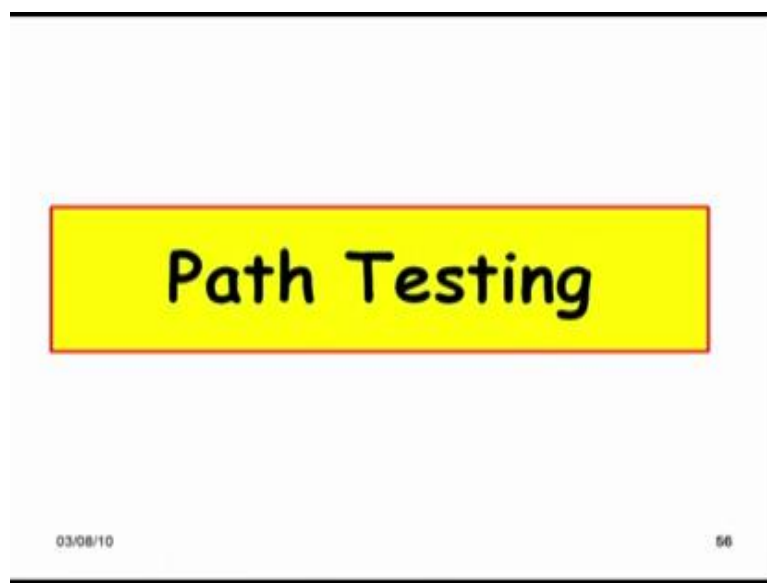


Lecture – 13
Path Testing

Welcome to this session and we will discuss about path testing.

(Refer Slide Time: 00:25)




In the path testing, we design the test cases or we **require** the test cases to achieve path coverage.

(Refer Slide Time: 00:26)

Path Coverage

- Design test cases such that:
 - All linearly independent paths in the program are executed at least once.
- Defined in terms of
 - Control flow graph (CFG) of a program.



We design the test cases such that () path coverage **is** achieved, but it may not always be () possible to design those test cases we will see why, but at least we should have test cases which may be randomly input, but we will require this, those random input numbers will achieve the required path coverage. So, what is the definition of path coverage? The definition of path coverage is that all linearly independent paths in the program are executed and by program we mean a unit and a unit is a function here.

So, in the function all the linearly independent paths are executed by the test cases for achieving path coverage but then that brings us to this question of what are linearly independent paths. The linearly independent paths are defined on a control flow graph of a program.

(Refer Slide Time: 01:50)

Path Coverage-Based Testing

- To understand the path coverage-based testing:
 - We need to learn how to draw control flow graph of a program.
- A control flow graph (CFG) describes:
 - The sequence in which different instructions of a program get executed.
 - The way control flows through the program.

And the control flow graph as we are discussing in the last session, it is a graph representation of the program where the statements of the program are nodes of this graph and it represents the sequence in which the control flows through the program, or the sequence in which the statements get executed. For a trivial program, where only linear transfer of control takes place, there are no branches, it will be just a linear graph, whereas in realistic programs it will be a complicated graph and we want to draw the control flow graph for a programs, so that we can define what are the linearly independent paths and the coverage required on them to be able to understand what is meant by path testing.

So, let us see how to draw a control flow graph for a program.

(Refer Slide Time: 03:01)

How to Draw Control Flow Graph?

- **Number all statements of a program.**
- Numbered statements:
 - Represent nodes of control flow graph.
- Draw an edge from one node to another node:
 - **If execution of the statement representing the first node can result in transfer of control to the other node.**

The first thing is that we have to number all node all the statements in the program and we have to draw one node in the graph for each program and we have to see if there is a transfer of control possible from one node to the other then we draw a an edge.

(Refer Slide Time: 03:20)

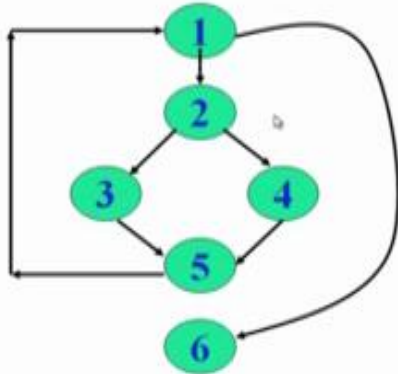
Example

```
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; }
```

So, using that for this program, we will have a graph like this, but then let us discuss about systematic technique would given an arbitrary program.

(Refer Slide Time: 02:28)

Example Control Flow Graph



How do I draw the CFG for that?

(Refer Slide Time: 03:41)

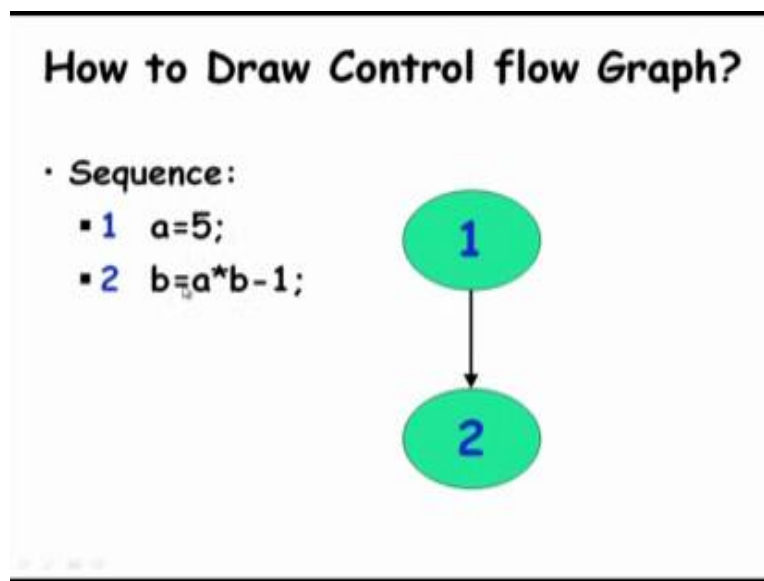
How to Draw Control flow Graph?

- Every program is composed of:
 - Sequence
 - Selection
 - Iteration
- If we know how to draw CFG corresponding these basic statements:
 - We can draw CFG for any program.

We need to know that every structured program consists of three types of statements, the sequence type of statements that is one statement when it is executes the next statement executes though they are sequence type of statement, selection type of statements these are if then else switch those are the selection statements and the iteration statements which are for while, do while etcetera those kind of loops.

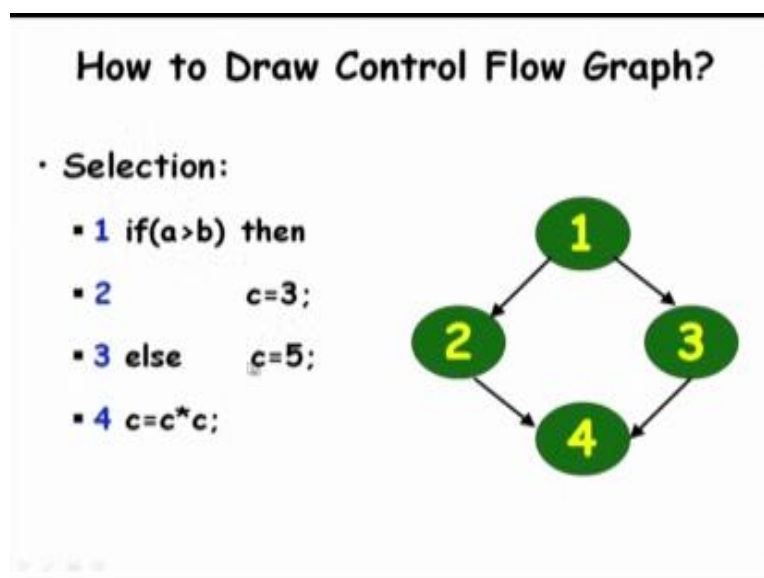
So, if we can draw the flow corresponding to this kind of statements the edges that are would be required in the CFG; control flow graph for these kind of statements then we can look through the program and for every statement, we can draw the edges given any arbitrary program we should be able to draw the CFG.

(Refer Slide Time: 04:51)



Now, let us look at the sequence this is the easiest. So, $a = 5$ as soon as it executes the next statement executes is a sequence type of statement and this is the easiest. So, as soon as one completes b executes. So, there is no hesitation and no ambiguity we just draw an edge from 1 to 2. Now, the selection is also not hard either here.

(Refer Slide Time: 05:16)



We have if a greater than b , c is 3 else $c = 5$ and this is the statement sequence type of statement following the selection. So, sequence type of statement is also very intuitive from 1 the control

transfers to either 2 or to 3 and as soon as either 2 or 2 or 3 completes control goes to the next statement. So, a selection statement is quite intuitive we can easily draw the CFG corresponding to that, but the one which is slightly complicated or which requires little bit of understanding is for iteration.

So, we have while some condition and two statements here in the loop and then there is a sequence statement. So, we have numbered these 1, 2, 3, 4 and from 1 control can come to 2, 2 to 3. So, as soon as the while evaluates to true the expression evaluates to true it control comes to 1 to 2, 2 to 3, but the important thing here to observe is that from 3 the control does not come to 4 after 3 the loop expression is evaluated. So, after 3 we have drawn this edge, so that at the end of iteration the loop expression is evaluated and if the loop expression becomes false then the control transfers to the next statement.

So, many do this mistake that they draw 3 to 4 control transfer, but control transfer from three to four does not happen it is should not be drawn. So, from three always the loop expression is evaluated and once the loop expression is evaluated then if it evaluates to false the edge is drawn here to 4 and in a control flow graph, we do not write the conditions that this evaluates to true or this evaluates to false that we do on a flow chart, but not in CFG as long as control can transfer from one node to the other we draw the edge.

Now, let us look at the definition of a path.

(Refer Slide Time: 07:59)

Path


- A path through a program:
 - A node and edge sequence from the starting node to a terminal node of the control flow graph.
 - There may be several terminal nodes for program.

A path is a node and edge sequence from the start node to a terminal node in the control flow graph is a path. So, this is any sequence of node edge pairs starting from the start node to a terminal node in the program is called as a path and also remember that there may be several terminal nodes in a program due to exit and such grudges.

(Refer Slide Time: 08:39)

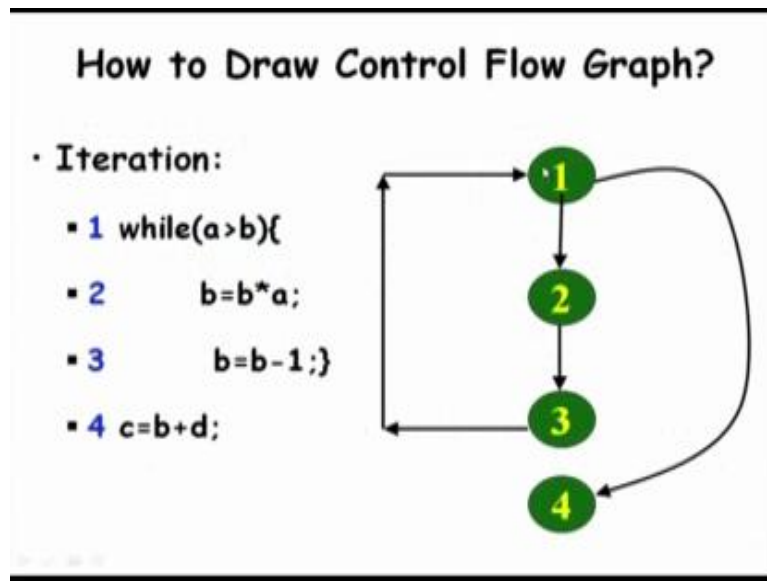
All Path Criterion

- In the presence of loops, the number paths can become extremely large:
 - This makes all path testing impractical



So, one thing is that in presence of loops the number of paths can become very large because it is any sequence of nodes from start node to the end node for the program that we had discussed.

(Refer Slide Time: 09:03)



Let us look at this iteration. So, 1-4 will be a path 1, 2, 3, 4 will be a path, 1, 2, 3, 1, 2, 3, 4 will be a path 1, 2, 3, 1, 2, 3, 1, 2, 3 will and 4 will be a path. So, there for each iteration of the loop will have a new path and if you require that all paths in the loop, we executed the number of test cases required will be huge and for any practical program we cannot achieve all path testing all path testing is impractical and for that reason we need to define this notion of a linearly independent path. So, which will achieve almost edge through testing as all path testing, but will require a manageable number of test cases.

So, let us look at what is a linearly independent set up path and then we will require coverage of this linearly independent set up paths. So, if we have p_1 to p_n is a path.

(Refer Slide Time: 10:17)

Linearly Independent Paths

- A path p is said to be a linear combination of paths p_1, \dots, p_n
 - if there are integers a_1, \dots, a_n such that $p = \sum a_i p_i$ (a_i could be negative, zero, or positive)
- A set of paths is linearly independent if no path in the set is a linear combination of any other paths in the set
 - A linearly independent path is any path *through the program* ("complete path") that introduces at least one *new edge* that is not included in any other linearly independent paths.

68

Then we say p is a linear combination of the path p_1 to p_n , if we have integers a_1 to a_n such that a linear combination of p_i $a_i p_i$ will result in the path p so that means, a path is represented as a set of nodes and edges. So, if one path is a linear addition is addition of two paths then that is not a linearly independent path. So, a set of paths is linearly independent if no path set with a linear combination of other paths in the set.

In other words, we say that each path in a linearly independent set of path as at least one edge which cannot be obtained by a linear combination of the other paths in the set. So, any path would not be in a linearly independent set of path would not be obtainable by a linear combination of other part paths in the set and there would be at least one edge that is not included in other paths. So, that is the definition of the linearly independent paths.

Now, let us look at example of a linearly independent set of path, if we look at the linearly independent set of paths, here if we have 1-4, if 1 is the start node and 4 is the terminal node 1-4 is a path 1, 2, 3, 4 is also a path, but they are not linearly independent path because the second one does not really introduce a new edge.

(Refer Slide Time: 12:47)

Linearly Independent Path

- Any path through the program that:
 - Introduces at least one new edge:
 - Not included in any other independent paths.

So, any path through the program that introduces at least one new edge not included in the other independent paths **we** will call it as an independent path.

(Refer Slide Time: 13:02)

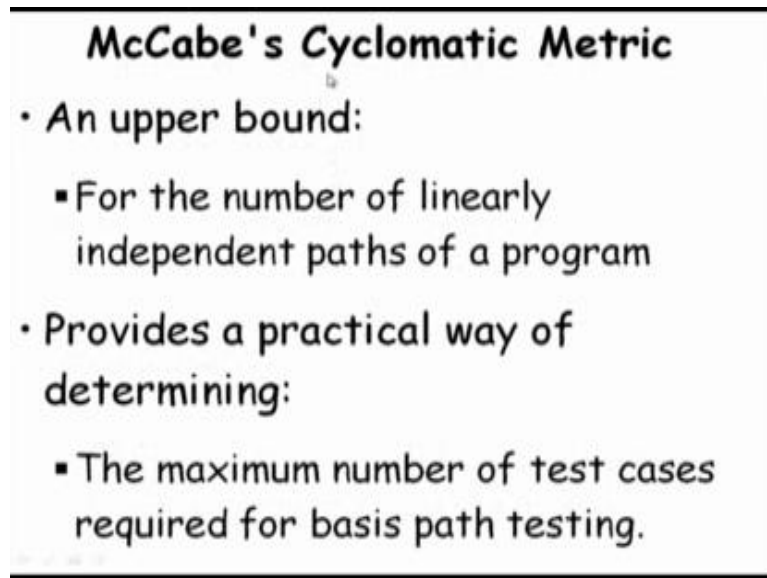
Linearly Independent path

- It is straight forward:
 - To identify the Linearly independent set of paths of simple programs.
- For complicated programs:
 - It is not easy to determine the number of independent paths.

Now, given a program small program of five lines or seven lines, we can identify the linearly independent set of paths by examining the control flow graph, but if the program has 20 or 30 nodes and 50 edges in that, it would become extremely complicated to look through the graph and find out what are the linearly independent set of paths.

You can spend 1 or 2 weeks just to identify what are the linearly independent set of paths for a program with 20 statements and therefore, nobody really tries to identify linearly independent set of paths, but we use this concept in doing the path testing.

(Refer Slide Time: 13:54)

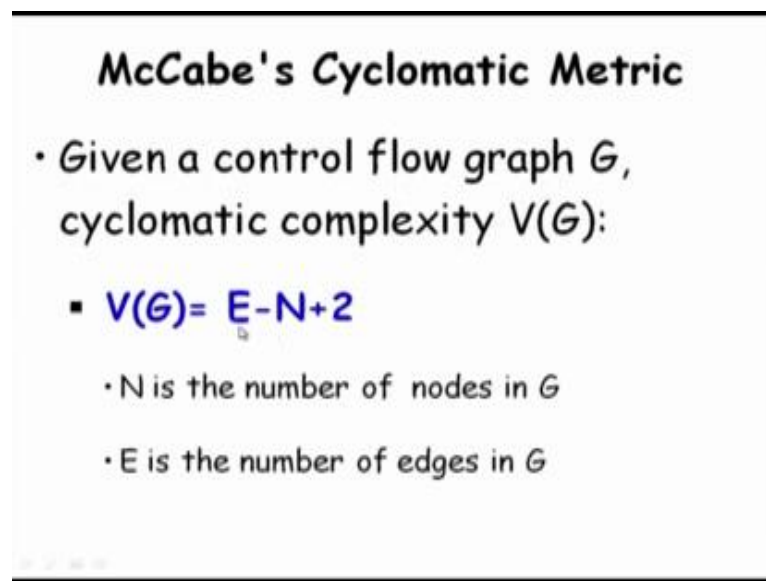


McCabe's Cyclomatic Metric

- An upper bound:
 - For the number of linearly independent paths of a program
- Provides a practical way of determining:
 - The maximum number of test cases required for basis path testing.

So, how do we do the path testing? We have this notion of McCabe's Cyclomatic Metric; the cyclomatic metric is very easy to compute given a control flow graph. We can compute the metric just by observing the graph or we can have tools which can easily compute the McCabe's Cyclomatic metric from the control flow graph or given the program code it can itself, if required compute the control flow graph and also the McCabe's metric and display the number to you and also McCabe metric as I was saying that you can compute very easily through a **visual** observation of the control flow graph.

(Refer Slide Time: 15:33)



McCabe's Cyclomatic Metric

- Given a control flow graph G , cyclomatic complexity $V(G)$:
 - $V(G) = E - N + 2$
 - N is the number of nodes in G
 - E is the number of edges in G

So, the McCabe's metric is actually is a bound on the number of linearly independent paths. So, if we know the McCabe's metric, we know at most how many paths to look for in the program if the McCabe's metric say 10 evaluates to 10, we would look for at most 10 linearly independent paths.

So, if we can compute the McCabe's metric it tells us at least how many test cases will be required to achieve path coverage. Now, let us see how to compute the McCabe's metric if we have the CFG, if we compute the number of edges in the graph minus the number of nodes plus 2 then we get the cyclomatic metric $e - n + 2$, we give us the cyclomatic metric. So, that would be easy to automate given program, we can easily draw the control flow graph and from the control flow graph you can compute the number of edges number of nodes plus 2 that will give the cyclomatic metric. So, for this graph the cyclomatic metric is there are 7 edges and 6 nodes, $7 - 6 + 2$ is 3, there are other ways of computing the cyclomatic metric as well.

(Refer Slide Time: 16:27)

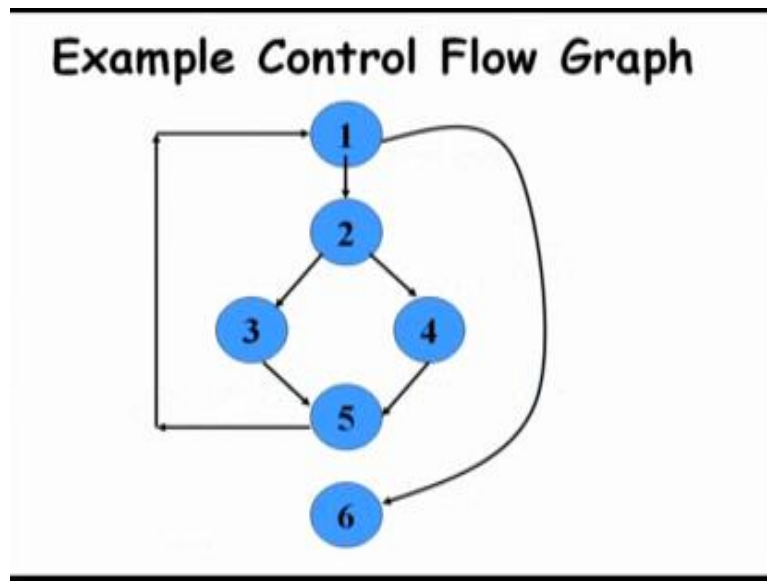
Cyclomatic Complexity

- Another way of computing cyclomatic complexity:
 - inspect control flow graph
 - determine number of bounded areas in the graph
- $V(G) = \text{Total number of bounded areas} + 1$
 - Any region enclosed by a nodes and edge sequence.

So, the way that the approach that we looked at now $e - n + 2$, it is easy to automate, we can also have a way to compute the cyclomatic metric just by usually observing a graph a CFG, you can just usually observe and say that what is it is cyclomatic complexity.

How do we do that? Here we look at the graph and find out how many total number of bounded areas are there, how many bounded area are there plus one that also gives us the cyclomatic metric and the definition of a bounded area is that a region enclosed by nodes and edges in sequence and this number should match exactly with the one computed by $e - n + 2$.

(Refer Slide Time: 17:18)




Now, let us look at this. So, how many bounded areas are there, let us see here there is one bounded area and there is another bounded area here. So, there are two bounded areas and therefore, number of bounded areas plus 1 is equal to 3 and that is exactly equal to $e - n + 2$ result that we have got that had also evaluated to 3, $7 - 6 + 2$.

(Refer Slide Time: 17:52)

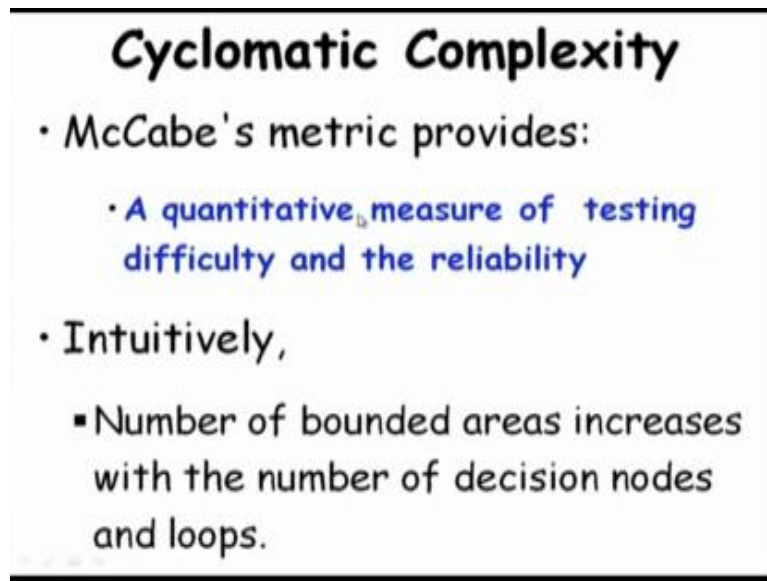
Example

- From a visual examination of the CFG:
 - Number of bounded areas is 2.
 - Cyclomatic complexity = $2 + 1 = 3$.



So, the number of bounded areas is 2 and the cyclomatic complexity is 3, the McCabe's metric it provides the testing difficulty and the reliability.

(Refer Slide Time: 17:58)



Cyclomatic Complexity

- McCabe's metric provides:
 - A quantitative measure of testing difficulty and the reliability
- Intuitively,
 - Number of bounded areas increases with the number of decision nodes and loops.

So, if we know that the cyclomatic metric for a program is 50 then we know that we need at least 50 test cases for this and also we know that this program is extremely complicated because requires 50 test cases and therefore, the testing effort required will be very high and also it will likely to have bugs even after testing with 50 test cases. So, 50 test cases, 50 McCabe's metric indicates that there are too many branches there and the code is not good. Normally the companies require that the code be have less than 10 McCabe's metric.

So, the first method of computing the cyclomatic metric $e - n + 2$ is good to automate, write a small program which will compute the cyclomatic complexity by $e - n + 2$. So, we gives an indication of the number of test cases that are to be designed to guarantee the coverage of independent paths and also there is another way of determining the cyclomatic metric, just look through the program do not even have to look at the CFG just visually inspect the program, find out how many branch expressions are there plus 1 that will also give you the cyclomatic metric.

(Refer Slide Time: 20:01)

Cyclomatic Complexity

- Knowing the number of test cases required:
 - Does not make it any easier to derive the test cases,
 - Only gives an indication of the minimum number of test cases required.

Look through the program find out how many branch expressions are there, how many if while those kinds of statements are there plus one that will also be the cyclomatic metric, but one thing that we need to remember is that knowing the cyclomatic metric. We know how many paths are there how many linearly independent paths are there, but we do not know the paths themselves and even after knowing the number of paths required. It will be hard to find those parts in the code in the CFG, but our testing would not required to find those paths. A path testing does not require us to find those paths and write test cases to execute it.

We normally execute the program with random test cases, but we try to measure we measure what is the branch what is the path coverage achieved and as long as we achieved high path coverage we say that path coverage path testing has been done. So, the cyclomatic complexity is a very important number. It tells us about the quality of the code what will be the final reliability of the program that once it is released and the testing effort and the minimum number of test cases required to test the program. So, a practical path testing, the tester proposes initial set of test data from experience and judgment.

(Refer Slide Time: 21:59)

Practical Path Testing

- The tester proposes initial set of test data :
 - Using his experience and judgment.
- A dynamic program analyzer used:
 - Measures which parts of the program have been tested
 - Result used to determine when to stop testing.

Then, we have a small tool called as a dynamic program analyzer. The dynamic program analyzer actually displays how many linearly independent paths have been covered and it is not hard to write such a program actually would like to give it as an assignment to write a dynamic program analyzer.

So, given a small code we can write a program to generate, its control flow graph the control flow graph nodes are actually the statements and we know that what will be the edges depending on the type of the statement and once we have the CFG as the program executes. We mark what are the edges that are taken by the test cases, what are the edges that are executed by the test cases, and as long as the test case execute at least one new edge we know that a new path has been executed.

Given a CFG which you can easily get from a program code, we know the number of paths by the McCabe's metric and then we can determine as the test case executes we can determine the number of paths that are covered by the test cases and then we can determine the percentage path coverage that has been achieved. But for very a small programs,

(Refer Slide Time: 23:53)

Derivation of Test Cases

- Draw control flow graph.
- Determine $V(G)$.
- Determine the set of linearly independent paths.
- Prepare test cases:
 - Force execution along each path.
 - Not practical for larger programs.

we can even prepare test cases to force execution along each path, but which is obviously, not practical for large programs. So, for small programs we first determine what the cyclomatic complexity is determine the set of linearly independent paths and then propose test cases design test cases which will execute this paths. So, this is just an example the same example code and then we see here that there are two decision statements and therefore, it is cyclomatic complexity is 3. So, we might expect up to three paths, but then there are actually two linearly independent paths.

(Refer Slide Time: 24:26)

Example

```
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;    }
```




So, 1, 2, 3, 5, 1, 6 and 1, 2, 4, 5, 1, 6, so these are two independent paths and any other paths are subsumed or linear combination of the paths of those two. So, we can then have the test cases x is equal to 1, y is equal to 1, x is equal to 1, y is equal to 2, etcetera which will execute those test cases for a small program 3 line or 4 line program.

(Refer Slide Time: 25:11)

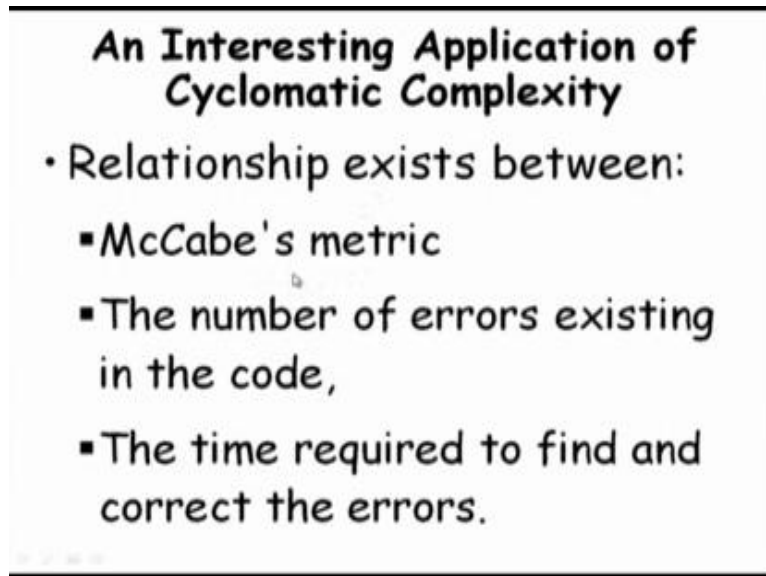
Derivation of Test Cases

- Number of independent paths: 3
 - 1,6 test case (x=1, y=1)
 - 1,2,3,5,1,6 test case(x=1, y=2)
 - 1,2,4,5,1,6 test case(x=2, y=1)



We can easily draw the CFG and find out the McCabe's metric which gives us upper bound on the number of test cases and then we write those test cases and check whether the paths are executed. Now, let us look at interesting application of the cyclomatic complexity.

(Refer Slide Time: 25:56)

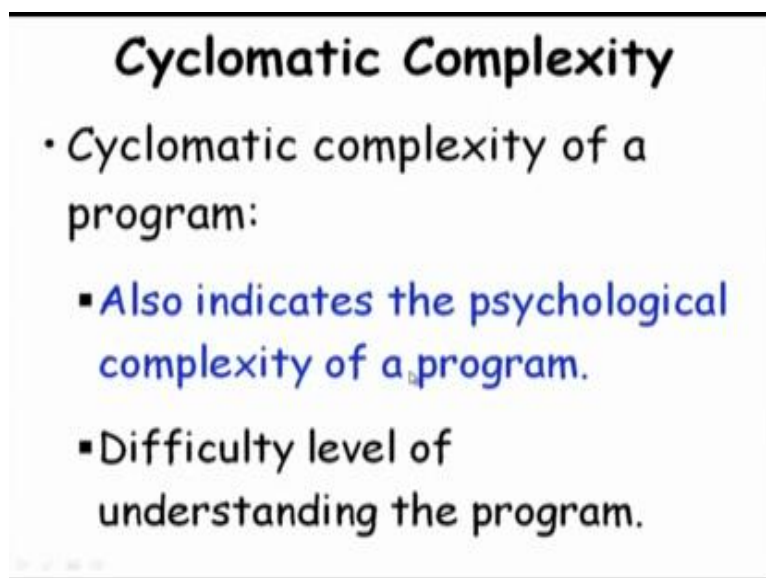


An Interesting Application of Cyclomatic Complexity

- Relationship exists between:
 - McCabe's metric
 - The number of errors existing in the code,
 - The time required to find and correct the errors.

We have already seen that McCabe's metric for a code indicates that after the code as been tested, how many errors would execute in that, how many errors will be present in that code it also indicates that given this code to a third person how much **effort** he has to make understand the code. So, the McCabe's metric is a measure of the psychological complexity of the program or the difficulty level of this program.

(Refer Slide Time: 26:31)



Cyclomatic Complexity

- Cyclomatic complexity of a program:
 - Also indicates the psychological complexity of a program.
 - Difficulty level of understanding the program.

So, given independent person who is not associated with this code development give it to him and

ask him to understand the code, the effort that you would have to put to understand this code would co-relate with the cyclomatic complexity. If the cyclomatic complexity is 50, he would have to spend substantial effort in understanding the code and if it is 10 or 5 or 1, he would have to spend very little time in understanding the code.

Why is that? What is the reason intuitively that he would have to spend lot of time in understanding a piece of code? If the cyclomatic complexity is large the answer is that to understand a code, we actually trace all the linearly independent paths in the code, we just look at the output and try to see how which inputs produce these or in other approach we look at the input and see what is the output produced in both these types of understanding of the program. We actually unconsciously traverse paths in the program.

So, our understanding the code would require us to traverse all the paths through the program and if the paths are more obviously, will have to spend long time traversing all paths in the code. So, we will stop this session at this point we will continue in the next session.

Thank you.