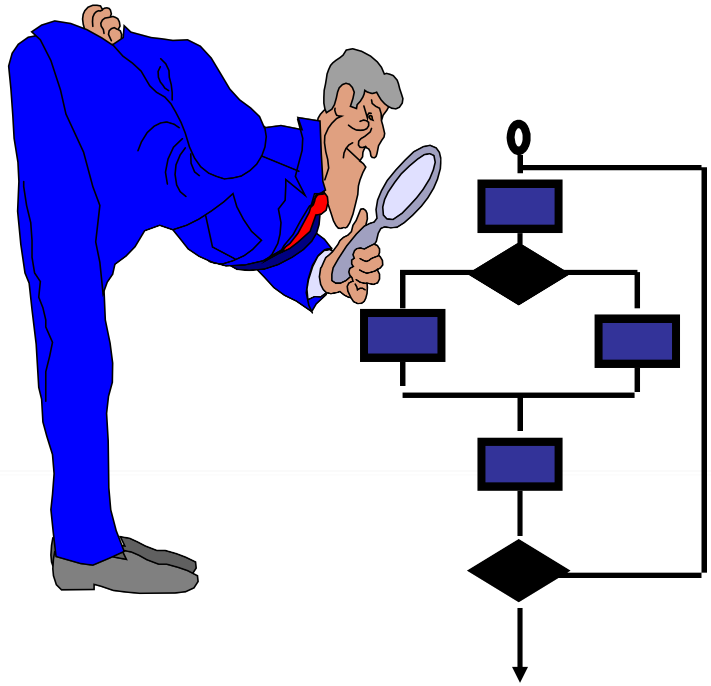
White-Box Testing

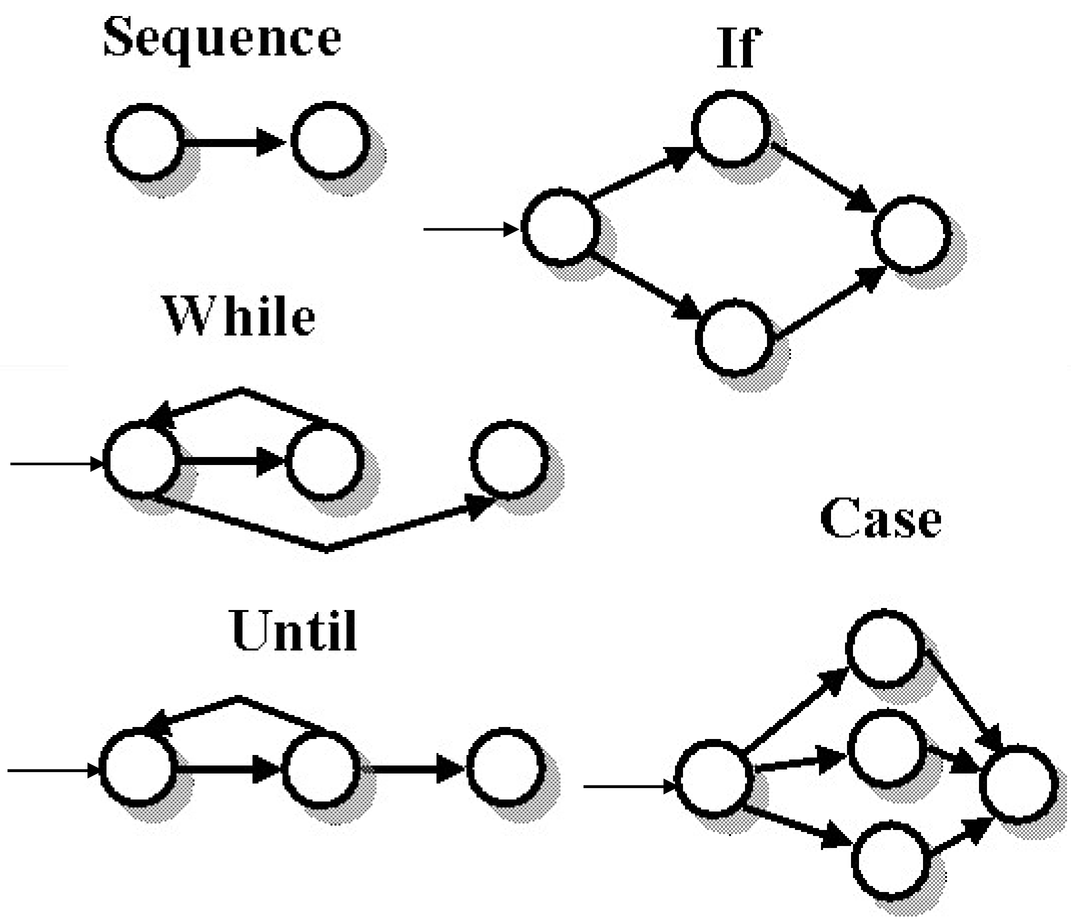


... our goal is to ensure that all statements and conditions have been executed at least once ...

Basis Path testing

- White Box testing technique
- Enables the test case designer to derive a logical complexity measure of a procedural design and use this measure as a guide for defining a basis set of execution paths.
- Guaranteed to execute every statement at least once during testing.

Flow graph notations

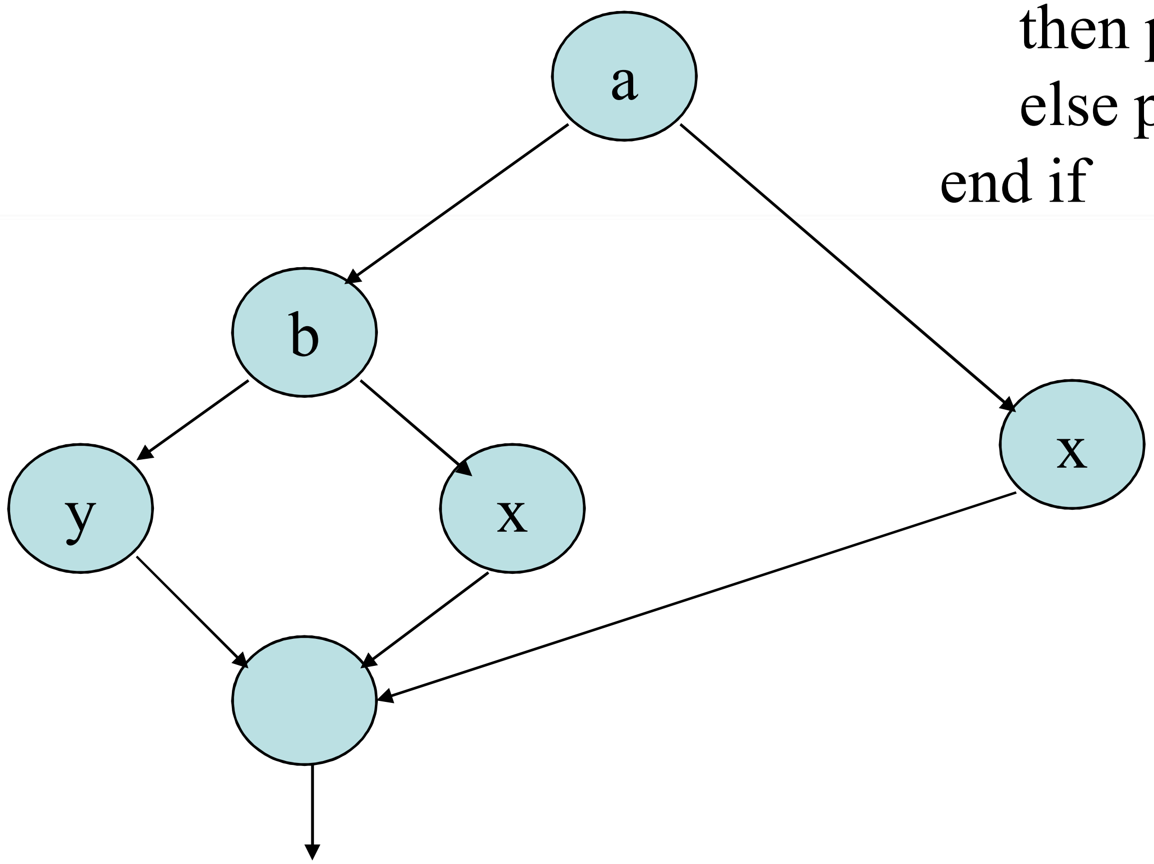


Flow graph notations

- Representation of control flow
- Circle – called a flow graph node- represents one or more procedural statements
- Edges or links – represents flow of control. An edge must terminate at a node even if that node does not represent any procedural statement.
- Regions – areas bounded by edges and nodes are called regions.
- Compound condition – a separate node is created for each of the condition
- Predicate node – each node that contains a condition. Characterized by two or more edges emanating from it.

Compound logic

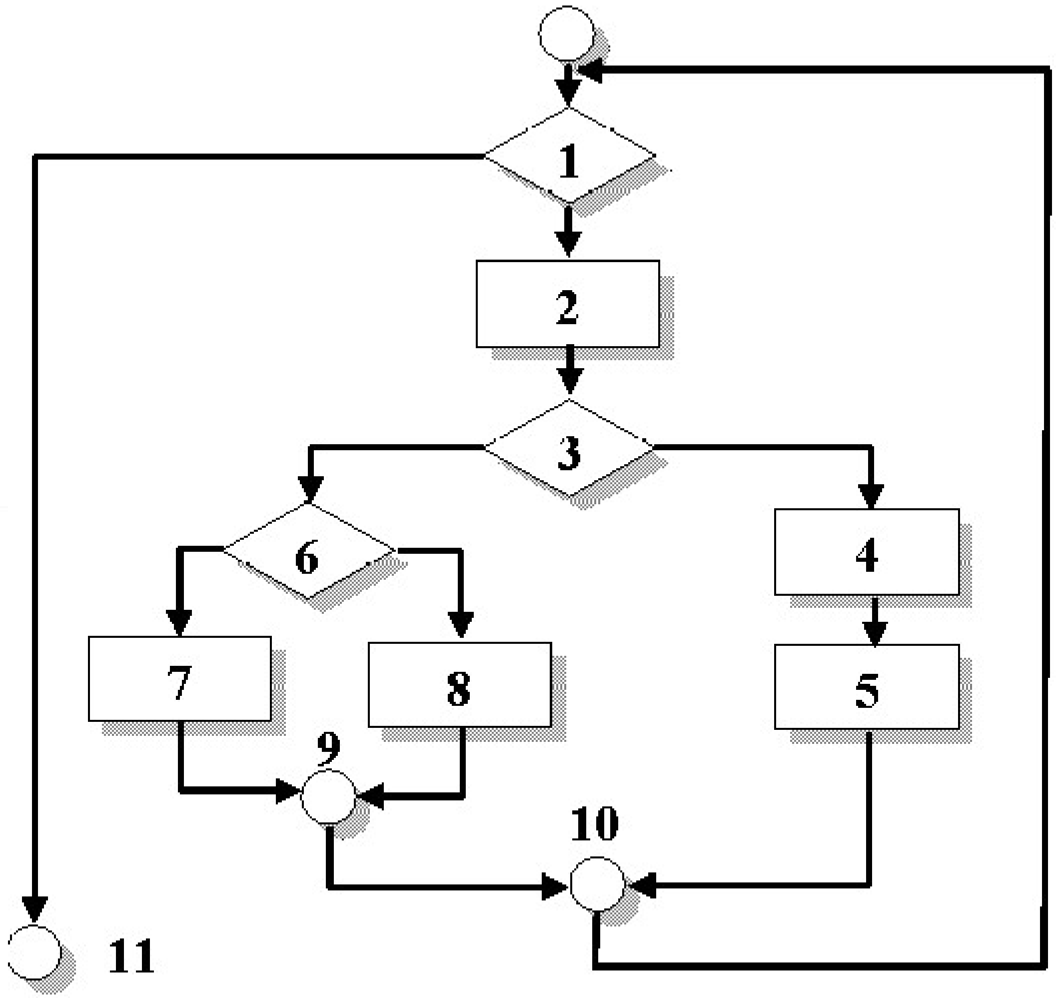
if a OR b
    then procedure x
    else procedure y
end if



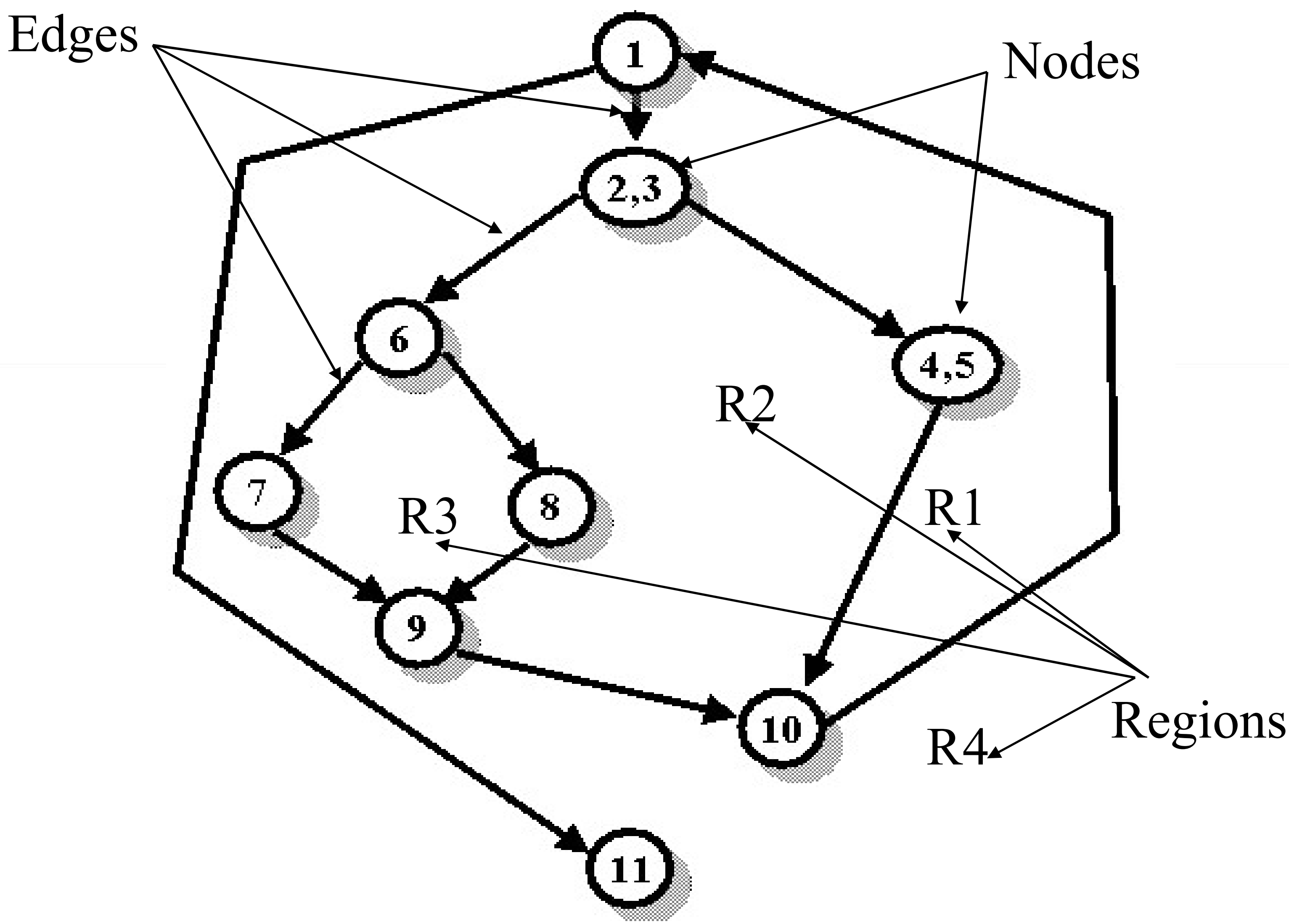
Independent Program Paths

- Any path through the program that introduces at least one new set of processing statements or a new condition.
- 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

Flow Chart



Flow Graph

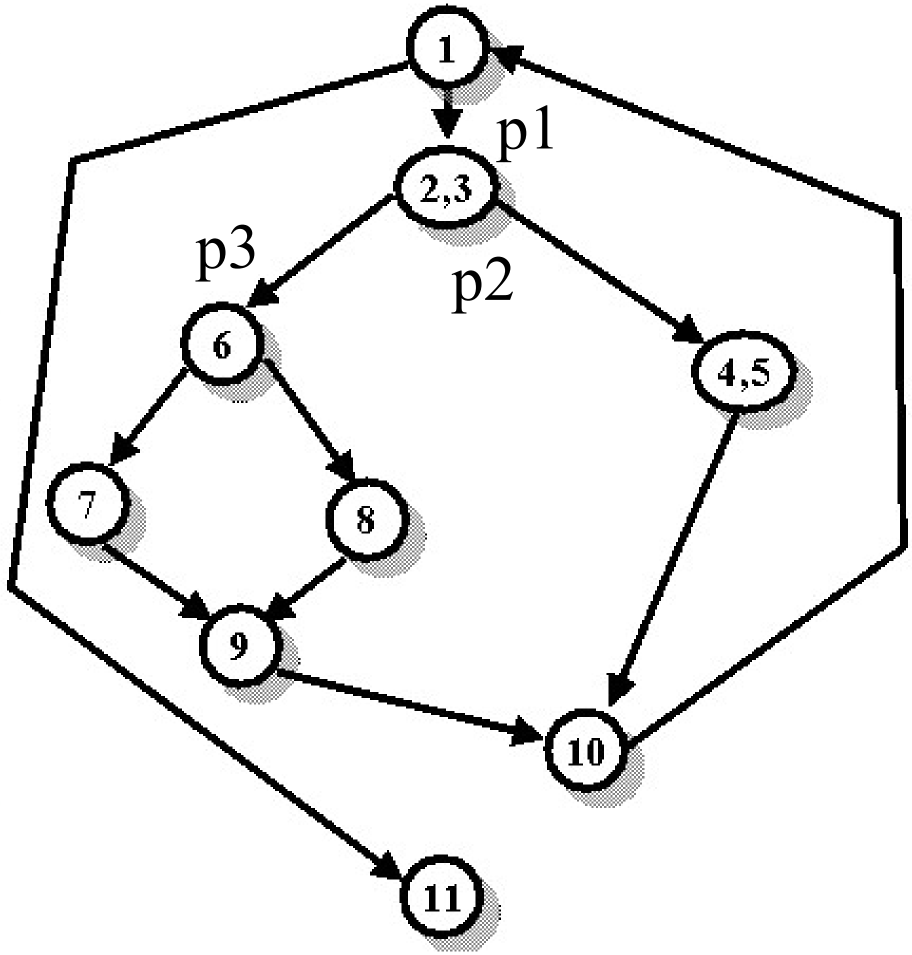


Cyclomatic complexity

- A software metric that provides a quantitative measure of the logical complexity of a program.
- When used in the context of basis path testing, it defines the number of independent paths in the basis set of a program.
- Cyclomatic complexity is computed as

1. The number of regions correspond.
2. For a flow graph G, Cyclomatic complexity V(G) is defined as V(G) = E-N+2 where E is number of edges and N is number of nodes.
3. For a flow graph G, Cyclomatic complexity V(G) is defined as V(G) =P+1 where P is number of predicate nodes.

Cyclomatic Complexity



1. V(G)=Number of Regions = ***4***
2. V(G) =E-N+2

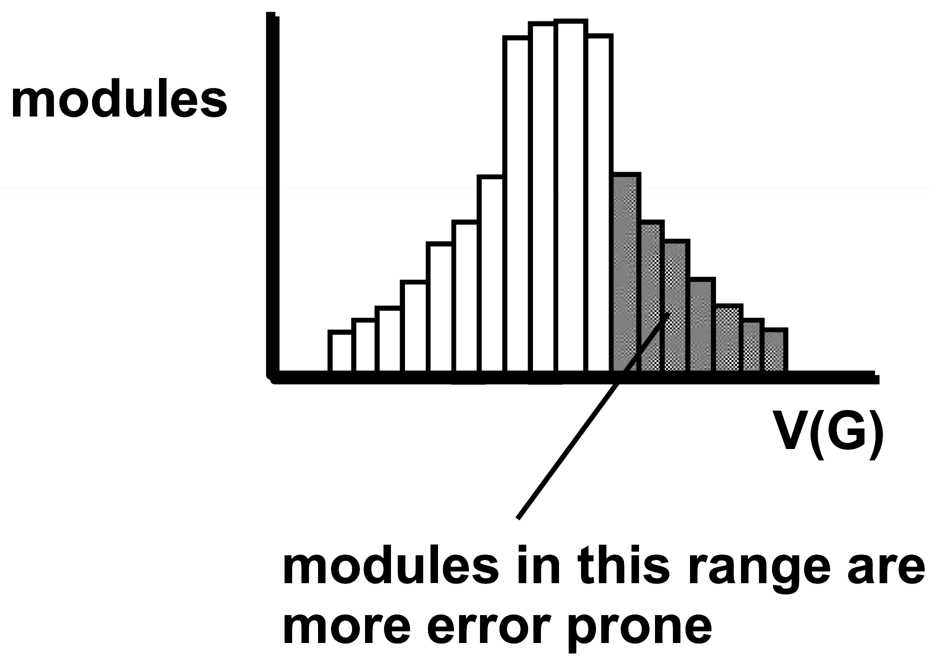
   =11-9+2

   = ***4***
3. V(G) = P+1

   = ***4***

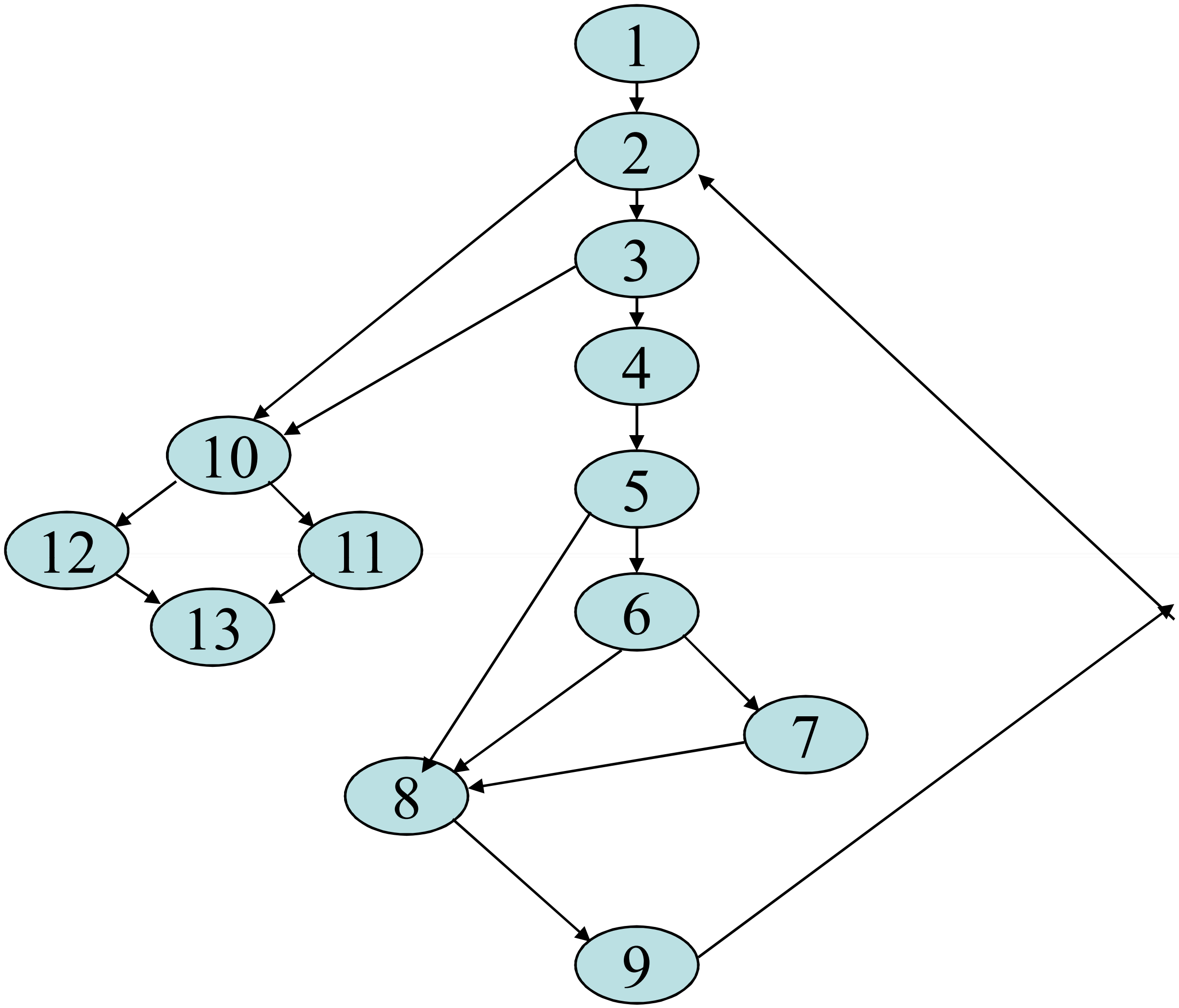
Cyclomatic Complexity

A number of industry studies have indicated that the higher V(G), the higher the probability or errors.

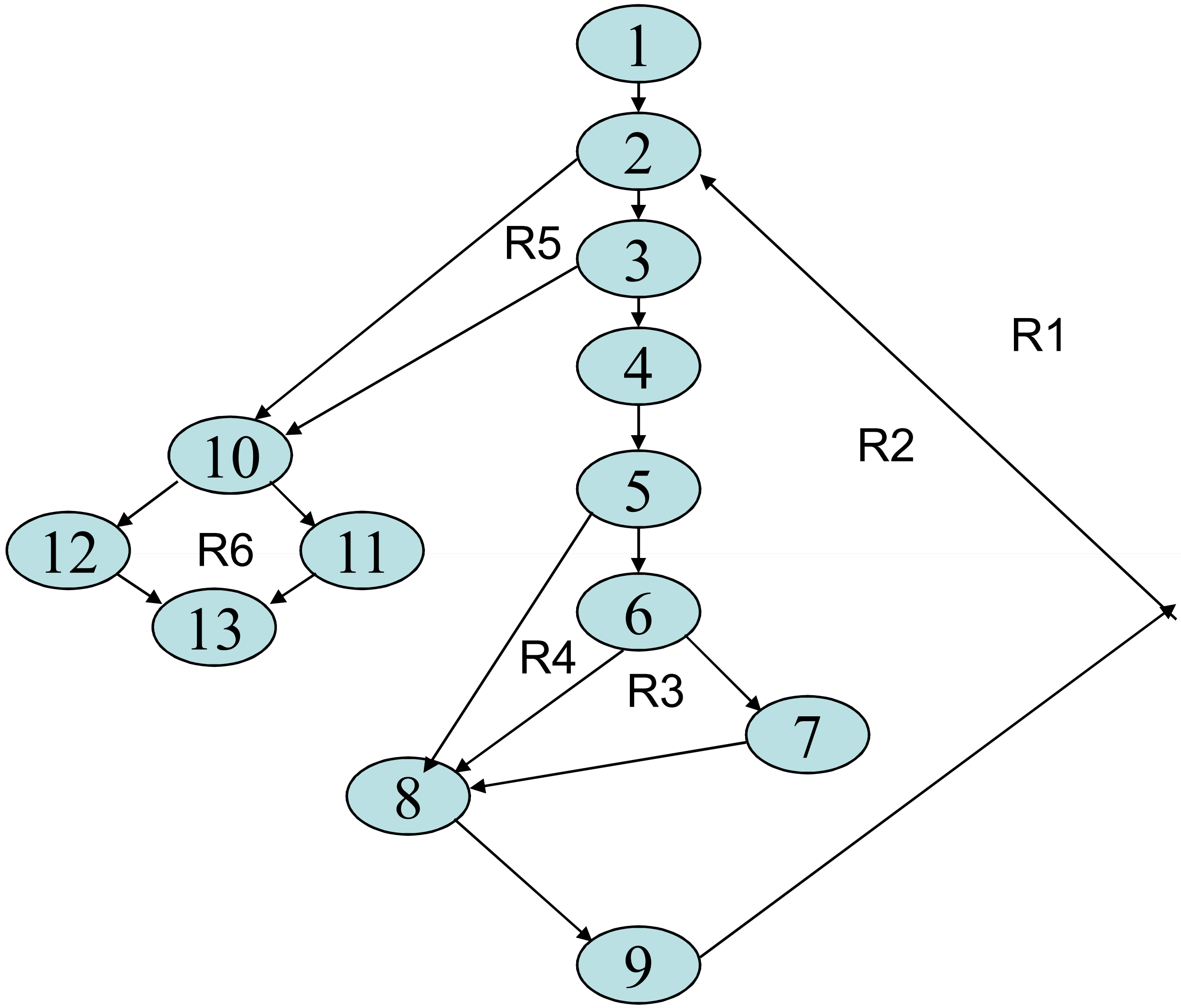


```
PROCEDURE average;
INTERFACE RETURNS average, total.input, total.valid;
INTERFACE ACCEPTS values, minimum, maximum;
Minimum, maximum, sum IS SCALER;
TYPE i IS INTEGER;
    i=1;
1   total.input = total.valid=0;          2
    sum=0;
    DO WHILE value[i] <> -999 AND total.input < 100          3
4   Increment total.input by 1;          5
    IF value[I] >= minimum AND value[i] <= maximum          6
                THEN total.valid by 1;
                        sum = sum + value[I];          7
                ELSE skip
8   ENDIF
    Increment i by 1;
    ENDDO          9
    IF total.valid > 0          10
      THEN average= sum/total.valid;          11
       ELSE average = -999;          12
    ENDIF
    END average
```

Flow Graph

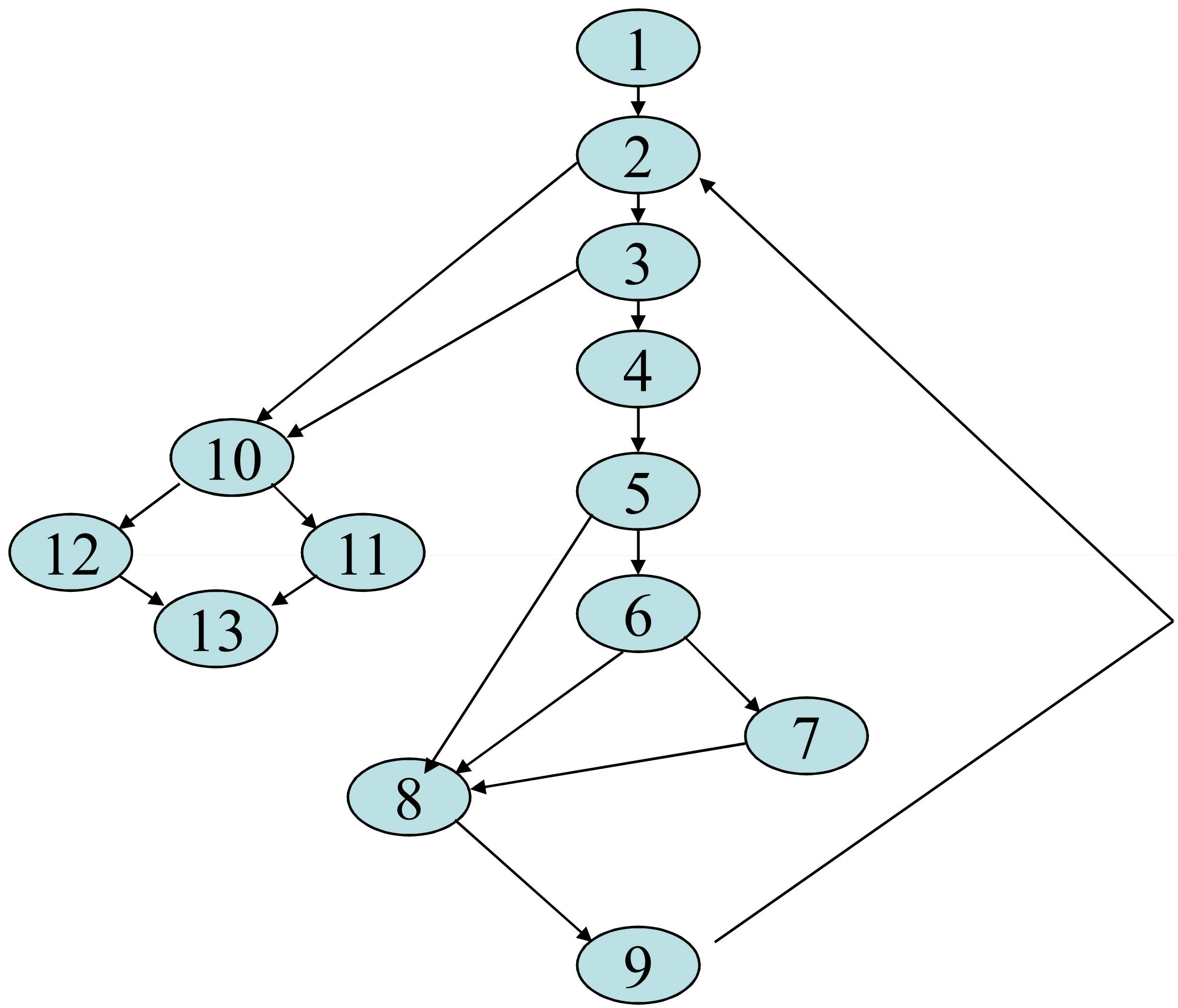


Cyclomatic Complexity



Cyclomatic Complexity = Number of regions = 6

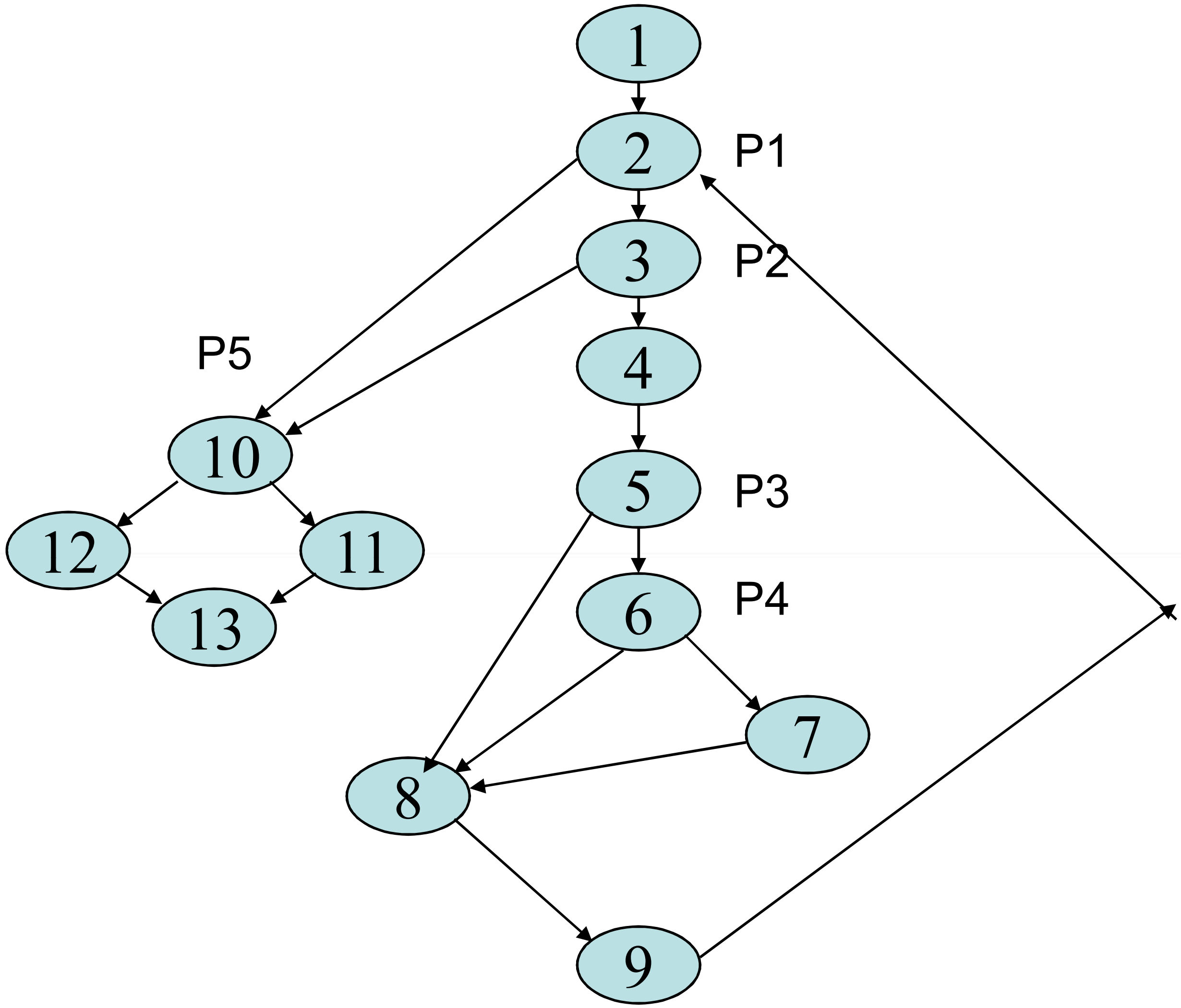
Cyclomatic Complexity



E1 -> 1-2
E2 -> 2-3
E3 -> 3-4
E4 -> 4-5
E5 -> 5-6
E6 -> 5-8
E7 -> 6-8
E8 -> 6-7
E9 -> 7-8
E10 -> 8-9
E11 -> 9-2
E12->2-10
E13-> 3-10
E14->10-11
E15->10-12
E16->12-13
E17->11-13

Cyclomatic complexity V(G)= E-N+2 = 17 -13 + 2 = 6

Cyclomatic Complexity



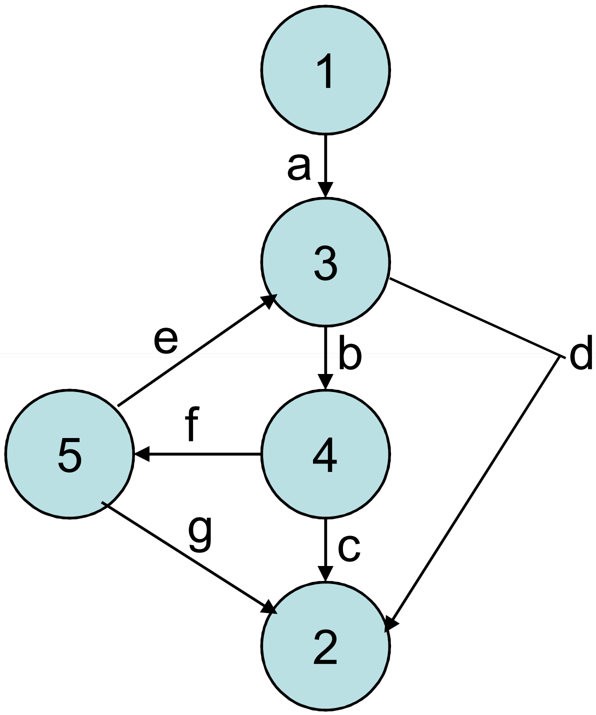
Cyclomatic Complexity V(G)=P+1 = 5 +1 = 6

Basis Set

- Path 1: 1-2-10-11-13
- Path 2: 1-2-10-12-13
- Path 3: 1-2-3-10-11-13 / 1-2-3-10-12-13
- Path 4: 1-2-3-4-5-8-9-2-…
- Path 5: 1-2-3-4-5-6-8-9-2-…
- Path 6: 1-2-3-4-5-6-7-8-9-2-…

Graph Metrices

- Flow Graph



Graph Metrix

node \ Connected to node	1	2	3	4	5
1			**a**		
2					
3		**d**		**b**	
4		**c**			**f**
5		**g**	**e**		

Graph Metrices

- A square matrix whose number of rows and columns are equal to number of nodes in the flow graph
- Each row and column corresponds to identified node
- Each matrix entry corresponds to an edge(I.e. connections between nodes)
- Adding a *link weight* to each matrix entry , it can become a powerful tool for evaluating program control structure during testing.
- Link weight can be *1* or *0*
- *Link weights* can be assigned other interesting properties
  1. *Probability that a link (edge) will execute*
  2. *Processing time expended during traversal of a link*
  3. *Memory required during traversal of a link*
  4. *Resources required during traversal of a link*

```java
package com.codign.sample.pathexample;

public class PathExample {

    public int returnInput(int x, boolean condition1,
                                  boolean condition2,
                                  boolean condition3) {
        if (condition1) {
            x++;
        }
        if (condition2) {
            x--;
        }
        if (condition3) {
            x=x;
        }
        return x;
    }
}
```

Control Structure testing: Condition Testing

- Exercise the logical conditions contained in a program module.
- E1 <relational operator> E2
- Elements in a condition includes
  - Boolean operator
  - Boolean variable
  - A pair of parenthesis
  - A relational operator
  - An arithmetic expression

Control Structure testing: Condition Testing

- When a condition is incorrect, then at least one component of the condition is incorrect
- The errors include
  - Boolean operator errors
  - Boolean variable errors
  - Parenthesis errors
  - Relational operator errors
  - Arithmetic expression errors

Control Structure testing: Data Flow Testing

- A program unit accepts inputs, performs computations, assigns new values to variables, and returns results.
- One can visualize of "flow" of data values from one statement to another.
- A data value produced in one statement is expected to be used later.
  - Example
    - Obtain a file pointer ……. use it later.
  - If the later use is never verified, we do not know if the earlier assignment is acceptable.
- Two motivations of data flow testing
    - The memory location for a variable is accessed in a "desirable" way.
    - Verify the correctness of data values "defined" (i.e. generated) – observe that all the "uses" of the value produce the desired results.
- Idea: A programmer can perform a number of tests on data values.
  - These tests are collectively known as data flow testing.

Control Structure testing: Data Flow Testing

- Selects test paths of a program according to the locations of definitions and uses of variables in the program
- For a statement with ***S*** as its statement number

DEF(S)= {X|Statement S contains a definition of X)

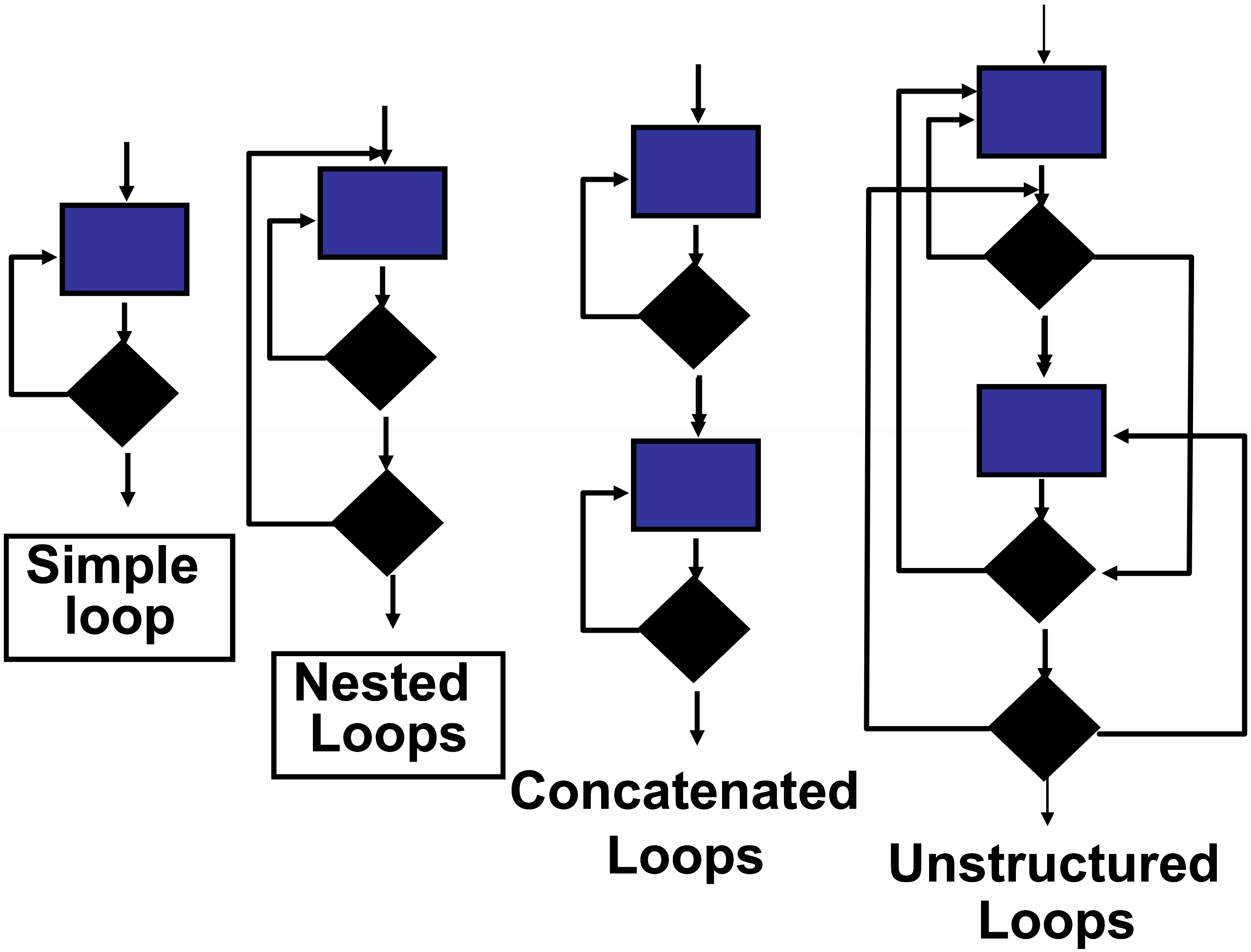
USE(S)={X|Statement S contains a use of X)

assumptions – every statement is assigned a unique Statement number and each functions does not modify its parameters or global variables

If statement S is an If or loop statement then its DEF set is empty and its use set is based on condition of statement S

- The definition of variable X at statement S is said to be live at statement s' if there exists a path from statement S to statement S' that contains no other definition of X
- A DU chain of variable X is of the form [X,S,S'] where S and S' are statement numbers, X is in DEF(S) and the definition of X is in statement S live at statement S'.
- DU testing strategy – every DU chain should be covered at least once.
- DU testing strategy does not guarantee the coverage of all branches of a program. Like in a IF statement whose *then part* does not have a definition of any variable and the else part does not exists.

Loop Testing



Loop Testing: Simple Loops

Minimum conditions—Simple Loops

1. skip the loop entirely
2. only one pass through the loop
3. two passes through the loop
4. m passes through the loop $m < n$
5. (n-1), n, and (n+1) passes through the loop

where n is the maximum number of allowable passes

Loop Testing: Nested Loops

Nested Loops

- Start at the innermost loop. Set all outer loops to their minimum iteration parameter values.
- Test the min+1, typical, max-1 and max for the innermost loop, while holding the outer loops at their minimum values.
- Move out one loop and set it up as in step 2, holding all other loops at typical values. Continue this step until the outermost loop has been tested.

Concatenated Loops

If the loops are independent of one another
then treat each as a simple loop
else* treat as nested loops
endif*

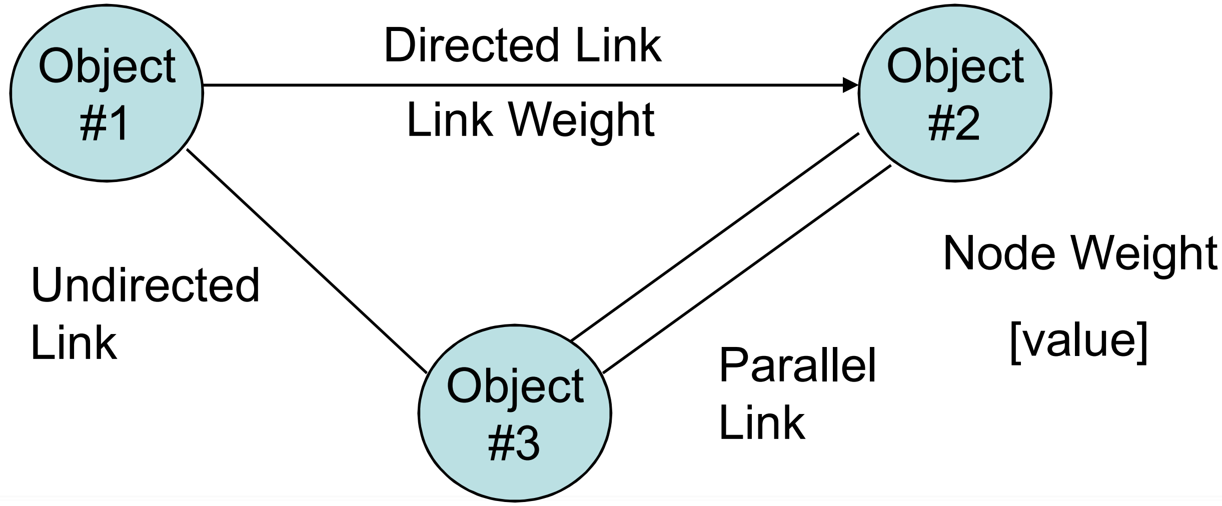
for example, the final loop counter value of loop 1 is used to initialize loop 2.

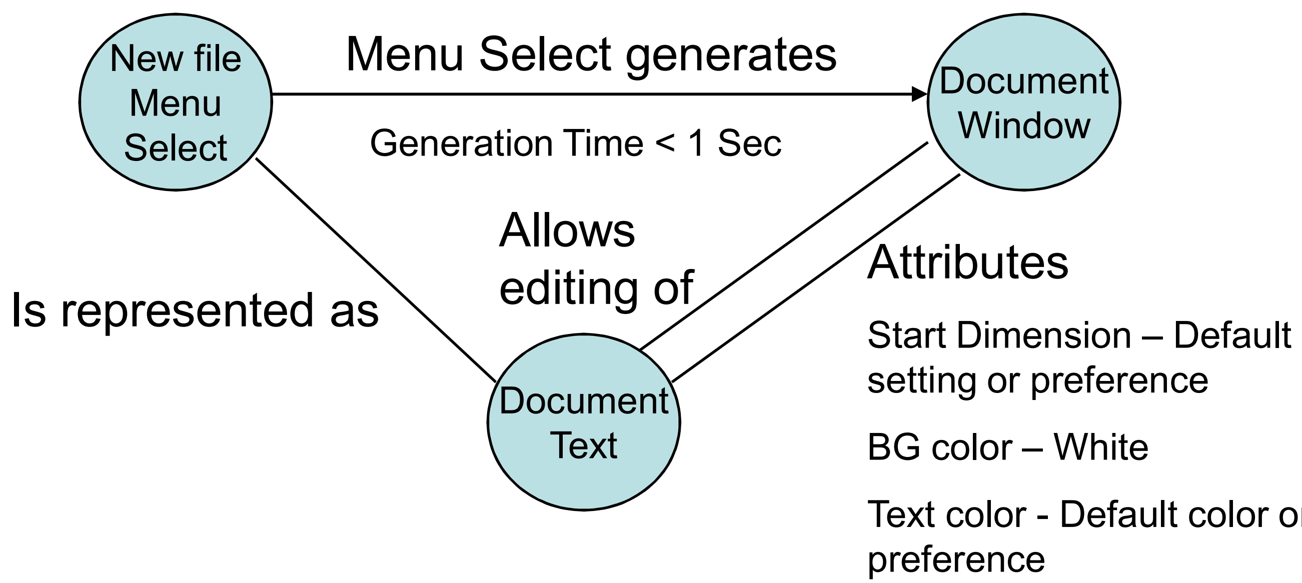
Black Box testing

- Tends to be applied during later stages of testing
- Focused on information domain

Graph Based Testing Methods

- Create a graph of important objects and their relationship.
- Devise a series of test cases that will cover the graph so that each object and relationship is exercised and errors are uncovered.
- Graph –
  - Collection of nodes that represents objects.
  - Links that represents the relationship between the objects.
  - Node weight that describe the properties of a node.
  - Link weight that describe some characteristic of a link.

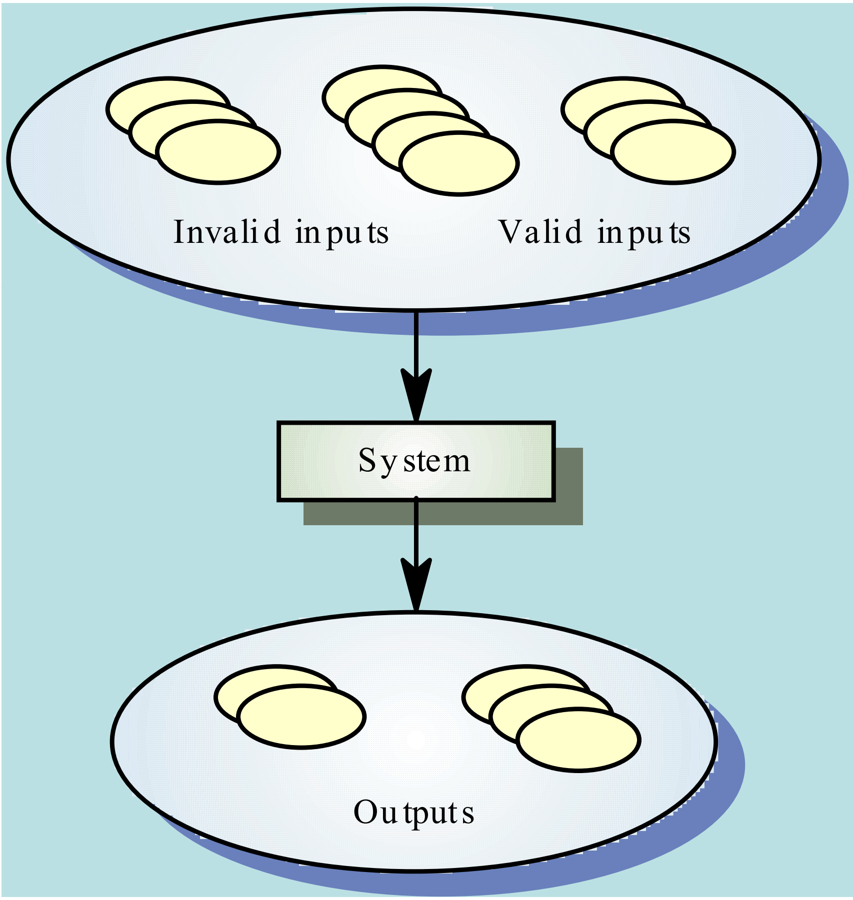

Object #1
Directed Link
Link Weight
Object #2
Undirected Link
Node Weight
[value]
Object #3
Parallel Link


New file Menu Select
Menu Select generates
Generation Time < 1 Sec
Document Window
Allows editing of
Is represented as
Attributes
Start Dimension – Default setting or preference
BG color – White
Text color - Default color or preference
Document Text

Equivalence partitioning

- Input data and output results often fall into different classes where all members of a class are related
- Each of these classes is an equivalence partition where the program behaves in an equivalent way for each class member
- Test cases should be chosen from each partition (or class)

Equivalence partitioning



Guidelines for equivalence classes

1. If an input condition specifies range,
   - one valid and two invalid equivalence classes are needed
2. If a condition requires a specific value,
   - then one valid and two invalid equivalence classes are needed
3. If an input condition specifies a member of a set,
   - one valid and one invalid equivalence class are needed
4. If an input condition is Boolean,
   - one valid and one invalid class are needed

Example: ATM

- Consider data maintained for ATM
  - User should be able to access the bank using PC and modem
  - User should provide six-digit password
  - Need to follow a set of typed commands

Data format

- Software accepts
  - Area code:
    - blank or three-digit
  - Prefix:
    - three-digit number not beginning with 0 or 1
  - Suffix:
    - four digits number
  - Password: six digit alphanumeric value
  - Command:
    - {“check”, “deposit,” “ bill pay”, “transfer” etc.}

Input conditions for ATM

- Input conditions: phone number
  - Area code:
    - Boolean: (the area code may or may not be present)
    - Range: values defined between 200-999
    - Specific value: no value > 905
  - Prefix: range –specific value >200
  - Suffix: value (four-digit length)
  - Password:
    - Boolean: password may or may not be present
    - Valid-value: six char string
  - Command: set containing commands noted previously

Input condition	Example	Test cases
Range of values	the item count can be from 1 to 999 • one valid equivalence class (1 < item count < 999) • two invalid equivalence classes (item count < 1 and item count > 999)	1. Item count = 874 2. Item count = 0 3. Item count = 1234
number of values	one through six owners can be listed for the automobile • one valid equivalence class nbr of owner 1 to 6 • two invalid equivalence classes no owners more than 6 owners	1. Owner list = { A, B} 2. Owner list = Empty 3. Owner List = {A,B,C,D,E,F,G }
a "must be" situation	first character of the identifier must be a letter • one valid equivalence class (it is a letter) • one invalid equivalence class (it is not a letter)	1. Input = "Correct" 2. Input = "5Incorrect"
set of input values where the program handles each differently	type of vehicle must be BUS, TRUCK, TAXICAB, PASSENGER, or MOTORCYCLE • a valid equivalence class for each type of vehicle • an invalid equivalence class no vehicle TRAILER	1. Vehicle = BUS 2. Vehicle = TRUCK 3. Vehicle = TAXICAB 4. Vehicle = PASSENGER 5. Vehicle = MOTORCYCLE 6. Vehicle = Empty 7. Vehicle = TRAILER

Equivalence partitions

Equivalence class	Valid range	Invalid range
Foreground color	Red (1) Blue (2) Black (3) White (4)	Lavendel is a beautiful color but also very long (5) Purple (6)
Background color	Yellow (7) Brown (8)	White (9) Black (10)
Outlining	Left (11) Right (12)	Center (13)

Valid test cases:

	TC 1	TC 2	TC 3	TC 4
Foreground color	Red	Blue	Black	White
Background color	Yellow	Brown	Brown	Brown
Outlining	Left	Right	Right	Right
Covers	1, 7, 11	2, 8, 12	3, 8, 12	4, 8, 12

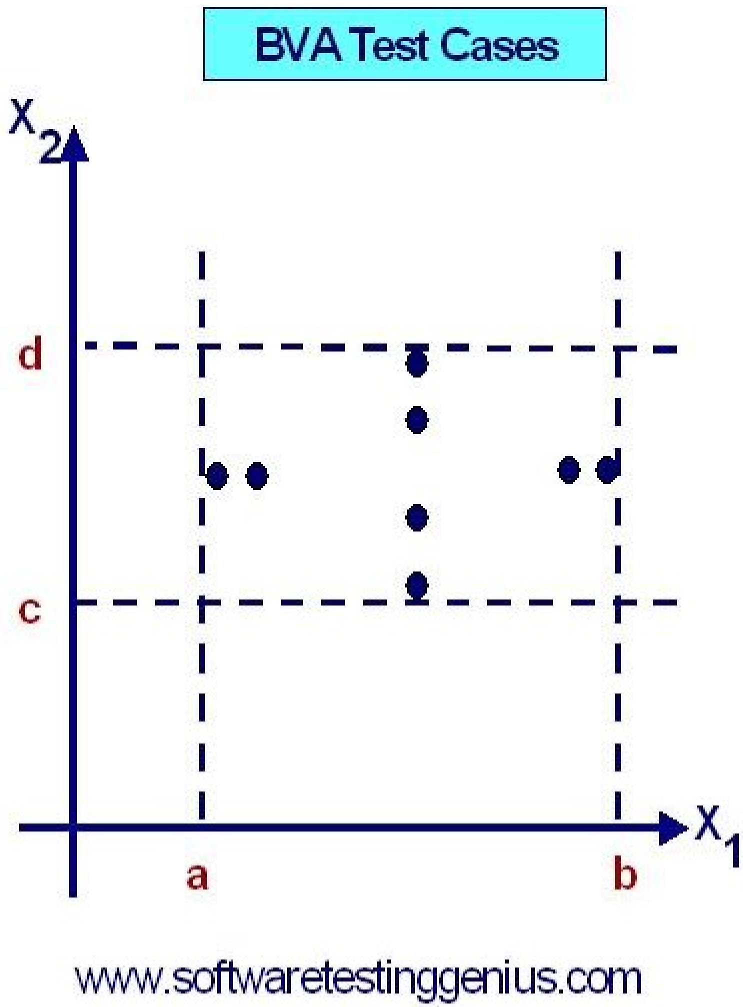
Invalid test cases:

	TC 5	TC 6	TC 7	TC 8	TC 9
Foreground color	Lavendel is a beautiful color but also very long	Purple	Red	Red	Red
Background color	Yellow	Yellow	White	Black	Yellow
Outlining	Left	Left	Left	Left	Center
Covers	5, 7, 11	6, 7, 11	1, 9, 11	1, 10, 11	1, 7, 13

Boundary Value Analysis (BVA)

- Complements equivalence partitioning

1. Focuses is on the boundaries of the input - If input condition specifies a range bounded by a certain values, say, a and b, then test cases should include
   - The values for a and b
   - The values just above and just below a and b
2. If an input condition specifies any number of values, test cases should be exercise
   - the minimum and maximum numbers,
   - the values just above and just below the minimum and maximum values
3. Apply guidelines 1 and 2 for output conditions.
4. If internal data structure has prescribed boundaries (e.g. an array has a defined limit of 100 entries), be certain to design a test case to exercise the data structure at its boundaries.



BVA Test Cases
X2
d
c
a
b
X1
www.softwaretestinggenius.com

Example 2: Equivalence Partitioning

Consider a component, *generate_grading*, with the following specification:

The component is passed an exam mark (out of 75) and a coursework (c/w) mark (out of 25), from which it generates a grade for the course in the range 'A' to 'D'. The grade is calculated from the overall mark which is calculated as the sum of the exam and c/w marks, as follows:

greater than or equal to 70	-	*'A'*
greater than or equal to 50, but less than 70	-	*'B'*
greater than or equal to 30, but less than 50	-	*'C'*
less than 30	-	*'D'*

Where a mark is outside its expected range then a fault message ('FM') is generated. All inputs are passed as integers.

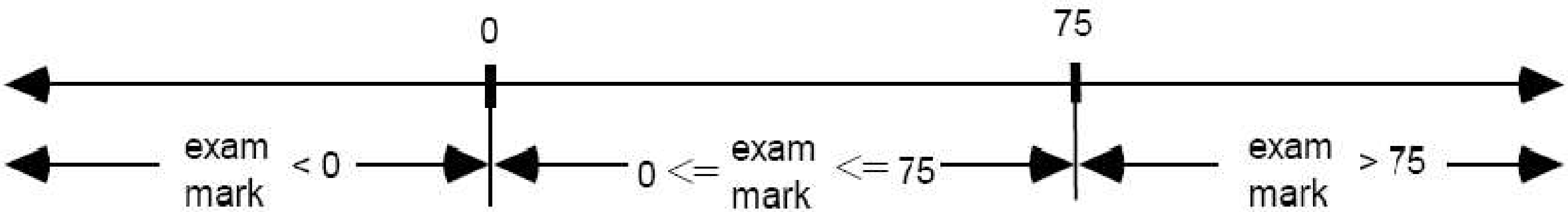
Valid partitions

- The valid partitions can be
  - 0<=exam mark <=75
  - 0<=coursework <=25

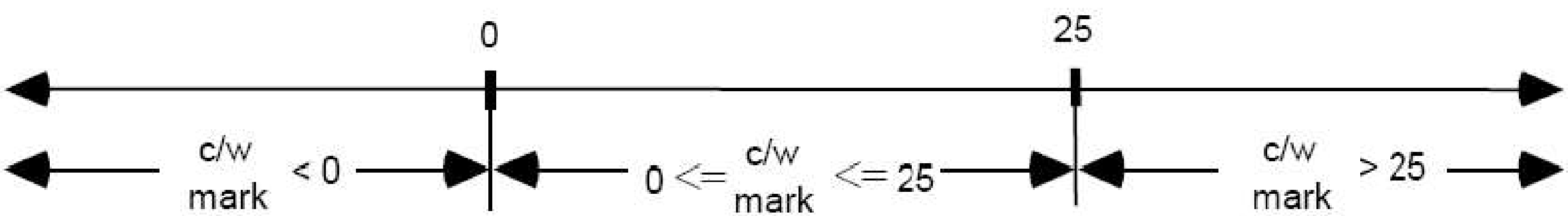
Invalid partitions

- The most obvious partitions are
  - Exam mark > 75
  - Exam mark < 0
  - Coursework mark > 25
  - Coursework mark <0

Exam mark and c/w mark



And for the input, coursework mark, we get:



Less obvious invalid input EP

- invalid INPUT EP should include

exam mark = real number (a number with a fractional part)
exam mark = alphabetic
coursework mark = real number
coursework mark = alphabetic

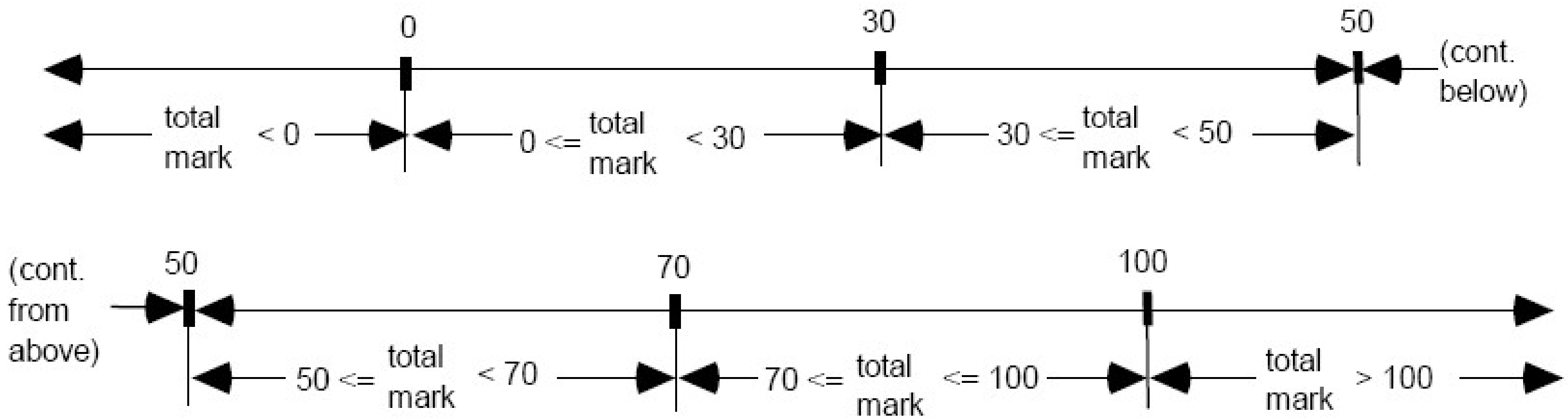
Partitions for the OUTPUTS

- EP for valid OUTPUTS should include

'A'	is induced by	70 <= total mark <= 100
'B'	is induced by	50 <= total mark < 70
'C'	is induced by	30 <= total mark < 50
'D'	is induced by	0 <= total mark < 30
'Fault Message'	is induced by	total mark > 100
'Fault Message'	is induced by	total mark < 0

The EP & Boundary values

- The EP and boundaries for total mark



Unspecified Outputs

- Three unspecfied Outputs can be identified (very subjective)
  - Output = “E”
  - Output = “A+”
  - Output = “null”

Total PE

0 <= exam mark <= 75
exam mark > 75
exam mark < 0
0 <= coursework mark <= 25
coursework mark > 25
coursework mark < 0
exam mark = real number
exam mark = alphabetic
coursework mark = real number
coursework mark = alphabetic
70 <= total mark <= 100
50 <= total mark < 70
30 <= total mark < 50
0 <= total mark < 30
total mark > 100
total mark < 0
output = 'E'
output = 'A+'
output = 'null'

exam mark (INPUT) (test cases 1, 2,3)

Test Case	1	2	3
Input (exam mark)	44	-10	93
Input (c/w mark)	15	15	15
total mark (as calculated)	59	5	108
Partition tested (of exam mark)	0 <= e <= 75	e < 0	e > 75
Exp. Output	'B'	'FM'	'FM'

Test Case 4-6 (coursework)

Test Case	4	5	6
Input (exam mark)	40	40	40
Input (c/w mark)	8	-15	47
total mark (as calculated)	48	25	87
Partition tested (of c/w mark)	0 <= c <= 25	c < 0	c > 25
Exp. Output	'C'	'FM'	'FM'

test case for Invalid inputs (7 to 10)

The test cases corresponding to partitions derived from possible invalid inputs are:

Test Case	7	8	9	10
Input (exam mark)	48.7	q	40	40
Input (c/w mark)	15	15	12.76	g
total mark (as calculated)	63.7	not applicable	52.76	not applicable
Partition tested	exam mark = real number	exam mark = alphabetic	c/w mark = real number	c/w mark = alphabetic
Exp. Output	'FM'	'FM'	'FM'	'FM'

Test cases for invalid outputs:11 to 13 test case

The test cases corresponding to partitions derived from the valid outputs are:

Test Case	11	12	13
Input (exam mark)	-10	12	32
Input (c/w mark)	-10	5	13
total mark (as calculated)	-20	17	45
Partition tested (of total mark)	t < 0	0 <= t < 30	30 <= t < 50
Exp. Output	'FM'	'D'	'C'

Test cases for invalid outputs:2

Test Case	14	15	16
Input (exam mark)	44	60	80
Input (c/w mark)	22	20	30
total mark (as calculated)	66	80	110
Partition tested (of total mark)	50 <= t < 70	70 <= t <= 100	t > 100
Exp. Output	'B'	'A'	'FM'

Test cases for invalid outputs:3

The test cases corresponding to partitions derived from the invalid outputs are:

Test Case	17	18	19
Input (exam mark)	-10	100	null
Input (c/w mark)	0	10	null
total mark (as calculated)	-10	110	null+null
Partition tested (output)	'E'	'A+'	'null'
Exp. Output	'FM'	'FM'	'FM'

Minimal Test cases:1

Test Case	1	2	3	4
Input (exam mark)	60	40	25	15
Input (c/w mark)	20	15	10	8
total mark (as calculated)	80	55	35	23
Partition (of exam mark)	$0 <= e <= 75$	$0 <= e <= 75$	$0 <= e <= 75$	$0 <= e <= 75$
Partition (of c/w mark)	$0 <= c <= 25$	$0 <= c <= 25$	$0 <= c <= 25$	$0 <= c <= 25$
Partition (of total mark)	$70 <= t <= 100$	$50 <= t < 70$	$30 <= t < 50$	$0 <= t < 30$
Exp. Output	'A'	'B'	'C'	'D'

Test Case	5	6	7	8
Input (exam mark)	-10	93	60.5	q
Input (c/w mark)	-15	35	20.23	g
total mark (as calculated)	-25	128	80.73	-
Partition (of exam mark)	$e < 0$	$e > 75$	e = real number	e = alphabetic
Partition (of c/w mark)	$c < 0$	$c > 25$	c = real number	c = alphabetic
Partition (of total mark)	$t < 0$	$t > 100$	$70 <= t <= 100$	-
Exp. Output	'FM'	'FM'	'FM'	'FM'

Minimal Test cases:2

Test Case	9	10	11
Input (exam mark)	-10	100	'null'
Input (c/w mark)	0	10	'null'
total mark (as calculated)	-10	110	null+null
Partition (of exam mark)	e < 0	e > 75	-
Partition (of c/w mark)	0 <= c <= 25	0 <= c <= 25	-
Partition (of total mark)	t < 0	t > 100	-
Partition (of output)	'E'	'A+'	'null'
Exp. Output	'FM'	'FM'	'FM'