Integration Testing :- (Refer Roger B.page :- 459 - 464)

Validation Testing:- (Refer Roger B.page :- 467 - 469)

System Testing:- (Refer Roger B.page :- 470 - 472)

White Box Testing:- White-box testing, sometimes called glass-box testing, is a test-case design philosophy that uses the control structure described as part of component-level design to derive test cases. Using white-box testing methods, you can derive test cases that

(1) guarantee that all independent paths within a module have been exercised at

least once

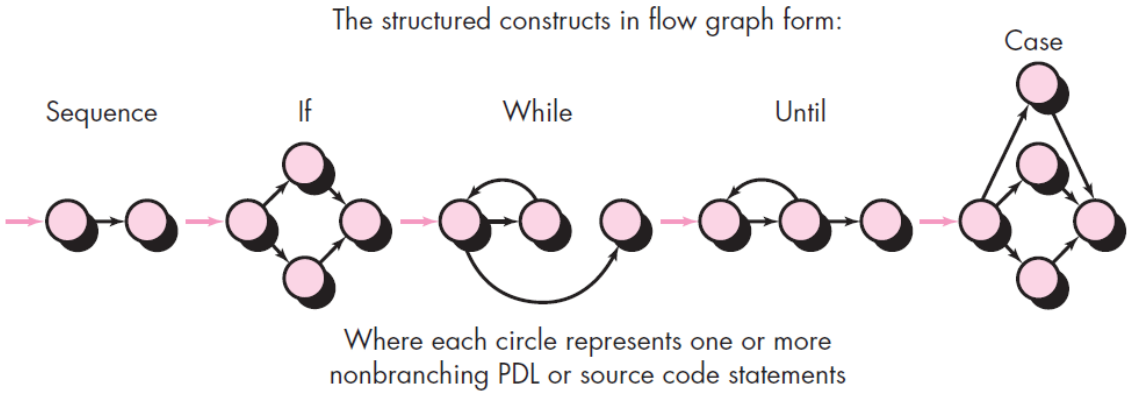
(2) exercise all logical decisions on their true and false sides,

(3) execute all loops at their boundaries and within their operational bounds, and (4) exercise

internal data structures to ensure their validity.

Basis Path Testing:- Basis path testing is a white-box testing technique first proposed by Tom McCabe. The basis path method 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. Test cases derived to exercise the basis set are guaranteed to execute every statement in the program at least one time during testing.

1. **Flow Graph:-** Before we consider the basis path method, a simple notation for the representation of control flow, called a flow graph (or program graph) must be introduced.The flow graph depicts logical control flow using the notation illustrated in Figure



To illustrate the use of a flow graph, consider the procedural design representation in Figure 18.1a. Here, a flowchart is used to depict program control structure. Figure 18.1b maps the flowchart into a corresponding flow graph (assuming that no compound conditions are contained in the decision diamonds of the flowchart). Referring to Figure 18.1b, each circle, called a flow graph node, represents one or more procedural statements. A sequence of process boxes and a decision diamond can map into a single node. The arrows on the flow graph, called edges or links, represent flow of control and are analogous to flowchart arrows. An edge must terminate at a node, even if the node does not represent any procedural statements (e.g., see the flow graph symbol for the if-then-else construct). Areas bounded by edges and nodes are called regions. When counting regions, we include the area outside the graph as a region

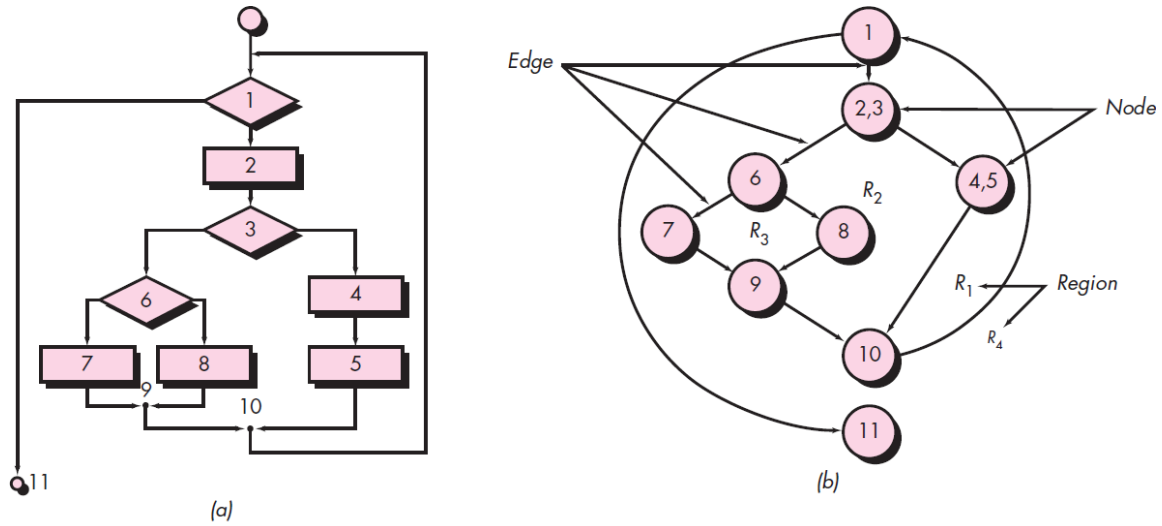


Figure 18.1

When compound conditions are encountered in a procedural design, the generation of a flow graph becomes slightly more complicated. A compound condition occurs when one or more Boolean operators (logical OR, AND, NAND, NOR) is present in a conditional statement. Referring to Figure 18.2, the program design language (PDL) segment translates into the flow graph shown. Note that a separate node is created for each of the conditions a and b in the statement IF a OR b. Each node that contains a condition is called a predicate node and is characterized by two or more edges emanating from it.

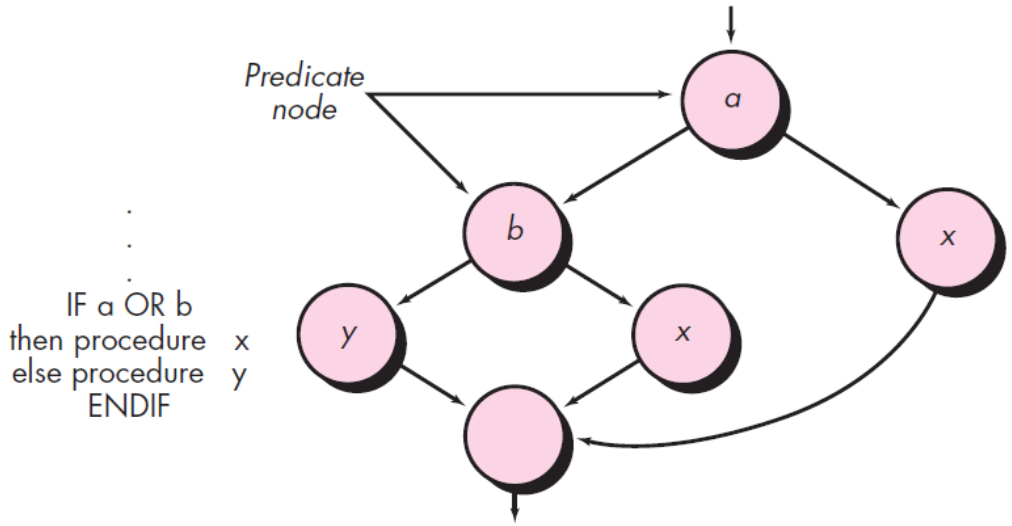


Figure 18.2

Independent Program Paths:- An independent path is any path through the program that introduces at least one new set of processing statements or a new condition. When stated in terms of a flow graph, an independent path must move along at least one edge that has not been traversed before the path is defined. For example, a set of independent paths for the flow graph illustrated in Figure 18.1b is

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.

Paths 1 through 4 constitute a basis set for the flow graph in Figure 18.2b. That is, if you can design tests to force execution of these paths (a basis set), every statement in the program will have been guaranteed to be executed at least one time and every condition will have been executed on its true and false sides.

It should be noted that the basis set is not unique. In fact, a number of different basis sets can be derived for a given procedural design.

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 you 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 has a foundation in graph theory and provides you with an extremely useful software metric. Complexity is computed in one of three ways:
1. The number of regions of the flow graph corresponds to the cyclomatic complexity.
2. Cyclomatic complexity V(G) for a flow graph G is defined as
V(G) = E - N + 2
where E is the number of flow graph edges and N is the number of flow
graph nodes.
3. Cyclomatic complexity V(G) for a flow graph G is also defined as
V(G) = P + 1
where P is the number of predicate nodes contained in the flow graph G.
Referring once more to the flow graph in Figure 18.1b, the cyclomatic complexity can
be computed using each of the algorithms just noted:
1. The flow graph has four regions.
2. V(G) = 11 edges - 9 nodes + 2 = 4.
3. V(G) =3 predicate nodes + 1 =4.
Therefore, the cyclomatic complexity of the flow graph in Figure 18.1b is 4.

Example:= Binary Search(Already Done in Class)

Control Structure Testing:- These broaden testing coverage and improve the quality of white-box testing.

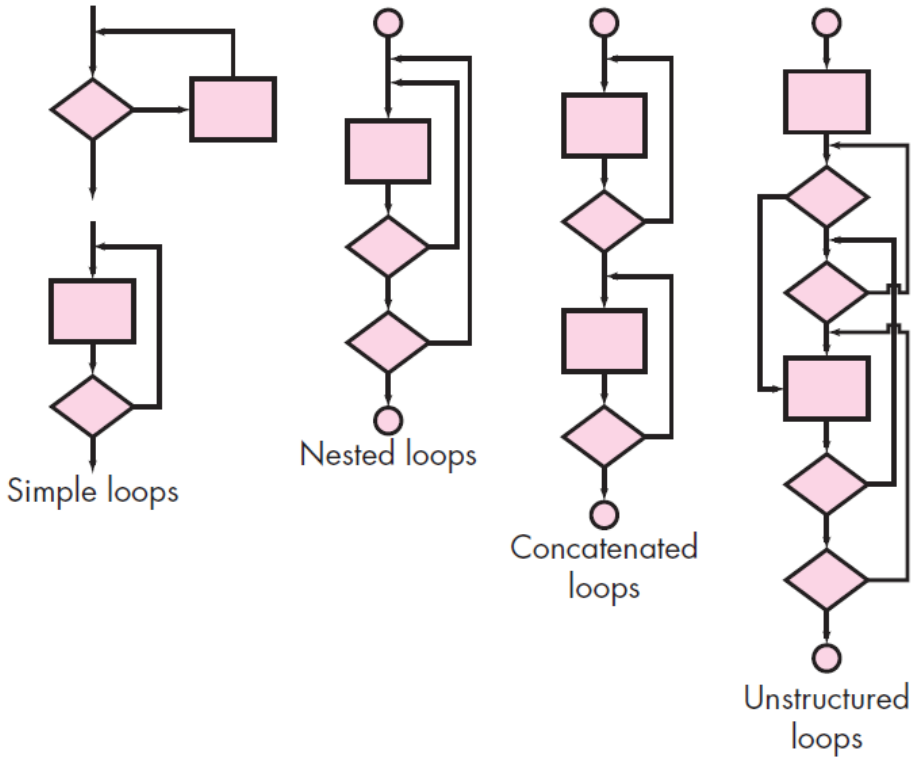
Condition Testing:- Condition testing [Tai89] is a test-case design method that exercises the logical conditions contained in a program module. A simple condition is a Boolean variable or
a relational expression, possibly preceded with one NOT (¬) operator. A relational expression takes the form
E1 <relational-operator> E2
where E1 and E2 are arithmetic expressions and <relational-operator> is one of the
following: <, <=, =, ≠ (nonequality), >, or >=.
A compound condition is composed of two or more simple conditions, Boolean operators, and parentheses. We assume that Boolean operators allowed in a compound condition include OR (), AND (&),and NOT (¬). A condition without relational expressions is referred to as a Boolean expression. If a condition is incorrect, then at least one component of the condition is incorrect.
Therefore, types of errors in a condition include Boolean operator errors (incorrect/missing/extra Boolean operators), Boolean variable errors, Boolean parenthesis errors, relational operator errors, and arithmetic expression errors. The condition testing method focuses on testing each condition in the program to ensure that it does not contain errors.

Loop Testing:- Loops are the cornerstone for the vast majority of all algorithms implemented in software. And yet, we often pay them little heed while conducting software tests.

Loop testing is a white-box testing technique that focuses exclusively on the validity of loop constructs. Four different classes of loops can be defined: simple loops, concatenated loops, nested loops, and unstructured loops

Simple loops. The following set of tests can be applied to simple loops, where *n* is the maximum number of allowable passes through the loop.

1. Skip the loop entirely.
2. Only one pass through the loop.
3. Two passes through the loop.
4. *m* passes through the loop where $m < n$.
5. n -1, n, n + 1 passes through the loop.



Nested loops. If we were to extend the test approach for simple loops to nested loops, the number of possible tests would grow geometrically as the level of nesting increases. This would result in an impractical number of tests. Beizer [Bei90] suggests an approach that will help to reduce the number of tests:

1. Start at the innermost loop. Set all other loops to minimum values.

2. Conduct simple loop tests for the innermost loop while holding the outer loops at their minimum iteration parameter (e.g., loop counter) values. Add other tests for out-of-range or excluded values.
3. Work outward, conducting tests for the next loop, but keeping all other outer loops at minimum values and other nested loops to "typical" values.
4. Continue until all loops have been tested.

Concatenated loops. Concatenated loops can be tested using the approach defined for simple loops, if each of the loops is independent of the other. However, if two loops are concatenated and the loop counter for loop 1 is used as the initial value for loop 2, then the loops are not independent. When the loops are not independent, the approach applied to nested loops is recommended.

Unstructured loops. Whenever possible, this class of loops should be redesigned to reflect the use of the structured programming constructs

Black Box Testing Techniques:-

Black-box testing, also called behavioral testing, focuses on the functional requirements of the software. That is, black-box testing techniques enable you 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 whitebox 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 database access,
(4) behavior or performance errors, and
(5) initialization and termination errors.
Tests are designed to answer the following questions:

- How is functional validity tested?
- How are 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?

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 test cases to be executed before the general error is observed.
In this method, input domain of a program is partitioned into a finite number of equivalence classes such that one can reasonably assume, but not be absolutely sure, that the test of a representative value of each class is equivalent to a test of any other value.

Two steps are required to implementing this method:

1. The equivalence classes are identified by taking each input condition and partitioning it into valid and invalid classes. For example, if an input condition specifies a range of values from 1 to 999, we identify one valid equivalence class [1<item<999]; and two invalid equivalence classes [item<1] and [item>999].

2. Generate the test cases using the equivalence classes identified in the previous step. This is performed by writing test cases covering all the valid equivalence classes. Then a test case is written for each invalid equivalence class so that no test contains more than one invalid class. This is to ensure that no two invalid classes mask each other.

   Most of the time, equivalence class testing defines classes of the input domain. However, equivalence classes should also be defined for output domain. Hence, we should design equivalence classes based on input and output domain.

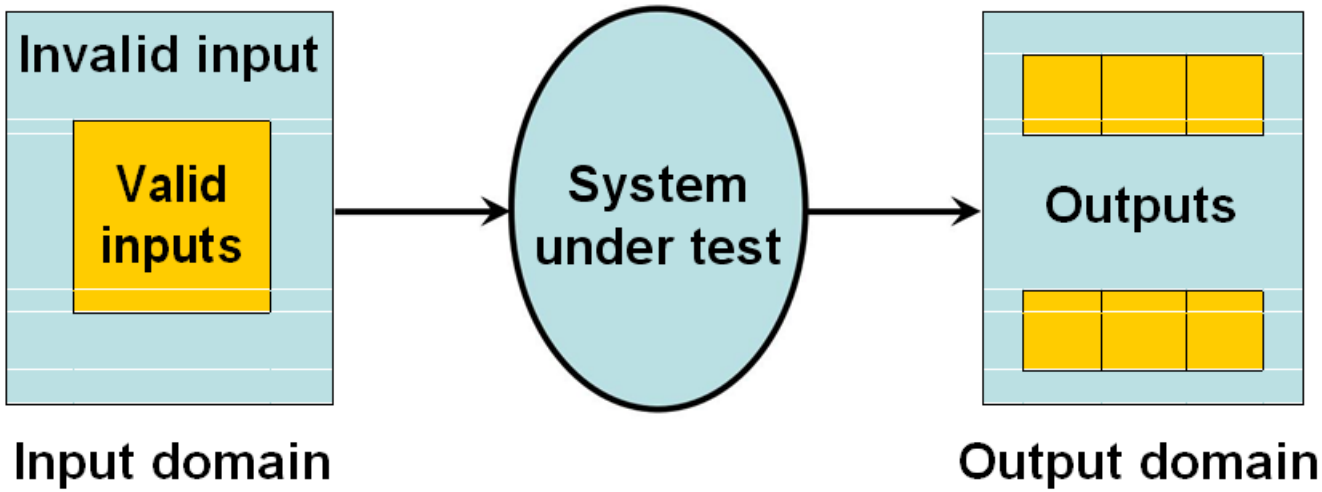


Fig. 7: Equivalence partitioning

Example- 8.I

Consider a program for the determination of the nature of roots of a quadratic equation. Its input is a triple of positive integers (say a,b,c) and values may be from interval [0,100]. The program output may have one of the following words.

[Not a quadratic equation; Real roots; Imaginary roots; Equal roots] Design the boundary value test cases. Identify the equivalence class test cases for output and input domains.

Solution

Output domain equivalence class test cases can be identified as follows:

O_1={<a,b,c>:Not a quadratic equation if a = 0}

O_1={<a,b,c>:Real roots if (b^2-4ac)>0}

O_1={<a,b,c>:Imaginary roots if (b^2-4ac)<0}

O_1={<a,b,c>:Equal roots if (b^2-4ac)=0}`

The number of test cases can be derived form above relations and shown below:

Test case	***a***	***b***	***c***	***Expected output***
1	0	50	50	Not a quadratic equation
2	1	50	50	Real roots

3	50	50	50	Imaginary roots
4	50	100	50	Equal roots

We may have another set of test cases based on input domain.

$I_1 = \{a: a = 0\}$

$I_2 = \{a: a < 0\}$

$I_3 = \{a: 1 \leq a \leq 100\}$

$I_4 = \{a: a > 100\}$

$I_5 = \{b: 0 \leq b \leq 100\}$

$I_6 = \{b: b < 0\}$

$I_7 = \{b: b > 100\}$

$I_8 = \{c: 0 \leq c \leq 100\}$

$I_9 = \{c: c < 0\}$

$I_{10} = \{c: c > 100\}$

Test Case	***a***	***b***	***c***	***Expected output***
1	0	50	50	Not a quadratic equation
2	-1	50	50	Invalid input
3	50	50	50	Imaginary Roots
4	101	50	50	invalid input
5	50	50	50	Imaginary Roots
6	50	-1	50	invalid input
7	50	101	50	invalid input
8	50	50	50	Imaginary Roots

9	50	50	-1	invalid input
10	50	50	101	invalid input

Here test cases 5 and 8 are redundant test cases. If we choose any value other than nominal, we may not have redundant test cases. Hence total test cases are 10+4=14 for this problem.

Example – 8.2

Consider a program for determining the Previous date. Its input is a triple of day, month and year with the values in the range

$1 \leq month \leq 12$

$1 \leq day \leq 31$

$1900 \leq year \leq 2025$

The possible outputs would be Previous date or invalid input date. Identify the equivalence class test cases for output & input domains.

Solution

Output domain equivalence class are:

O_1={<D,M,Y>: Previous date if all are valid inputs}

O_1={<D,M,Y>: Invalid date if any input makes the date invalid}

Test case	***M***	***D***	***Y***	***Expected output***
1	6	15	1962	14 June, 1962
2	6	31	1962	Invalid date

We may have another set of test cases which are based on input domain.

I_1={month: $1 \leq m \leq 12$}

I_2={month: $m < 1$}

I_3={month: $m > 12$}

I_4={day: $1 \leq D \leq 31$}

I_5={day: $D < 1$}

I_6={day: $D > 31$}

I_7={year: $1900 \leq Y \leq 2025$}

I_8={year: $Y < 1900$}

I_9={year: $Y > 2025$}

Inputs domain test cases are :

Test Case	***M***	***D***	***Y***	***Expected output***
1	6	15	1962	14 June, 1962
2	-1	15	1962	Invalid input
3	13	15	1962	invalid input
4	6	15	1962	14 June, 1962
5	6	-1	1962	invalid input
6	6	32	1962	invalid input
7	6	15	1962	14 June, 1962
8	6	15	1899	invalid input (Value out of range)
9	6	15	2026	invalid input (Value out of range)

Example 8.3 Consider a simple program to classify a triangle. Its inputs is a triple of positive integers (say x, y, z) and the date type for input parameters ensures that these will be integers greater than 0 and less than or equal to 100. The program output may be one of the following words:

[Scalene; Isosceles; Equilateral; Not a triangle] Identify the equivalence class test cases for output and input domain.

Solution

Output domain equivalence classes are:

O_1={<*x*,*y*,*z*>: Equilateral triangle with sides *x*,*y*,*z*}

O_1={<*x*,*y*,*z*>: Isosceles triangle with sides *x*,*y*,*z*}

O_1={<*x*,*y*,*z*>: Scalene triangle with sides *x*,*y*,*z*}

O_1={<*x*,*y*,*z*>: Not a triangle with sides *x*,*y*,*z*}

The test cases are:

Test case	*x*	*y*	*z*	*Expected Output*
1	50	50	50	Equilateral
2	50	50	99	Isosceles
3	100	99	50	Scalene
4	50	100	50	Not a triangle

Input domain based classes are:

$I_1=\{x: x < 1\}$

$I_2=\{x: x > 100\}$

$I_3=\{x: 1 \leq x \leq 100\}$

$I_4=\{y: y < 1\}$

$I_5=\{y: y > 100\}$

$I_6=\{y: 1 \leq y \leq 100\}$

$I_7=\{z: z < 1\}$

$I_8=\{z: z > 100\}$

$I_9=\{z: 1 \leq z \leq 100\}$

Some inputs domain test cases can be obtained using the relationship amongst x,y and z.

$I_{10}=\{< x,y,z >: x = y = z\}$

$I_{11}=\{< x,y,z >: x = y, x \neq z\}$

$I_{12}=\{< x,y,z >: x = z, x \neq y\}$

$I_{13}=\{< x,y,z >: y = z, x \neq y\}$

I_{14}={< x,y,z >: x ≠ y, x ≠ z, y ≠ z}

I_{15}={< x,y,z >: x = y + z}

I_{16}={< x,y,z >: x > y +z}

I_{17}={< x,y,z >: y = x +z}

I_{18}={< x,y,z >: y > x + z}

I_{19}={< x,y,z >: z = x + y}

I_{20}={< x,y,z >: z > x +y}

Test cases derived from input domain are:

Test case	***x***	***y***	***z***	***Expected Output***
1	0	50	50	Invalid input

2	101	50	50	Invalid input
3	50	50	50	Equilateral
4	50	0	50	Invalid input
5	50	101	50	Invalid input
6	50	50	50	Equilateral
7	50	50	0	Invalid input
8	50	50	101	Invalid input
9	50	50	50	Equilateral
10	60	60	60	Equilateral
11	50	50	60	Isosceles
12	50	60	50	Isosceles
13	60	50	50	Isosceles

Test case	x	y	z	Expected Output
14	100	99	50	Scalene
15	100	50	50	Not a triangle
16	100	50	25	Not a triangle
17	50	100	50	Not a triangle
18	50	100	25	Not a triangle
19	50	50	100	Not a triangle
20	25	50	100	Not a triangle

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. Boundary value analysis leads to a selection of test cases that exercise bounding values.

Boundary value analysis is a test-case design technique that complements equivalence partitioning. Rather than selecting any element of an equivalence class, BVA leads to the selection of test cases at the "edges" of the class. Rather than focusing solely on input conditions, BVA derives test cases from the output domain as well

Guidelines for BVA are similar in many respects to those provided for equivalence partitioning:

1. If an input condition specifies a range bounded by values *a* and *b,* test cases should be designed with values *a* and *b* and 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
versus 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., a table has a defined limit of 100 entries), be certain to design a test case to exercise the data structure at its boundary.

Consider a program with two input variables x and y. These input variables have specified boundaries as:

$$a \leq x \leq b$$

$$c \leq y \leq d$$

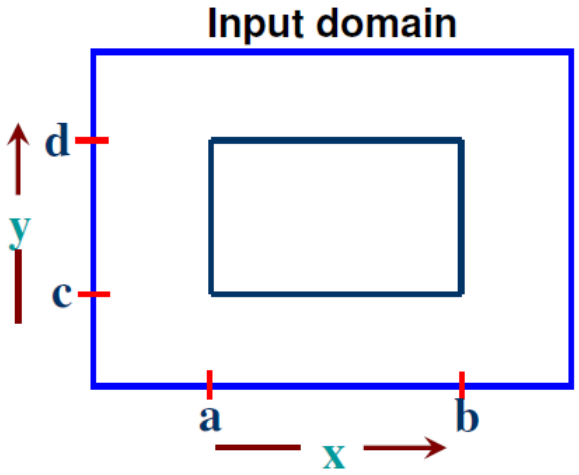


Fig.4: Input domain for program having two input variables

The boundary value analysis test cases for our program with two inputs variables (x and y) that may have any value from 100 to 300 are: (200,100),

(200,101), (200,200), (200,299), (200,300), (100,200), (101,200), (299,200) and (300,200). This input domain is shown in Fig. 8.5. Each dot represent a test case and inner rectangle is the domain of legitimate inputs. Thus, for a program of n variables, boundary value analysis yield **4n + 1** test cases.

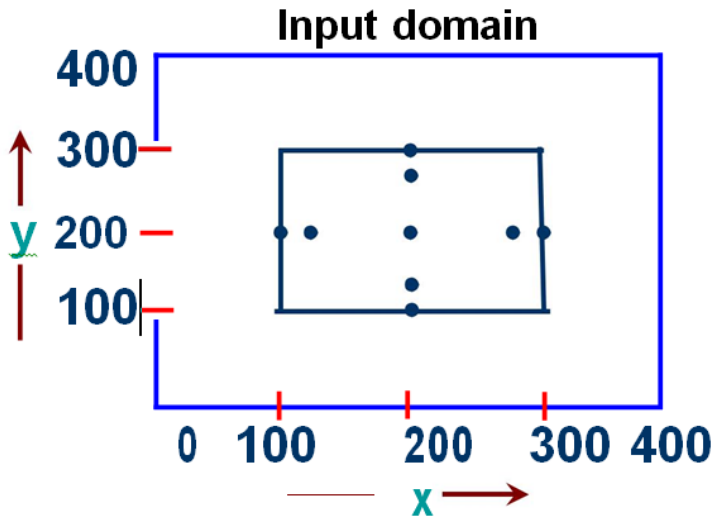


Fig. 5: Input domain of two variables x and y with boundaries [100,300] each

Example- 8.I

Consider a program for the determination of the nature of roots of a quadratic equation. Its input is a triple of positive integers (say a,b,c) and values may be from interval [0,100]. The program output may have one of the following words.

[Not a quadratic equation; Real roots; Imaginary roots; Equal roots] Design the boundary value test cases.

Solution

Quadratic equation will be of type:

$$ax^2+bx+c=0$$

Roots are real if $(b^2-4ac)>0$

Roots are imaginary if $(b^2-4ac)<0$

Roots are equal if $(b^2-4ac)=0$

Equation is not quadratic if a=0

The boundary value test cases are :

Test Case	***a***	***b***	***c***	***Expected output***
1	0	50	50	Not Quadratic
2	1	50	50	Real Roots
3	50	50	50	Imaginary Roots
4	99	50	50	Imaginary Roots
5	100	50	50	Imaginary Roots
6	50	0	50	Imaginary Roots

7	50	1	50	Imaginary Roots
8	50	99	50	Imaginary Roots
9	50	100	50	Equal Roots
10	50	50	0	Real Roots
11	50	50	1	Real Roots
12	50	50	99	Imaginary Roots
13	50	50	100	Imaginary Roots

Example – 8.2

Consider a program for determining the Previous date. Its input is a triple of day, month and year with the values in the range

$$1 \leq \text{month} \leq 12$$

$1 \leq day \leq 31$

$1900 \leq year \leq 2025$

The possible outputs would be Previous date or invalid input date. Design the boundary value test cases.

Solution

The Previous date program takes a date as input and checks it for validity.

If valid, it returns the previous date as its output.

With single fault assumption theory, 4n+1 test cases can be designed and which are equal to 13.

The boundary value test cases are:

Test Case	*Month*	*Day*	*Year*	*Expected output*

1	6	15	1900	14 June, 1900
2	6	15	1901	14 June, 1901
3	6	15	1962	14 June, 1962
4	6	15	2024	14 June, 2024
5	6	15	2025	14 June, 2025
6	6	1	1962	31 May, 1962
7	6	2	1962	1 June, 1962
8	6	30	1962	29 June, 1962
9	6	31	1962	Invalid date
10	1	15	1962	14 January, 1962
11	2	15	1962	14 February, 1962
12	11	15	1962	14 November, 1962
13	12	15	1962	14 December, 1962

Example – 8.3

Consider a simple program to classify a triangle. Its inputs is a triple of positive integers (say x, y, z) and the date type for input parameters ensures that these will be integers greater than 0 and less than or equal to 100. The program output may be one of the following words:

[Scalene; Isosceles; Equilateral; Not a triangle] Design the boundary value test cases.

Solution

The boundary value test cases are shown below:

Test case	*x*	*y*	*z*	*Expected Output*
1	50	50	1	Isosceles

2	50	50	2	Isosceles
3	50	50	50	Equilateral
4	50	50	99	Isosceles
5	50	50	100	Not a triangle
6	50	1	50	Isosceles
7	50	2	50	Isosceles
8	50	99	50	Isosceles
9	50	100	50	Not a triangle
10	1	50	50	Isosceles
11	2	50	50	Isosceles
12	99	50	50	Isosceles
13	100	50	50	Not a triangle