UNIT-III
KCS-601
SOFTWARE ENGINEERING
PART-II

Syllabus

Unit-III: Software Design

Basic Concept of Software Design, Architectural Design, Low Level Design: Modularization, Design Structure Charts, Pseudo Codes, Flow Charts, Coupling and Cohesion Measures, Design Strategies: Function Oriented Design, Object Oriented Design, Top-Down and Bottom-Up Design.

Software Measurement and Metrics: Various Size Oriented Measures: Halestead's Software Science, Function Point (FP) Based Measures, Cyclomatic Complexity Measures: Control Flow Graphs

SOFTWARE MEASUREMENT AND METRICS

- **SOFTWARE METRICS-**

  Software Metrics are **Quantifiable Measures** that could be used to **measure different characteristics of a Software System** or the **Software-Development Process**.

  Metrics and Measurements are **necessary aspects of managing a Software- Development Project.**

  For effective monitoring, management needs to get information about the project-

  - How far it has progressed,
  - How much development has taken place,
  - How far behind schedule it is, and the
  - Quality of the development so far.

  Based on this information, decisions can be made about the project.

Definition

- **Software Metrics can be Defined as-**

  "The **continuous application of Measurement-Based techniques** to the **software-development process** and **its products to supply meaningful and timely management information**, together with the use of those techniques to improve that process and its products."

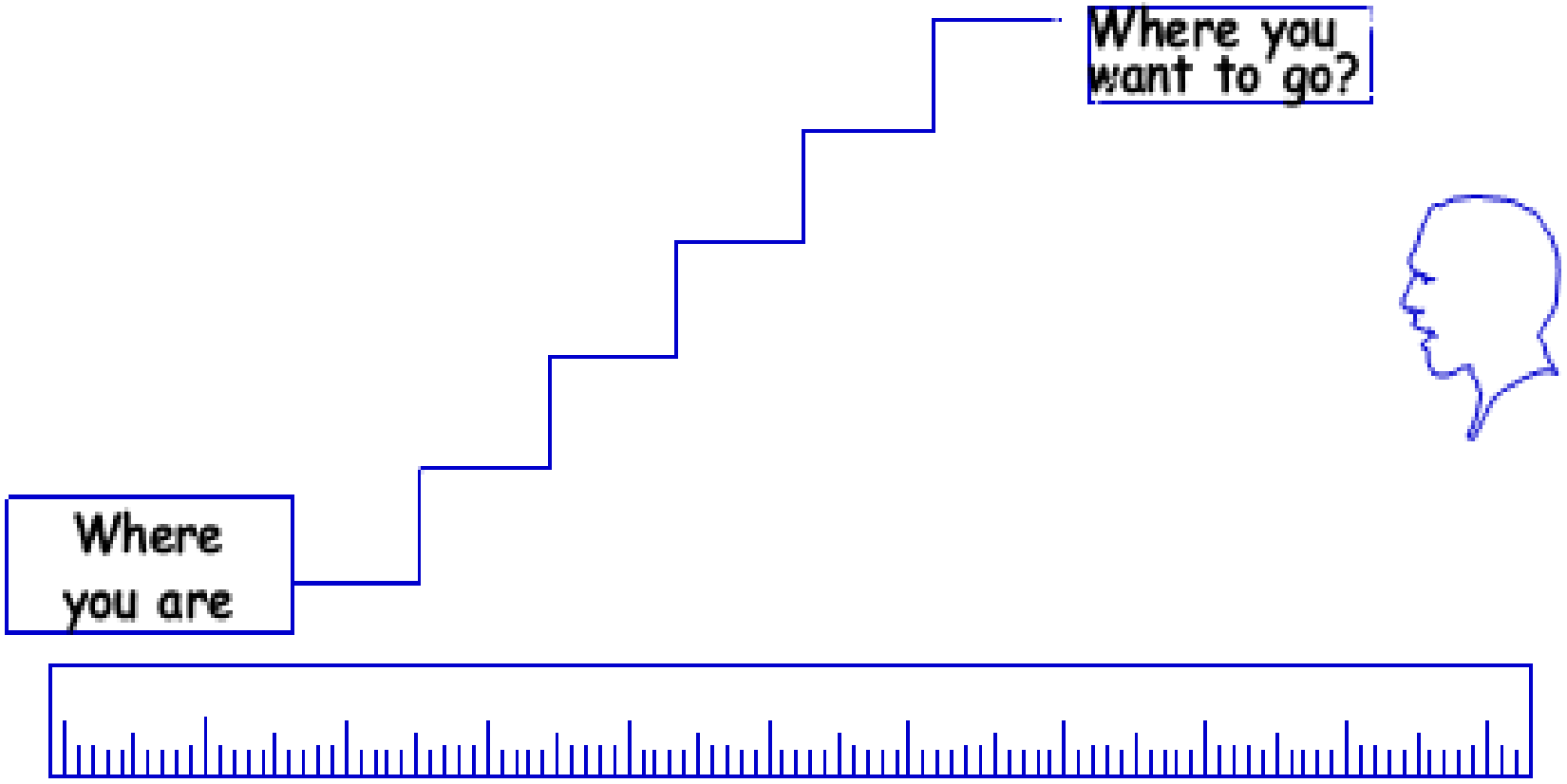
  Metrics Ensure that the **Final Product is of High Quality** and the **Productivity of the Project stays High**.

  In this sequence metrics for intermediate products of requirements and design is to predict or get some idea about the metrics of the final product.

  Several Metrics have been designed for coding;

  - Size, Complexity, Style, and Reliability.

SOFTWARE MEASUREMENT AND METRICS



SOFTWARE MEASUREMENT AND METRICS

Direct Metrics	Indirect Metrics
Size (LOC or FP)	Efficiency
Cost	Productivity
Effort	Reliability
Errors	Functionality
Defects	Complexity
	Maintainability

Categories of Metrics

- There are three categories of software metrics, which are as follows:

  Product Metrics- Product Metrics describe the **characteristics of the product**, such as **size, complexity, design features, performance, efficiency, reliability, portability**, etc.

  Process Metrics- Process Metrics describe the **effectiveness and quality of the processes** that **produce the software product.**

  - Examples are-
    - Effort required in the process.
    - Time to produce the product.
    - Effectiveness of defect removal during development.
    - Number of defects found during testing.
    - Maturity of the process.

Categories of Metrics

- **Project Metrics-** Project Metrics describe the Project Characteristics and Execution.

  Examples are-

  - Number of software developers.
  - Staffing pattern over the life-cycle of the software.
  - Cost and schedule.
  - Productivity.

Attributes of Effective Software Metrics

- Hundreds of metrics have been proposed for computer software, but not all provide practical support to the software engineer.
- Some demand measurement that is too complex, others are so obscure that few real-world professionals have any hope of understanding them.
- The **Derived Metric and the Measures** that lead to it should be-
  - Simple and Computable.
  - Consistent and objective.
  - Consistent in the use of units and dimensions.
  - Programming-language independent.
  - An effective mechanism for high-quality feedback.

HALSTEAD'S SOFTWARE SCIENCE

- **Software Science is an Approach**—Based on **Halstead's Theories**—

  "Measuring Software Qualities on the Basis of Objective Code Measures".

- **It is An Analytical technique to measure-**

  Size, Development Effort, and Development Time.

- It is Based on **Information Theory**, which in turn is based on the following Measurable Quantities, defined for a given Program Coded in any Programming Language.

HALSTEAD'S SOFTWARE SCIENCE

- η_1, the number of unique, distinct operators appearing in the program;
- η_2, the number of unique, distinct operands appearing in the program;
- N_1, the total number of occurrences of operators in the program;
- N_2, the total number of occurrences of operands in the program.

❑ In the expression a = &b;

- **{a, b}** are operand and
- **{ =, &}** are operators.

Halstead's Metrics

Program Length, $N = N_1 + N_2$

Vocabulary, $n = n_1 + n_2$

Predicted Length, $N\hat{} = (n_1 * \log_2 n_1) + (n_2 * \log_2 n_2)$

Program Volume, $V = N * \log_2 n$

Potential volume, $V^* = (2 + n_2^*) \log_2 (2 + n_2^*)$

- program with minimum size

n_1 / n_2 - Number of unique operators / operands

N_1 / N_2 -Total occurrences of operators /operands

Halstead's Metrics

Program Level, $L = V^*/V$

- **Ranges 0-1, highest possible level is 1**

Estimated Level, $L^{\wedge} = 2\ n_2 / (n_1\ N_2)$

Difficulty, $D = 1/L$

Estimated Difficulty, $D^{\wedge} = 1/L^{\wedge}$

Purity ratio: $PR = \hat{N}/N$

Program effort: $E=D*V$

- This is a good measure of program understandability

Predicted number of bugs $B = V/3000$

Language level, $\lambda = L^* V^* = L^2 V$

Halstead's Metrics

n1 is number of distinct operators

n2 is number of distinct operands

N1 is total number of operators

N2 is total number of operands

From these numbers, five measures are derived :

Measure	Symbol	Formula
Program length	N	N = N1 + N2
Program vocabulary	n	n = n1 + n2
Volume	V	V = N * ($\log_2$ n)
Difficulty	D	D = (n1/2) * (N2/n2)
Effort	E	E = D * V

Counting rules for C language

1. Comments are not considered.
2. The identifier and function declarations are not considered.
3. All the variables and constants are considered operands.
4. Global variables used in different modules of the same program are counted as multiple occurrences of the same variable.

Counting rules for C language

5. Local variables with the same name in different functions are counted as unique operands.

6. Functions calls are considered as operators.

7. All looping statements e.g., do {…} while ( ), while ( ) {…}, for ( ) {…}, all control statements e.g., if ( ) {…}, if ( ) {…} else {…}, etc. are considered as operators.

8. In control construct switch ( ) {case:…}, switch as well as all the case statements are considered as operators.

Counting rules for C language

9. The reserve words like return, default, continue, break, sizeof, etc., are considered as operators.

10. All the brackets, commas, and terminators are considered as operators.

11. GOTO is counted as an operator and the label is counted as an operand.

12. The unary and binary occurrence of "+" and "-" are dealt separately. Similarly "*" (multiplication operator) are dealt with separately.

Counting rules for C language

13. In the array variables such as “array-name [index]” “array-name” and “index” are considered as operands and [ ] is considered as operator.

14. In the structure variables such as “struct-name, member-name” or “struct-name -> member-name”, struct-name, member-name are taken as operands and ‘.’, ‘->’ are taken as operators. Some names of member elements in different structure variables are counted as unique operands.

15. All the hash directive are ignored.

Example-1

Example

```
if (k < 2)
{
   if (k > 3)
      x = x*k;
}
```

- Distinct operators: if ( ) { } > < = * ;
- Distinct operands: k 2 3 x
- $n_1 = 10$
- $n_2 = 4$
- $N_1 = 13$
- $N_2 = 7$

- n_1 - number of distinct operators
- n_2 - number of distinct operands
- N_1 - total number of operators
- N_2 - total number of operands

Example-2

Take a program, for example this little bit of one.

```
int f=1, n=7;
for (int i=1; i<=n; i+=1)
  f*=i;
```

An "operator" is a fixed symbol or reserved word in a language and "operand" is everything else: variable names function names, numeric constants, etc. Comments don't count.

Define n1 to be the number of unique operators, in the example there are 10:

```
int  =  ,  ;  for  (  <=  +=  )  *=
```

Define n2 to be the number of unique operands, in the example there are 5:

```
f  1  n  7  i
```

Define N1 to be the number of operators, in the example there are 16:

```
int  =  ,  =  ;  for  (  int  =  ;  <=  ;
+=  )  *=  ;
```

Define N2 to be the number of operands, in the example there are 12:

```
f  1  n  7  i  1  i  n  i  1  f  i
```

Example-3

```
main( )
{
 int a, b, c, avg;
 scanf("%d %d %d", &a, &b, &c);
 avg = (a+b+c)/3;
 printf("avg = %d", avg);
}
```

The unique operators are: main,(),{},int,scanf,&/, printf

The unique operands are: a, b, c, &a, &b, &c, a+b+c, avg, 3, "%d %d %d", "avg = %d"

Therefore, $\eta_1 = 12$, $\eta_2 = 11$

Calculated Program Length (or Estimated Length) = $(12*\log_2 12 + 11*\log_2 11) = (12*3.58 + 11*3.45) = (43+38) = 81$

Volume = Length*log(23) = 81*4.52 = 366

Example-4

Line	
01	SUBROUTINE SORT(X,N)
02	INTEGER X(100), N, I, J, SAVE, IM1
03	C THIS ROUTINE SORTS ARRAY X INTO ASCENDING
04	C ORDER
05	IF (N.LT.2) GOTO 200
06	DO 210 I = 2, N
07	IM1 = I-1
08	DO 220 J = 1, IM1
09	IF (X(1) .GE. X(J)) GOTO 220
10	SAVE = X(1)
11	X(I) = X(J)
12	X(J) = SAVE
13	220 CONTINUE
14	210 CONTINUE
15	200 RETURN
16	END

Example-4

Operator	Occurrences	Operands	Occurrences
SUBROUTINE	1	SORT	1
()	10	X	8
,	8	N	4
INTEGER	1	100	1
IF	2	I	6
.LT.	1	J	5
GOTO	2	SAVE	3
DO	2	IM1	3
=	6	2	2
-	1	200	2
.GE.	1	210	2
CONTINUE	2	1	2
RETURN	1	220	3
End of line	13	-	-
$n_1 = 14$	$N_1 = 51$	$n_2 = 13$	$N_2 = 42$

Example-4

$N_1 = 51$; $N_2 = 42$; Calculate

- Program Length
- Vocabulary
- Program Volume
- Estimated Statement Count
- Predicted Length
- Potential Volume
- Program Level
- Estimated Level
- Difficulty
- Estimated Difficulty
- Effort
- Time

Example-4

$N_1 = 51; \quad N_2 = 42$

Program Length = $N_1 + N_2 = 93$

Vocabulary, $n = n_1 + n_2 = 14+13 = 27$

Program Volume, $V = N * \log_2 n = 93 * \log_2 27 = 442$ bits

Estimated Statement Count $S_s = N/C = 93/7=13$

- C is constant 7 for FORTRAN

Example-4

$N_1 = 51;$ $N_2 = 42$

Predicted Length, $N\hat{} = (n_1 \log_2 n_1) + (n_2 \log_2 n_2)$

$= 14 * \log_2 14 + 13 * \log_2 13$

$= 14 * 3.81 + 13 * 3.70 = 101.45$

n_2^*,Unique input output parameter= 3

- X: array holding the integer to be sorted, used as an input & output
- N: the size of the array to be sorted

Example-4

Potential volume, program with min. size ,

- $V^* = (2 + n_2^*) \log_2 (2 + n_2^*)$

  $= 5 \log_2 5 = 11.6$

Program Level, $L = V^*/V$

$= 11.6/442 = 0.026$

Estimated Level, $L^\wedge = 2 n_2 / (n_1 N_2) = (2*13)/(14*42) = 0.044$

Difficulty, $D = 1/L = 1/0.026 = 38.5$

Estimated Difficulty, $D^\wedge = 1/L^\wedge = 1/0.044 = 22.72$

Example-4

Effort, E = V/ L^ = V * D^
= 442/0.044
= 10,045

Time, T = E/â
= 10045/18
= 558 sec.
= 10 minutes

Advantages & Drawbacks of Halstead

- **Advantages of Halstead :**

  Do not require in-depth and control flow analysis of Program.

  Predicts Effort, rate of error and time.

  Useful in scheduling projects.

  Simple to calculate

  Measure overall quality of programs

  Predicts maintenance effort.

- **Drawbacks of Halstead :**

  It depends on usage of operator and operands in completed code.

  It has no use in predicting complexity of program at design level.

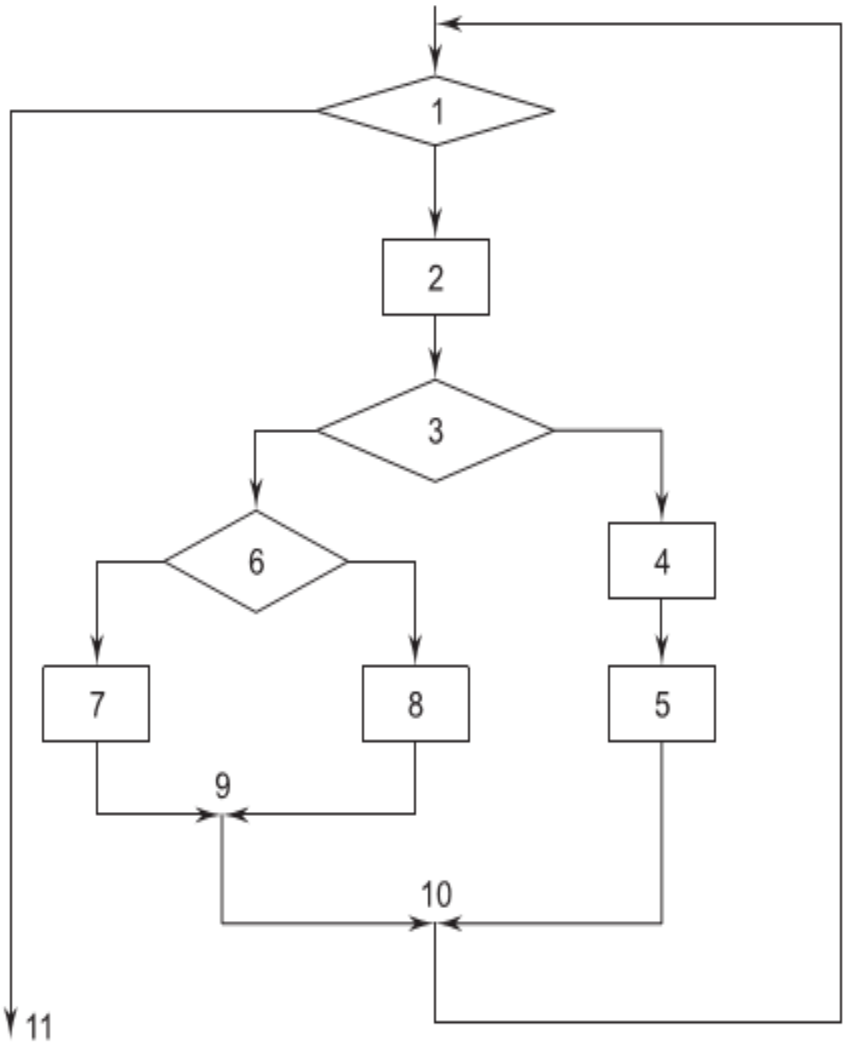
Primitive Measures of Halstead

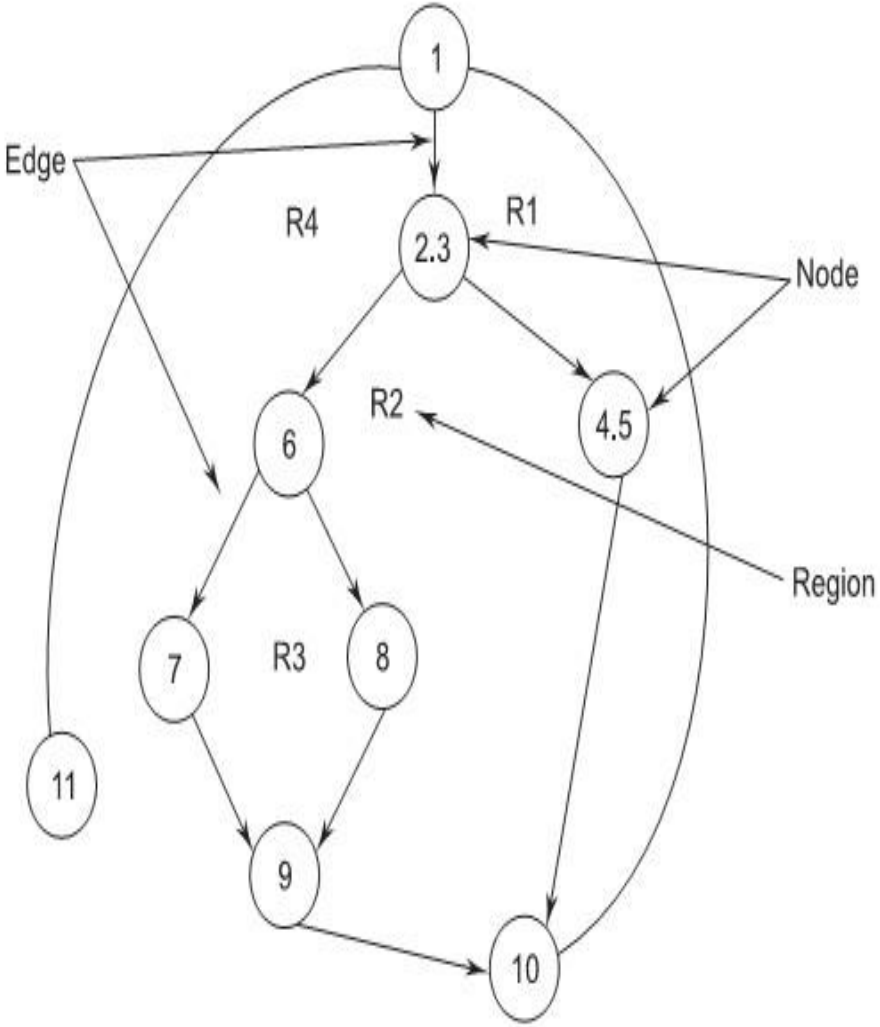
- Halstead used these Primitive Measures to develop expression for-
  - Overall program length
  - Potential minimum volume for an algorithm
  - Actual volume(no of bits required to specify a program)
  - Program level
  - Language level

CYCLOMATIC COMPLEXITY

- Cyclomatic Complexity is a software metric that Provides a **Quantitative Measure of the logical Complexity of a Program.**
- When used in the context of the basis Path-testing method, the value computed for cyclomatic complexity defines the **Number of independent Paths in the basis set of a program**, and provides us with an upper bound for the **Number of Tests** that must be conducted to ensure that all statements have been executed at least once.

CYCLOMATIC COMPLEXITY (Flow-chart & Flow-graph)





Cyclomatic Complexity Computation

- **Cyclomatic Complexity has a foundation in graph Theory and is computed in one of three ways-**

  1-The **Number of Regions** corresponds to the cyclomatic complexity.

  2-Cyclomatic Complexity, **V(G)**, for a flow-graph G, is defined as-

  - Where-
    - **E = Number of flow-graph edges**
    - **N = Number of flow-graph nodes.**

$$V(G) = E - N + 2,$$

  3-Cyclomatic complexity, V(G) for a flow-graph G, is also defined as **V (G) = P + 1,**

  - Where P = Number of predicate nodes (decision) contained in flow graphs G.

Example

- **A Set of independent paths for the above given flow graph-**
  - **Path 1: 1-11.**
  - **Path 2: 1-2-3-4-5-10-1-11.**
  - **Path 3: 1-2-3-6-8-9-10-1-11.**
  - **Path 4: 1-2-3-6-7-9-10-1-11.**
- Note that each new Path introduces a **New Edge**.
- The Path 1-2-3-4-5-10-1-2-3-6-8-9-10-1-11 is not considered to be an **independent path** because it is simply a **combination of already specified paths** and does not traverse any new edges.

Example

- **The flow-graph has Four Regions-**

  V(G) = 11 edges - 9 nodes + 2 = 4.

  V(G) = 3 predicate nodes + 1 = 4.

- Therefore, the cyclomatic complexity of the flow-graph is 4.
- Cyclomatic complexity V(G) of a flow graph G is equal to the number of predicate (Decision) nodes plus one. **V(G)=P+1, P=**Where is the number of predicate nodes contained in the flow graph G

CYCLOMATIC COMPLEXITY

- Cyclomatic Complexity Comes under White Box Testing. **(Introduced by Thomas McCabe in 1976).**
- It's used to **Measure the Complexity of the Software Process.**
- It's used to measure the **How Many no of Test Cases are used to test the application in all Possible Ways.**

Cyclomatic Complexity(Value)	Complexity level and Risk
1-10	a simple program, without much risk
11-20	more complex, moderate risk
21-50	more complex, moderate risk
Greater than 50	untestable program , very high risk

Control Flow Graph (CFG)
How to Draw Control Flow Graph?

- A control flow graph (CFG) describes-

  The **Sequence** in which **different instructions of a Program get Executed.**

  The **Way Control Flows through the Program**.

- **Number all the Statements of a Program.**
- Numbered Statements:

  Represent Nodes of the Control Flow Graph.

- An Edge From one Node to Another Node exists-

  If execution of the statement Representing the first node.

  - Can result in transfer of control to the other node.

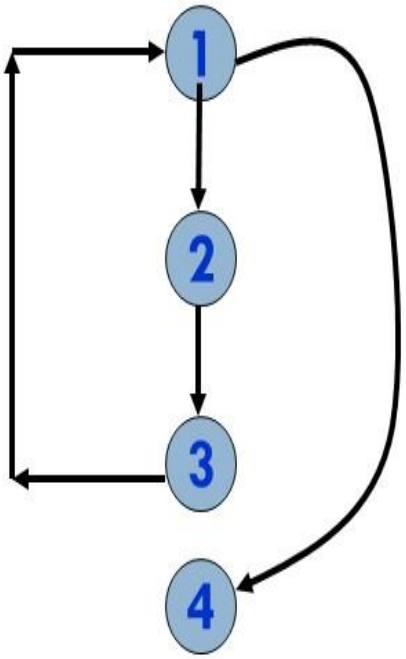
How to Draw Control flow graph?

Sequence:

- 1 a=5;
- 2 b=a*b-1;

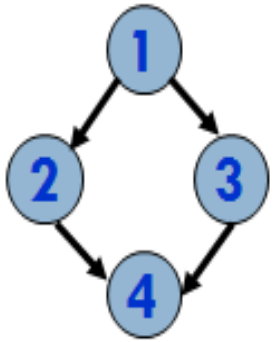
Iteration:

- 1 while(a>b){
- 2 b=b*a;
- 3 b=b-1;}
- 4 c=b+d;



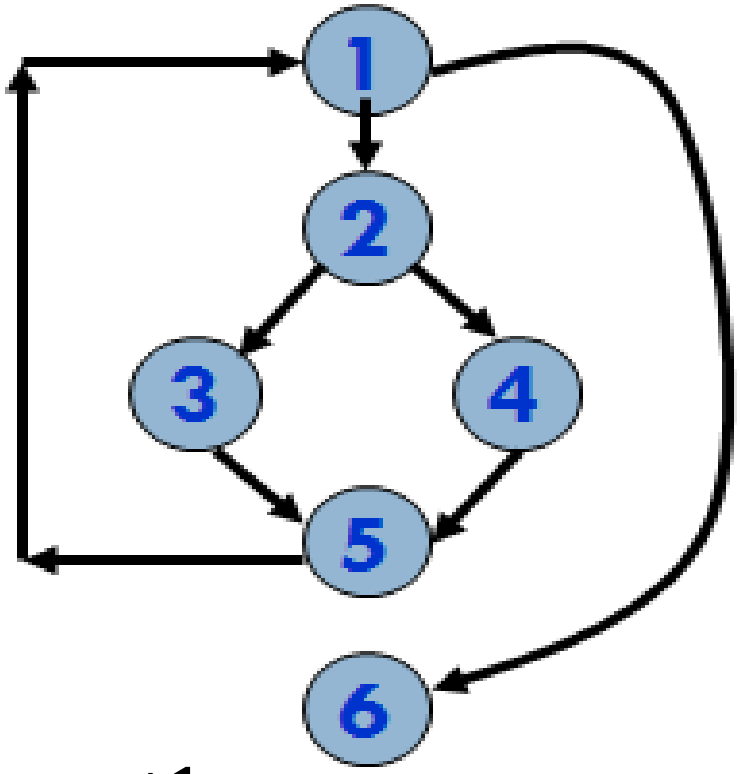
Selection:

- 1 if(a>b) then
- 2 c=3;
- 3 else c=5;
- 4 c=c*c;



How to Draw Control flow graph?

- 1 while (x != y){
- 2 if (x>y) then
- 3 x=x-y;
- 4 else y=y-x;
- 5 }
- 6 return x;



1-V(G) = Total number of bounded areas +1

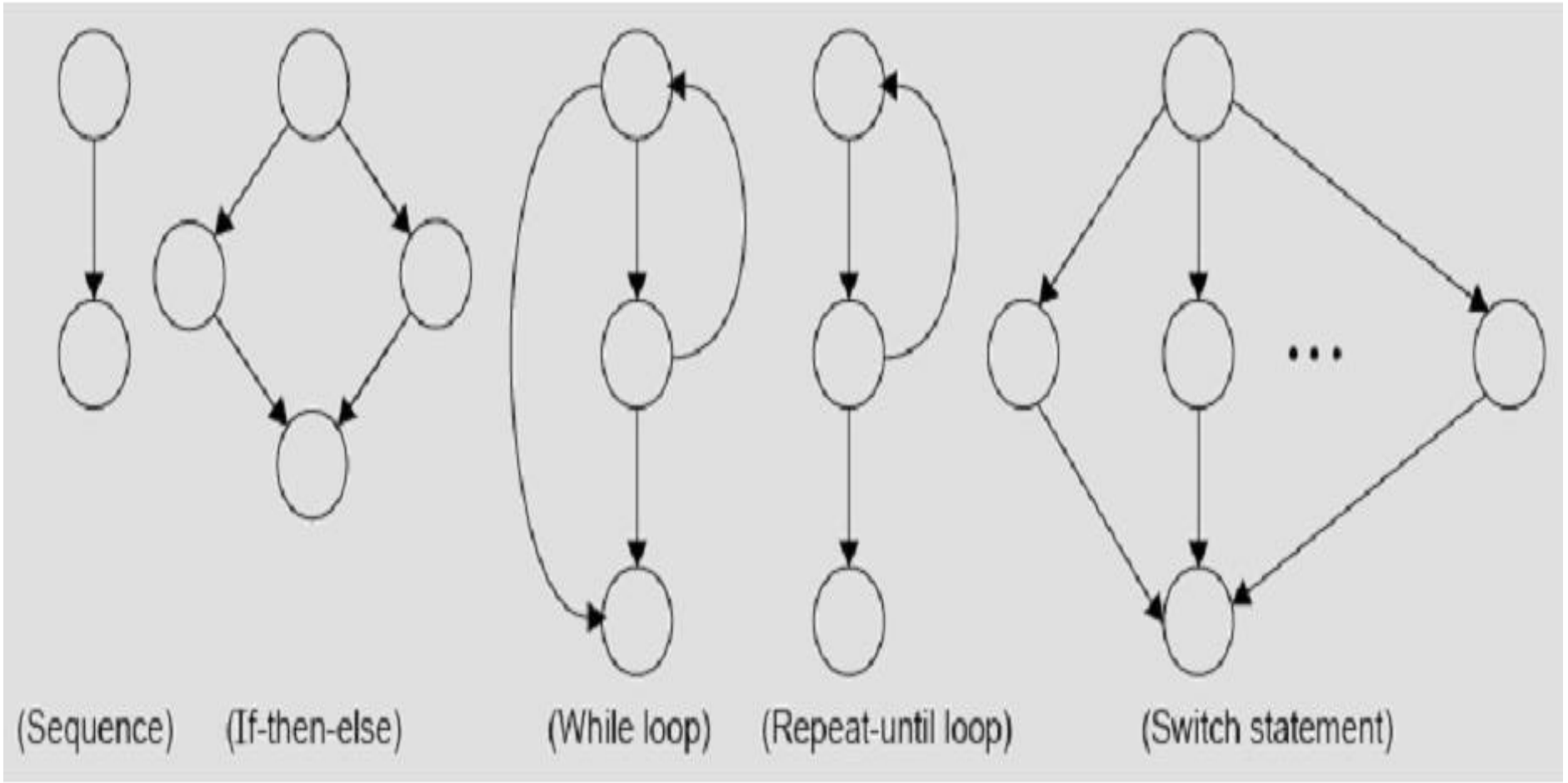
Any Region Enclosed by a nodes and edge sequence.

the number of bounded areas is 2.

cyclomatic complexity = 2+1=3.

2-Cyclomatic Complexity =7-6+2 = 3

Flow Graph


(Sequence)
(If-then-else)
(While loop)
(Repeat-until loop)
(Switch statement)

Flow Graph

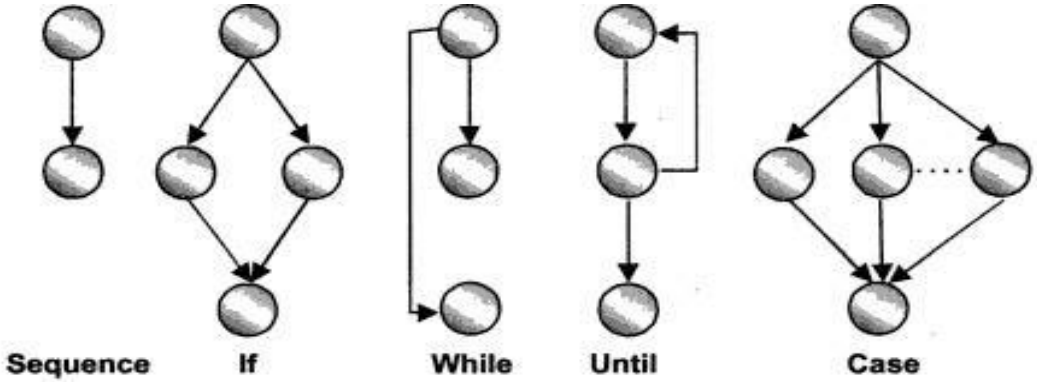


Figure 6.24 Flow Graph Notation

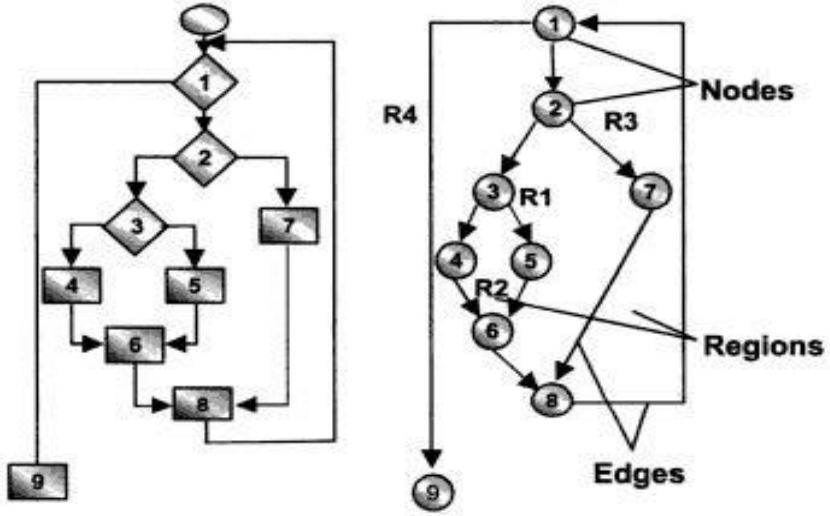
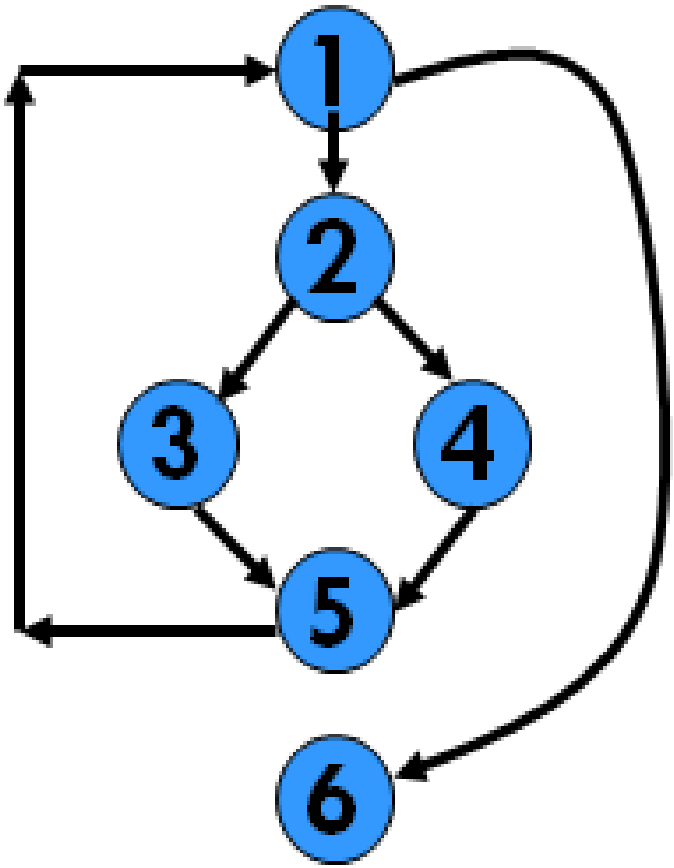


Figure 6.25 Creating Flow Graph

Example-1

```
1 while (x != y){
2    if (x>y) then
3         x=x-y;
4    else y=y-x;
5 }
6 return x;
```



Derivation of Test Cases-

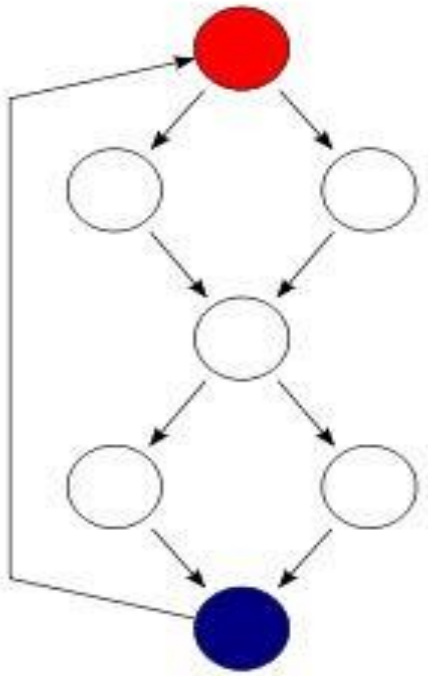
Number of independent paths: 3

- 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)

Example-2

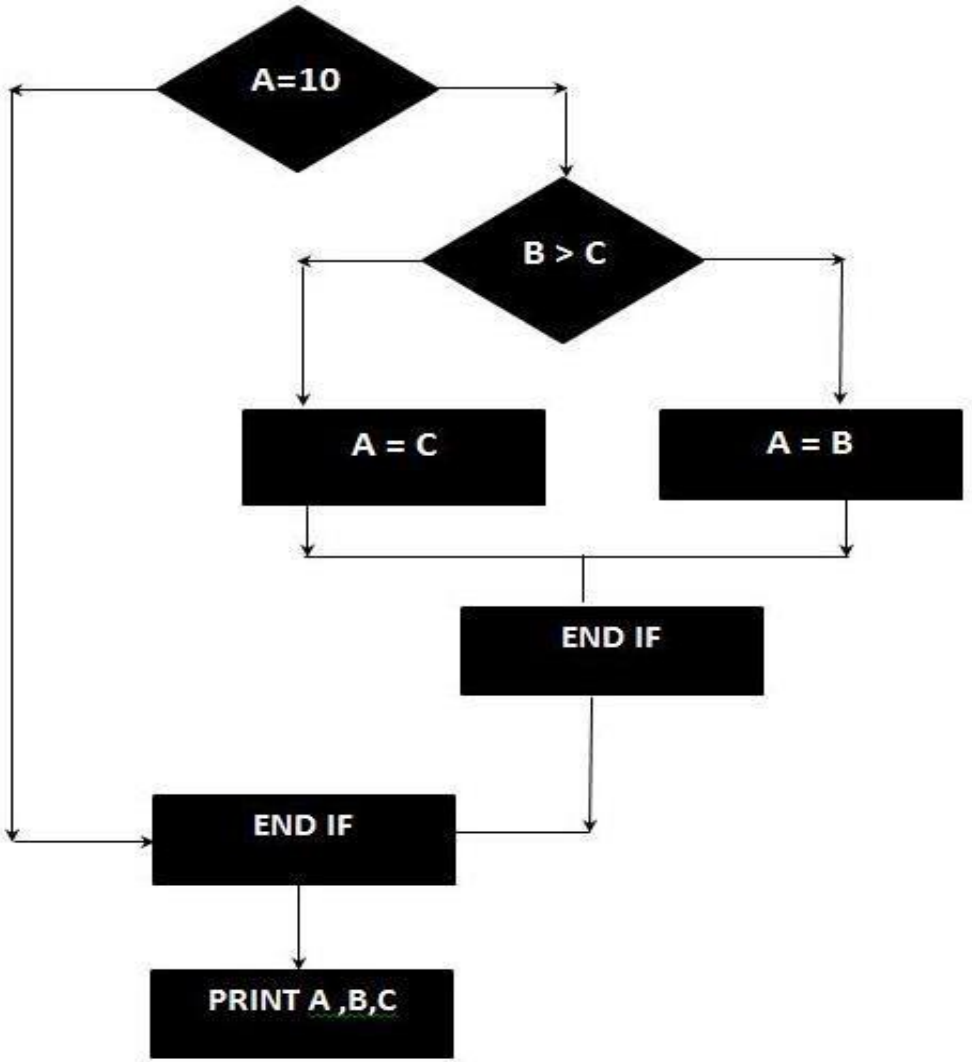
```
if( c1() )
    f1();
else
    f2();

if( c2() )
    f3();
else
    f4();
```



Example-3

```
IF A = 10 THEN
 IF B > C THEN
   A = B
 ELSE
   A = C
 ENDIF
ENDIF
Print A
Print B
Print C
```



Example-4

1. ***Draw the flow graph of the program:*** A flow graph is constructed using symbols previously discussed. For example, a program to find the greater of two numbers is given below.

```
procedure greater;
integer: a, b, c = 0;
1   enter the value of a;
2   enter the value of b;
3   if a > b then
4   c = a;
    else
5   c = b;
6   end greater
```

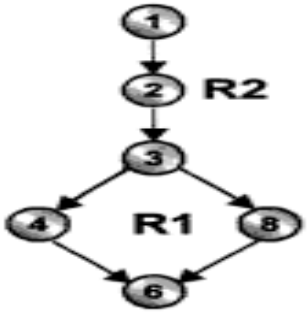


Figure 6.26
Flow Graph to Find the Greater between Two Numbers

The flow graph for the above program is shown in Figure 6.26.

2. ***Determine the cyclomatic complexity of the program using the flow graph:*** The cyclomatic complexity for flow graph depicted in Figure 6.26 can be calculated as follows:

CC = 2 regions
Or
CC = 6 edges - 6 nodes + 2 = 2
Or
CC = 1 predicate node + 1 = 2

3. ***Determine all the independent paths present in the program using the flow graph:*** For the flow graph shown in Figure 6.26, the independent paths are:

P1 = 1-2-3-4-6

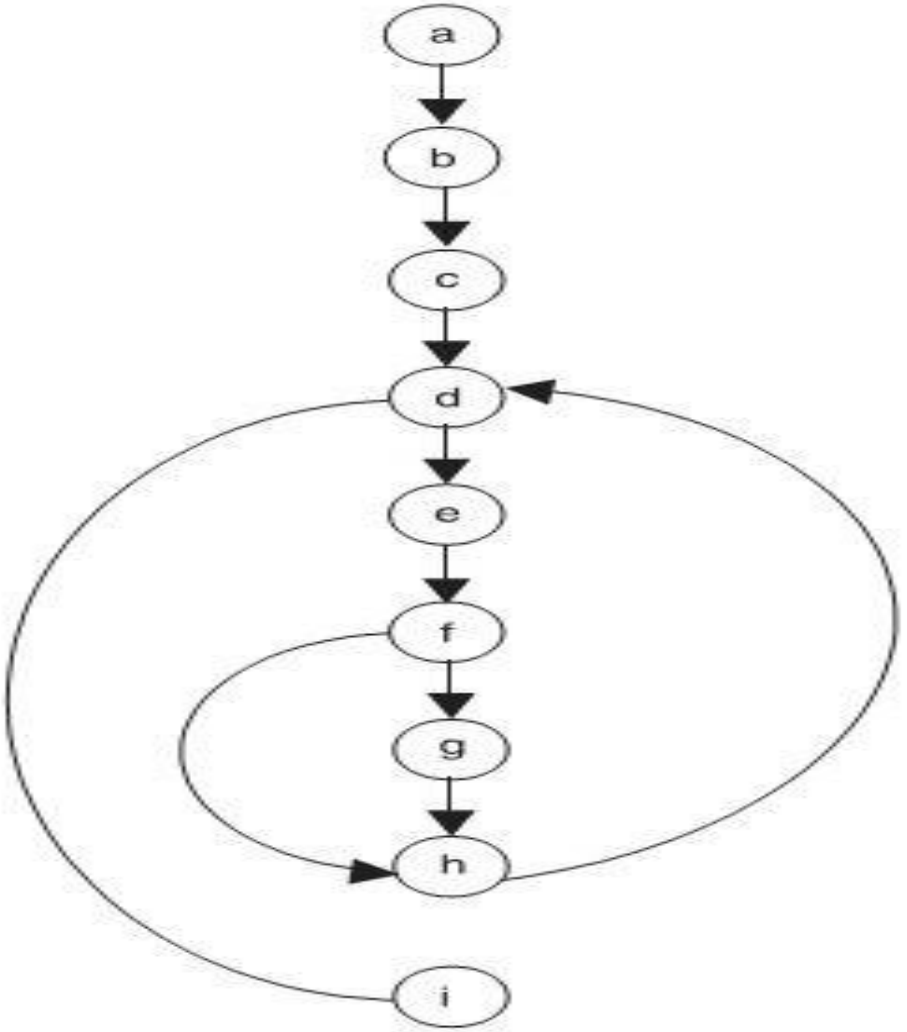
P2 = 1-2-3-5-6

4. ***Prepare test cases:*** Test cases are prepared to implement the execution of all the independent paths in the basis set. Each test case is executed and compared with the desired results.

Example-5

a. Read N
b. MAX = 0
c. I = 1
d. While I <= N
e. Read X(I)
f. If X(I) > MAX
g. THEN MAX = X(I)
h. I = I + 1
i. PRINT MAX

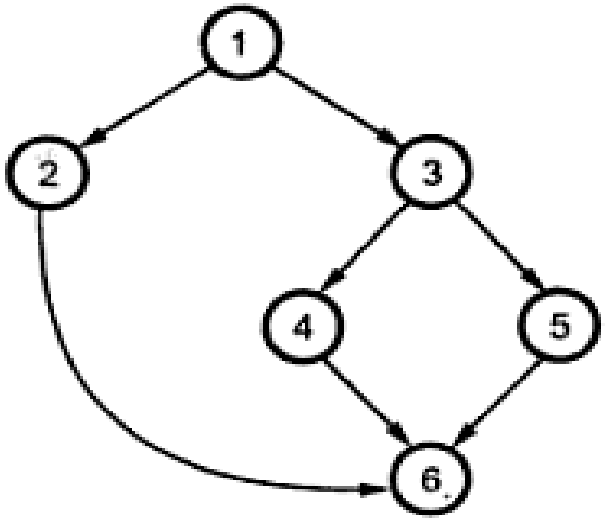
(*a*) Program Logic



(*b*) Program Flow Graph

Example-6

```
{
1.  if  (a  <  b)
2.  F1  (  )  ;
    else
    {
3.  if  (a  <  c)
4.  F2  (  )  ;
    else
5.  F3  (  );
    }
6.  }
```



Cyclomatic Complexity

- **Cyclomatic Complexity of a Program:**

  Also indicates the Psychological Complexity of a Program.

  Measures the Difficulty level of understanding the Program.

- Important From **Maintenance Perspective-**

  Limit Cyclomatic Complexity.

  - Of modules to Some Reasonable Value.

  Good Software Development Organizations:

  - Restrict cyclomatic complexity of functions to a maximum of ten or so.