Software Engineering: A Practitioner's Approach, 6/e

Chapter 14
Software Testing Techniques

Testability

- Operability—it operates cleanly
- Observability—the results of each test case are readily observed
- Controllability—the degree to which testing can be automated and optimized
- Decomposability—testing can be targeted
- Simplicity—reduce complex architecture and logic to simplify tests
- Stability—few changes are requested during testing
- Understandability—of the design

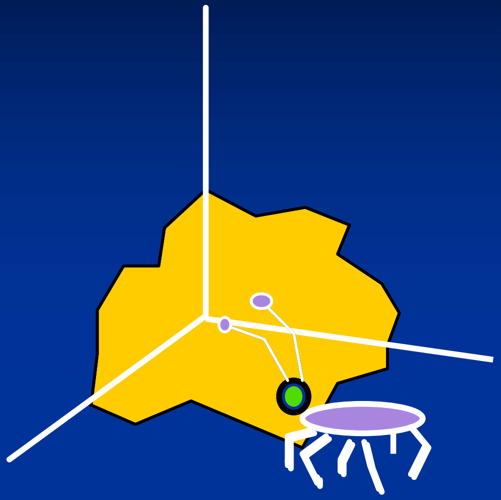
What is a “Good” Test?

- A good test has a high probability of finding an error
- A good test is not redundant.
- A good test should be “best of breed”
- A good test should be neither too simple nor too complex

Test Case Design

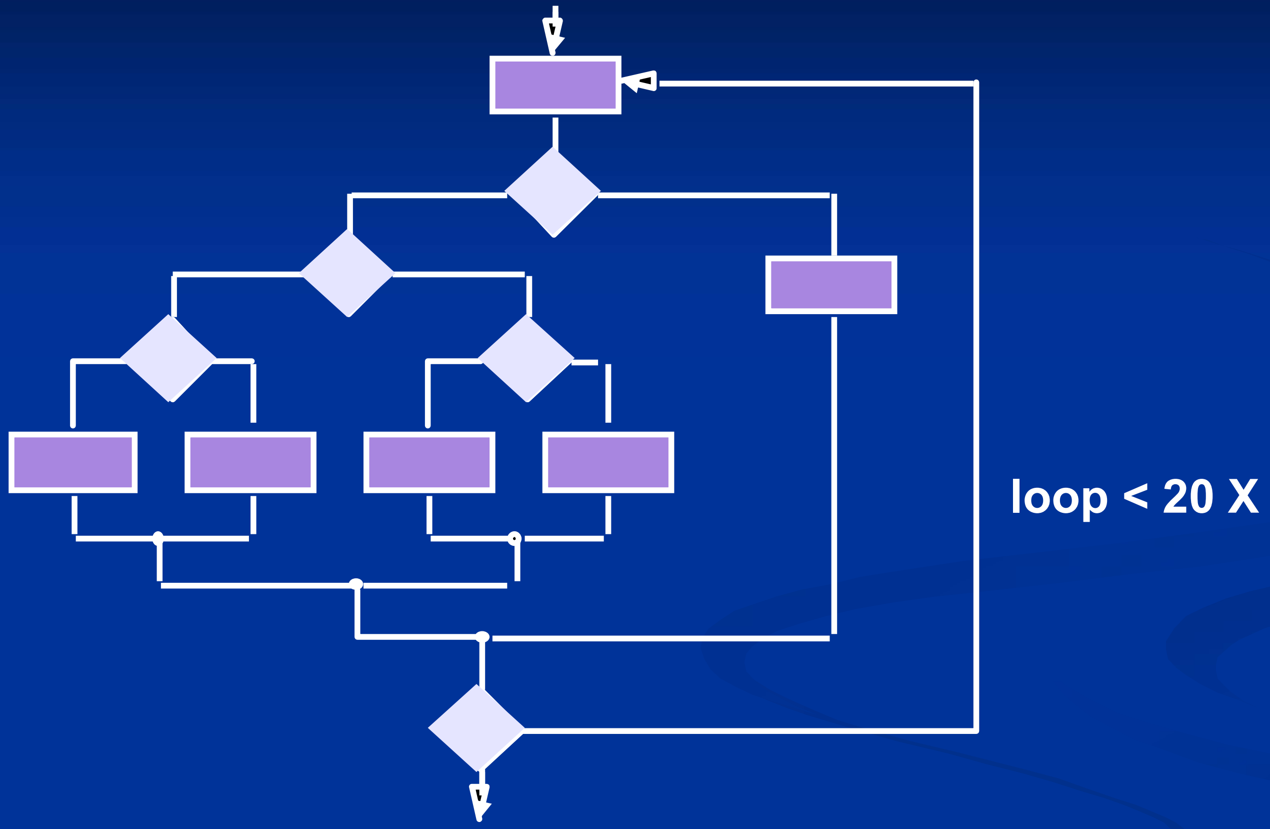
"Bugs lurk in corners and congregate at boundaries ..."

Boris Beizer



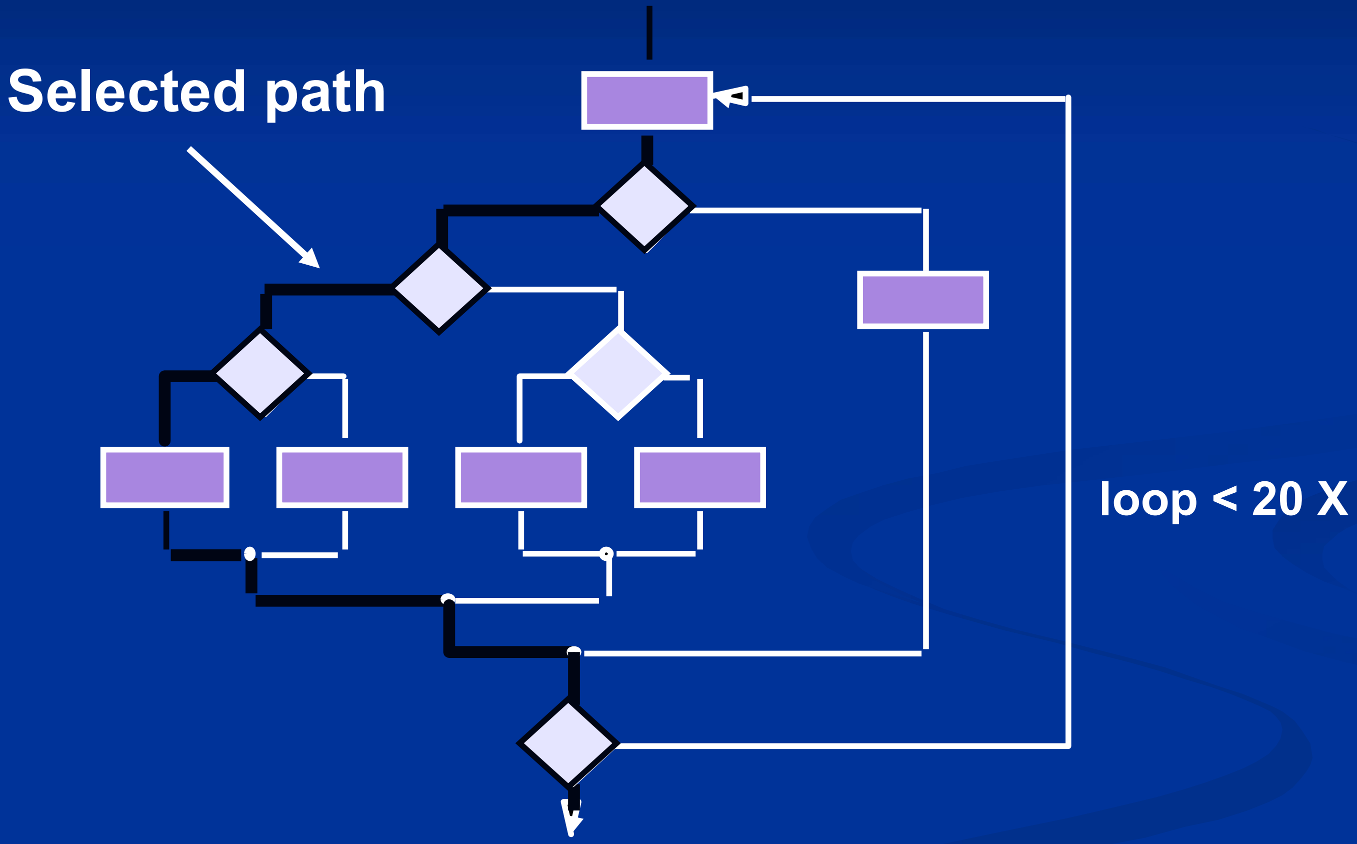
OBJECTIVE	**to uncover errors**
CRITERIA	**in a complete manner**
CONSTRAINT	**with a minimum of effort and time**

Exhaustive Testing

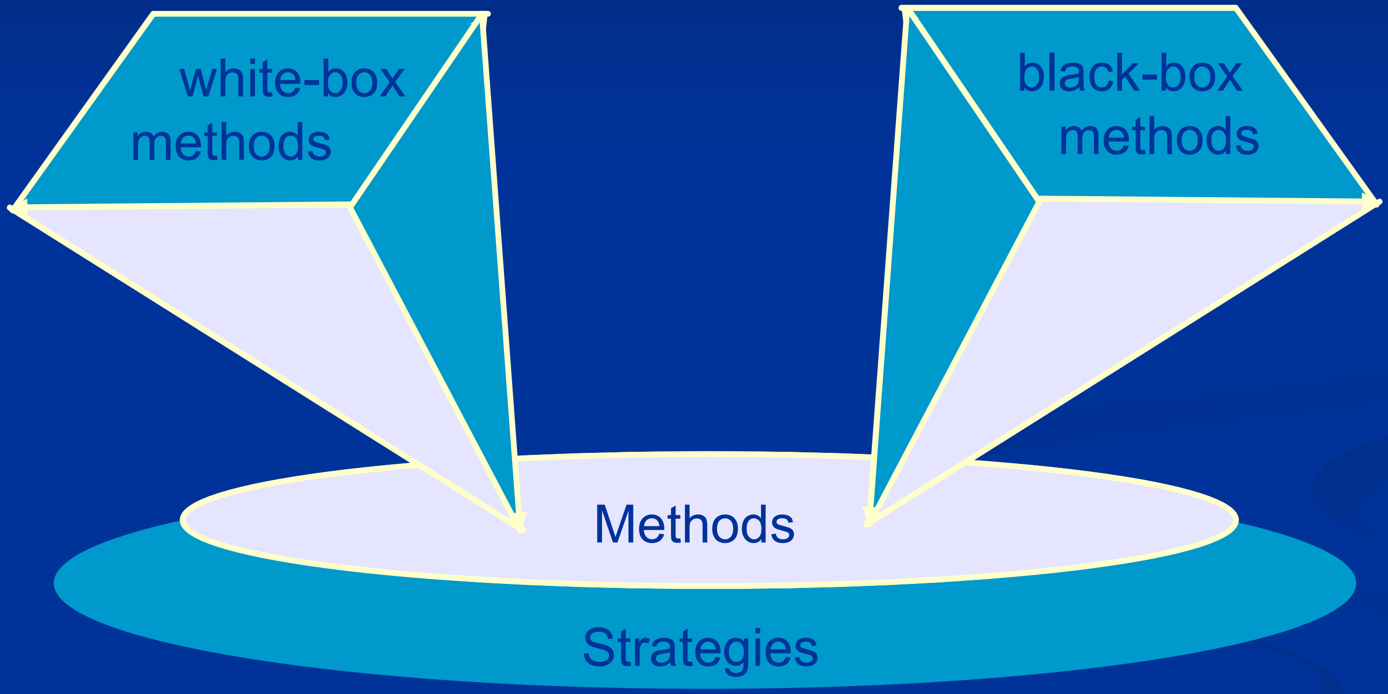


There are 10^{14} possible paths! If we execute one test per millisecond, it would take 3,170 years to test this program!!

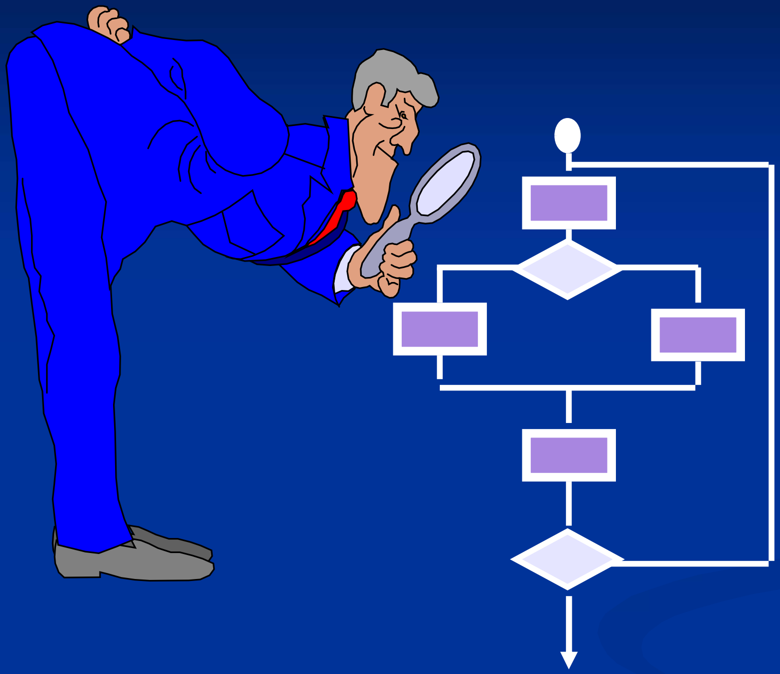
Selective Testing



Software Testing



White-Box Testing



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

Why Cover?

- logic errors and incorrect assumptions are inversely proportional to a path's execution probability
- we often believe that a path is not likely to be executed; in fact, reality is often counter intuitive
- typographical errors are random; it's likely that untested paths will contain some

Basis Path Testing

- **Basis path testing,** a structured testing or white box testing technique used for designing test cases intended to examine all possible paths of execution at least once.

Basis Path Testing

- In Basis Path Testing, Code will be given.
- Steps:

Step 1: Draw the flow graph.

Step 2: Determine the Cyclomatic Complexity V(G)

V(G)=E-N+2 OR

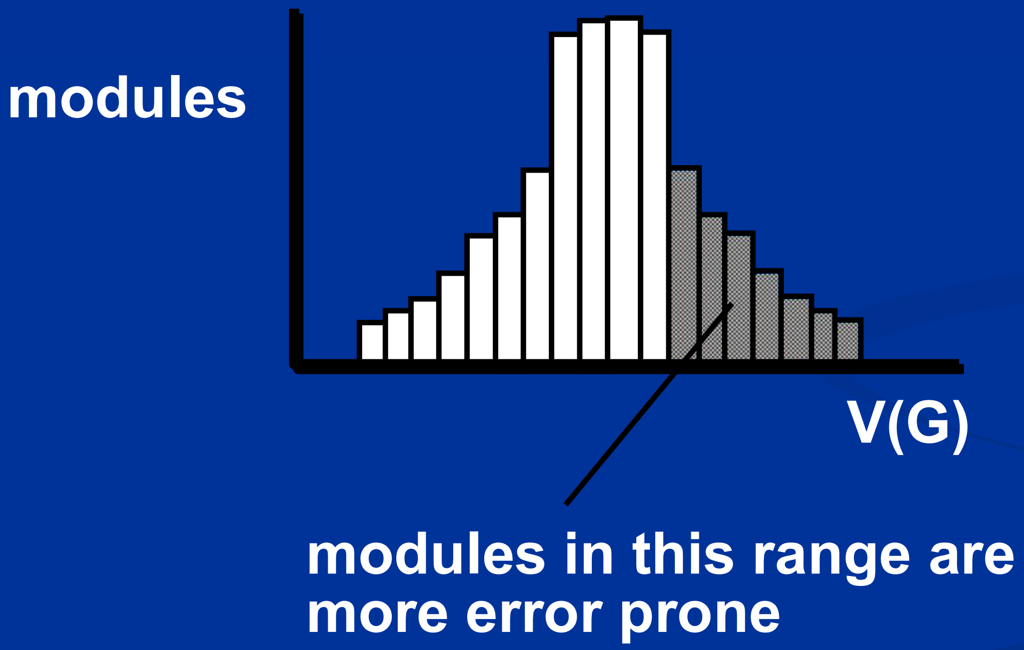
V(G)=No. of Predicate nodes +1 OR

V(G)=No. of Bounded regions +1

Step 3: Determine the independent paths. Number of independent paths is equivalent to the Cyclomatic Complexity of the code.

Cyclomatic Complexity

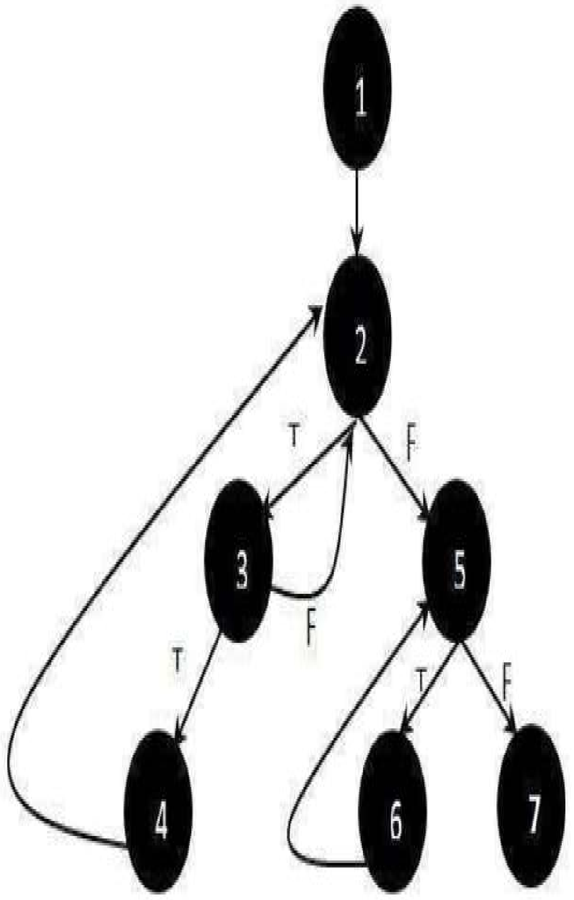
A number of industry studies have indicated that the higher V(G), the higher the probability or errors.



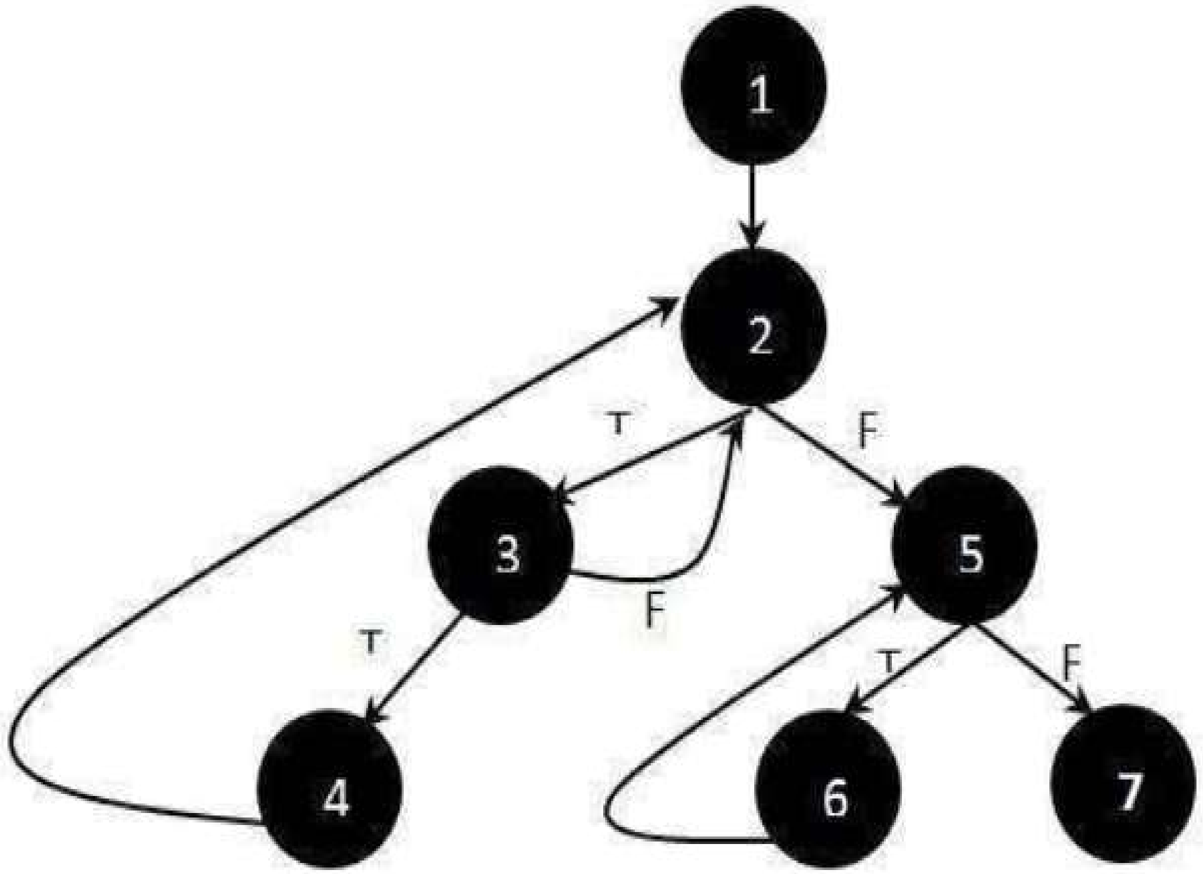
Example:

```
Function fn_delete_element (int value, int array_size, int array[])
{
        1  int i;
           location = array_size + 1;

        2  for i = 1 to array_size
        3  if ( array[i] == value )
        4        location = i; end if;
            end for;

        5  for i = location to array_size
        6        array[i] = array[i+1]; end for;
        7  array_size --;
}
```

Step 1 : Draw the Flow Graph of the Function/Program under consideration as shown below.



Step 1 : Draw the Flow Graph of the Function/Program under consideration as shown below.



Step 2: Determine Cyclomatic Complexity.

V(G)=E-N+2=9-7+2 =4

V(G)=No. of Predicate nodes +1 = 3+1= 4

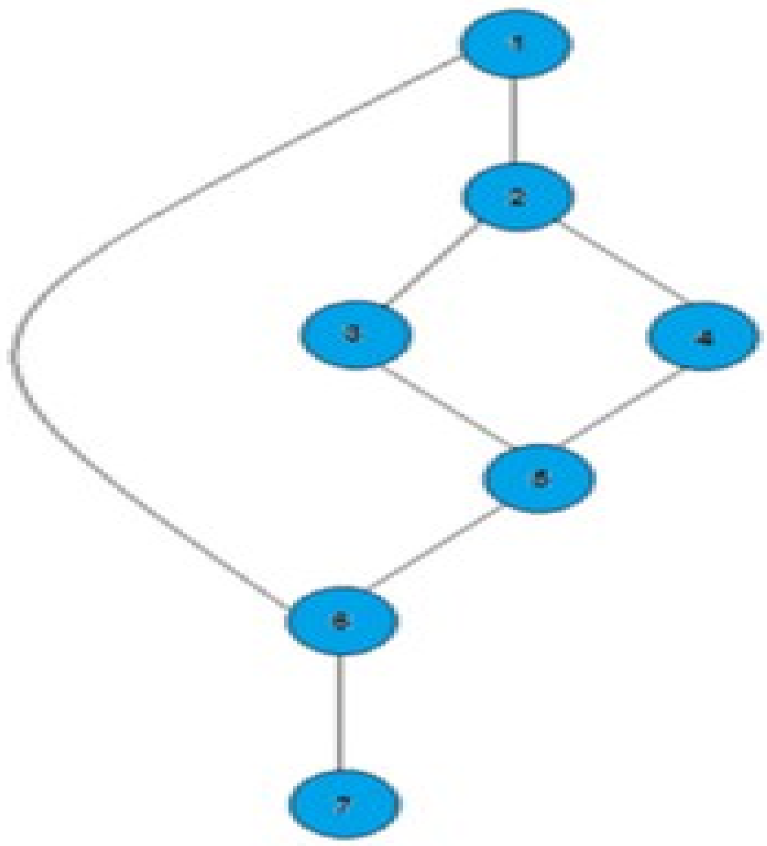
V(G)=No. of Bounded regions +1 = 3=1=4

Step 3: Determine the independent paths.

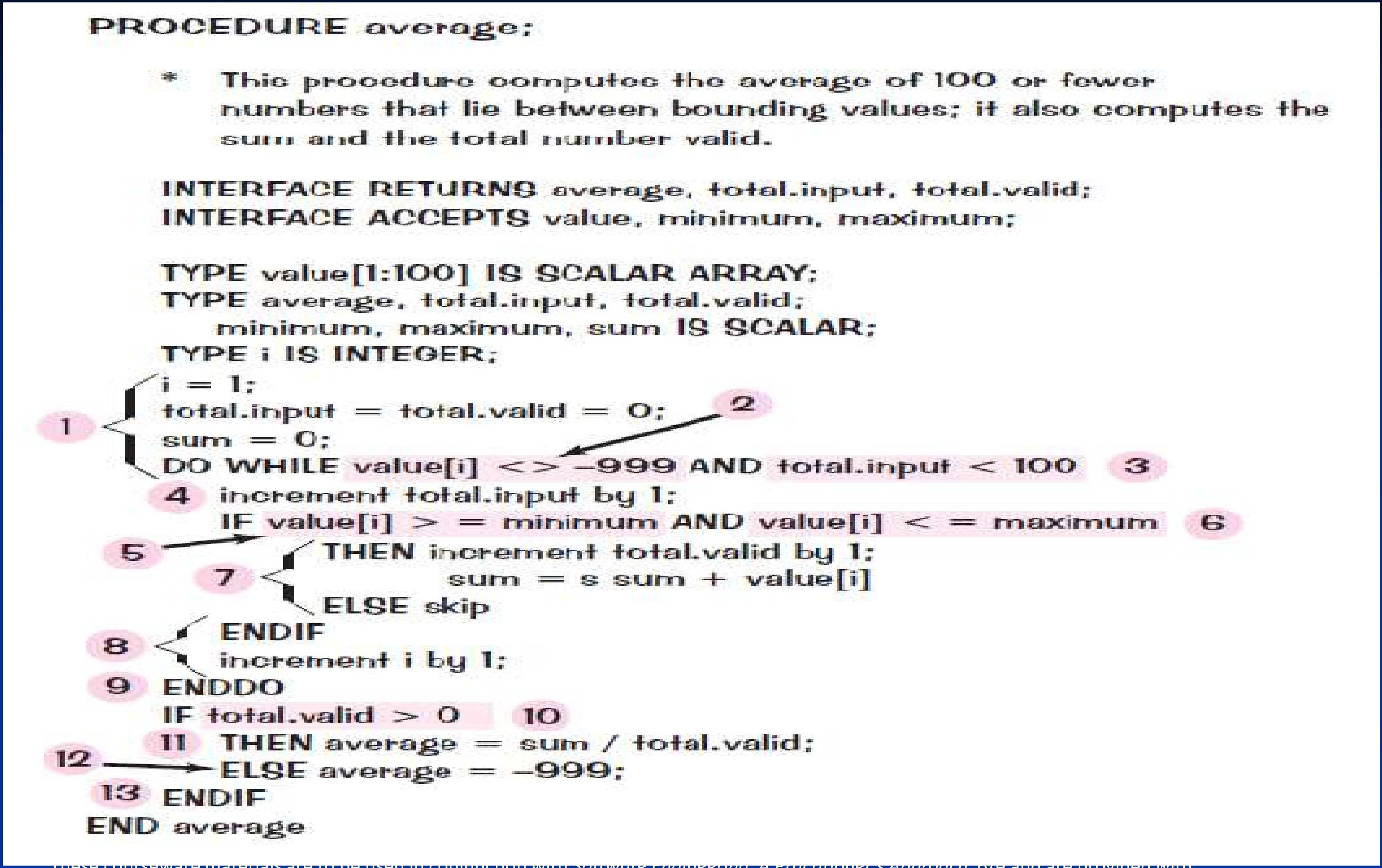
As V(G)=4, there will be 4 linearly independent paths.

Path 1:	1 - 2 - 5- 7
Path 2:	1 - 2 - 5 - 6 - 5- 7
Path 3:	1 - 2 - 3 - 2 - 5 - 7
Path 4:	1 - 2 - 3 - 4 - 2 - 5 - 7

Find the Cyclomatic Complexity and Determine independent Paths



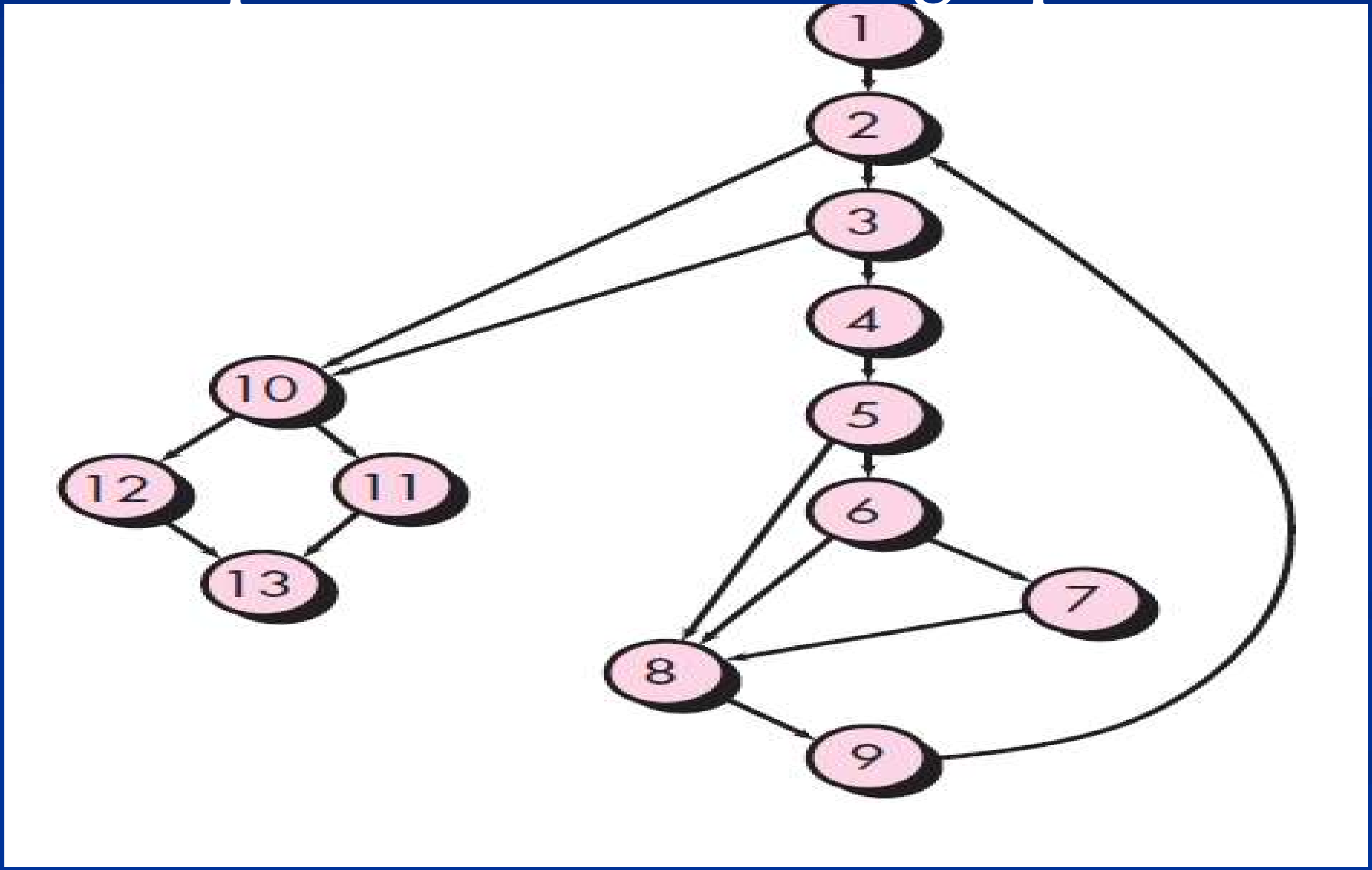
```
1.   If A= 50
2.   THEN IF B>C
3.   THEN A =B
4.   ELSE A=C
5.   ENDIF
6.   ENDIF
7.   Print A
```



```
PROCEDURE average;

    *   This procedure computes the average of 100 or fewer
        numbers that lie between bounding values; it also computes the
        sum and the total number valid.

    INTERFACE RETURNS average, total.input, total.valid;
    INTERFACE ACCEPTS value, minimum, maximum;

    TYPE value[1:100] IS SCALAR ARRAY;
    TYPE average, total.input, total.valid;
        minimum, maximum, sum IS SCALAR;
    TYPE i IS INTEGER;
 1  i = 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
  7         THEN increment total.valid by 1;
                    sum = s sum + value[i]
            ELSE skip
  8     ENDIF
        increment i by 1;
  9 ENDDO
    IF total.valid > 0        10
 11     THEN average = sum / total.valid;
 12     ELSE average = -999;
 13 ENDIF
END average
```

Basis Path Testing

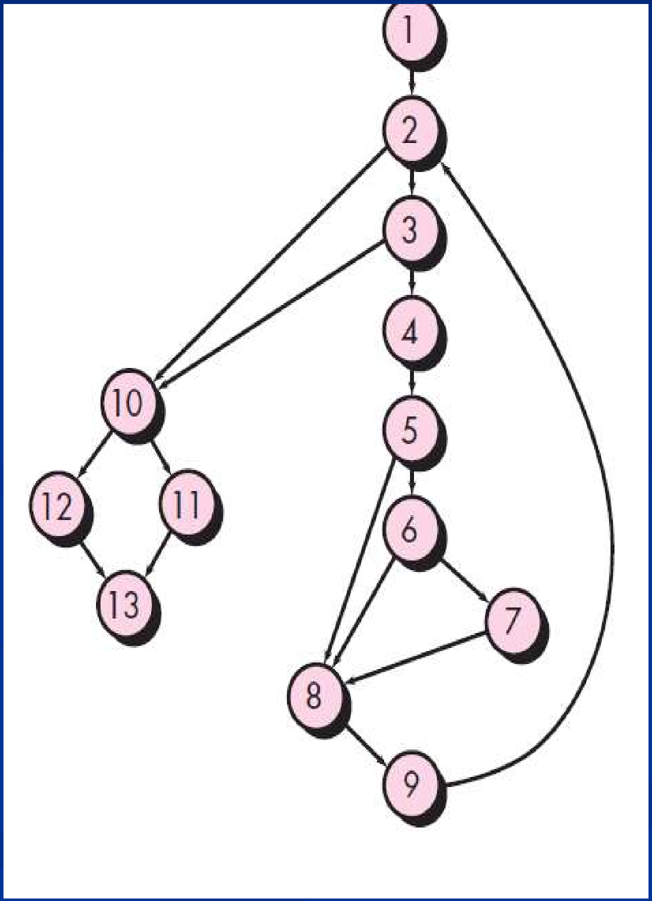
Step 1: Draw the flow graph



Basis Path Testing

- Step 2 : Determine cyclomatic complexity.

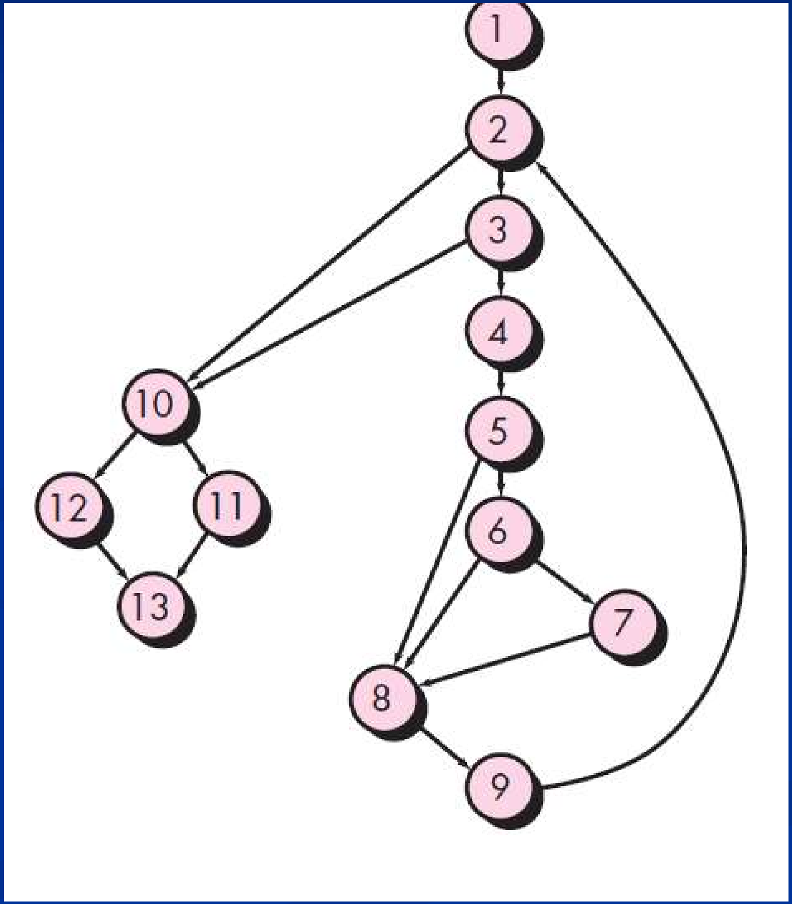
- $V(G)$ = 5 bounded regions +1 =6
- $V(G)$ = 17 edges - 13 nodes + 2 = 6
- $V(G)$ = 5 predicate nodes+ 1 = 6



Basis Path Testing

- Step 3: Determine linearly independent paths.

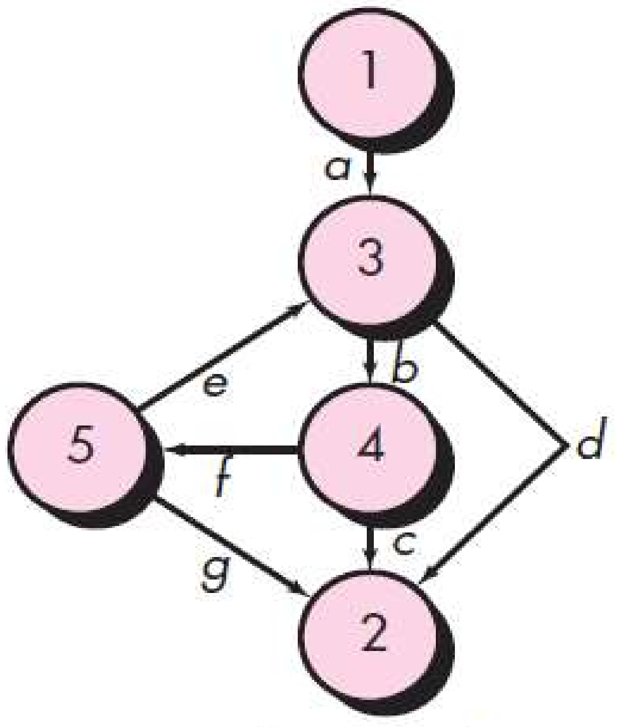
- Path 1: 1-2-10-11-13
- Path 2: 1-2-10-12-13
- Path 3: 1-2-3-10-11-13
- Path 4: 1-2-3-4-5-8-9-2-. . .
- Path 5: 1-2-3-4-5-6-8-9-2-. . .
- Path 6: 1-2-3-4-5-6-7-8-9-2-. . .



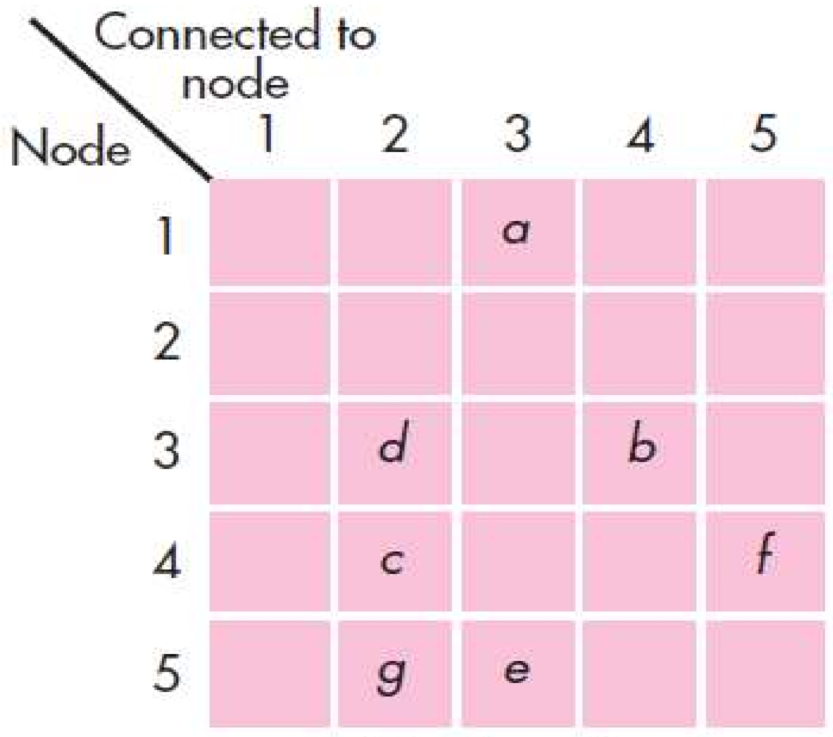
Graph Matrices

- A graph matrix is a square matrix whose size (i.e., number of rows and columns) is equal to the number of nodes on a flow graph
- Each row and column corresponds to an identified node, and matrix entries correspond to connections (an edge) between nodes.
- By adding a *link weight* to each matrix entry, the graph matrix can become a powerful tool for evaluating program control structure during testing

Graph Matrix



Flow graph



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

Graph matrix

Graph Matrix

The link weight provides additional information about control flow. In its simplest form, the link weight is 1 (a connection exists) or 0 (a connection does not exist). But link weights can be assigned other, more interesting properties:

- The probability that a link (edge) will be execute.
- The processing time expended during traversal of a link
- The memory required during traversal of a link
- The resources required during traversal of a link.

Control Structure Testing

- Condition testing — a test case design method that exercises the logical conditions contained in a program module
- Data flow testing — selects test paths of a program according to the locations of definitions and uses of variables in the program

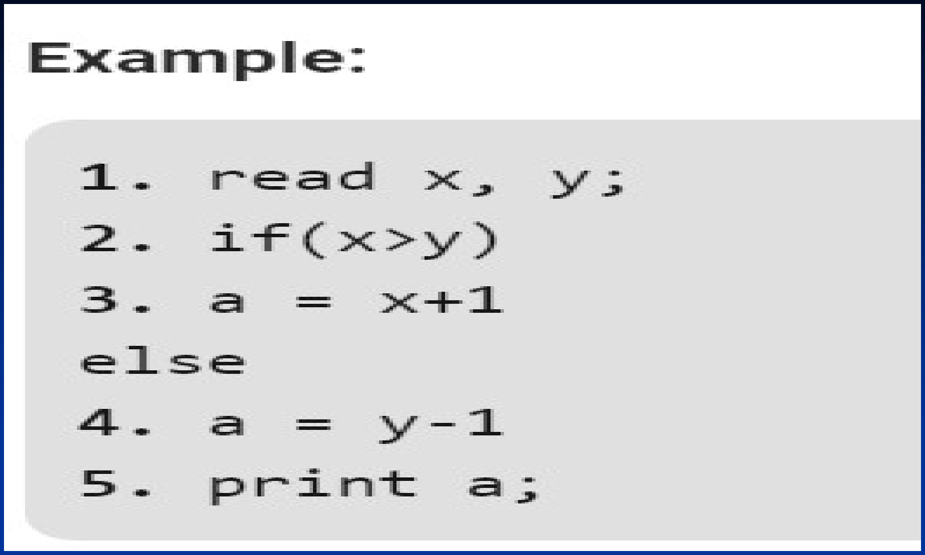
Data Flow Testing

The data flow testing method [Fra93] selects test paths of a program according to the locations of definitions and uses of variables in the program.

Assume that each statement in a program is assigned a unique statement number and that each function does not modify its parameters or global variables. For a statement with S as its statement number

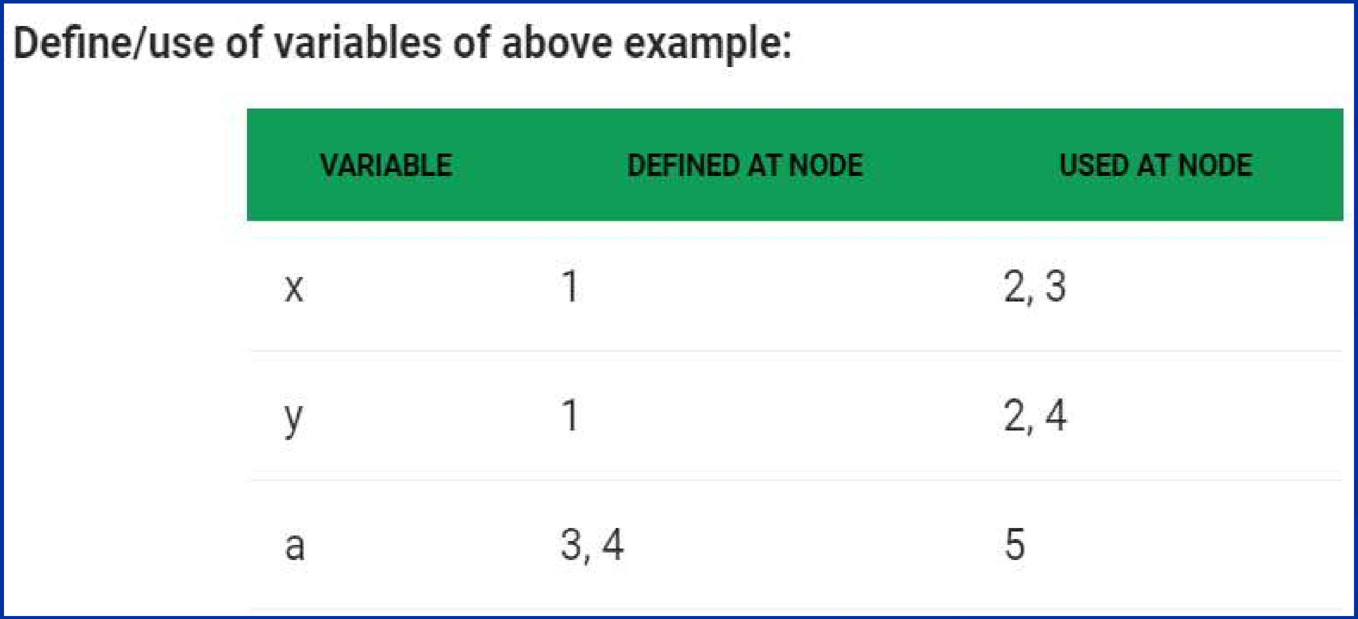
- DEF(S) = {X | statement S contains a definition of X}
- USE(S) = {X | statement S contains a use of X}

A definition-use (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 USE(S'), and the definition of X in statement S is live at statement S'



Example:

```
1.  read x, y;
2.  if(x>y)
3.  a = x+1
else
4.  a = y-1
5.  print a;
```



Define/use of variables of above example:

VARIABLE	DEFINED AT NODE	USED AT NODE
x	1	2, 3
y	1	2, 4
a	3, 4	5

DU Chains

[x,1,2]

[x,1,3]

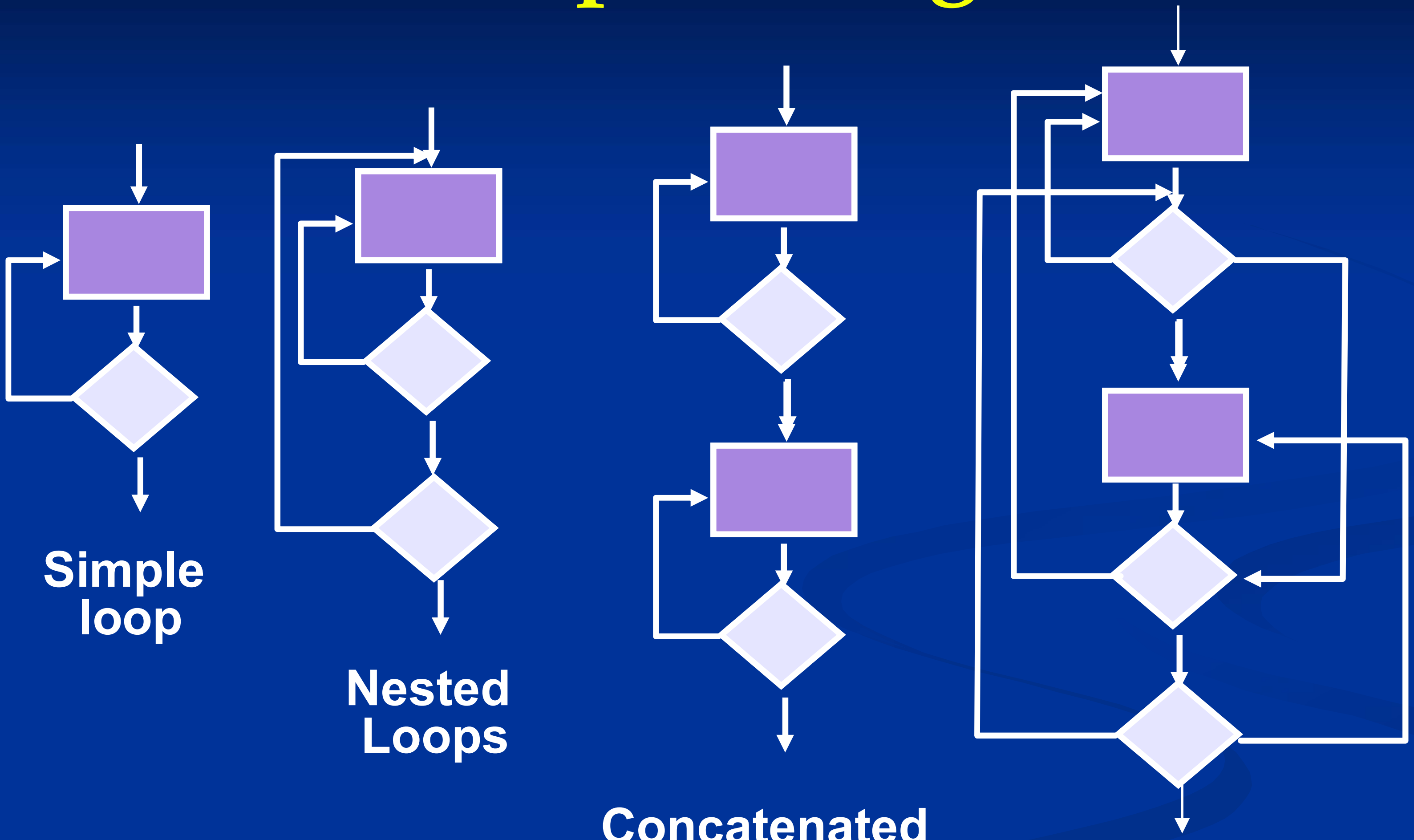
[y,1,2]

[y,1,4]

[a,3,5]

[a,4,5]

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, min+1, typical, max-1 and max for the innermost loop, while holding the outer loops at their minimum values.

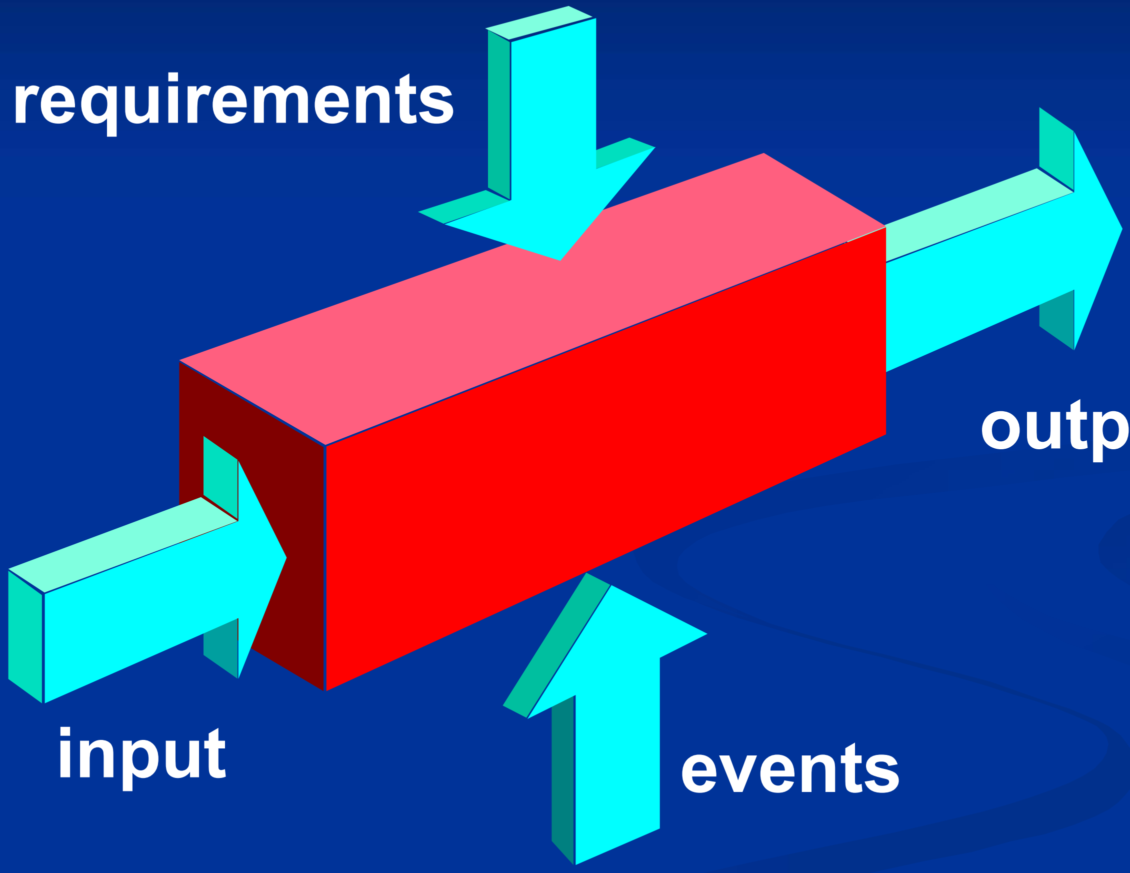
Move out one loop and set it up as in step 2, holding all other loops at typical values. Continue this step until the outermost loop has been tested.

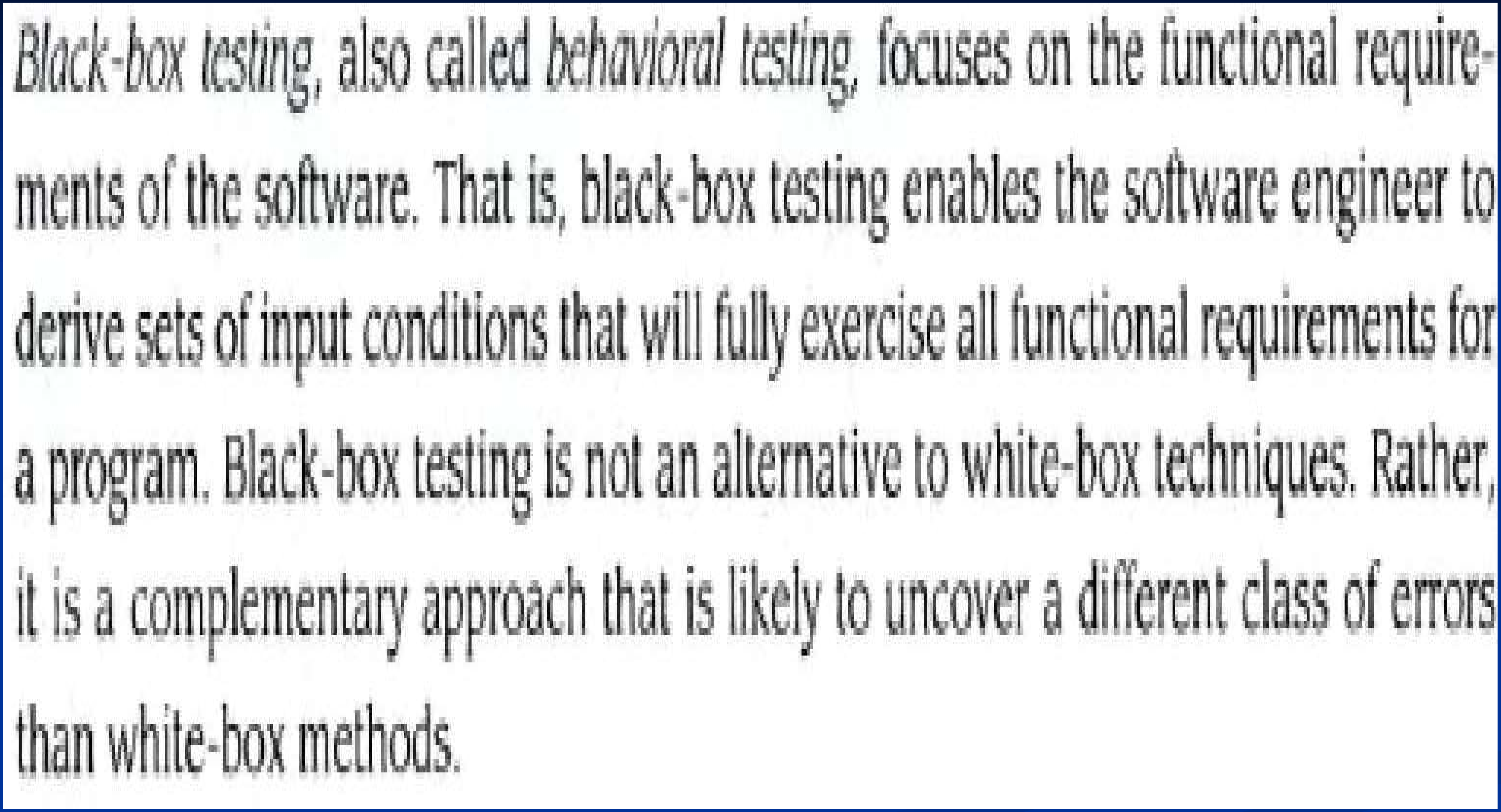
Concatenated Loops

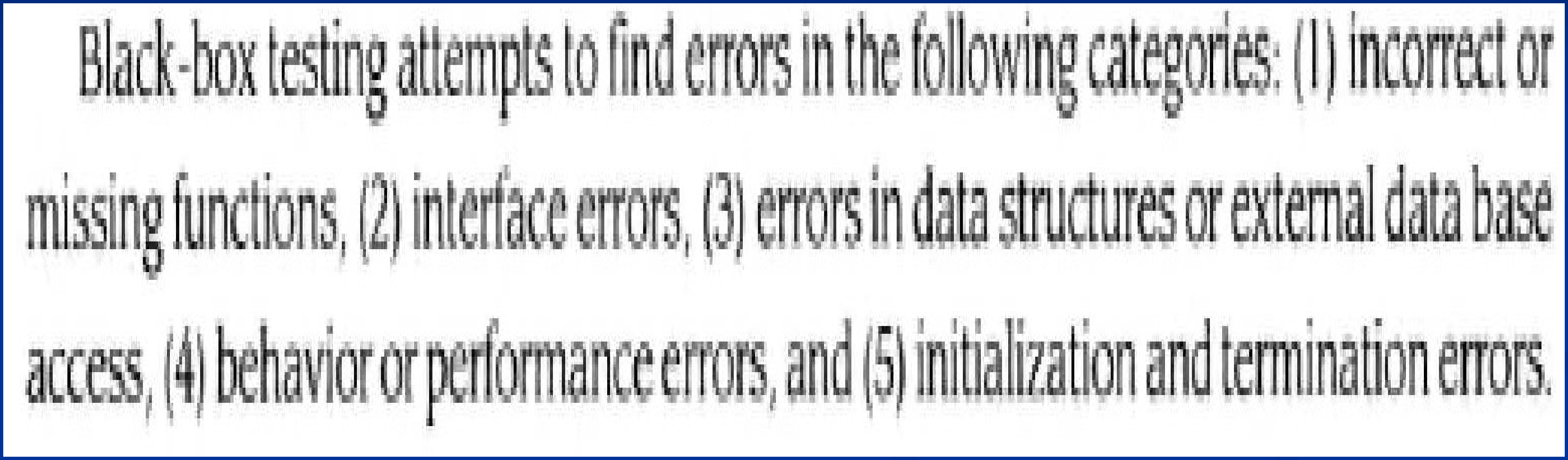
```
If the loops are independent of one another
   then treat each as a simple loop
   else* treat as nested loops
endif*
```

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

Black-Box Testing




Black-box testing, also called *behavioral testing,* focuses on the functional requirements of the software. That is, black-box testing enables the software engineer to derive sets of input conditions that will fully exercise all functional requirements for a program. Black-box testing is not an alternative to white-box techniques. Rather, it is a complementary approach that is likely to uncover a different class of errors than white-box methods.



Black-box testing attempts to find errors in the following categories: (1) incorrect or missing functions, (2) interface errors, (3) errors in data structures or external data base access, (4) behavior or performance errors, and (5) initialization and termination errors.

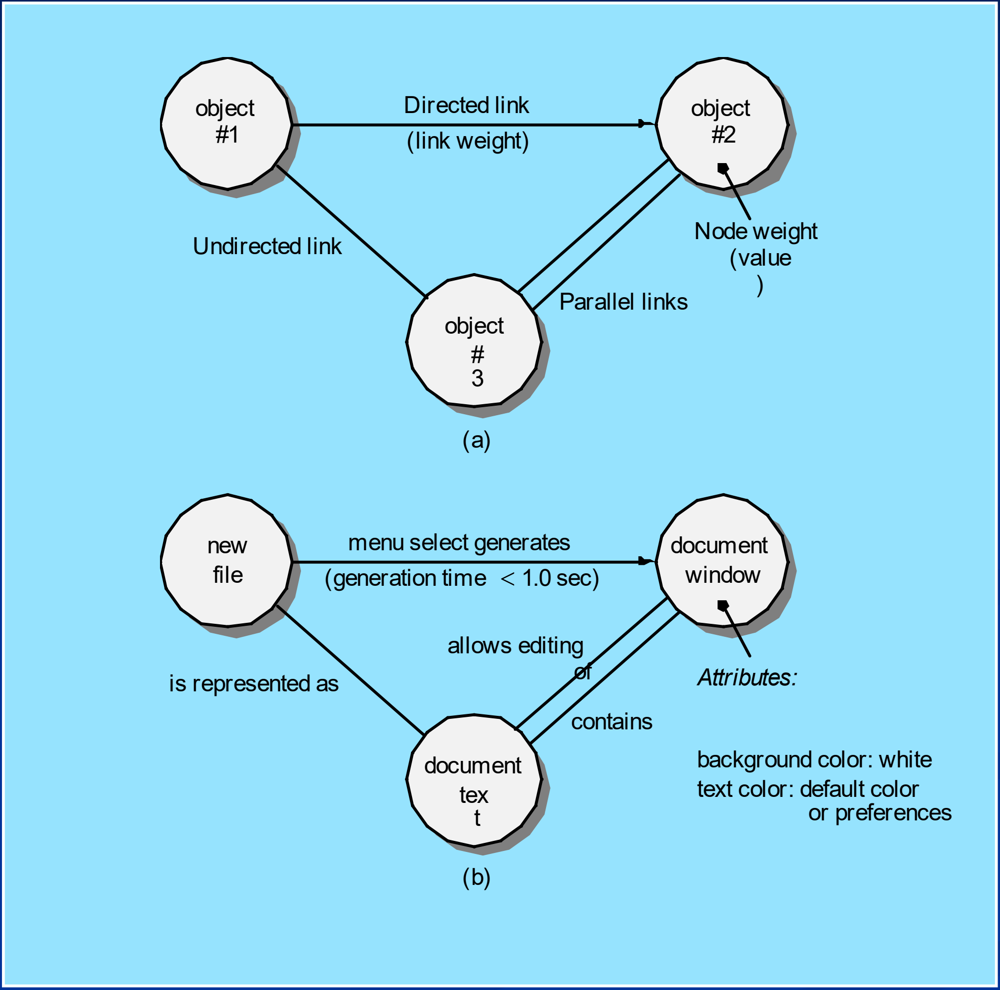
Black-Box Testing

- How is functional validity tested?
- How is system behavior and performance tested?
- What classes of input will make good test cases?
- Is the system particularly sensitive to certain input values?
- How are the boundaries of a data class isolated?
- What data rates and data volume can the system tolerate?
- What effect will specific combinations of data have on system operation?

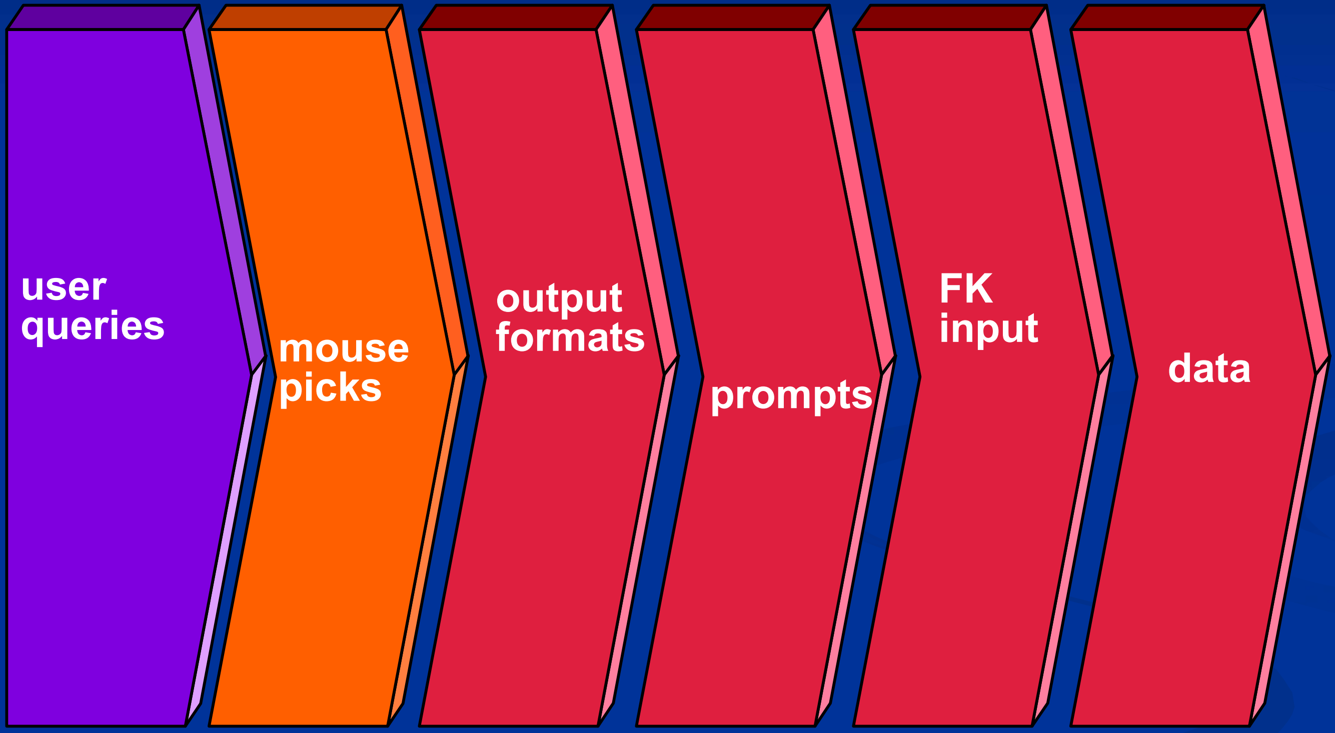
Graph-Based Methods

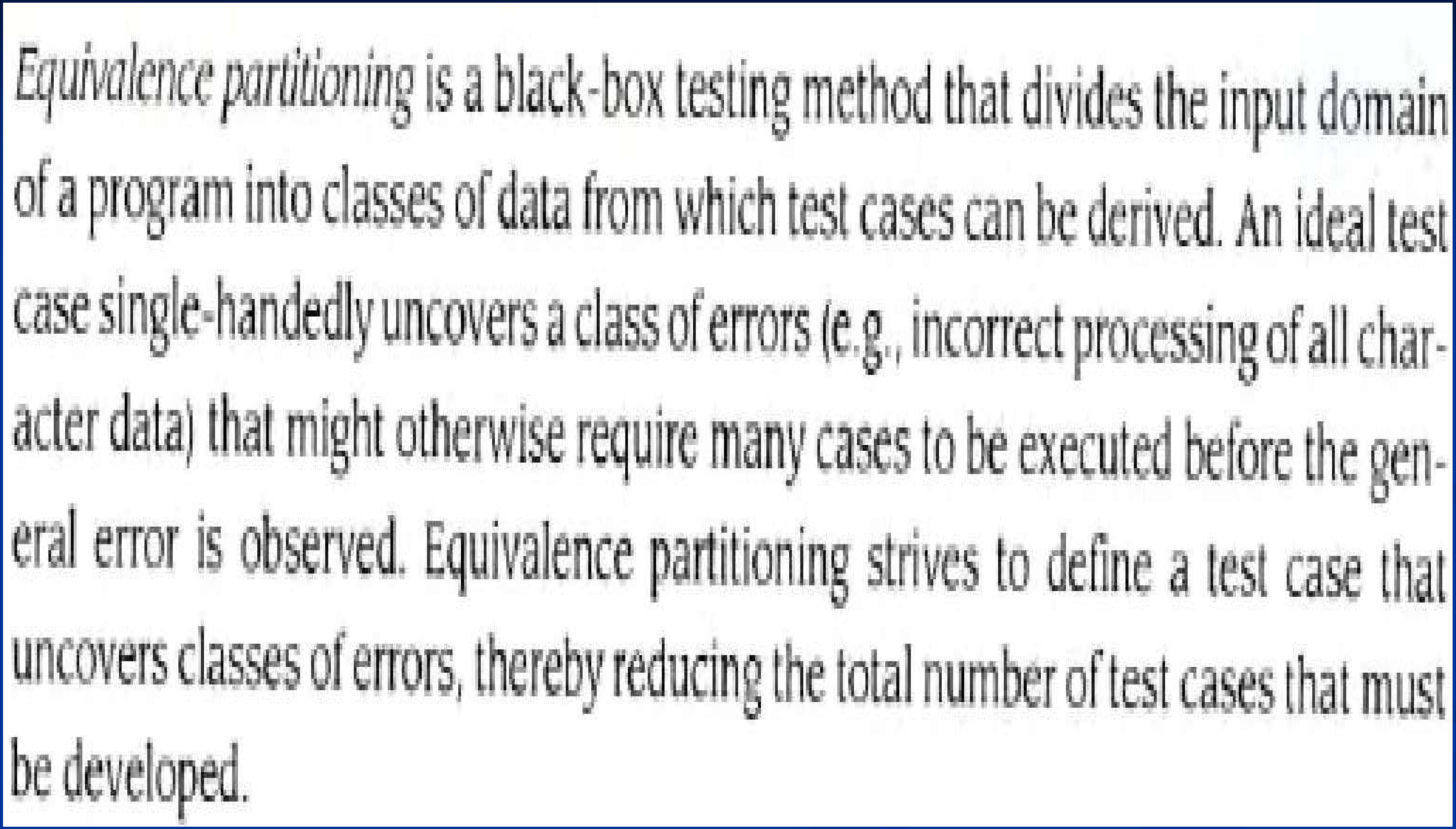
To understand the objects that are modeled in software and the relationships that connect these objects

In this context, we consider the term “objects” in the broadest possible context. It encompasses data objects, traditional components (modules), and object-oriented elements of computer software.



Equivalence Partitioning




Equivalence partitioning is a black-box testing method that divides the input domain of a program into classes of data from which test cases can be derived. An ideal test case single-handedly uncovers a class of errors (e.g., incorrect processing of all character data) that might otherwise require many cases to be executed before the general error is observed. Equivalence partitioning strives to define a test case that uncovers classes of errors, thereby reducing the total number of test cases that must be developed.

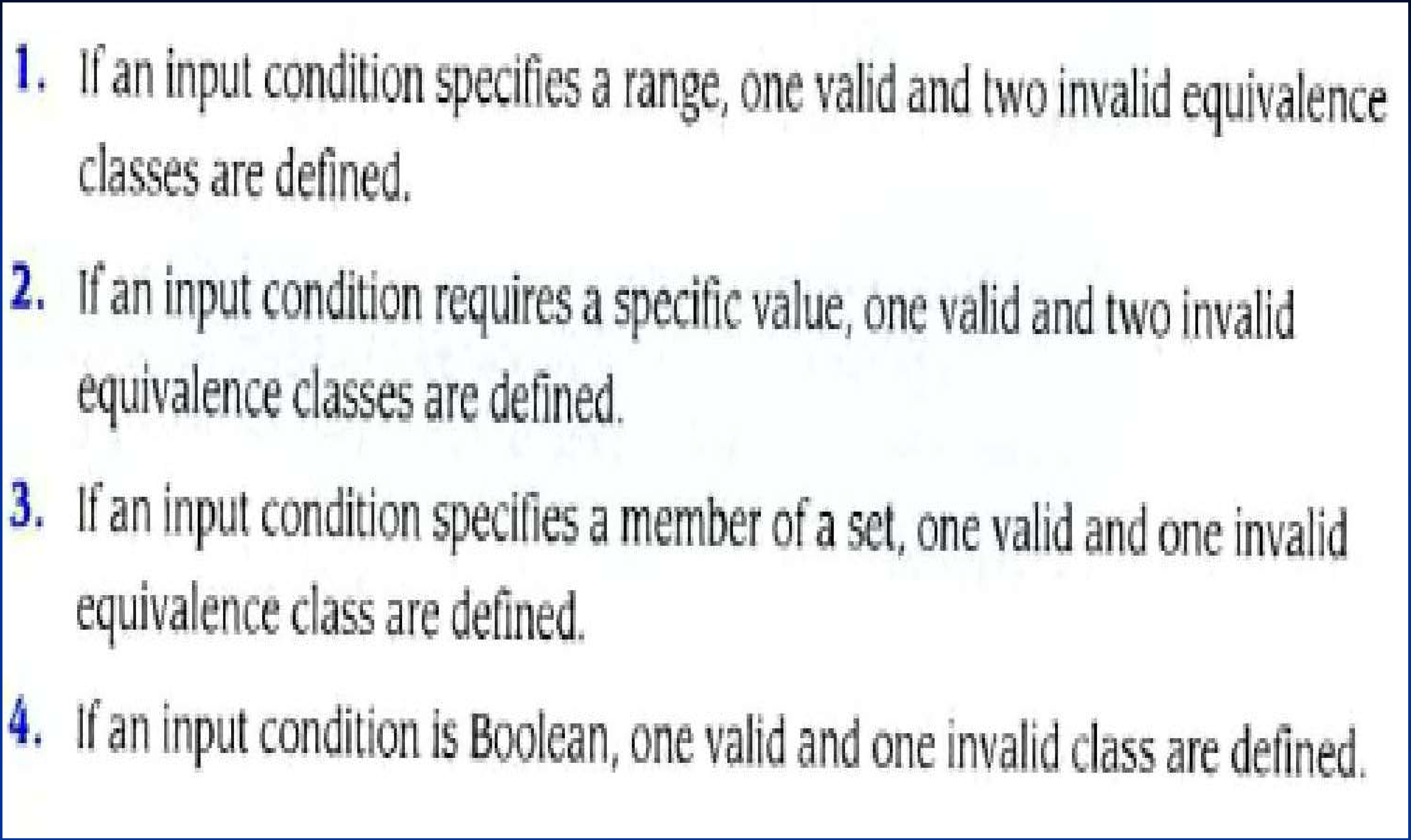
Sample Equivalence Classes

Valid data

- user supplied commands
- responses to system prompts
- file names
- computational data
  - physical parameters
  - bounding values
  - initiation values
- output data formatting
- responses to error messages
- graphical data (e.g., mouse picks)

Invalid data

- data outside bounds of the program
- physically impossible data
- proper value supplied in wrong place



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

Example#1: For a software that computes the square root of an input integer which can assume values in the range of 0 to 5000, there are three equivalence classes: The set of negative integers, the set of integers in the range of 0 and 5000, and the integers larger than 5000. Therefore, the test cases must include representatives for each of the three equivalence classes and a possible test set can be: {-5,500,6000}.

Example#2:
Design the black-box test suite for the following program. The program computes the intersection point of two straight lines and displays the result. It reads two integer pairs (m1, c1) and (m2, c2) defining the two straight lines of the form y=mx + c.

Example#2:

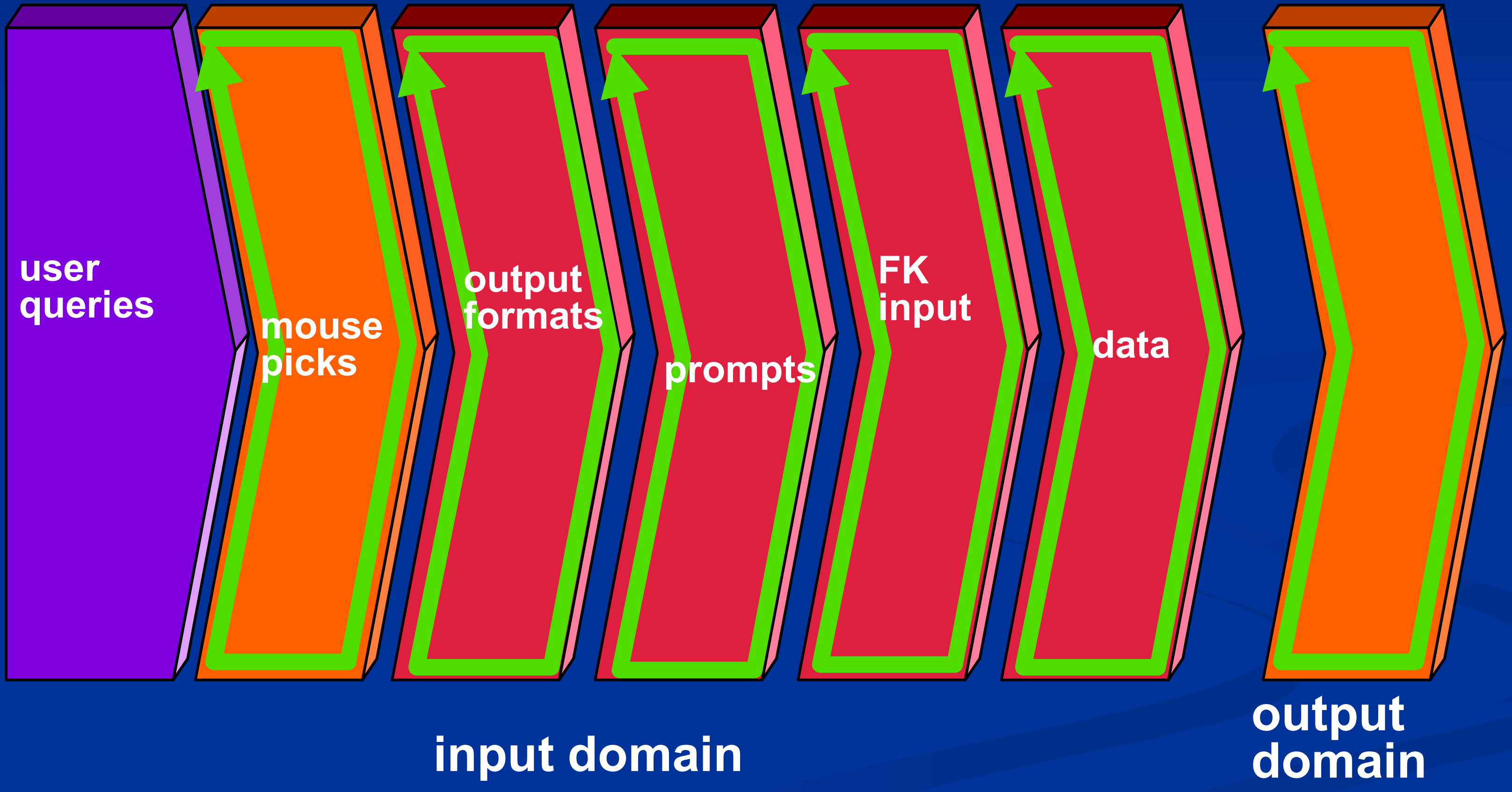
Design the black-box test suite for the following program. The program computes the intersection point of two straight lines and displays the result. It reads two integer pairs (m1, c1) and (m2, c2) defining the two straight lines of the form y=mx + c.

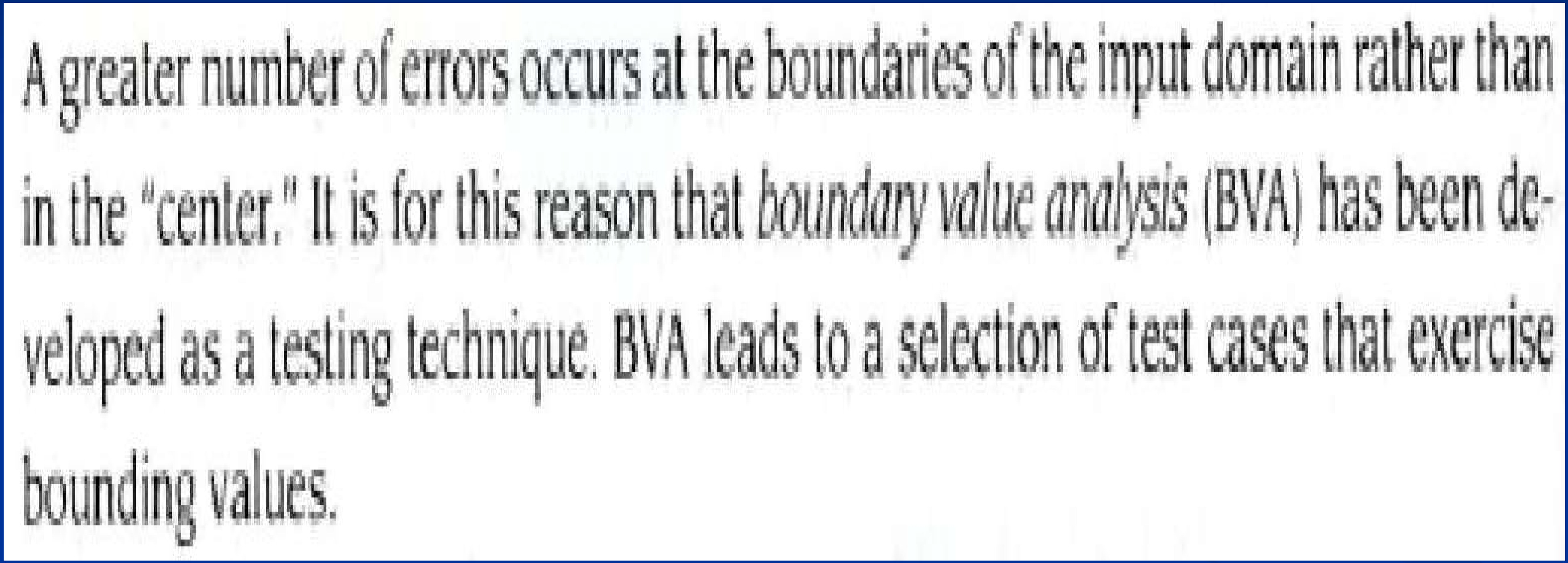
The equivalence classes are the following:

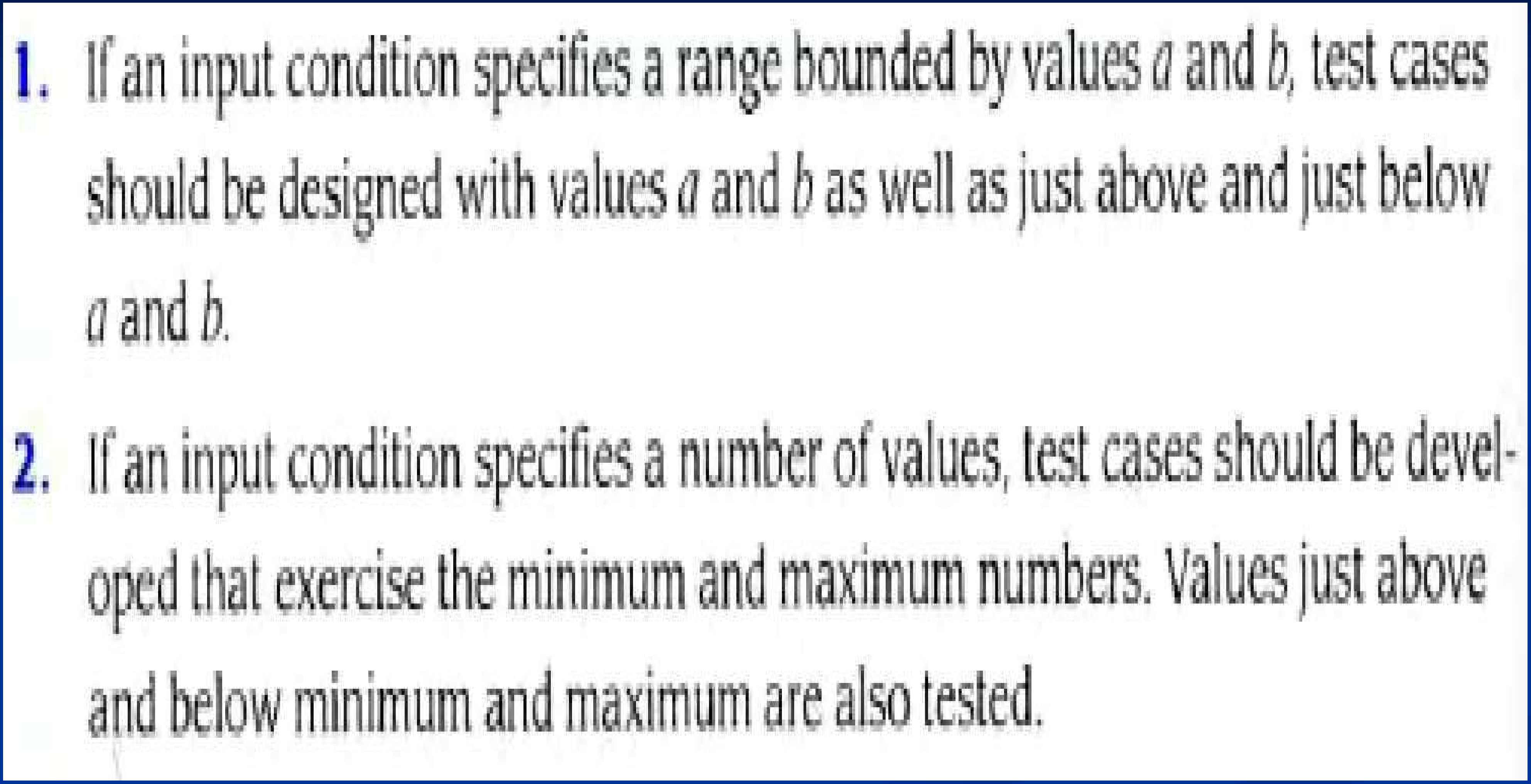
- Parallel lines (m1=m2, c1≠c2)
- Intersecting lines (m1≠m2)
- Coincident lines (m1=m2, c1=c2)

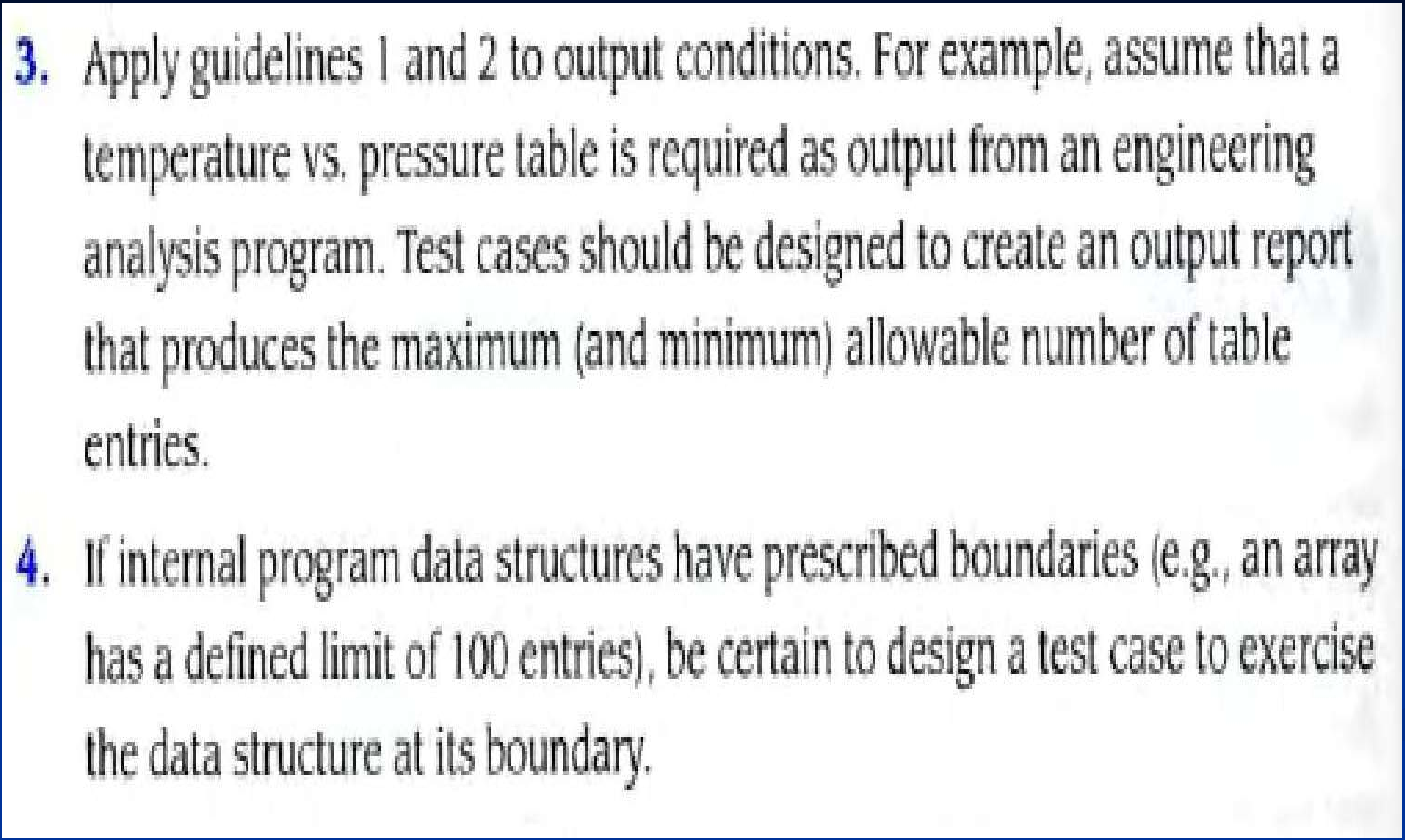
Now, selecting one representative value from each equivalence class, the test
suit (2, 2) (2, 5), (5, 5) (7, 7), (10, 10) (10, 10) are obtained.

Boundary Value Analysis




A greater number of errors occurs at the boundaries of the input domain rather than in the "center." It is for this reason that *boundary value analysis* (BVA) has been developed as a testing technique. BVA leads to a selection of test cases that exercise bounding values.


1. If an input condition specifies a range bounded by values *a* and *b*, test cases should be designed with values *a* and *b* as well as just above and just below *a* and *b*.
2. If an input condition specifies a number of values, test cases should be developed that exercise the minimum and maximum numbers. Values just above and below minimum and maximum are also tested.



3. Apply guidelines 1 and 2 to output conditions. For example, assume that a temperature vs. pressure table is required as output from an engineering analysis program. Test cases should be designed to create an output report that produces the maximum (and minimum) allowable number of table entries.

4. If internal program data structures have 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 boundary.

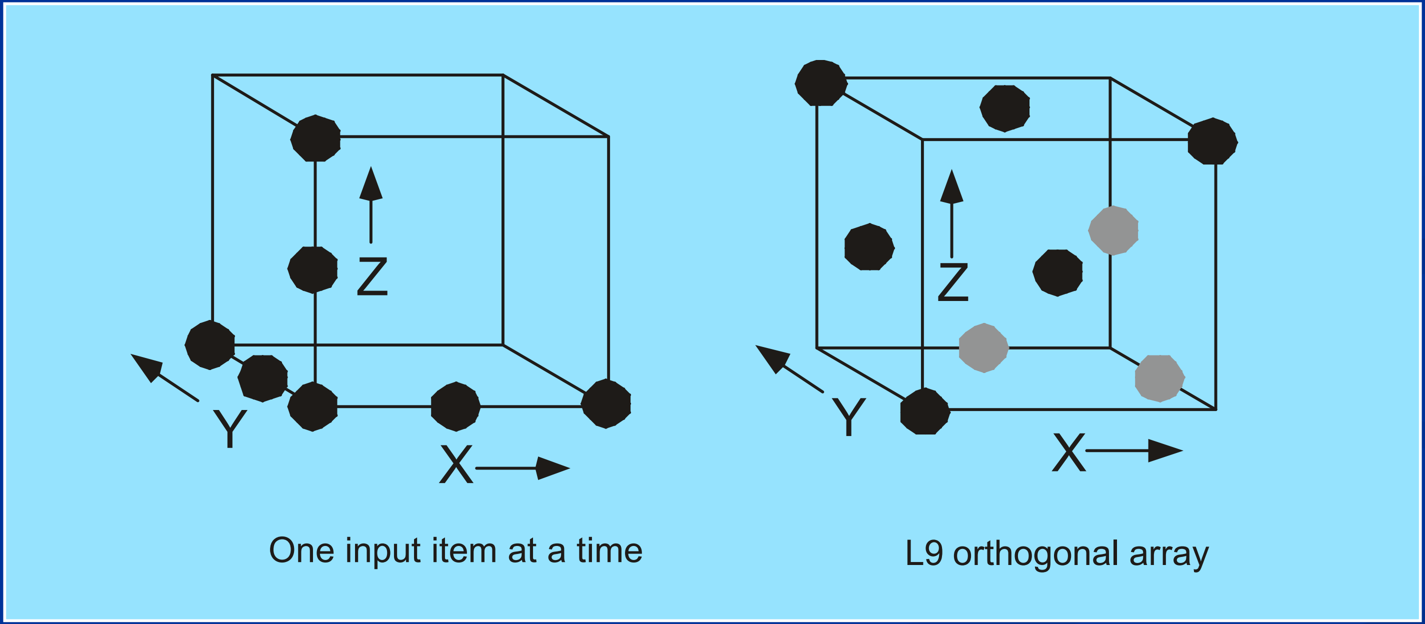
Example:
For a function that computes the square root of integer values in the range of 0 and 5000, the test cases must include the following values:
{0, -1,5000,5001}.

Comparison Testing

- Used only in situations in which the reliability of software is absolutely critical (e.g., human-rated systems)
  - Separate software engineering teams develop independent versions of an application using the same specification
  - Each version can be tested with the same test data to ensure that all provide identical output
  - Then all versions are executed in parallel with real-time comparison of results to ensure consistency

Orthogonal Array Testing

- Used when the number of input parameters is small and the values that each of the parameters may take are clearly bounded


Z
Y
X
One input item at a time
Z
Y
X
L9 orthogonal array