true and false values → condition coverage: 100%
  But the else branch is never executed → decision coverage: not 100%
- So: If there's a bug in the else block, you'll never catch it with only condition coverage.

With tests:

- **[A=0, B=1] → executes the then branch**
- **[A=1, B=0] → executes the then branch**

Better Alternatives

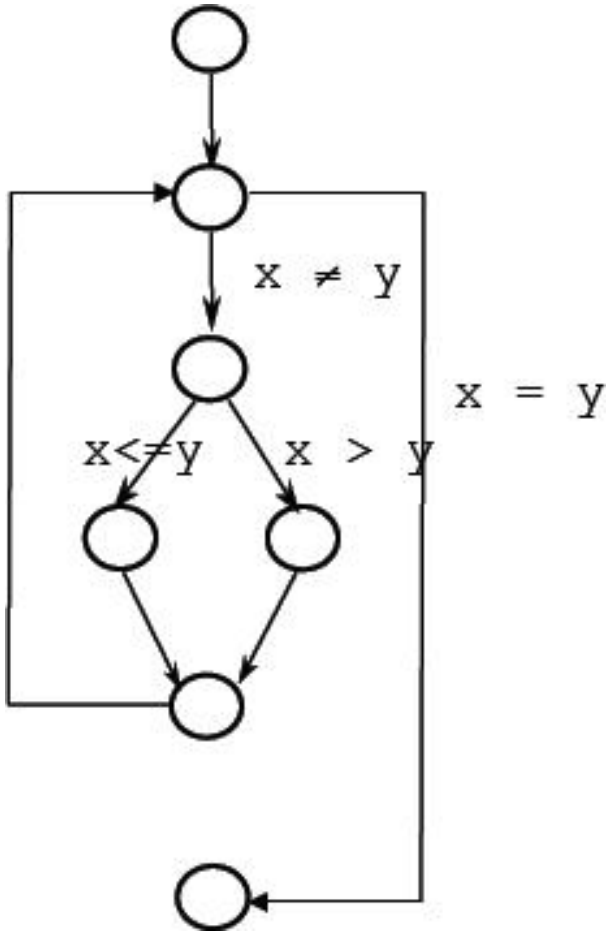
To fix this gap, consider:

1. Decision (Branch) Coverage – makes sure every if/else (true and false outcome) is taken.

2. Condition/Decision Coverage (C/D) – combines both.

3. Modified Condition/Decision Coverage (MC/DC) – required in critical systems (like avionics) because it ensures each condition independently affects the decision.

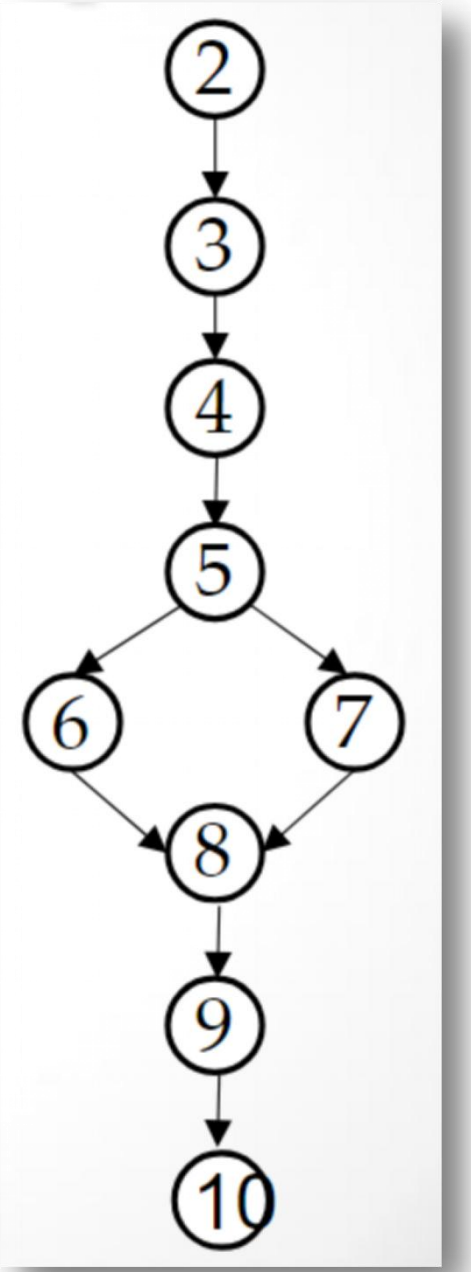
Coverage Type	Ensures All Conditions	Ensures All Branches	Notes
Condition	✅	❌	Might miss decision outcomes
Decision	❌	✅	Might miss individual condition effects
MC/DC	✅	✅ (with independence)	Very thorough, used in safety-critical systems



4. Path Coverage

- Path Coverage requires that all program paths will have been traversed at least once.
- In practice the number of paths is too large, if not infinite
  - (e.g., when we have loops)
- Some paths are infeasible
  - e.g., not practical given the system's business logic

1. Program ‘Simple Subtraction’
2. Input(x,y)
3. Output (x)
4. Output(y)
5. If x> y then Do
6. x-y = z
7. Else y-x = z
8. EndIf
9. Output(z)
10. Output “End Program”

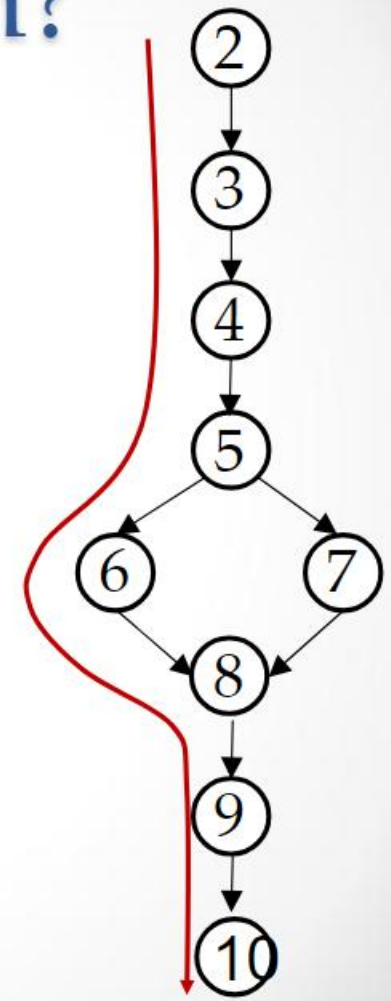


What is Path?

- A path through a program is a sequence of statements that starts at an entry, junctions, or decision and ends.
- A path may go through several junctions, processes, or decisions, on or more times.
- Paths consist of segments that has smallest segment is a link between 2 nodes.

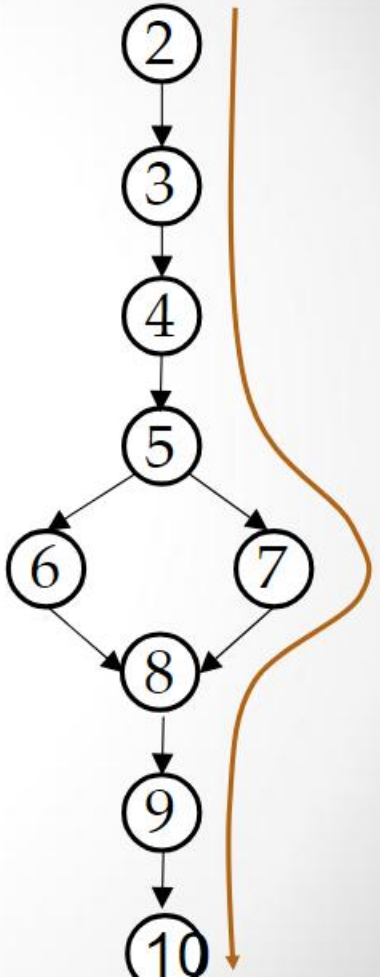
What is Path?

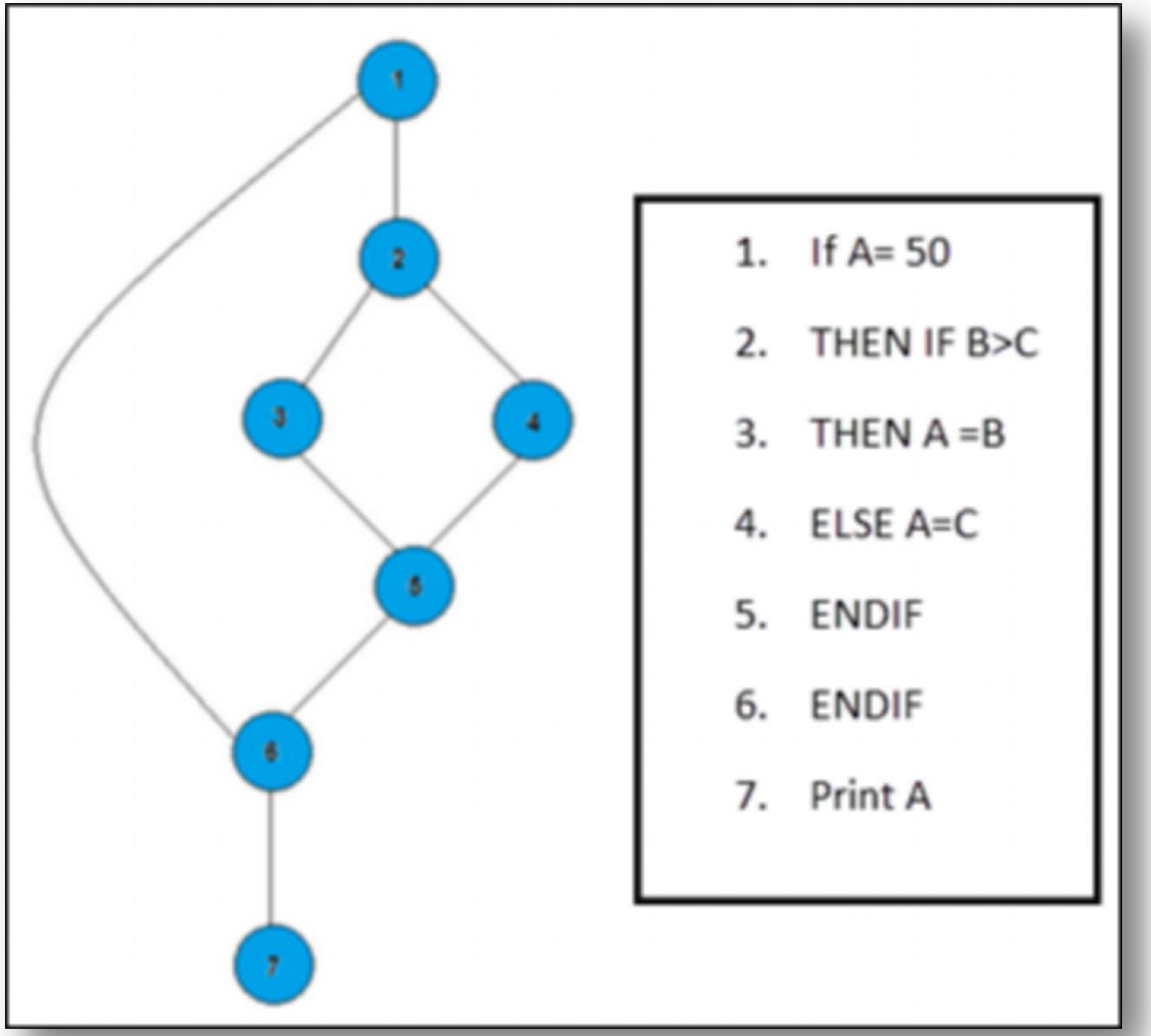
- For example:
  Path1 = 2-3-4-5-6-8-9-10



What is Path?

- For example:
  Path2 = 2-3-4-5-7-8-9-10





For this control graph, you can see three paths that require testing, namely:

1. **Path 1:** Test lines 1, 2, 3, 5, 6, and 7.
2. **Path 2:** Test lines 1, 2, 4, 5, 6, and 7.
3. **Path 3:** Test lines 1, 6, and 7.

Path Coverage – Dealing with Loops

- In practice, however, the number of paths can be too large, if not infinite (e.g., when we have loops) → Impractical
- A pragmatic heuristic: Look for conditions that execute loops
  - Ø Zero times
  - Ø A maximum number of times
  - Ø A average number of times (statistical criterion)
- For example, in the array search algorithm
  - Ø Skipping the loop (the table is empty)
  - Ø Executing the loop once or twice and then finding the element*
  - Ø Searching the entire table without finding the desired element

🧪 1. Zero Times – No Boxes to Check

- **What happens?** There are **no boxes** to look through.
- **In code?** The list or array is **empty**.
- **Why test it?** To make sure the code doesn't break when there's **nothing to loop through**.

`arr = []` → `len(arr) = 0` → `range(0)`

🧪 2. Once or Twice – Just a Few Boxes

- **What happens?** You look into **one or two boxes**, maybe find your toy quickly.
- **In code?** The loop runs just **1 or 2 times**.
- **Why test it?** To catch bugs that only happen **at the start** of the loop.

python

```
arr = [5]
target = 5
```

🧪 3. Many Times – Lots of Boxes

- **What happens?** You have to check through **a big line of boxes**.
- **In code?** The loop runs a **large number** of times.
- **Why test it?** To make sure the program can handle **long loops** without slowing down or crashing.

```
arr = list(range(10000))
target = 9999
```

🧪 4. All the Way to the End – Toy Not Found

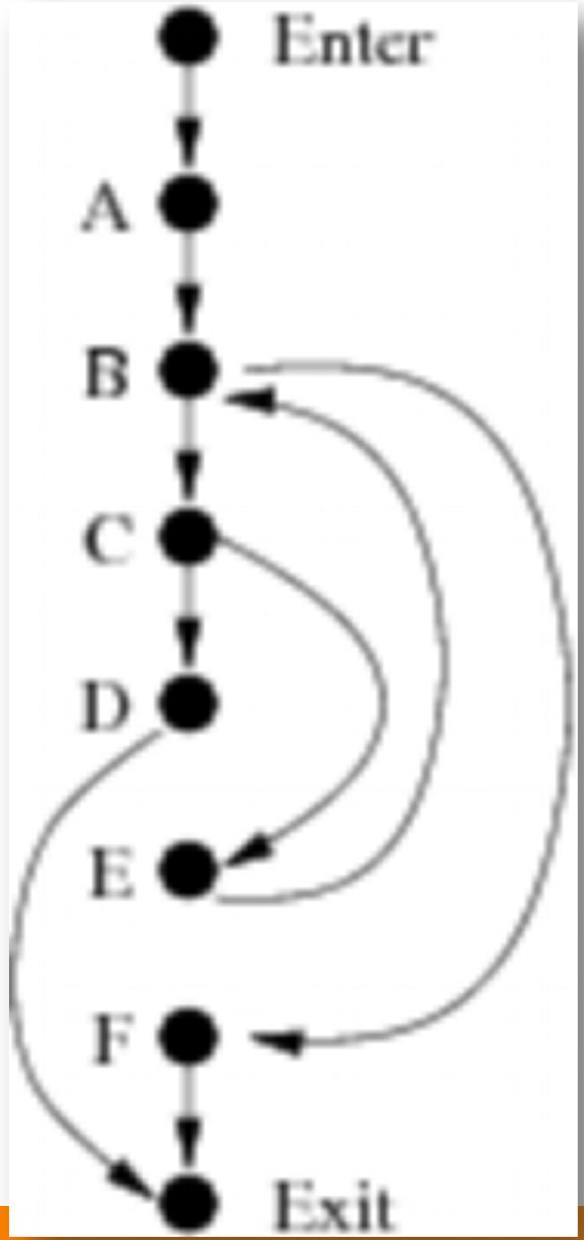
- **What happens?** You look through **every single box**, but the toy is **not there**.
- **In code?** The loop runs **until the end**, and doesn't find what it's looking for.
- **Why test it?** To make sure it behaves correctly in the **worst-case** scenario.

```
arr = [1, 2, 3, 4]
target = 10  # not in any box
```

What is McCabe's Cyclomatic Complexity?

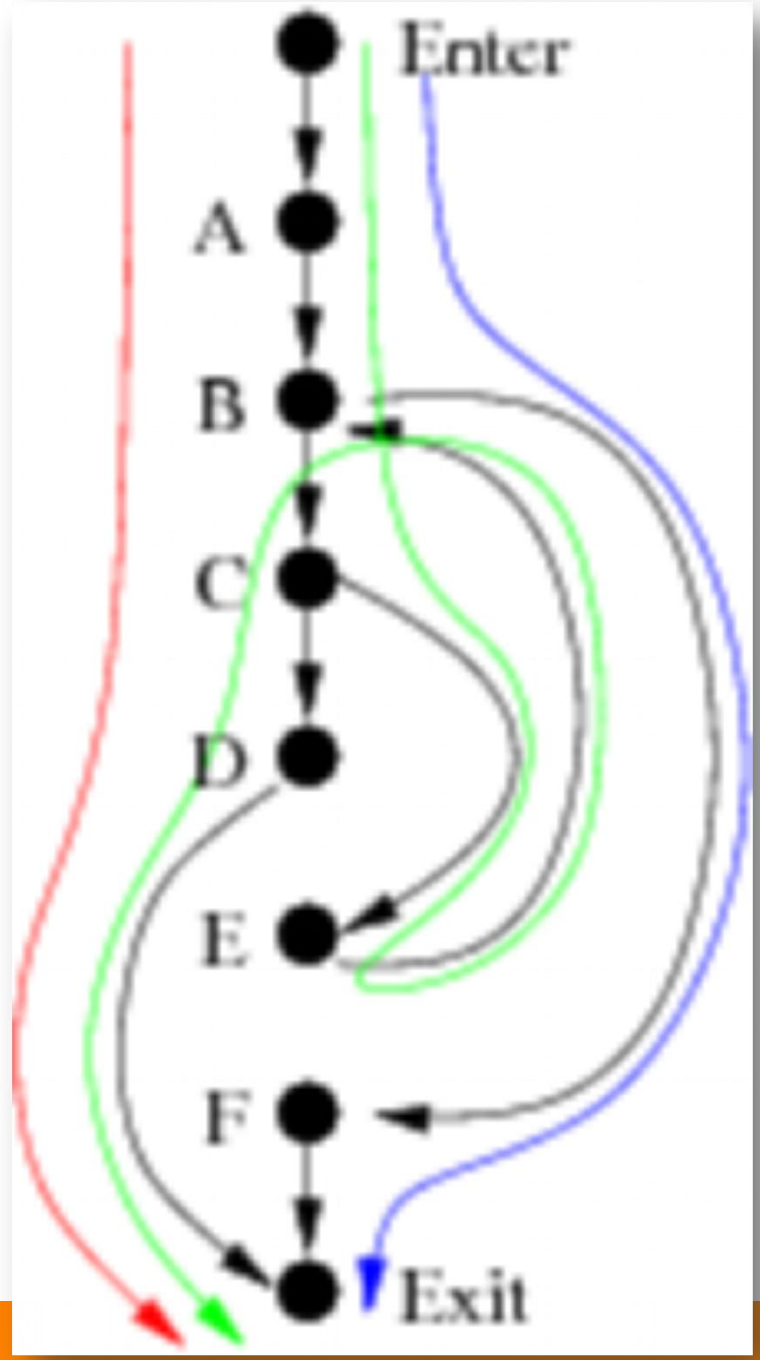
- A testing metric used for measuring the **complexity** of a software program.
- It is a quantitative measure of independent paths in the source code of a software program.
- Linearly independent path which introduces at least one new edge which edge not present in any order path that is already covered.
- Cyclomatic complexity can be calculated by using control flow graphs or with respect to functions, modules, methods or classes within a software program.

```
    public static boolean isPrime(int n) {
A       int i = 2;
B       while (i < n) {
C           if (n % i == 0) {
D               return false
            }
E           i++;
        }
F       return true;
    }
```



- A path through a flow graph starts at the Enter vertex and follows edges until the Exit vertex is reached. A set of paths are linearly independent if there none of them can be created by combining the others in some way. It may take a bit of staring, but in the case of our example above, there are at most three such paths

- Linearly independent path which introduces at least one new edge which edge not present in any order path that is already covered.



Cyclomatic Complexity

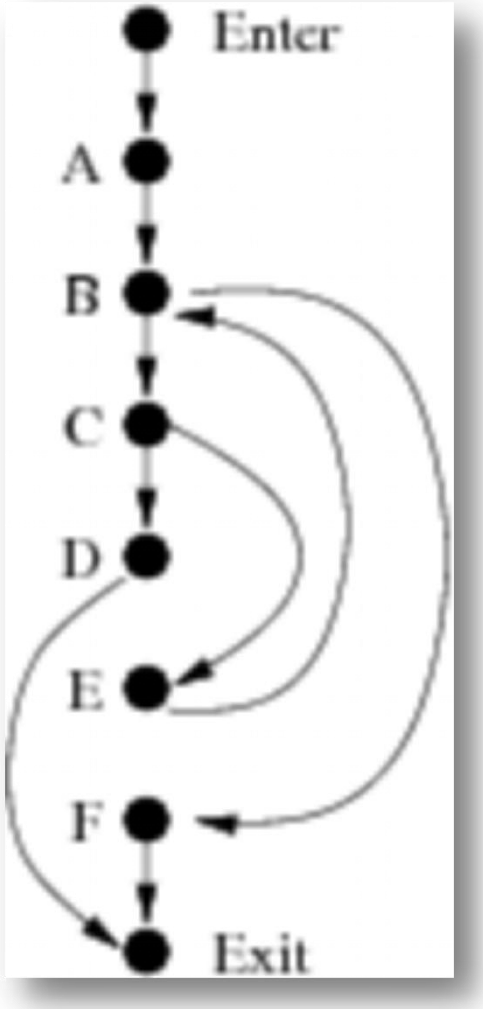
Cyclomatic complexity provides a quantitative measure of the logical complexity of a program

- It is the number of tests that must be conducted to assure that all statements have been executed at least once

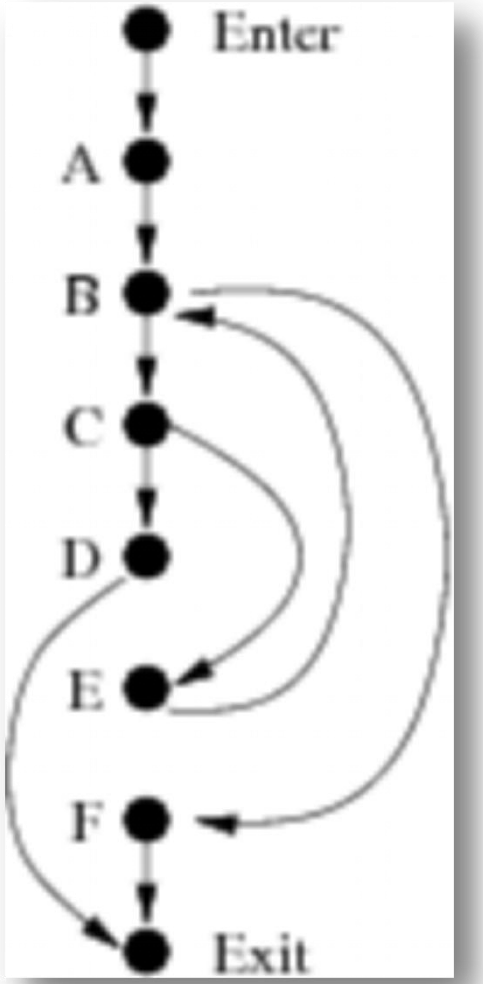
Graph theory tells us that for graphs with the properties that flow graphs have, the maximum number of linearly independent paths is e - n + 2, where e is the number of edges and n is the number of vertices.

Cyclomatic complexity, V(G) is given by:

*(1) V(G) = E - N + 2 where **E** is the number of **edges** and **N** number of **nodes***
*(2) V(G) = P + 1, where **P** is the number of **predicate***
(3) V(G) = number of regions (**4**)



In the case of the example,
n = 8, e = 9, so
NLIP = e - n + 2
NLIP = 9 - 8 + 2 = 3.



- Maximum number of linearly independent paths = d + 1, where d is the number of decision vertices.
- A decision vertex corresponds to a conditional branch in the code. Decision vertices are easy to spot — they are the ones that have multiple edges leaving them, or with out-degree greater than 1. In fact, that this property almost certainly requires that such vertices have out-degree exactly 2, which is certainly true of the standard conditional statements such as if or while. (But more on this in a moment.)
- Returning to our example, we see two decision vertices, and they correspond to the while and if statements. So again we get 3 linearly independent paths.

V(G) = Number of decision points + 1
(e.g., number of if, while, for, case, etc.)

Why might we care about linearly independent paths?

The main reason (and in fact the one put forward by McCabe) is that it gives us some idea about how much testing is going to be needed **(estimating test effort)**. As each path corresponds to a different combination of results of the conditions, it is reasonable that we should make sure we have at least one test case for each path. In some sense, the NLIP for a method is the minimum number of test cases needed to adequately test the method. In reality, the minimum number of test cases is almost certainly much larger than NLIP, but it's a good start.

Why is this useful?

- It gives you a baseline for the number of test cases you'll need.
- Each independent path can potentially uncover different bugs.
- If you don't test all the independent paths, you're more likely to miss edge cases or hidden bugs.

```python
def check(x):
    if x > 0:
        print("Positive")
    else:
        print("Non-positive")
```

- 1 decision (`if`) → V(G) = 1 + 1 = **2**
- ➡️ You need **2 test cases**: one where `x > 0` and one where `x <= 0`

Testing Phase

What content should be included in a software test plan?

- Testing activities and schedule
- Testing tasks and assignments
- Selected test strategy and test models
- Test methods and criteria
- Required test tools and environment
- Problem tracking and reporting
- Test cost estimation

Test Execution

- using manual approach
- using an automatic approach
- using a semi-automatic approach

Basic activities in test execution:

- Select a test case
- Set up the pre-conditions for a test case
- Set up test data
- Run a test case following its procedure
- Track the test operations and results
- Monitor the post-conditions of a test case & expected results
- Verify the test results and report the problems if there is any
- Record each test execution

How White Box Testing Techniques works?

- A major White box testing technique is Code Coverage analysis.
- Code Coverage analysis eliminates gaps in a Test Case suite.
- It identifies areas of a program that are not exercised by a set of test cases.
- Once gaps are identified, create test cases to verify untested parts of the code, thereby increasing the quality of the software product

White Box Testing Tools

Below is a list of top white box testing tools.

- EclEmma
- NUnit
- PyUnit
- HTMLUnit
- CppUnit

Advantages of White Box Testing

- Code optimization by finding hidden errors.
- White box tests cases can be easily automated.
- Testing is more thorough as all code paths are usually covered.
- Testing can start early in SDLC even if GUI is not available.

Disadvantages of White Box Testing

- White box testing can be quite complex and expensive.
- Developers who usually execute white box test cases detest it. The white box testing by developers is not detailed and can lead to production errors.
- White box testing requires professional resources with a detailed understanding of programming and implementation.
- White-box testing is time-consuming, bigger programming applications take the time to test fully.