Software Testing Techniques

(Source: Pressman, R. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 2005)

Overview

Test Case Design

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

Characteristics of Testable Software

Operable

- The better it works (i.e., better quality), the easier it is to test

Observable

- Incorrect output is easily identified; internal errors are automatically detected

Controllable

- The states and variables of the software can be controlled directly by the tester

Decomposable

- The software is built from independent modules that can be tested independently

Characteristics of Testable Software (continued)

Simple

- The program should exhibit functional, structural, and code simplicity

Stable

- Changes to the software during testing are infrequent and do not invalidate existing tests

Understandable

- The architectural design is well understood; documentation is available and organized

Test Characteristics

A good test has a high probability of finding an error

- The tester must understand the software and how it might fail

A good test is not redundant

- Testing time is limited; one test should not serve the same purpose as another test

A good test should be “best of breed”

- Tests that have the highest likelihood of uncovering a whole class of errors should be used

A good test should be neither too simple nor too complex

- Each test should be executed separately; combining a series of tests could cause side effects and mask certain errors

Two Unit Testing Techniques

Black-box testing

- Knowing the specified function that a product has been designed to perform, test to see if that function is fully operational and error free
- Includes tests that are conducted at the software interface
- Not concerned with internal logical structure of the software

White-box testing

- Knowing the internal workings of a product, test that all internal operations are performed according to specifications and all internal components have been exercised
- Involves tests that concentrate on close examination of procedural detail
- Logical paths through the software are tested
- Test cases exercise specific sets of conditions and loops

White-box Testing

White-box Testing

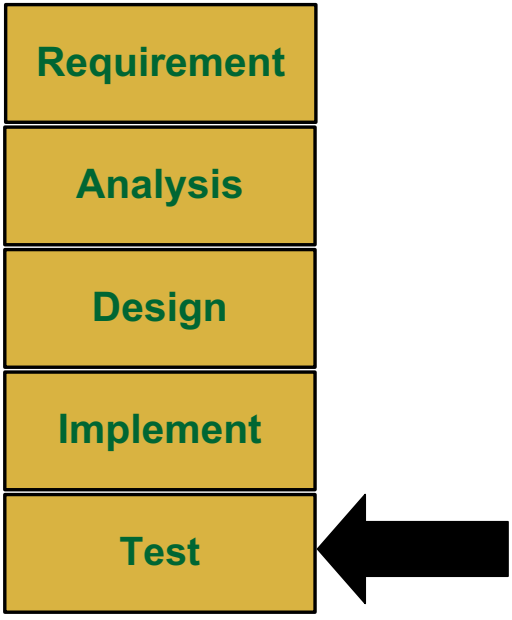
Uses the control structure part of component-level design to derive the test cases

These test cases

- Guarantee that all independent paths within a module have been exercised at least once
- Exercise all logical decisions on their true and false sides
- Execute all loops at their boundaries and within their operational bounds
- Exercise internal data structures to ensure their validity

"Bugs lurk in corners and congregate at boundaries"

Where are we now?



Evaluating the System

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.

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.

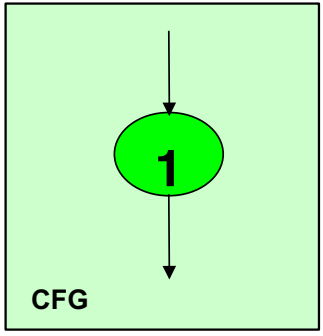
Path:

- A collection of *Nodes* linked with *Directed Edges.*

Simple Examples

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

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



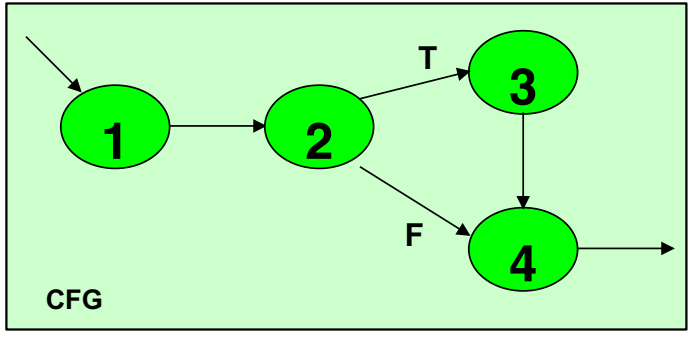

```
Statement1;
Statement2;

if X < 10 then
    Statement3;

Statement4;
```

1

2

3

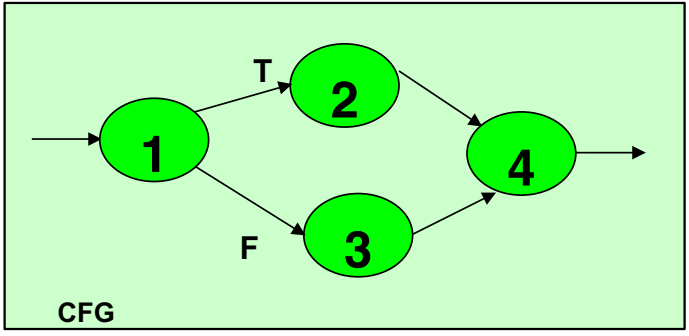
4



More Examples

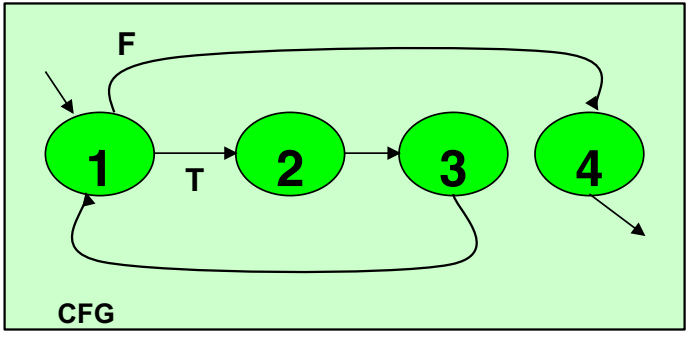
```
if X > 0 then
    Statement1;
else
    Statement2;
```

1
2
3




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

1
2
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.

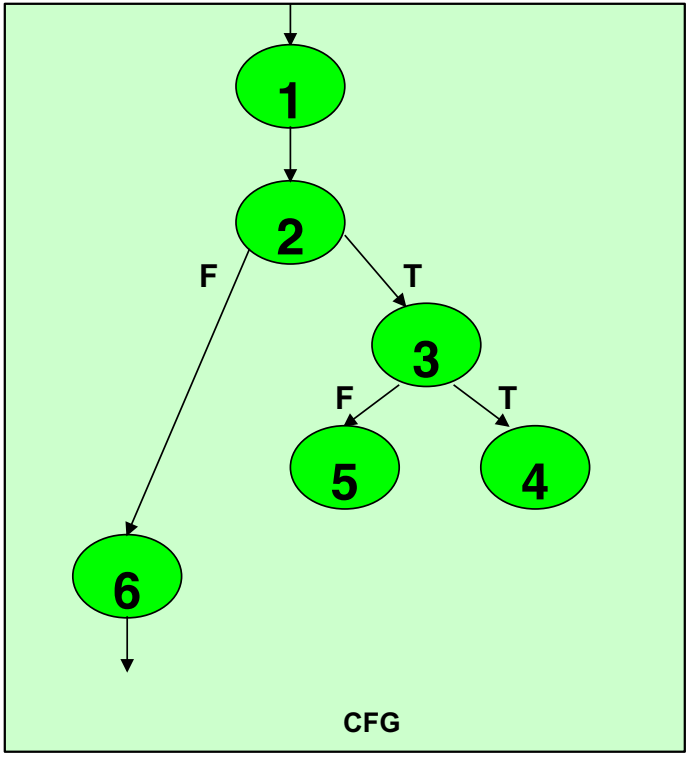
- Represents a logical closure.
- Example:

  Node 4 in the `if-then-else` example from previous slide.

Example: Minimum Element

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

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

1
2
3
4
5
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?

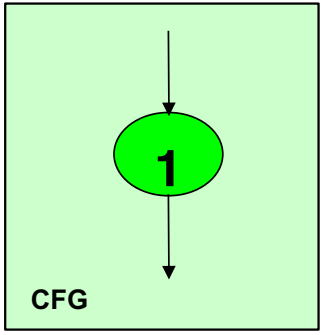
Translating the program into a CFG, an equivalent question is:

- 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:

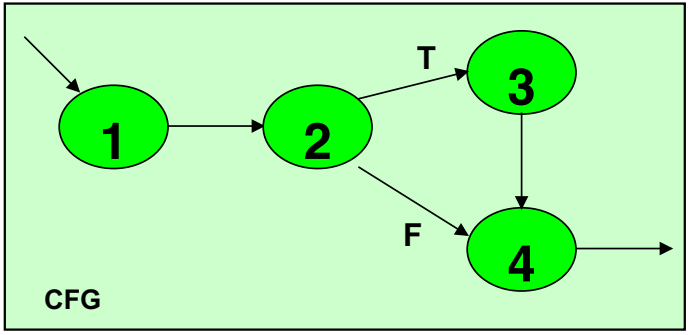
- Given a CFG, what is the set of paths that can cover all arcs and nodes?

Example



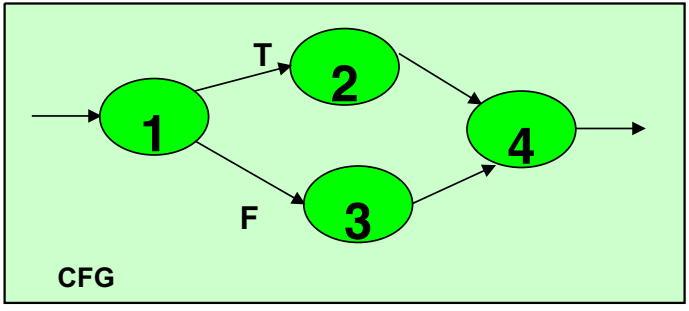
Only **one** path is needed:

- **[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

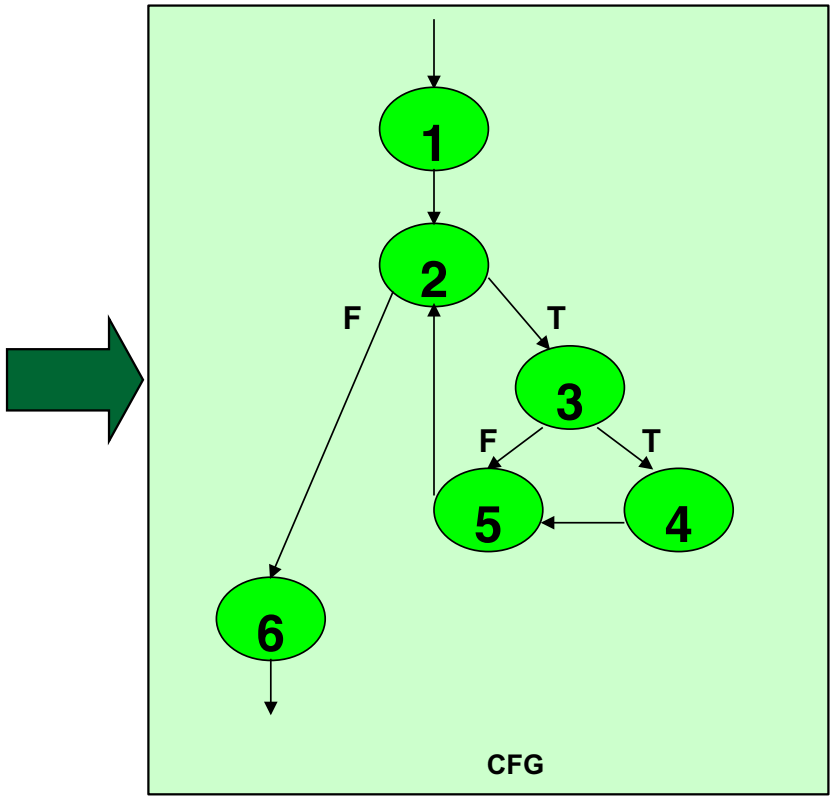
A generalized technique to find out the number of paths needed (known as *cyclomatic complexity*) to cover all arcs and nodes in CFG.

Steps:

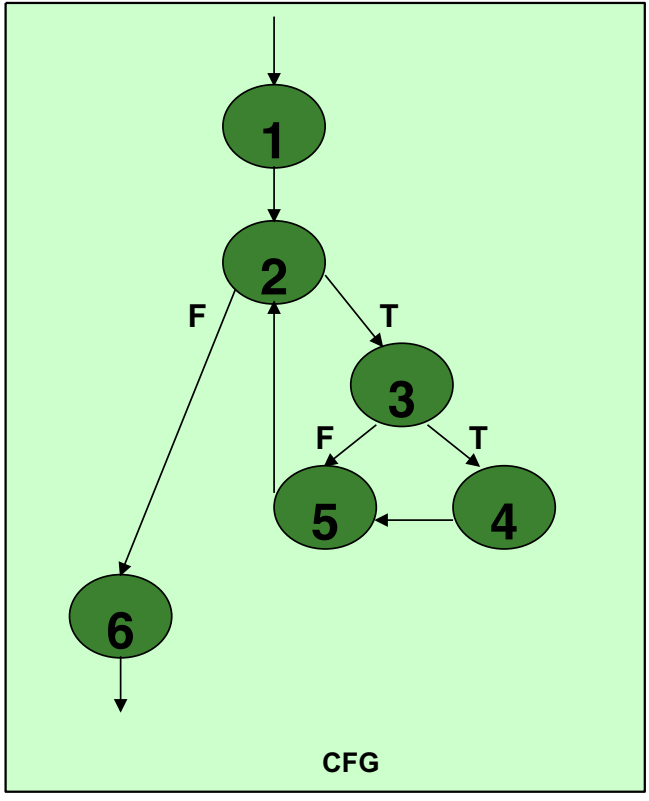
1. Draw the CFG for the code fragment.
2. Compute the *cyclomatic complexity number* ***C***, for the CFG.
3. Find at most ***C*** paths that cover the nodes and arcs in a CFG, also known as **Basic Paths Set;**
4. Design test cases to force execution along paths in the **Basic Paths Set.**

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
```



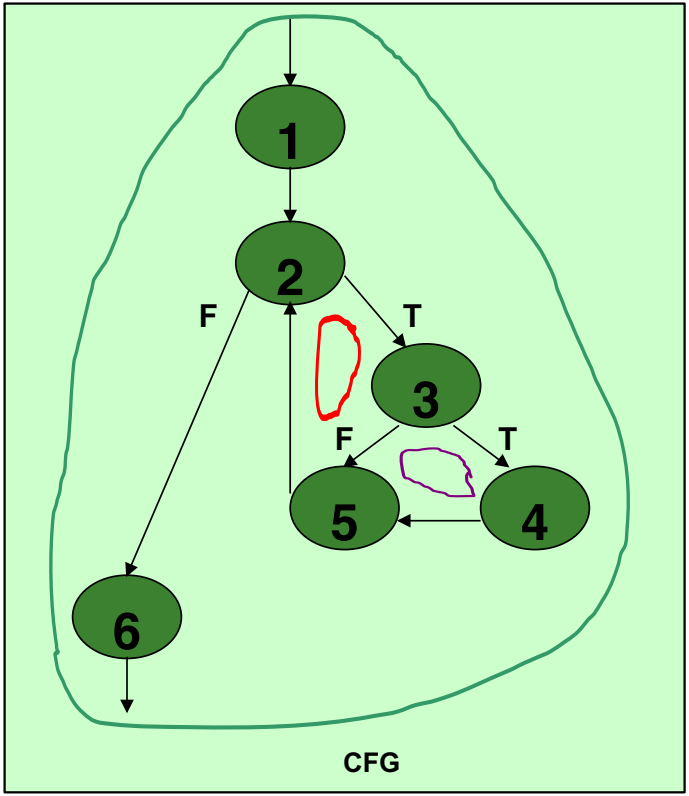
Path Base Testing: Step 2



Cyclomatic complexity =

- The number of 'regions' in the graph; OR
- The number of predicates + 1.
- Number of edges-Number of nodes+2 (E-N+2)

Path Base Testing: Step 2



Region: Enclosed area in the CFG.

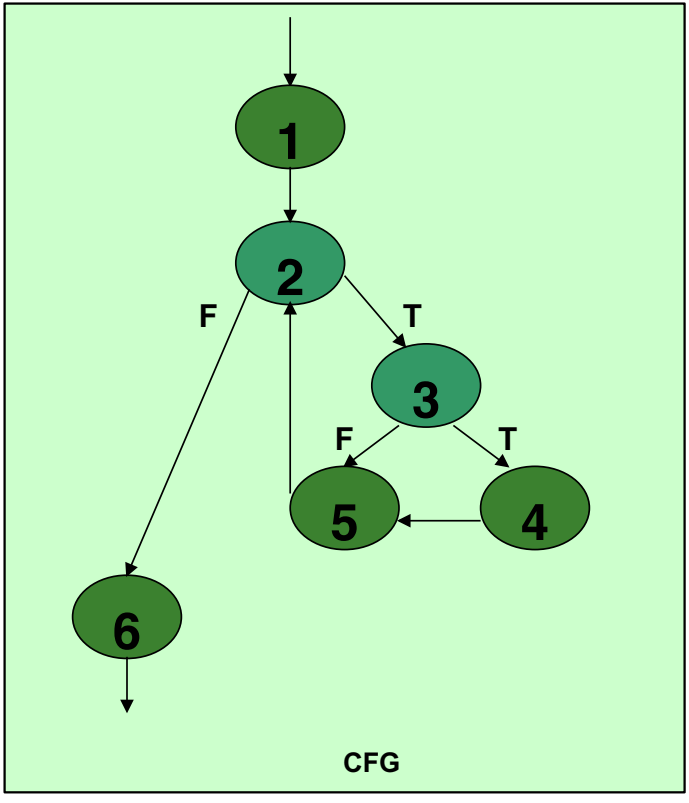
- Do not forget the outermost region.

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:

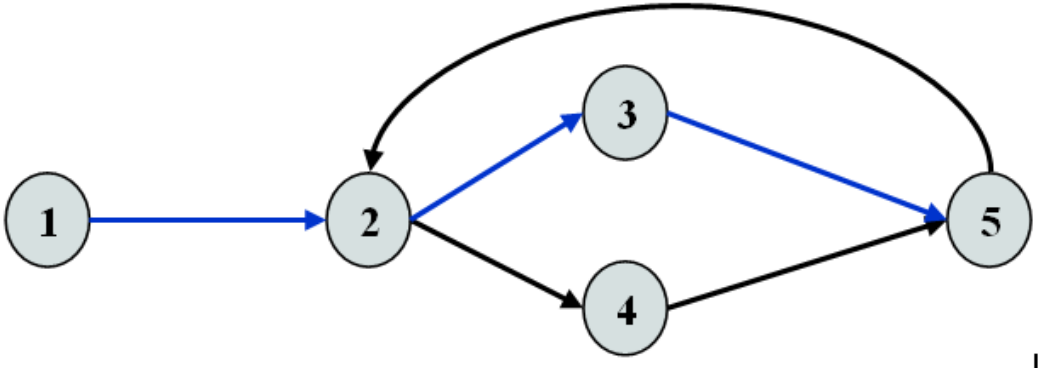
- An **executable** or **realizable path** through the graph from the start node to the end node that has not been traversed before.
- **Must** move along **at least one arc** that has not been yet traversed (an unvisited arc).
- The objective is to cover all statements in a program by independent paths.

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.

Example



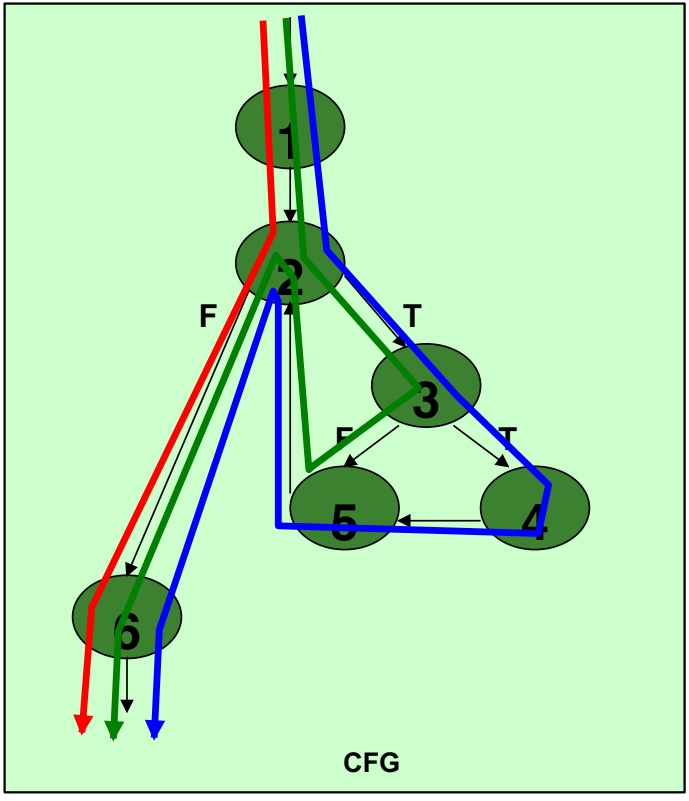
1-2-3-5 can be the first independent path; 1-2-4-5 is another; 1-2-3-5-2-4-5 is one more.

There are only these 3 independent paths. The basis path set is then having 3 paths.

Alternatively, if we had identified 1-2-3-5-2-4-5 as the first independent path, there would be no more independent paths.

The number of independent paths therefore can vary according to the order we identify them.

Path Base Testing: Step 3



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.

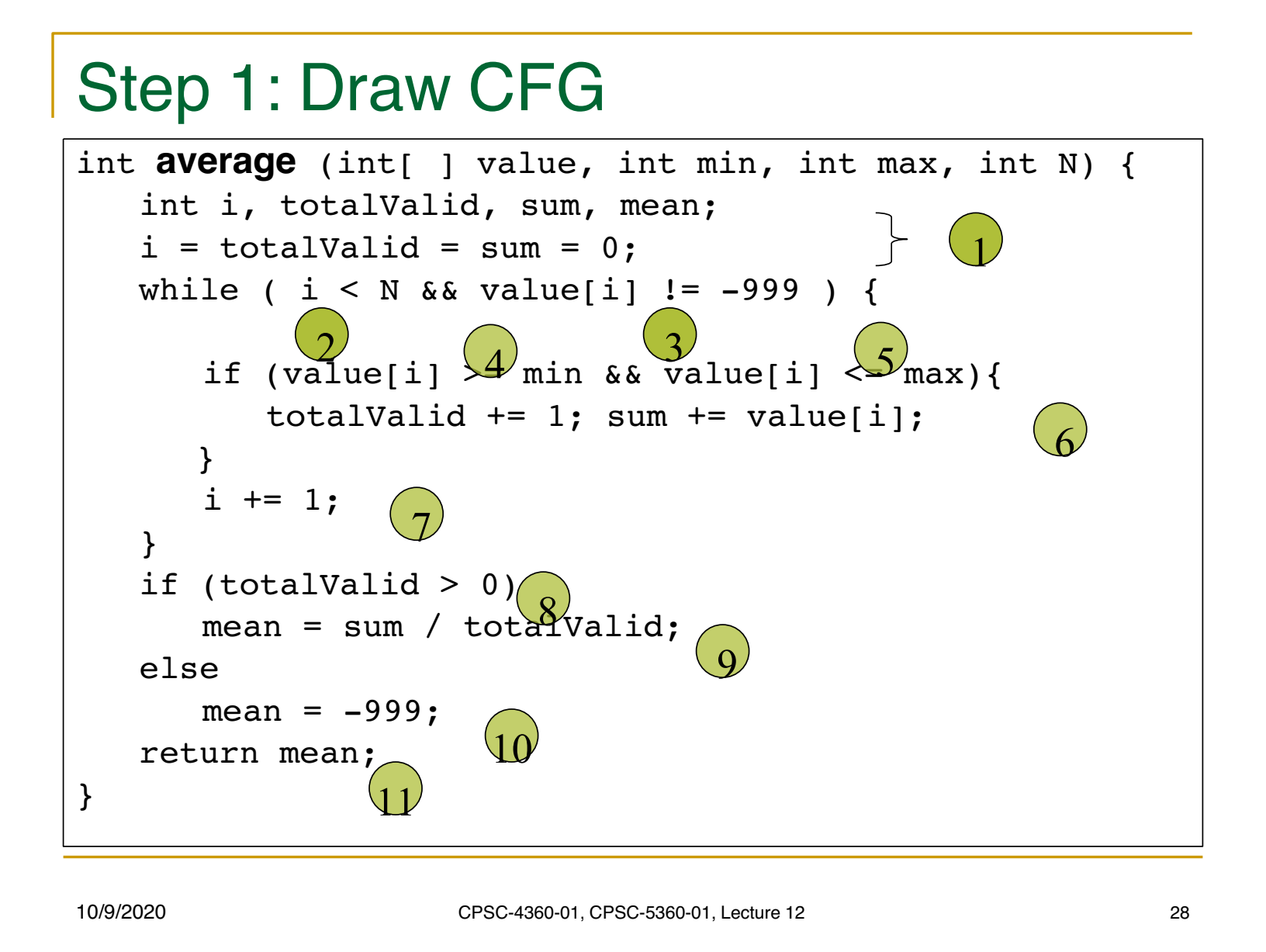
Try to verify that the test cases actually force execution along a desired path.

Another Example

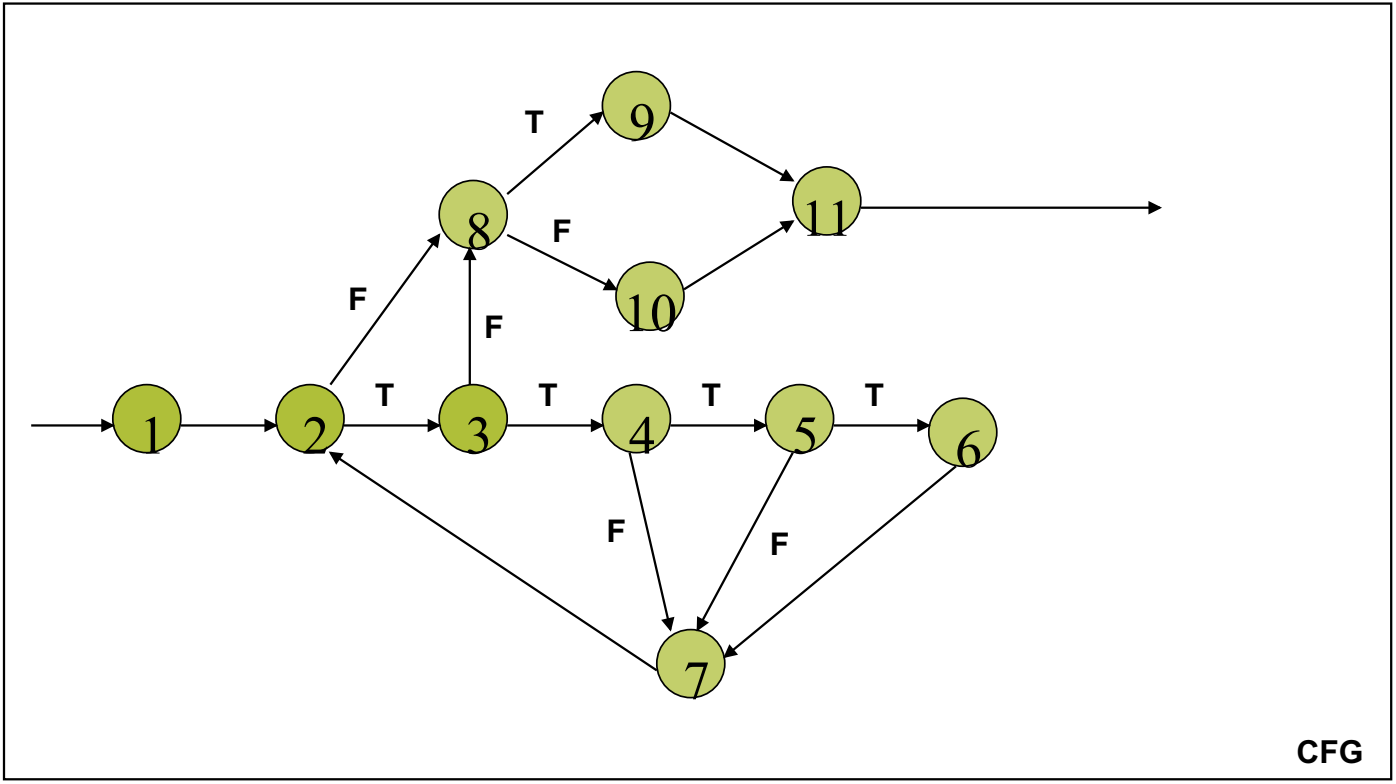
```
int average (int[ ] value, int min, int max, int N) {
    int i, totalValid, sum, mean;
    i = totalValid = sum = 0;
    while ( i < N && value[i] != -999 ) {
        if (value[i] >= min && value[i] <= max){
            totalValid += 1; sum += value[i];
        }
        i += 1;
    }
    if (totalValid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

Step 1: Draw CFG

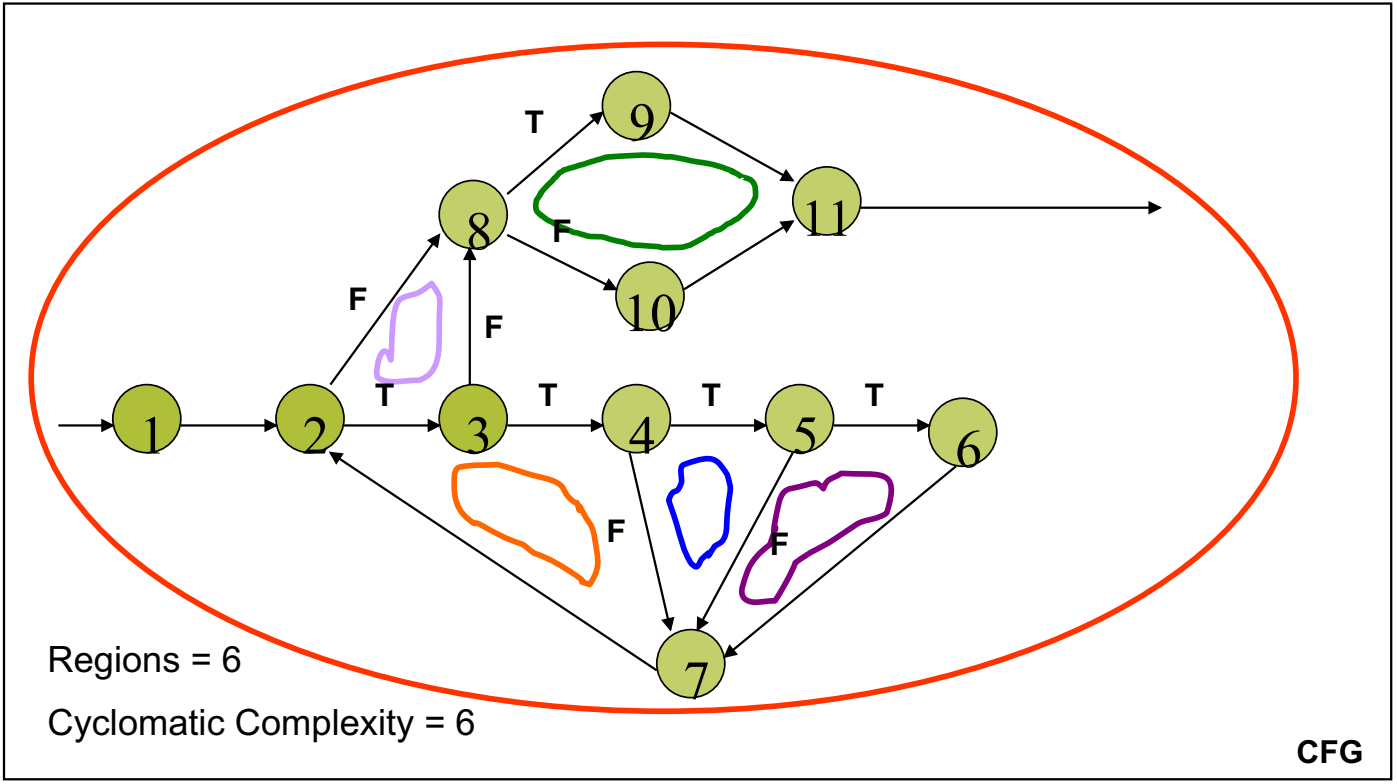
```
int average (int[ ] value, int min, int max, int N) {
    int i, totalValid, sum, mean;
    i = totalValid = sum = 0;
    while ( i < N && value[i] != -999 ) {
        if (value[i] >= min && value[i] <= max){
            totalValid += 1; sum += value[i];
        }
        i += 1;
    }
    if (totalValid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



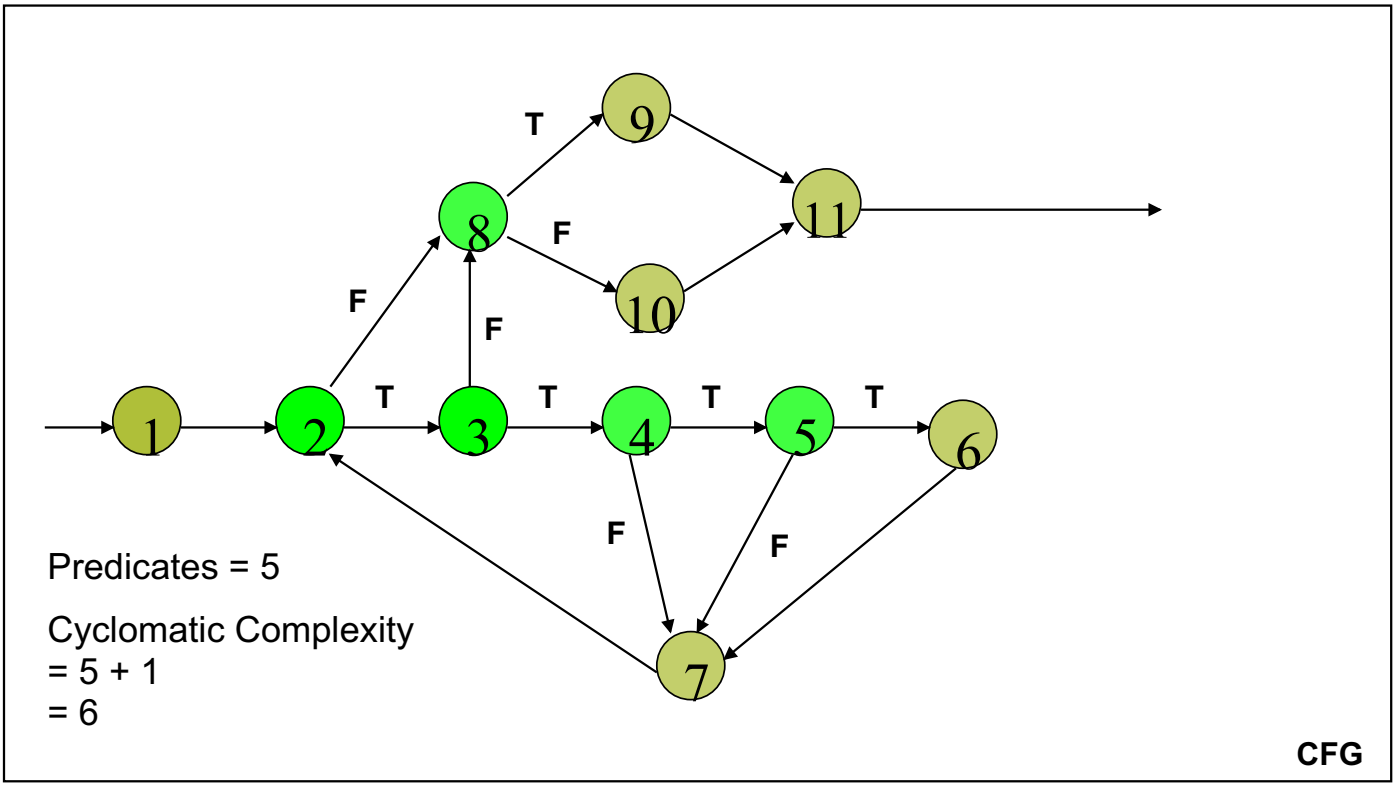
Step 1: Draw CFG



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:

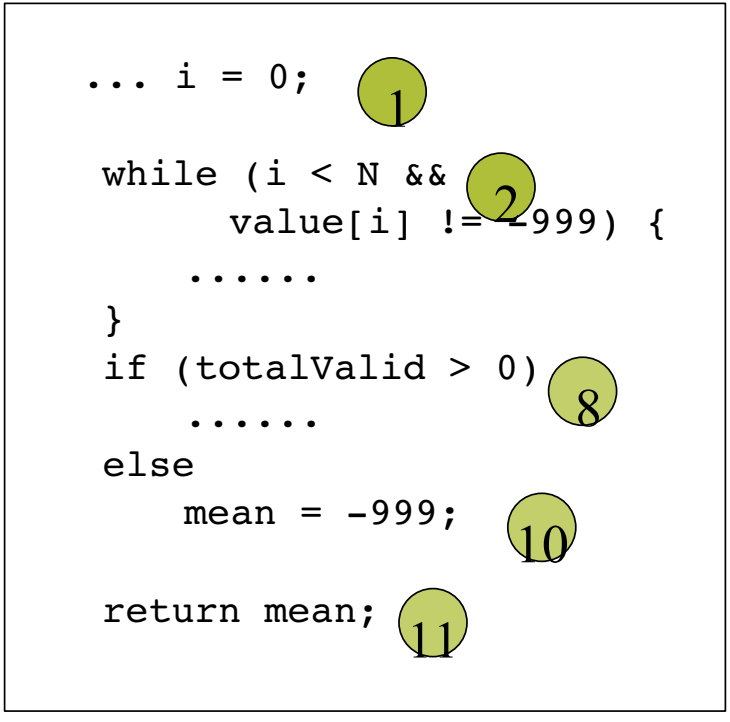
- [1 – 2 – 8 – 10 – 11]

Test Case:

- `value = {…}` irrelevant.
- `N = 0`
- `min, max` irrelevant.

Expected Output:

- `average = -999`



Step 4: Derive Test Cases

Path:

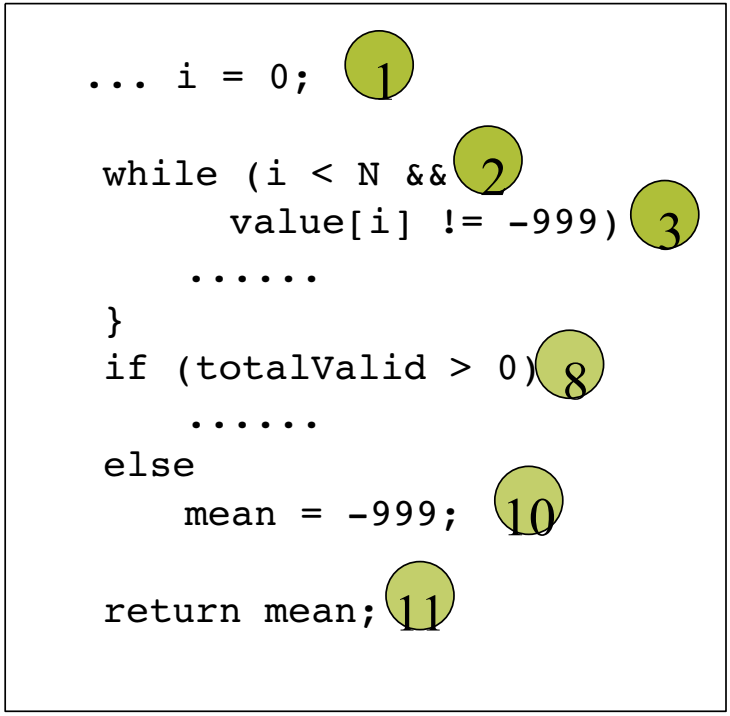
- [1 – 2 – 3 – 8 – 10 – 11]

Test Case:

- `value = {-999}`
- `N = 1`
- `min, max` irrelevant

Expected Output:

- `average = -999`



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.

Graph Matrices

To develop a software tool assists in basis path testing, a data structure, called a graph matrix.

Square Matrix: tabular representation of flow graph

Each node represents by number and each edge represents by letter

Add link weight to each entry which provides additional information about control flow

Link weight is 1 (a connection exists)

Link weight is 0 (a connection does not exists)

Graph Matrices

Link weights can be assigned other

- Probability of edge will be executed
- Processing time during traversal a link
- Memory required during traversal a link
- Resources required during traversal a link

Control Structure Testing : to improve quality of white box testing

Condition Testing

- Simple condition
- Compound condition

Data Flow Testing

- Selects test paths of a program according to variables

Loop Testing

- Simple Loops, Nested Loops, Concatenated Loops and Unstructured Loops

Black Box Testing

Black Box Testing: Introduction

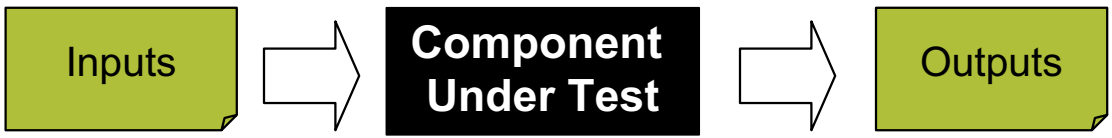
Test Engineers have no access to the source code or documentation of internal working.

The “Black Box” can be:

- A single unit.
- A subsystem.
- The whole system.

Tests are based on:

- Specification of the “Black Box”.
- Providing **inputs** to the “Black Box” and inspect the **outputs.**



Test Case Design

Two techniques will be covered for the black box testing in this course:

- Equivalence Partition;
- Boundary Value Analysis.

Equivalence Partition: Introduction

To ensure the correct behavior of a “black box”, both valid and invalid cases need to be tested.

Example:

- Given the method below:

```
boolean  isValidMonth(int  m)

Functionality:  check  m  is  [1..12]
Output:
        -  true  if  m  is  1  to  12
         -  false  otherwise
```

Is there a better way to test other than testing **all** integer values [-2^{31}, …, $2^{31}-1$] ?

Equivalence Partition

Experience shows that exhaustive testing is not feasible or necessary:

- Impractical for most methods.
- An error in the code would have caused the same failure for many input values:

  There is no reason why a value will be treated differently from others.

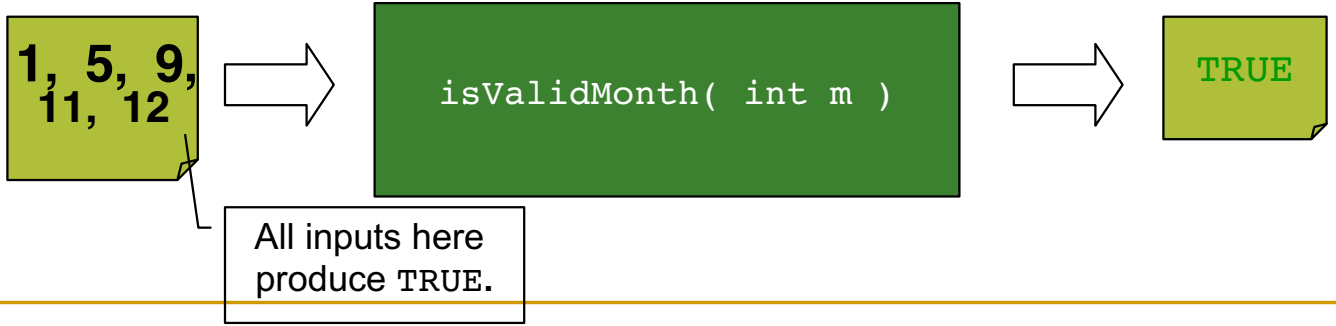
  E.g., if value `240` fails the testing, it is *likely* that `241` is likely to fail the test too.

A better way of choosing test cases is needed.

Equivalence Partition

Observations:

- For a method, it is common to have a number of inputs that produce similar outcomes.
- Testing one of the inputs *should be* as good as exhaustively testing all of them.
- So, pick only a few test cases from each "category" of input that produce the same output.



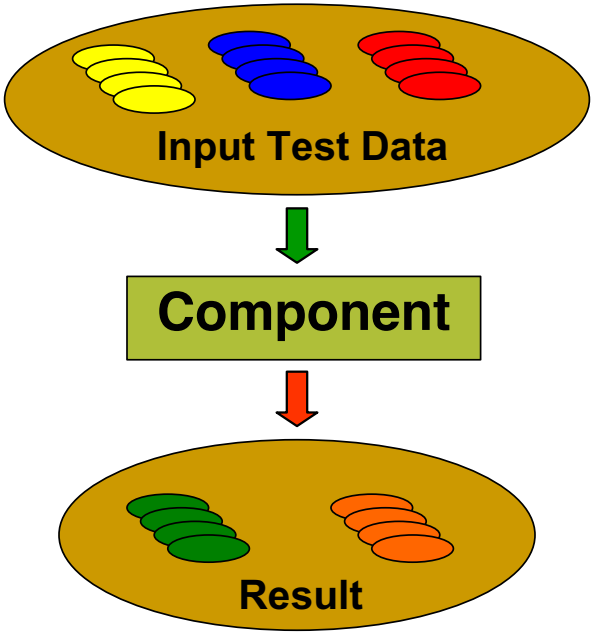
Equivalence Partition: Definition

Partition input data into *equivalence classes.*

Data in each equivalence class:

- Likely to be treated equally by a reasonable algorithm.
- Produce same output state, i.e., valid/invalid.

Derive test data for each class.



Example (*isValidMonth*)

For the *isValidMonth* example:

- Input value `[1 … 12]` should get a similar treatment.
- Input values lesser than `1`, larger than `12` are two other groups.

Three partitions:

- `[ -∞ ... 0 ]` should produce an **invalid** result
- `[ 1 … 12 ]` should produce a **valid** result
- `[ 13 … ∞ ]` should produce an **invalid** result

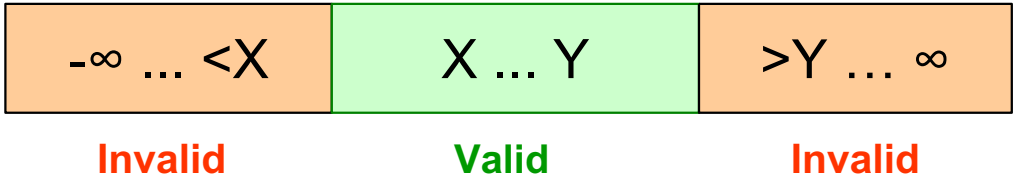
Pick one value from each partition as test case:

- E.g., `{-12, 5, 15}`
- Reduce the number of test cases significantly.

Common Partitions

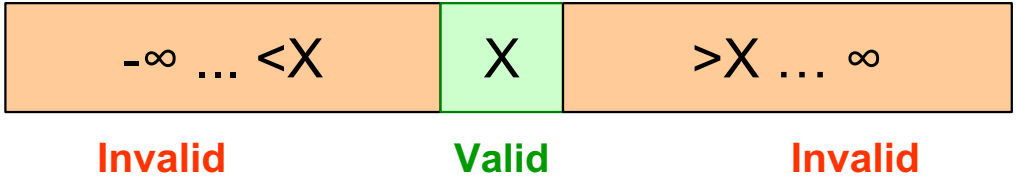
If the component specifies an input range, [X … Y].

- Three partitions:



If the component specifies a single value, [X].

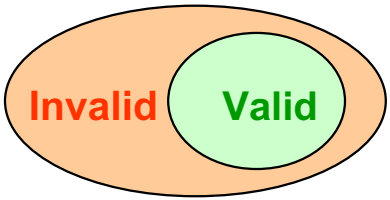
- Three partitions:



Common Partitions

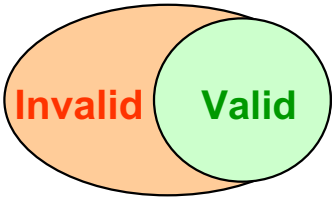
If the component specifies member(s) of a set:

- E.g., The traffic light color = {Green, Yellow, Red}; Vehicles have to stop moving when the traffic light is Red.
- Two partitions:



If the component specifies a boolean value.

- Two partitions:



Combination of Equivalence Classes

Number of test cases can grow very large when there are multiple parameters.

Example:

- Check a phone number with the following format:

  (XXX) XXX – XXXX

  Area Code (optional) **Prefix** **Suffix**

- In addition:

  Area Core Present: Boolean value [`True` or `False`]

  Area Code: 3-digit number [`200` … `999`] except `911`

  Prefix: 3-digit number not beginning with `0` or `1`

  Suffix: 4-digit number

Example (cont)

Area Core Present: Boolean value [`True` or `False`]

- Two Classes:
  [True], [False]

Area Code: 3-digit number [`200 … 999`] except `911`

- Five Classes:
  [-∞ …199], [200 … 910], [911], [912 … 999], [1000 … ∞]

Prefix: 3-digit number not beginning with `0` or `1`

- Three Classes:
  [-∞ … 199], [200 … 999], [1000 … ∞]

Suffix: 4-digit number

- Three Classes:
  [-∞ … -1], [0000 … 9999], [10000 … ∞]

Example (cont)

A thorough testing would require us to test all combinations of the equivalence classes for each parameter.

Hence, the total equivalence classes for the example should be the *multiplication* of equivalence classes for each input:

- 2 x 5 x 3 x 3 = 90 classes = 90 test cases (!)

For critical systems, all combinations should be tested to ensure a correct behavior.

A less stringent testing strategy can be used for normal systems.

Reduction of Test Cases

A reasonable approach to reduce the test cases is as follows:

- At least one test for each equivalence class for each parameter:

  Different equivalence class should be chosen for each parameter in each test to minimize the number of cases.

- Test all combinations (if possible) where a parameter may affect each other.
- A few other random combinations.

Example

As the Area Code has the most classes (i.e., 5), five test cases can be defined which simultaneously try out difference equivalence classes for each parameter:

Test Case	Area Code Present	Area Code	Prefix	Suffix
1	False	[-∞ …199]	[-∞ … 199]	[-∞ … -1]
2	True	[200 … 910]	[200 … 999]	[0000 … 9999]
3	True	[911]	[1000 … ∞]	[10000 … ∞]
4	True	[912 … 999]	[200 … 999]	[0000 … 9999]
5	True	[1000 … ∞]	[1000 … ∞]	[10000 … ∞]

E.g., Actual Test Case Data:
(True, 934, 222, 4321)

Example (cont)

In this example, Area Code Present affects the interpretation of Area Code, so all combinations between these two parameters should be tested.

- 2 x 5 = 10 test cases.
- Five combinations have already been tested in the previous test cases, five more would be needed.

Not counting extra random combinations, this strategy reduces the number of test cases to only 10.

Boundary Value Analysis: Introduction

It has been found that most errors are caught at the **boundary** of the equivalence classes.

Not surprising, as the end points of the boundary are usually used in the code for checking:

- E.g., checking `K` is in range [`X` … `Y`):

```
if (K >= X && K <= Y)
    ...
```

Easy to make mistake on the comparison.

Hence, when choosing test data using equivalence classes, boundary values should be used.

Nicely complement the Equivalence Class Testing.

Using Boundary Value Analysis

If the component specifies a range, [`X` … `Y`]

- Four values should be tested:
  - Valid: `X` and `Y`
  - Invalid: Just below `X` (e.g., `X – 1`)
  - Invalid: Just above `Y` (e.g., `Y + 1`)
- E.g., [`1` … `12`]
  - Test Data: `{0, 1, 12, 13}`

Similar for open interval (`X` … `Y`), i.e., `X` and `Y` not inclusive.

- Four values should be tested:
  - Invalid: `X` and `Y`
  - Valid: Just **above** `X` (e.g., `X + 1`)
  - Valid: Just **below** `Y` (e.g., `Y – 1`)
- E.g., (`100` … `200`)
  - Test Data: `{100, 101, 199, 200}`

Using Boundary Value Analysis

If the component specifies a number of values:

- Define test data using [`min` value … `max` value]:
  - Valid: `min, max`
  - Invalid: Just below `min` (e.g., `min – 1`)
  - Invalid: Just above `max` (e.g., `max + 1`)
- E.g., `values = {2, 4, 6, 8} → [2 … 8]`
  - Test Data: `{1, 2, 8, 9}`

If a data structure has prescribed boundaries:

- define test data to exercise the data structure at those boundaries.
- E.g.,
  - String: Empty String, String with 1 character
  - Array : Empty Array, Array with 1 element, Full Array

Boundary value analysis is not applicable for data with no meaningful boundary, e.g., the set *color* `{Red, Green, Yellow}`.

Functionality Testing

Previous examples have mostly numerical parameters and simplistic functionality, where it is easy to see how Equivalence Class Testing and Boundary Value Analysis can be applied.

The following example is to illustrate how functionality testing of a method can be accomplished by the black box testing techniques discussed.

This requires the method to be well specified:

- The Precondition, Postcondition and Invariant should be available.
- The Invariant: A property that is preserved by the method, i.e., `true` before and after the execution of method.

Functionality Testing

For a well specified method (or component).

- Use the Precondition:

  Define Equivalence Classes.

  Apply Boundary Value Analysis if possible to choose test data from the equivalence classes.

- Use the Postcondition and the Invariant:

  Derive expected results.

Example (Searching)

boolean Search(
List aList, int key)

Precondition:

- **aList has at least one element**

Postcondition:

- **true if key is in the aList**
- **false if key is not in aList**

Equivalence Classes

Sequence with a single value:

- key found.
- key not found.

Sequence of multi values:

- key found:
  - First element in sequence.
  - Last element in sequence.
  - “Middle” element in sequence.
- key not found.

Test Data

Test Case	aList	Key	Expected Result
1	[123]	123	True
2	[123]	456	False
3	[1, 6, 3, -4, 5]	1	True
4	[1, 6, 3, -4, 5]	5	True
5	[1, 6, 3, -4, 5]	3	True
6	[1, 6, 3, -4, 5]	123	False

Example (Stack – Push Method)

```
void push (Object obj) throws FullStackException
```

Precondition:

- `! full()`

Postconditions:

- `if !full() on entry  then`
  `top() == obj && size() == old size() + 1`
  `else throw FullStackException`

Invariant:

- `size() >= 0 && size() <= capacity()`

Common methods in the `Stack` class:

- `full(), top(), size(), capacity()`

Test Data

Precondition: stack is not full (i.e., boolean).

- Two equivalence classes can be defined.
- Valid Case: `Stack` is not full.

  Input: a non-full stack, an object `obj`

  Expected result:

  ```
  top() == obj
  size() == old size() + 1
  0 <= size() <= capacity()
  ```

- Invalid Case: `Stack` is full.

  Input: a full stack, an object `obj`

  Expected result:

  `FullStackException` is thrown

  ```
  0 <= size() <= capacity()
  ```

Example

A HR department of company is following employment process where applications are sorted out based on a person's age. A person under 15 as well as senior citizen older than 55 is not eligible to hire. Person less than 18 can be hired on a part-time basis only. Adult persons above 18 can hire as full time employees. Design Test Cases using Equivalence Partitioning and Boundary Value Analysis for above scenario.

Answer

Equivalence Partitioning

- Age Below 15
- Age 16 to 18
- Age 18 to 55
- Age above 55

Boundary Value Analysis:

- Test case values for Attribute Age: 14, 15, 16, 17, 18, 19, 54, 55, 56