Software Engineering
(CSE 355)

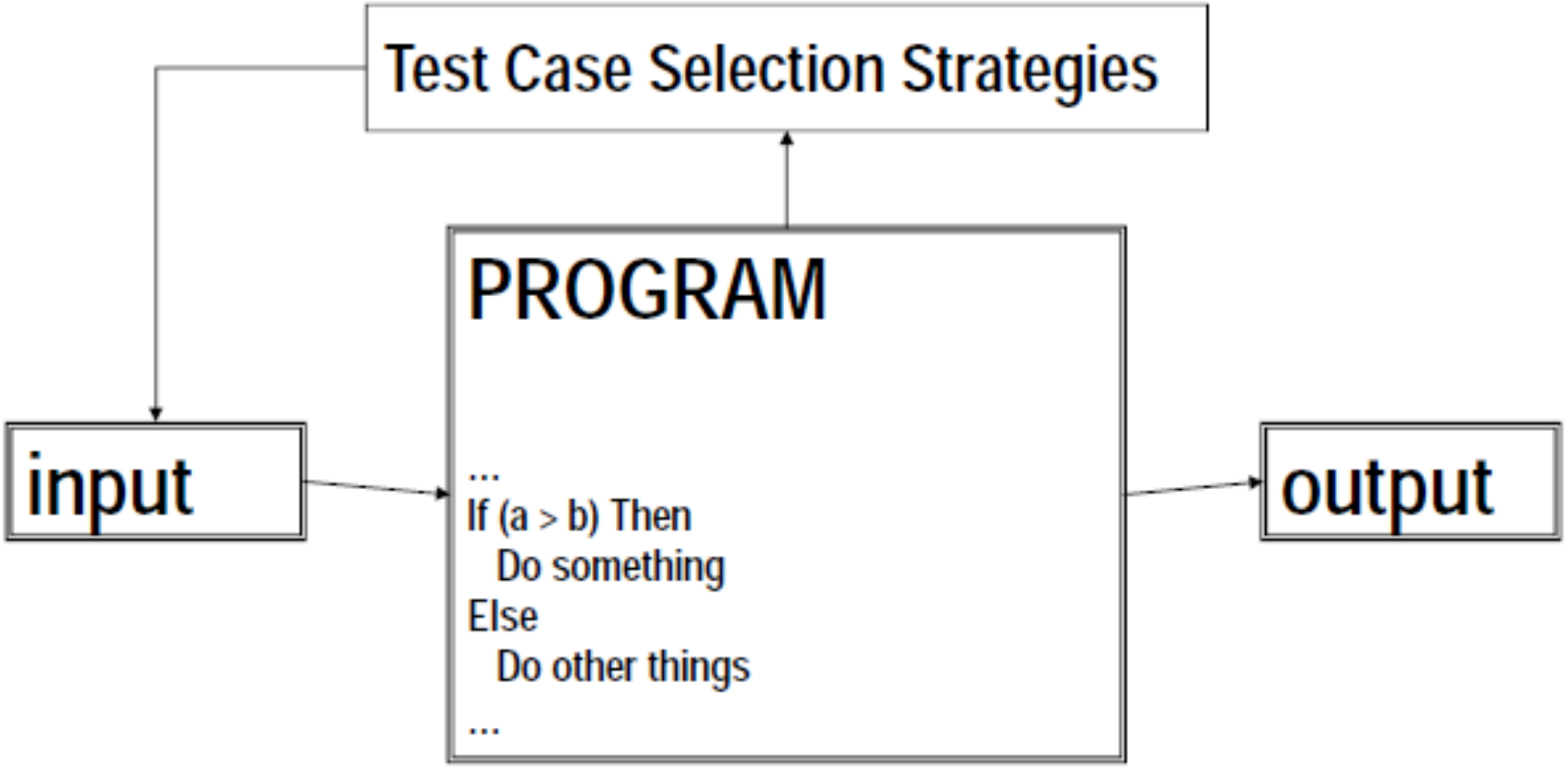
Lecture 9
(White Box Testing)

White Box Testing

White Box Testing: Introduction

- Test Engineers have access to the source code.
- Typical at the Unit Test level as the programmers have knowledge of the internal logic of code.
- Tests are based on coverage of:
  - Code statements;
  - Branches;
  - Paths;
  - Conditions.
- Most of the testing techniques are based on *Control Flow Graph* (denoted as CFG) of a code fragment.

White Box Testing – A common view



Control Flow Graph: Introduction

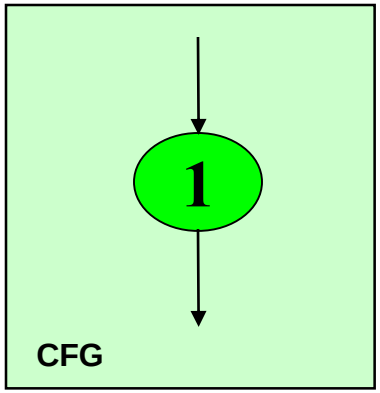
- An abstract representation of a structured program/function/method.
- Consists of two major components:
  - ***Node*:**
    - Represents a stretch of sequential code statements with no branches.
  - ***Directed Edge*** (also called *arc*):
    - Represents a branch, alternative path in execution.
- Path:
  - A collection of *Nodes* linked with *Directed Edges.*

Simple Examples

```
int a;
float b;
printf("Hello World");
scanf("%d", &a);
```

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

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



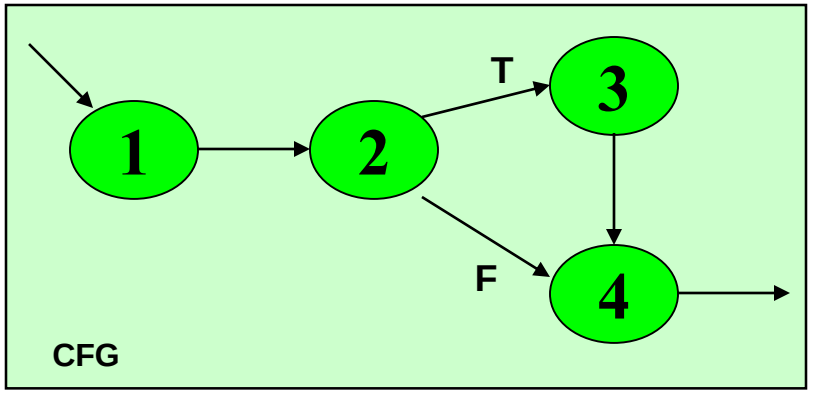

```
Statement1;
Statement2;

if X < 10 then
   Statement3;

Statement4;
```

1

2

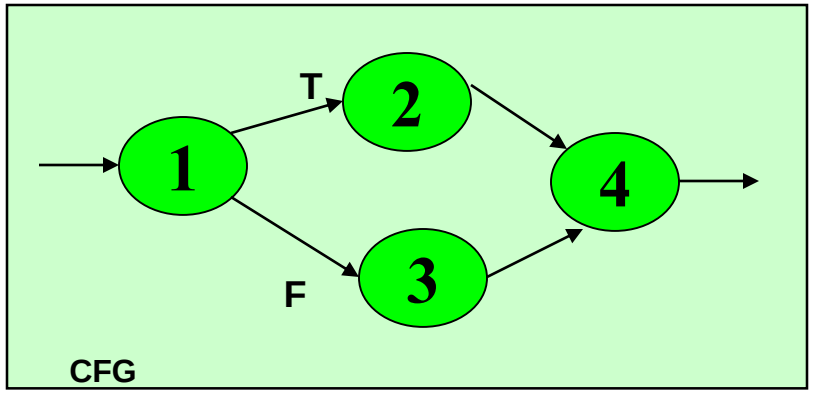
3

4



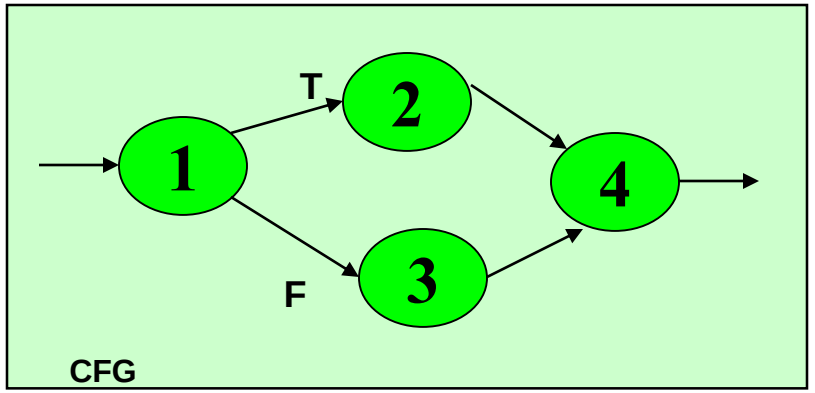
More Examples

```
if X > 0 then          1
   Statement1;         2
else
   Statement2;         3
```

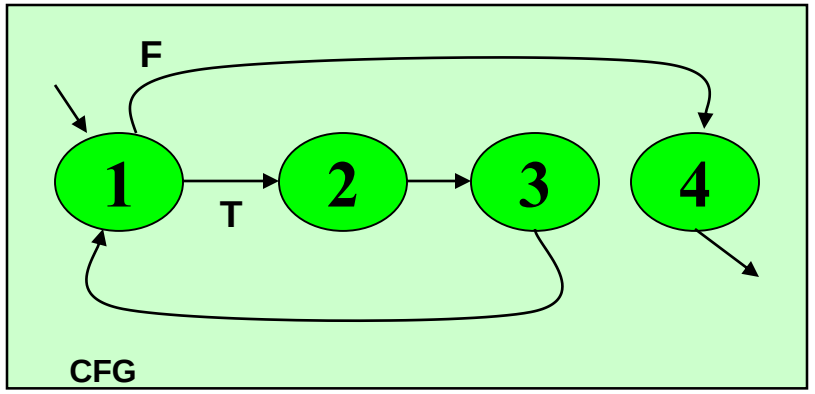


More Examples

```
if X > 0 then          1
   Statement1;         2
else
   Statement2;         3
```




```
while X < 10 {         1
   Statement1;         2
   X++; }              3
```



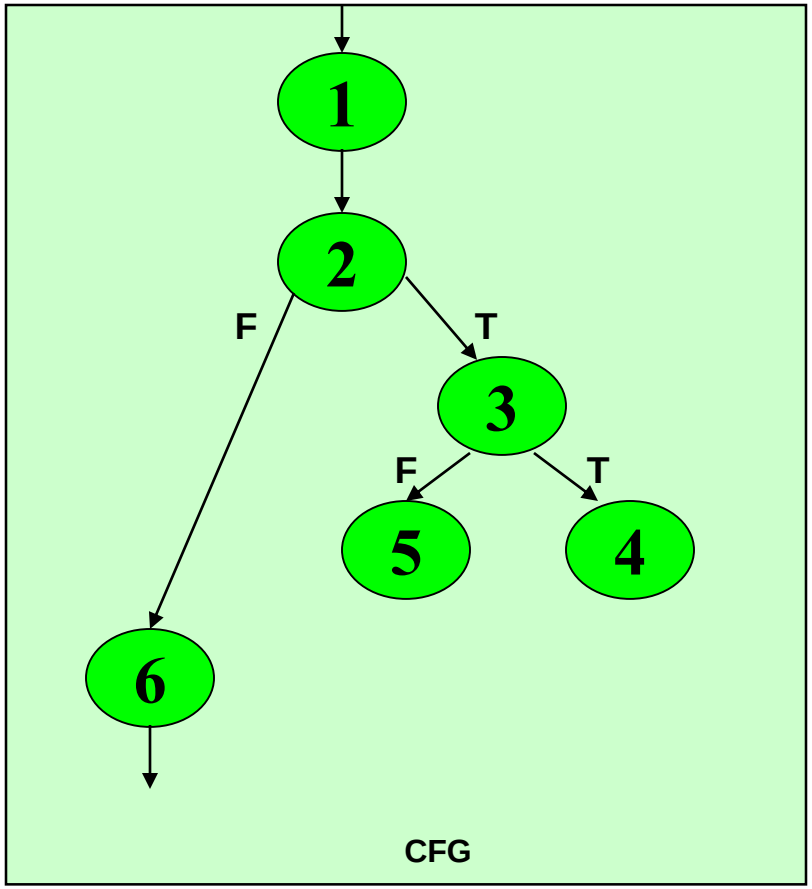
Question: Why is there a node **4** in both CFGs?

Notation Guide for CFG

- A CFG should have:
  - 1 entry arc (known as a directed edge, too).
  - 1 exit arc.
- All nodes should have:
  - At least 1 entry arc.
  - At least 1 exit arc.
- **A Logical Node** that does not represent any actual statements can be added as a joining point for several incoming edges.
  - Represents a logical closure.
  - Example:
    - Node 4 in the `if-then-else` example from previous slide.

Example: Minimum Element

```
min = A[0];                    1
I = 1;

while (I < N) {                2
      if (A[I] < min)          3
          min = A[I];          4
      I = I + 1;               5
}
print min                      6
```

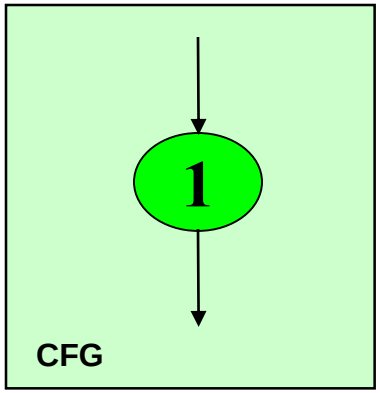


Note: The CFG is **INCOMPLETE**. Try to complete it

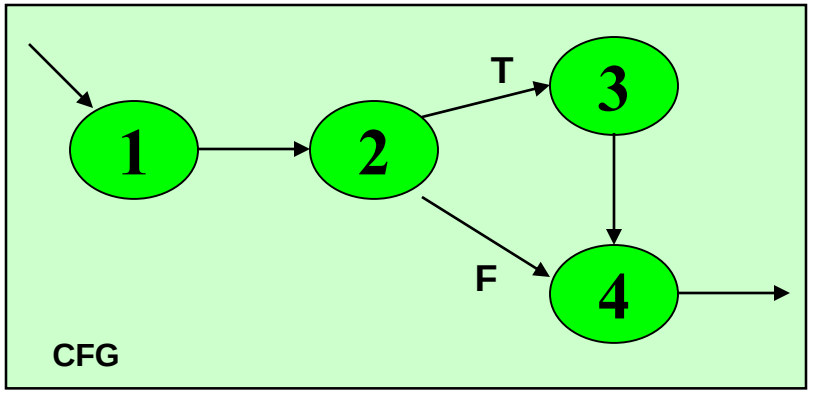
Number of Paths through CFG

- Given a program, how do we exercise all statements and branches at least once?
- Translating the program into a CFG, an equivalent question is:
  - Given a CFG, how do we cover all arcs and nodes at least once?
- Since a path is a trail of nodes linked by arcs, this is similar to ask:
  - Given a CFG, what is the set of paths that can cover all arcs and nodes?

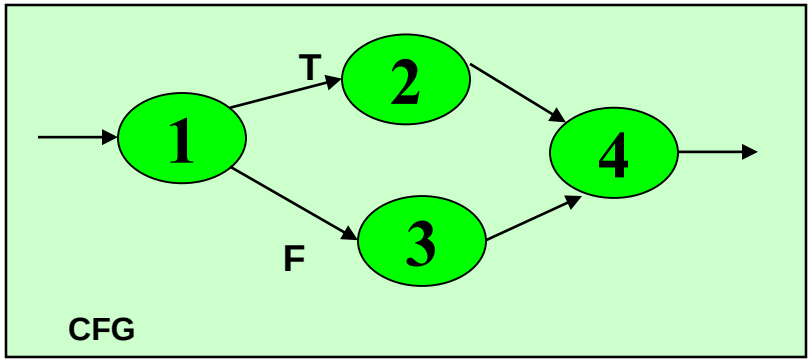
Example



- Only **one** path is needed:
  - **[1]**



- **Two** paths are needed:
  - **[1 – 2 – 4]**
  - **[1 – 2 – 3 – 4]**



- **Two** paths are needed:
  - **[1 – 2 – 4]**
  - **[1 – 3 – 4]**

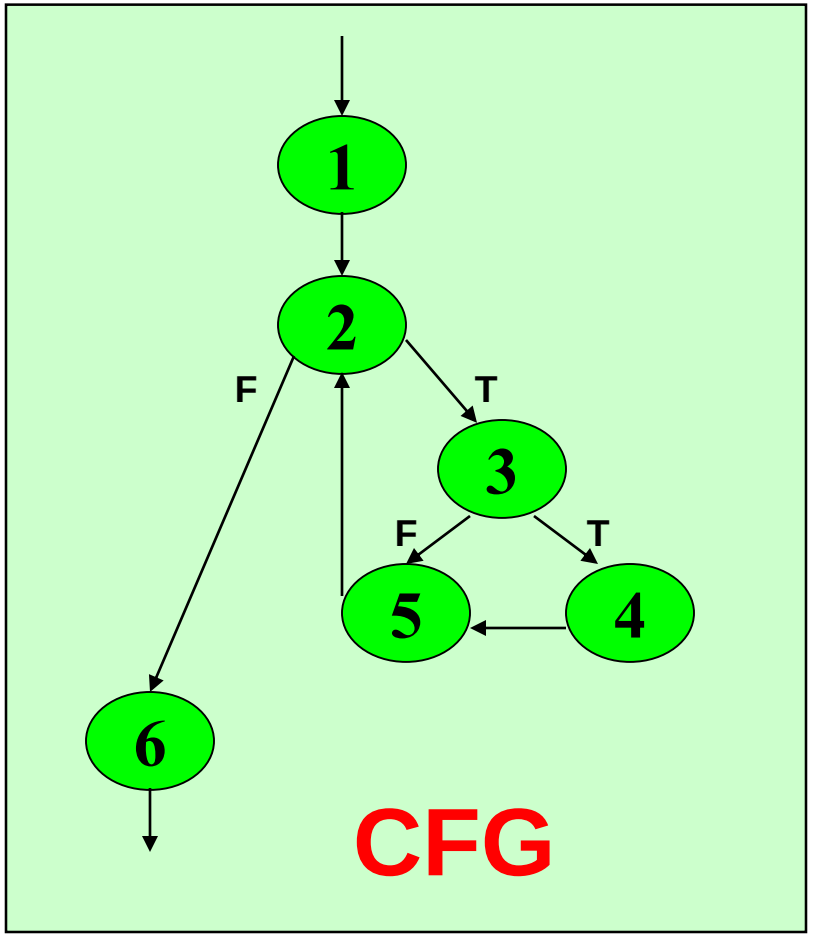
White Box Testing: Path Based

- A generalized technique to find out the number of paths needed (known as *cyclomatic complexity*) to cover all arcs and nodes in CFG.
- Steps:
  1. Draw the CFG for the code fragment.
  2. Compute the *cyclomatic complexity number* ***C***, for the CFG.
  3. Find at most ***C*** paths that cover the nodes and arcs in a CFG, also known as **Basic Paths Set;**
  4. Design test cases to force execution along paths in the **Basic Paths Set.**

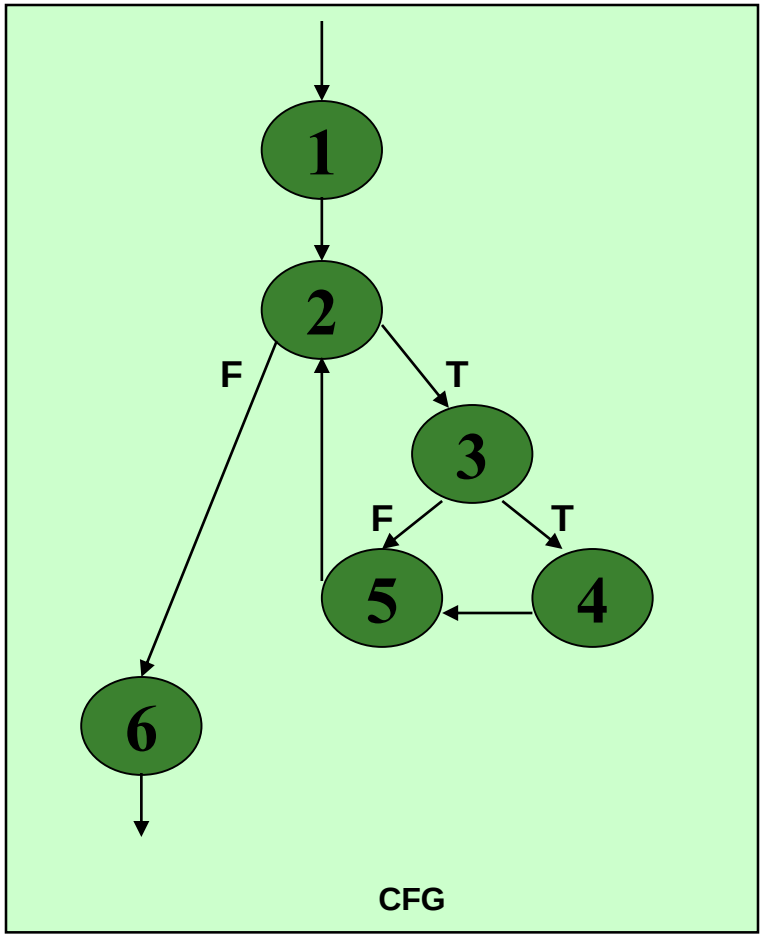
Path Based Testing: Step 1

```
min = A[0];
I = 1;

while (I < N) {
      if (A[I] < min)
         min = A[I];
      I = I + 1;
}
print min
```

1 2 3 4 5 6

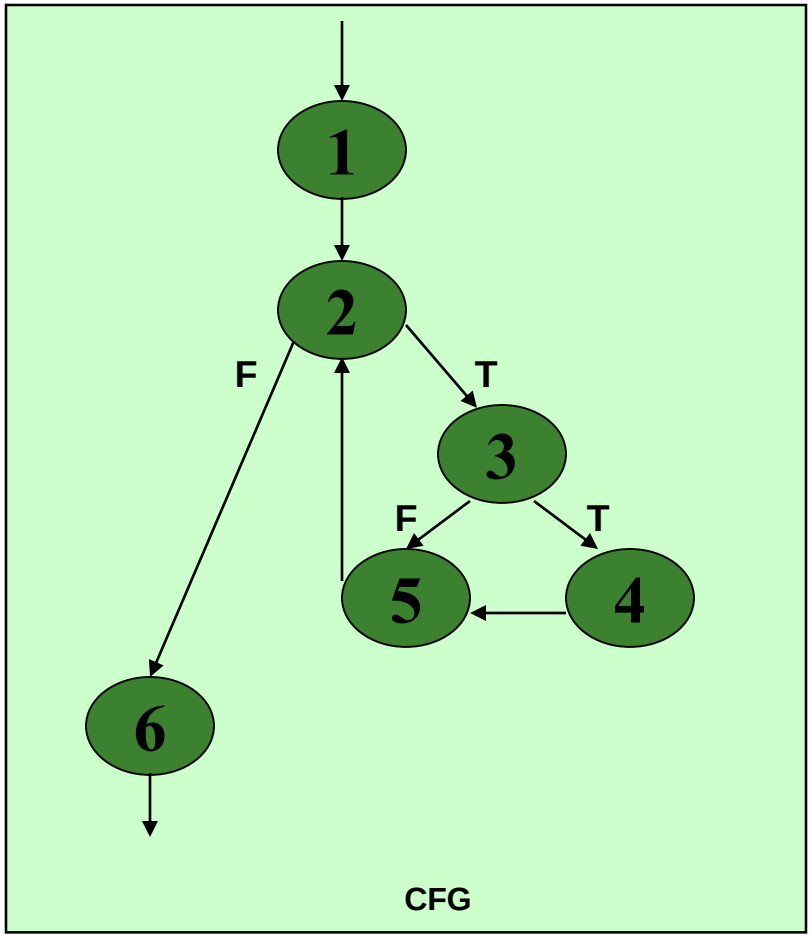


Path Base Testing: Step 2



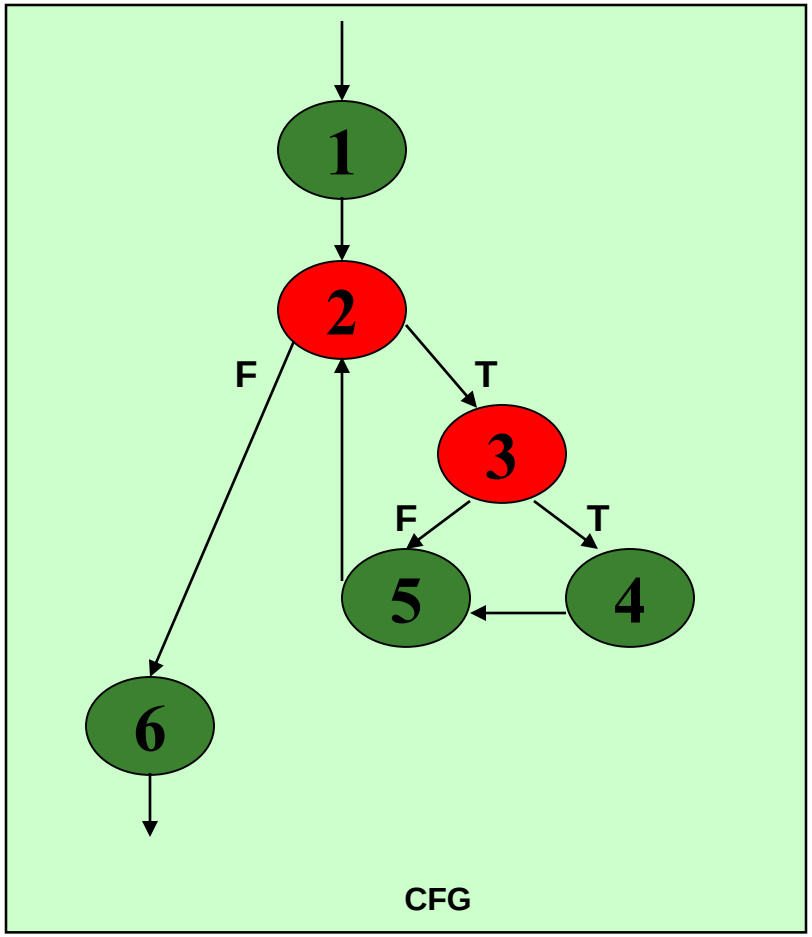
- Cyclomatic complexity =
  - The number of ‘regions’ in the graph; OR
  - The number of predicates + 1.

Path Base Testing: Step 2



- **Region**: Enclosed area in the CFG.
  - Do not forget the outermost region.
- In this example:
  - 3 Regions (see the circles with different colors).
  - Cyclomatic Complexity = 3
- Alternative way in next slide.

Path Base Testing: Step 2



- **Predicates**:
  - Nodes with multiple exit arcs.
  - Corresponds to branch/conditional statement in program.
- In this example:
  - Predicates = 2
    - (Node 2 and 3)
  - Cyclomatic Complexity

    = 2 + 1

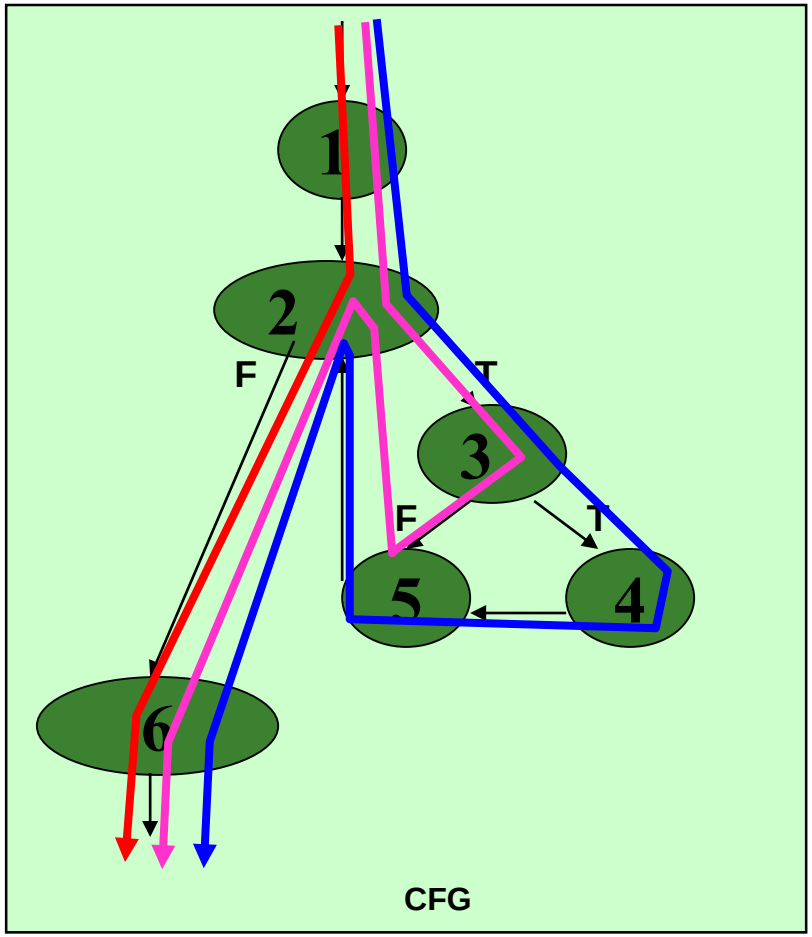
    = 3

Path Base Testing: Step 3

- Independent path:
  - An **executable** or **realizable path** through the graph from the start node to the end node that has not been traversed before.
  - **Must** move along **at least one arc** that has not been yet traversed (an unvisited arc).
  - The objective is to cover all statements in a program by independent paths.
- The number of independent paths to discover <= cyclomatic complexity number.
- Decide the Basis Path Set:
  - It is the maximal set of *independent paths* in the flow graph.
  - **NOT** a unique set.

Path Base Testing: Step 3

```
min = A[0];                    } 1
I = 1;
while (I < N) {                } 2
      if (A[I] < min)          } 3
         min = A[I];           } 4
      I = I + 1;               } 5
}
print min                      } 6
```



- Cyclomatic complexity = 3.
- Need at most **3** independent paths to cover the CFG.
- In this example:
  - [1 – 2 – 6]
  - [1 – 2 – 3 – 5 – 2 – 6]
  - [1 – 2 – 3 – 4 – 5 – 2 – 6]

Path Base Testing: Step 4

- Prepare a test case for each independent path.
- In this example:
  - Path: [1 – 2 – 6]
    - Test Case: A = { 5, …}, N = 1
    - Expected Output: 5
  - Path: [1 – 2 – 3 – 5 – 2 – 6]
    - Test Case: A = { 5, 9, … }, N = 2
    - Expected Output: 5
  - Path: [1 – 2 – 3 – 4 – 5 – 2 – 6]
    - Test Case: A = { 8, 6, … }, N = 2
    - Expected Output: 6
- These tests will result a complete decision and statement coverage of the code.

Try to verify that the test cases actually force execution along a desired path.

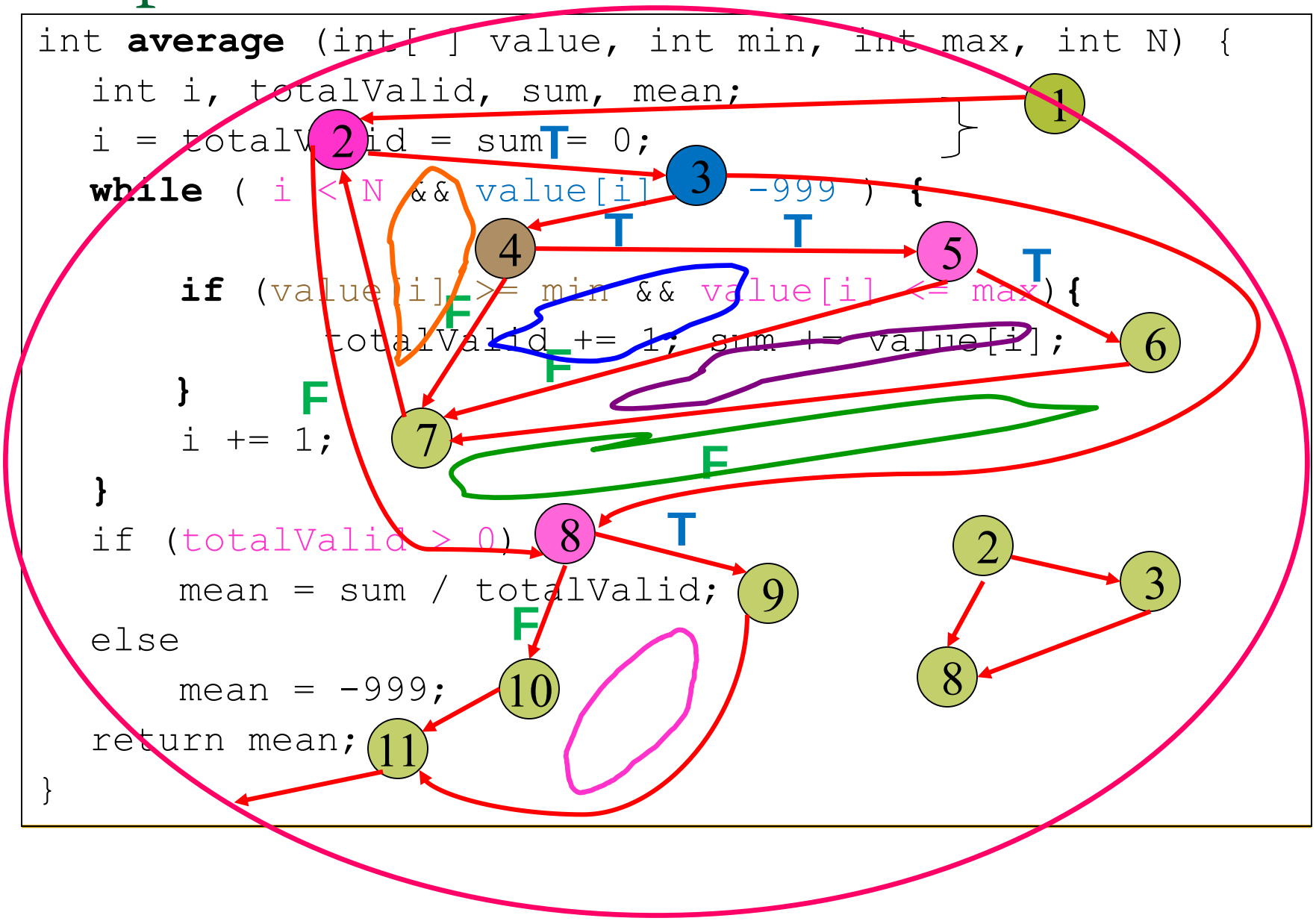
Another Example

```
int average (int[ ] value, int min, int max, int N) {
   int i, totalValid, sum, mean;
   i = totalValid = sum = 0;
   while ( i < N && value[i] != -999 ) {
       if (value[i] >= min && value[i] <= max){
              totalValid += 1; sum += value[i];
       }
       i += 1;
   }
   if (totalValid > 0)
       mean = sum / totalValid;
   else
       mean = -999;
   return mean;
}
```

Step 1: Draw CFG

```
int average (int[ ] value, int min, int max, int N) {
   int i, totalValid, sum, mean;
   i = totalValid = sum = 0;                          } 1
   while ( i < N && value[i] != -999 ) {
                2          4          3          5
        if (value[i] >= min && value[i] <= max){
                totalValid += 1; sum += value[i];   6
        }
        i += 1;    7
   }
   if (totalValid > 0)  8
        mean = sum / totalValid;  9
   else
        mean = -999;    10
   return mean;  11
}
```

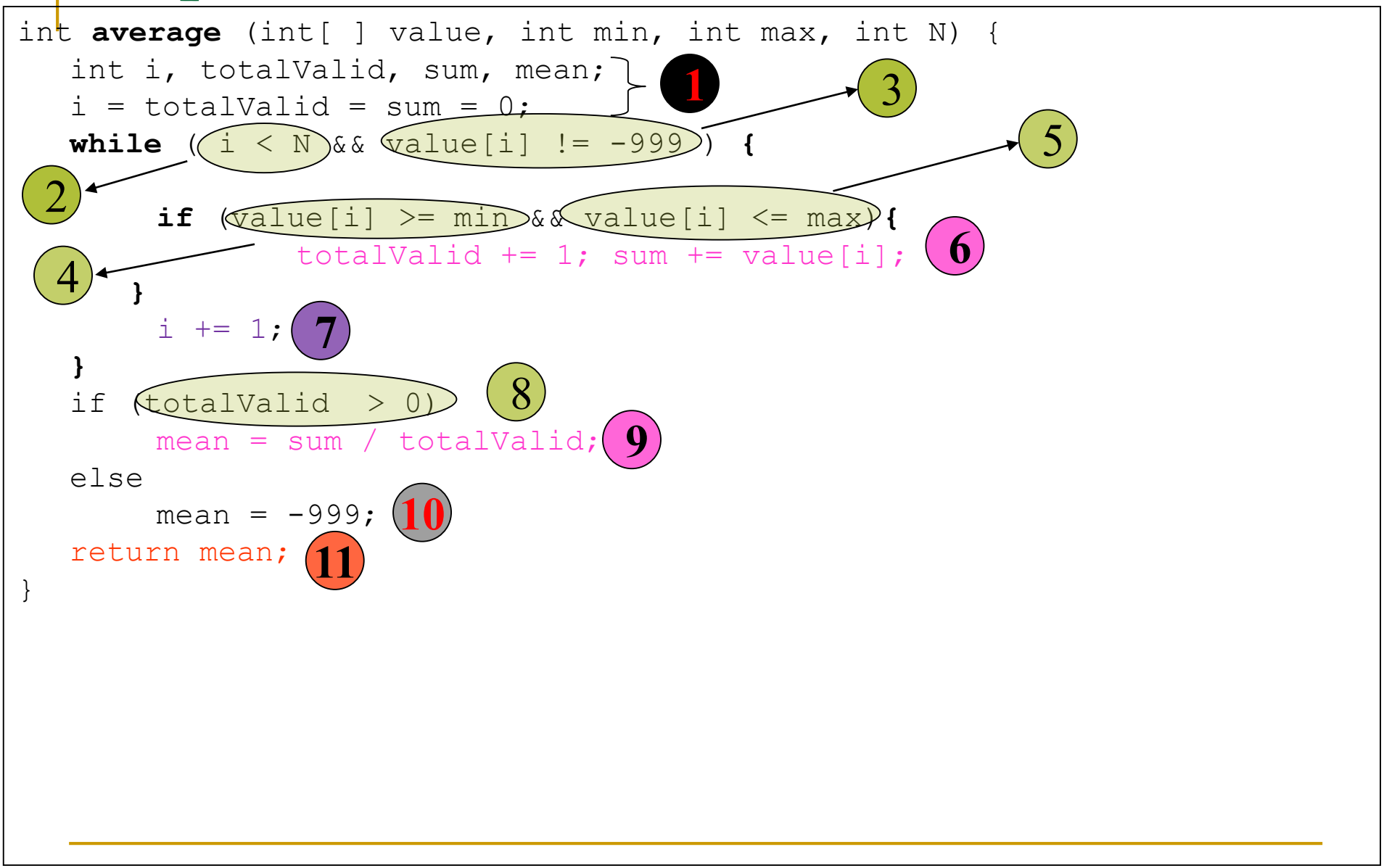
Step 1: Draw CFG

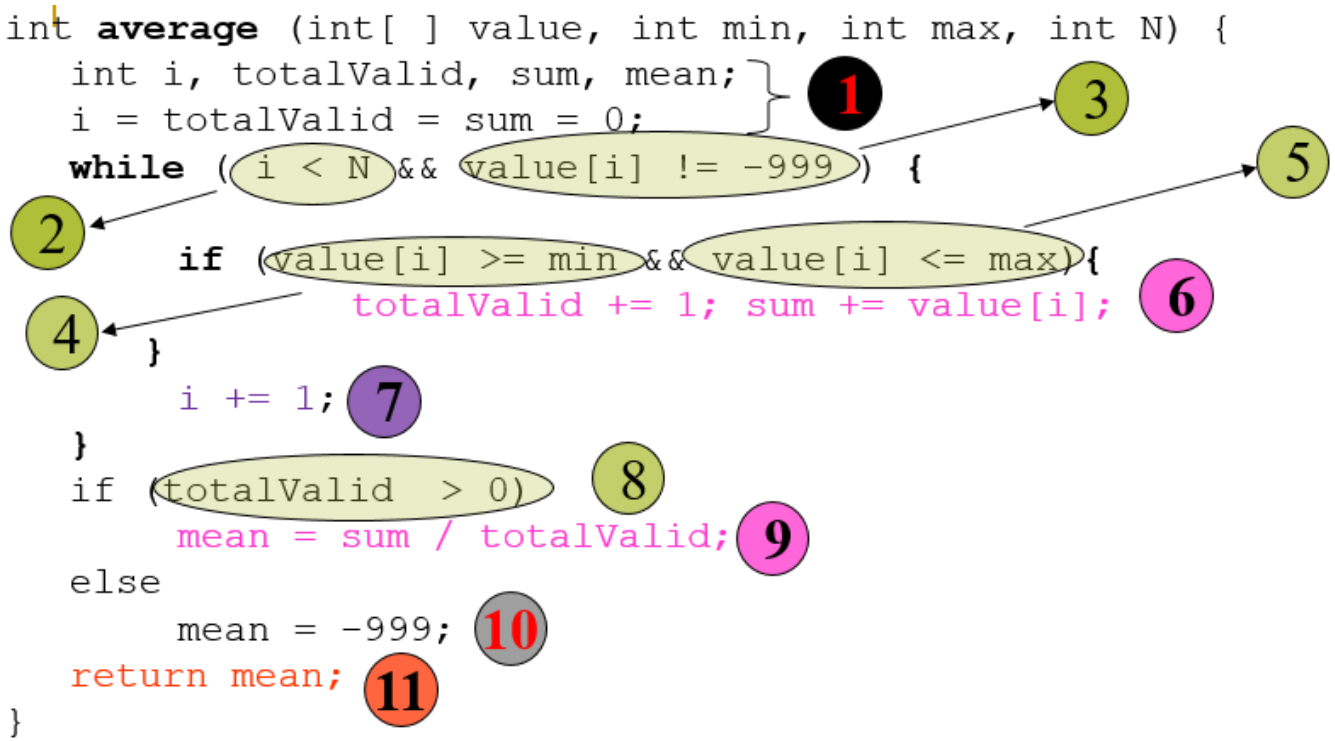


Step 1: Draw CFG

```
int average (int[ ] value, int min, int max, int N) {
   int i, totalValid, sum, mean;
   i = totalValid = sum = 0;
   while ( i < N && value[i] != -999 ) {

        if (value[i] >= min && value[i] <= max){
                totalValid += 1; sum += value[i];
        }
        i += 1;
   }
   if (totalValid  > 0)
        mean = sum / totalValid;
   else
        mean = -999;
   return mean;
}
```

Step 1: Draw CFG

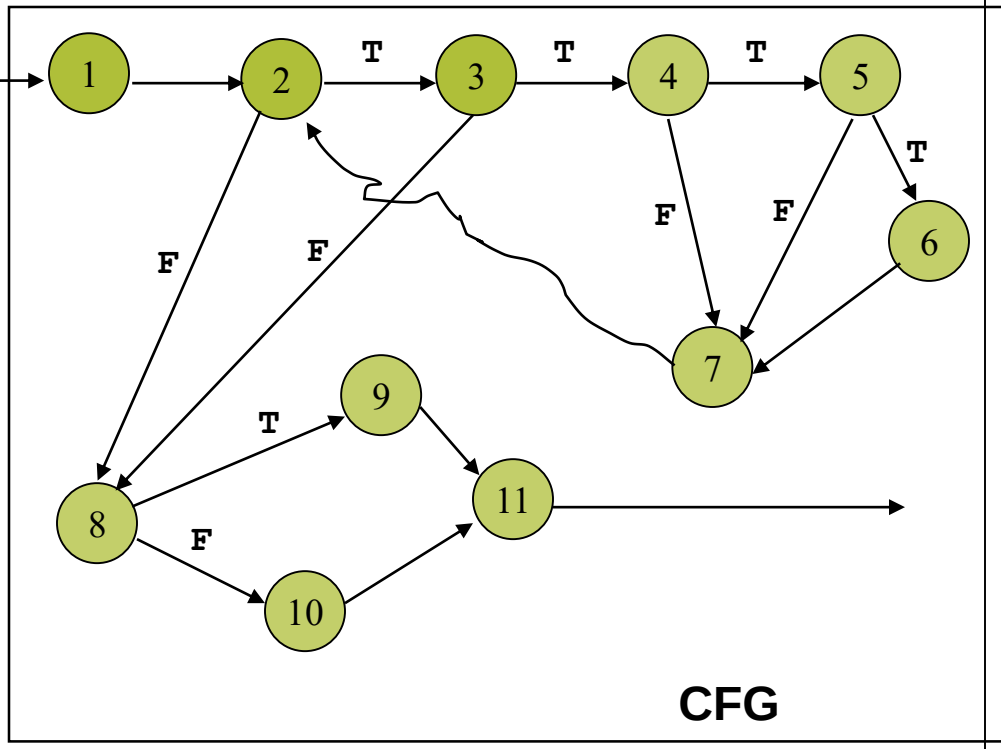



int average (int[ ] value, int min, int max, int N) {
int i, totalValid, sum, mean;
i = totalValid = sum = 0;
while (i < N && value[i] != -999) {
if (value[i] >= min && value[i] <= max){
totalValid += 1; sum += value[i];
}
i += 1;
}
if (totalValid > 0)
mean = sum / totalValid;
else
mean = -999;
return mean;
}
1
2
3
4
5
6
7
8
9
10
11

Step 1: Draw CFG

```
int average (int[ ] value, int min, int max, int N) {
    int i, totalValid, sum, mean;
    i = totalValid = sum = 0;
    while ( i < N && value[i] != -999 ) {
        if (value[i] >= min && value[i] <= max){
            totalValid += 1; sum += value[i];
        }
        i += 1;
    }
    if (totalValid > 0)
        mean = sum / totalValid;
    else
        mean = -999;
    return mean;
}
```

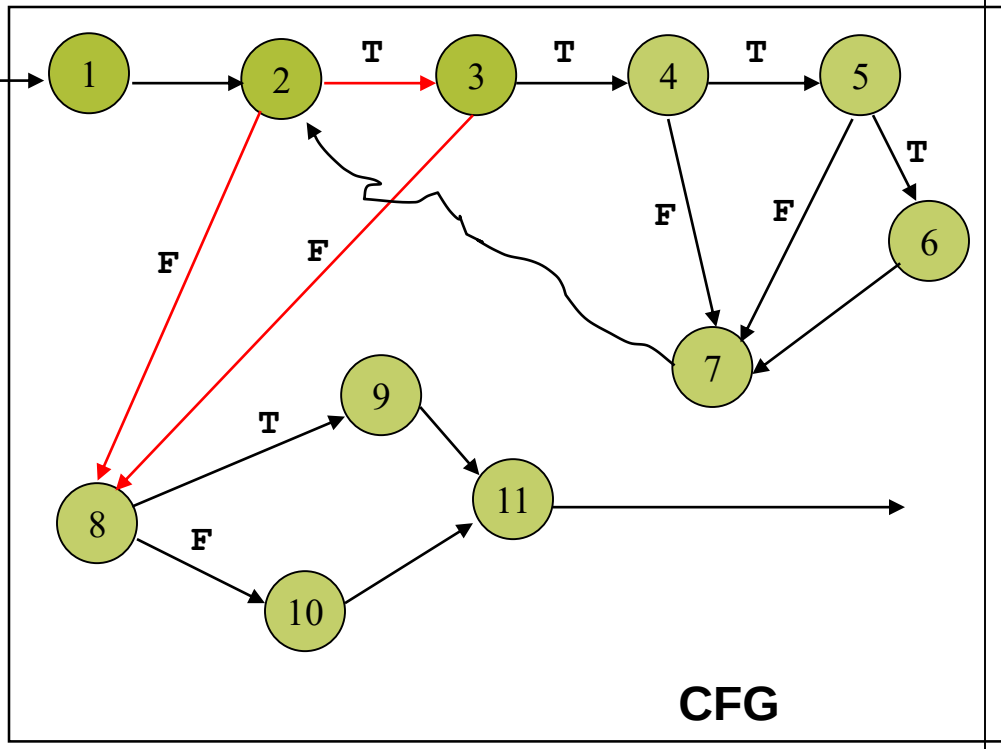
1 2 3 4 5 6 7 8 9 10 11



Step 1: Draw CFG

```
int average (int[ ] value, int min, int max, int N) {
    int i, totalValid, sum, mean;
    i = totalValid = sum = 0;
    while ( i < N && value[i] != -999 ) {
        if ( value[i] >= min && value[i] <= max){
            totalValid += 1; sum += value[i];
        }
        i += 1;
    }
    if ( totalValid > 0)
        mean = sum / totalValid;
    else
        mean = -999;
    return mean;
}
```

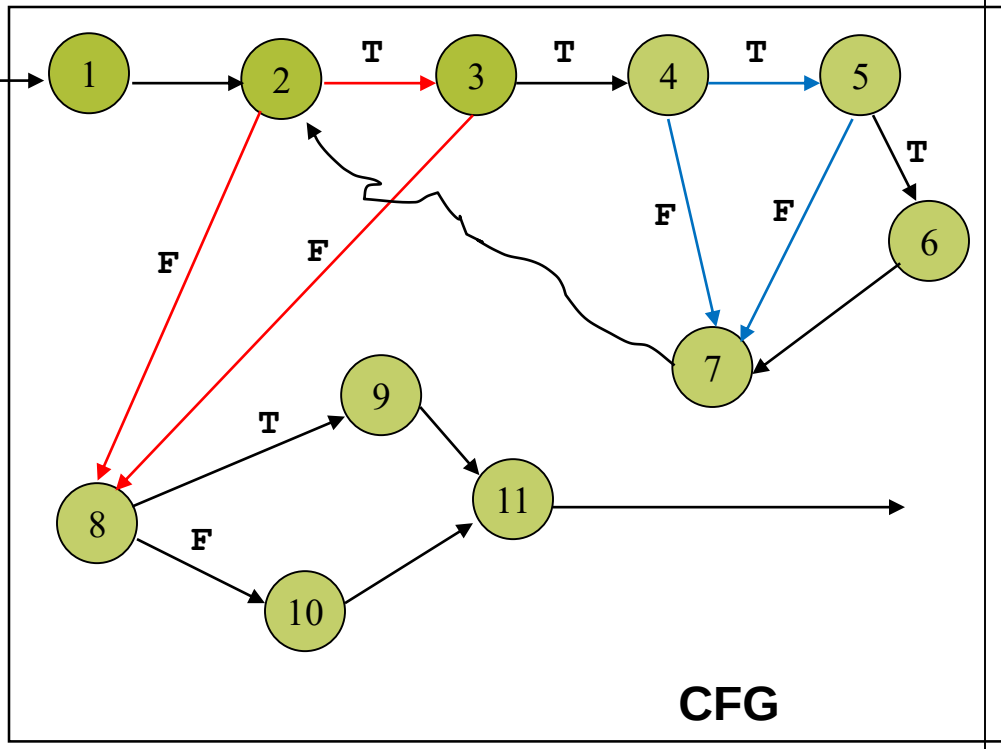
1 2 3 4 5 6 7 8 9 10 11



Step 1: Draw CFG

```
int average (int[ ] value, int min, int max, int N) {
    int i, totalValid, sum, mean;
    i = totalValid = sum = 0;
    while ( i < N && value[i] != -999 ) {
        if ( value[i] >= min && value[i] <= max){
            totalValid += 1; sum += value[i];
        }
        i += 1;
    }
    if ( totalValid > 0)
        mean = sum / totalValid;
    else
        mean = -999;
    return mean;
}
```

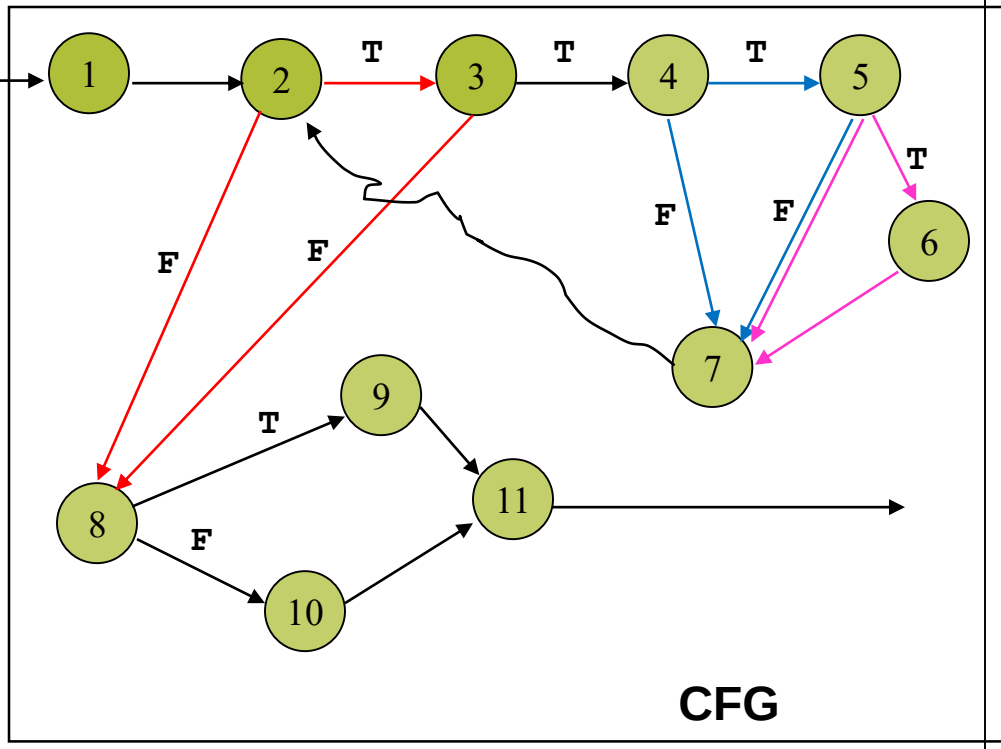
1 2 3 4 5 6 7 8 9 10 11



Step 1: Draw CFG

```
int average (int[ ] value, int min, int max, int N) {
    int i, totalValid, sum, mean;
    i = totalValid = sum = 0;
    while ( i < N && value[i] != -999 ) {
        if ( value[i] >= min && value[i] <= max){
            totalValid += 1; sum += value[i];
        }
        i += 1;
    }
    if ( totalValid > 0)
        mean = sum / totalValid;
    else
        mean = -999;
    return mean;
}
```

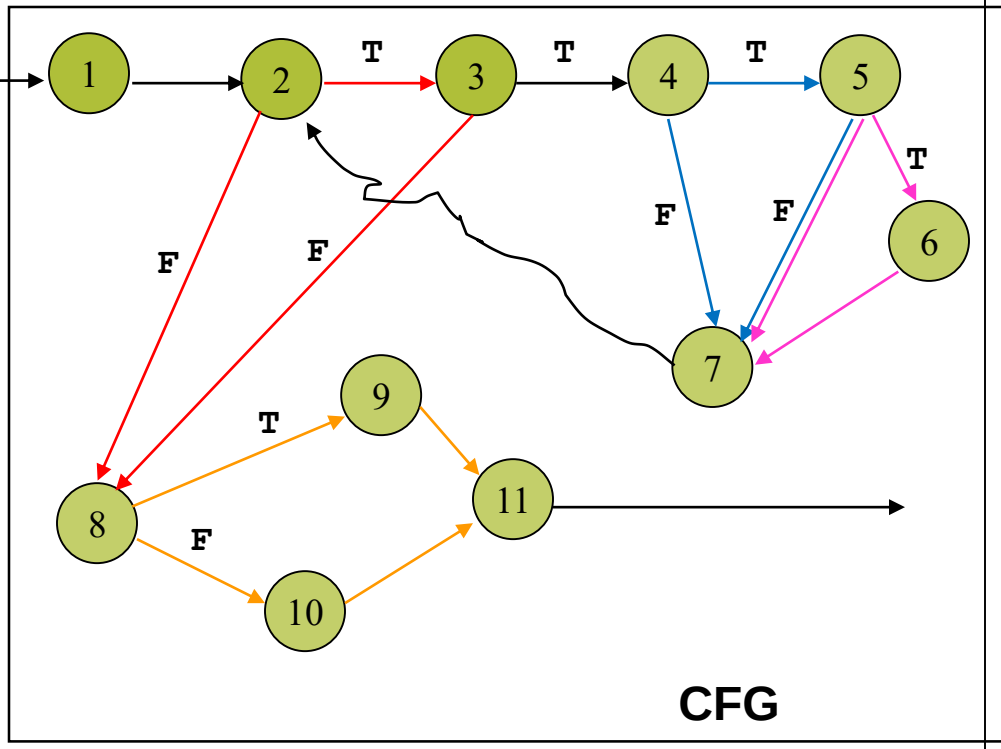
1 2 3 4 5 6 7 8 9 10 11



Step 1: Draw CFG

```
int average (int[ ] value, int min, int max, int N) {
    int i, totalValid, sum, mean;
    i = totalValid = sum = 0;
    while ( i < N && value[i] != -999 ) {
        if ( value[i] >= min && value[i] <= max){
            totalValid += 1; sum += value[i];
        }
        i += 1;
    }
    if ( totalValid > 0)
        mean = sum / totalValid;
    else
        mean = -999;
    return mean;
}
```

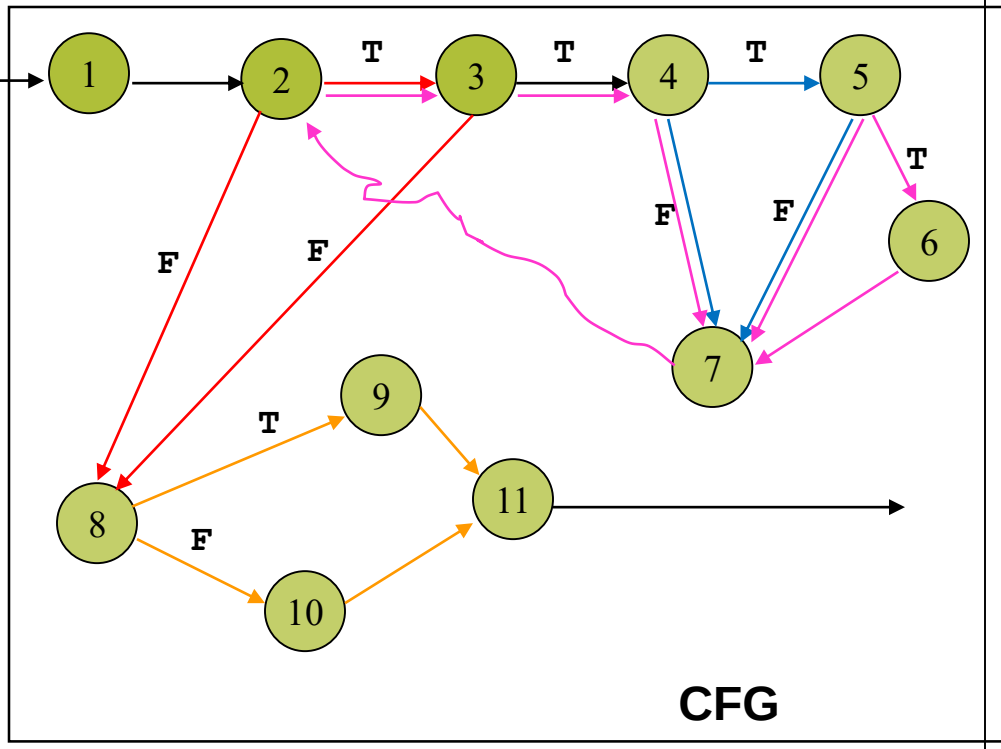
1 2 3 4 5 6 7 8 9 10 11



Step 1: Draw CFG

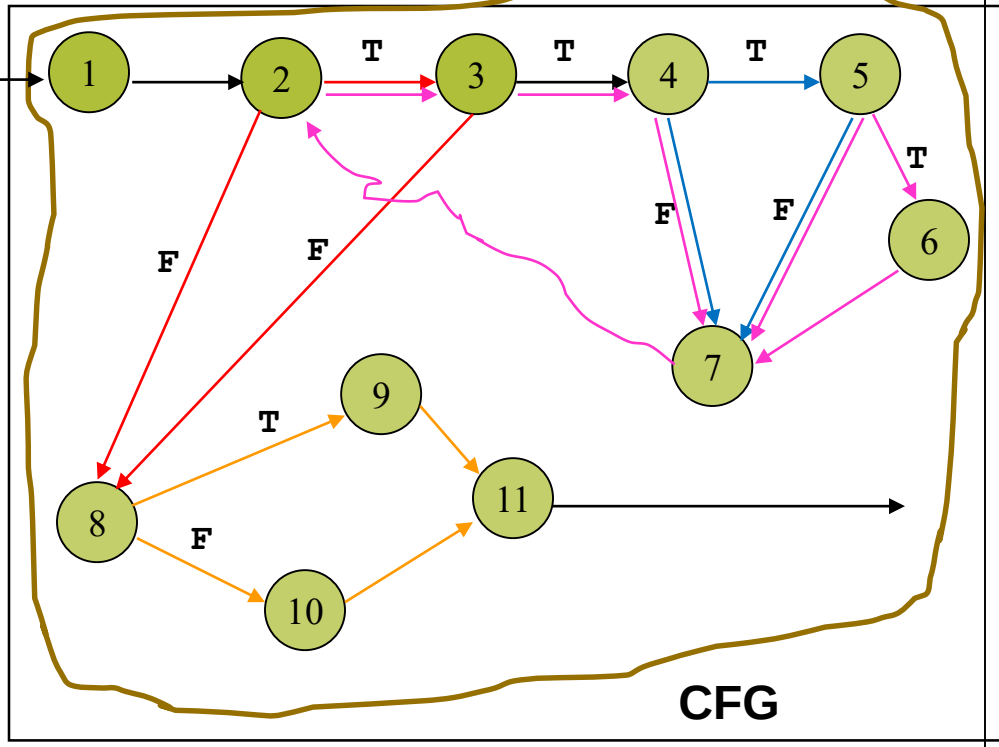
```
int average (int[ ] value, int min, int max, int N) {
    int i, totalValid, sum, mean;
    i = totalValid = sum = 0;
    while ( i < N && value[i] != -999 ) {
        if ( value[i] >= min && value[i] <= max ){
            totalValid += 1; sum += value[i];
        }
        i += 1;
    }
    if ( totalValid > 0)
        mean = sum / totalValid;
    else
        mean = -999;
    return mean;
}
```

1 2 3 4 5 6 7 8 9 10 11

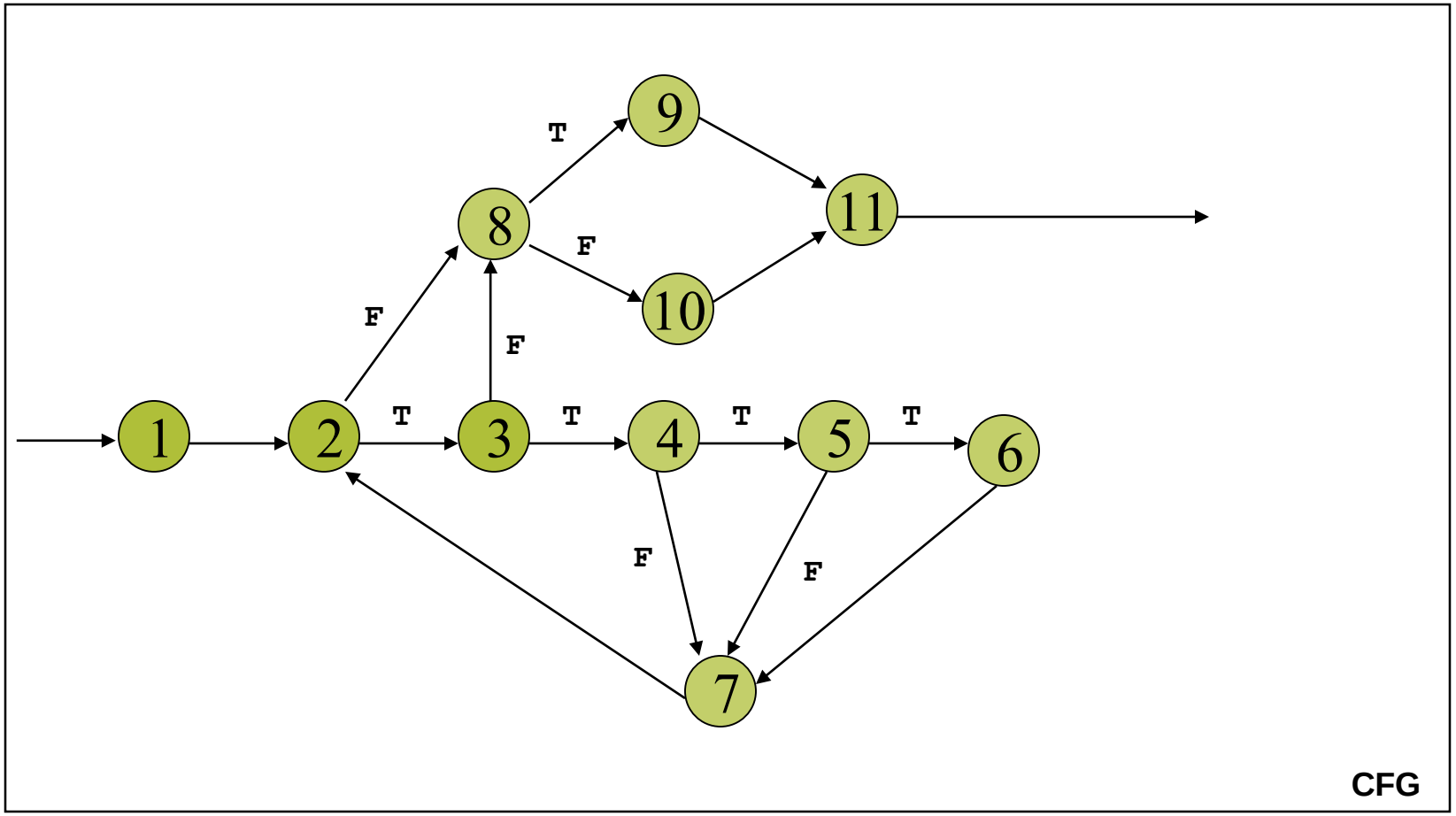


Step 1: Draw CFG

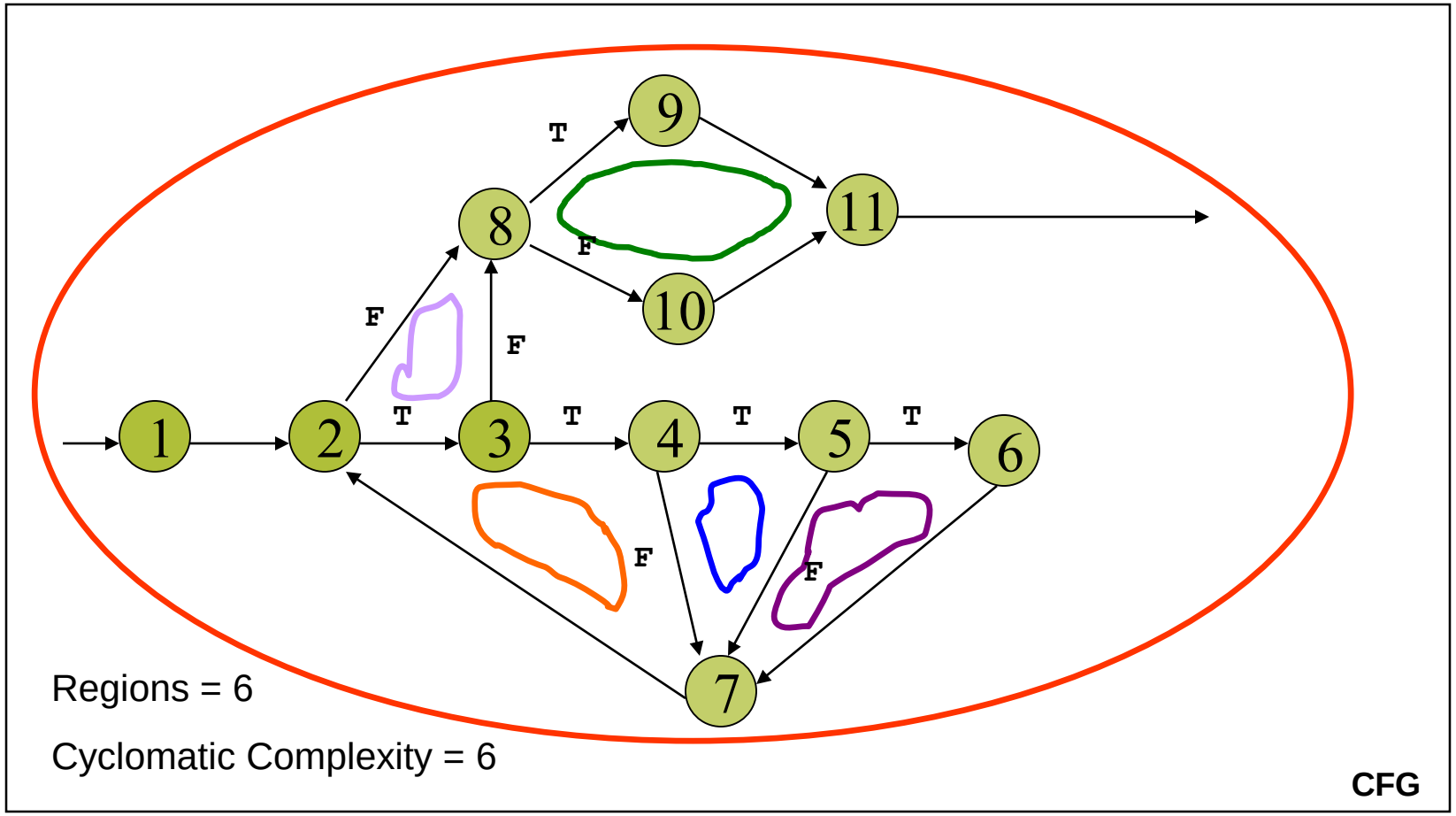
```
int average (int[ ] value, int min, int max, int N) {
    int i, totalValid, sum, mean;
    i = totalValid = sum = 0;
    while ( i < N && value[i] != -999 ) {
        if (value[i] >= min && value[i] <= max){
            totalValid += 1; sum += value[i];
        }
        i += 1;
    }
    if (totalValid > 0)
        mean = sum / totalValid;
    else
        mean = -999;
    return mean;
}
```



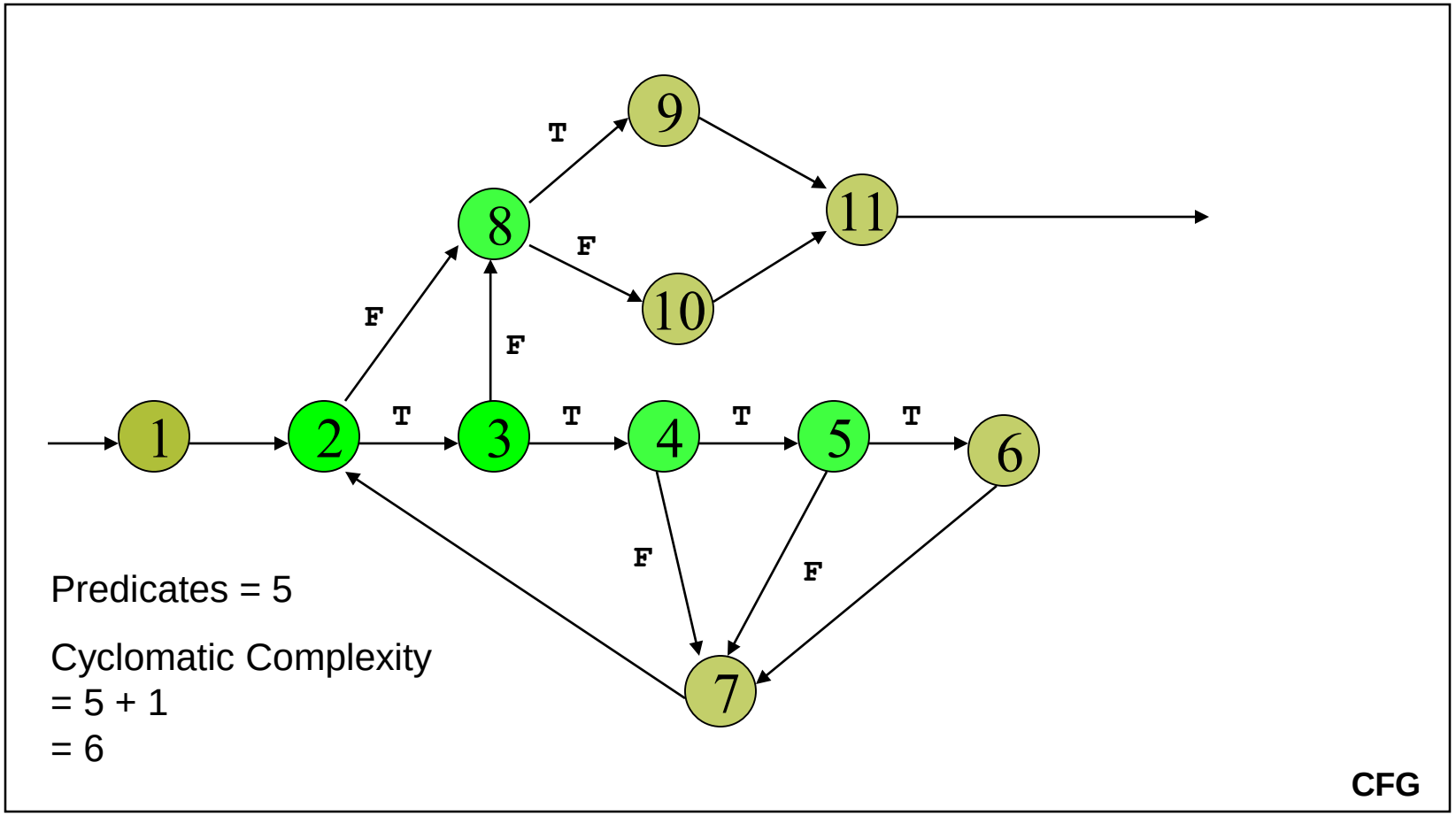
Step 1: Draw CFG



Step 2: Find Cyclomatic Complexity



Step 2: Find Cyclomatic Complexity



Step 3: Find Basic Path Set

- Find at most 6 independent paths.
- Usually, simpler path == easier to find a test case.
- However, some of the simpler paths are not possible (not realizable):
  - Example: [1 – 2 – 8 – 9 – 11].
    - Not Realizable (i.e., impossible in execution).
    - Verify this by tracing the code.
- Basic Path Set:
  - [1 – 2 – 8 – 10 – 11].
  - [1 – 2 – 3 – 8 – 10 – 11].
  - [1 – 2 – 3 – 4 – 7 – 2 – 8 – 10 – 11].
  - [1 – 2 – 3 – 4 – 5 – 7 – 2 – 8 – 10 – 11].
  - [1 – (2 – 3 – 4 – 5 – 6 – 7) – 2 – 8 – 9 – 11].
- In the last case, (…) represents possible repetition.

Step 4: Derive Test Cases

- Path:
  - [1 – 2 – 8 – 10 – 11]
- Test Case:
  - `value = {…}` irrelevant.
  - `N = 0`
  - `min, max` irrelevant.
- Expected Output:
  - `average = -999`

```
... i = 0;                    1

while (i < N &&               2
       value[i] != -999) {
     ......
}
if (totalValid > 0)           8
     ......
else
     mean = -999;             10

return mean;                  11
```

Step 4: Derive Test Cases

- Path:
  - [1 – 2 – 3 – 8 – 10 – 11]
- Test Case:
  - `value = {-999}`
  - `N = 1`
  - `min, max` irrelevant
- Expected Output:
  - `average = -999`

```
... i = 0;                    1

while (i < N &&               2
       value[i] != -999)      3
     ......
}
if (totalValid > 0)           8
     ......
else                          10
     mean = -999;
                              11
return mean;
```

Step 4: Derive Test Cases

- Path:
  - [1 – 2 – 3 – 4 – 7 – 2 – 8 – 10 – 11]
- Test Case:
  - A single value in the `value[ ]` array which is smaller than *min.*
  - `value = { 25 }, N = 1, min = 30, max` irrelevant.
- Expected Output:
  - `average = -999`

- Path:
  - [1 – 2 – 3 – 4 – 5 – 7 – 2 – 8 – 10 – 11]
- Test Case:
  - A single value in the `value[ ]` array which is larger than *max.*
  - `value = { 99 }, N = 1, max = 90, min` irrelevant.
- Expected Output:
  - `average = -999`

Step 4: Derive Test Cases

- Path:
  - [1 – 2 – 3 – 4 – 5 – 6 – 7 – 2 – 8 – 9 – 11]
- Test Case:
  - A single valid value in the `value[ ]` array.
  - `value = { 25 }, N = 1, min = 0, max = 100`
- Expected Output:
  - `average = 25`

OR

- Path:
  - [1 – 2 – 3 – 4 – 5 – 6 – 7 – 2 – 3 – 4 – 5 – 6 – 7 – 2 – 8 – 9 – 11]
- Test Case:
  - Multiple valid values in the `value[ ]` array.
  - `value = { 25, 75 }, N = 2, min = 0, max = 100`
- Expected Output:
  - `average = 50`

Summary: Path Base White Box Testing

- A simple test that:
  - Cover all statements.
  - Exercise all decisions (conditions).
- The cyclomatic complexity is an **upperbound** of the independent paths needed to cover the CFG.
  - If more paths are needed, then either cyclomatic complexity is wrong, or the paths chosen are incorrect.
- Although picking a complicated path that covers more than one unvisited edge is possible all times, it is not encouraged:
  - May be hard to design the test case.

Summary

- Test Case Design
  - White Box
    - Control Flow Graph
    - Cyclomatic Complexity
    - Basic Path Testing
  - Black Box
    - Equivalence Classes
    - Boundary Value Analysis