Unit-2

Types of Testing

Que 1 : What is Black- box testing ?

- Black-box technique is one of the major techniques in dynamic testing for designing effective test cases.
- This technique considers only the functional requirements of the software or module.
- In other words, the structure or logic of the software is not considered.
- Therefore, this is also known as functional testing.
- The software system is considered as a black box, taking no notice of its internal structure, so it is also called as black-box testing technique.
- in black-box technique, test cases are designed based on functional specifications.
- Input test data is given to the system, which is a black box to the tester, and results are checked
  against expected outputs after executing the software

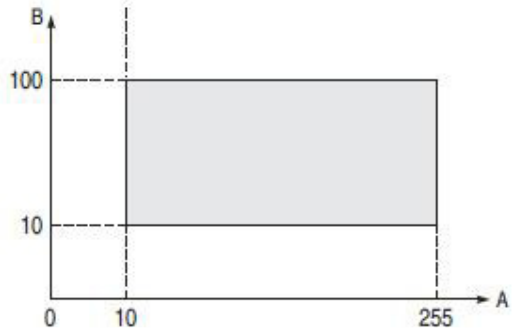
Que 2 : Why Black-box testing is required ?

Black-box testing attempts to find errors in the following categories

- To test the modules independently
- To test the functional validity of the software so that incorrect or missing functions can be recognized
- To look for interface errors.
- To test the system behaviour and check its performance
- To test the maximum load or stress on the system
- To test the software such that the user/customer accepts the system within defined acceptable limits

Que 3 : Black-box testing Techniques

BOUNDARY VALUE ANALYSIS (BVA)

- An effective test case design requires test cases to be designed such that they maximize the probability of finding errors.
- BVA technique addresses this issue.
- With the experience of testing team, it has been observed that test cases designed with boundary input values have a high chance to find errors.
- It means that most of the failures crop up due to boundary values.
- BVA is considered a technique that uncovers the bugs at the boundary ofinput values.
- Here, boundary means the maximum or minimum value takenby the input domain. For example, if *A* is an integer between 10 and 255, then boundary checking can be on 10(9,10,11) and on 255(256,255,254).
- Similarly,*B* is another integer variable between 10 and 100, then boundary checkingcan be on 10(9,10,11) and 100(99,100,101), as shown in Fig.
- 

BVA offers several methods to design test cases as discussed in the following sections.

BOUNDARY VALUE CHECKING (BVC)

In this method, the test cases are designed by holding one variable at its extreme value and other variables at their nominal values in the input domain.

The variable at its extreme value can be selected at:

1. Maximum value (Max)
2. Minimum value (Min)
3. Value just below the maximum value (Max−)
4. Value just above the minimum value (Min+)

Example 1:

A Program computes a^b a lies in range [1,10] and b lies in range [1,5] deign test cases for BVC.

Extremevalues for A and B are as follow

Extreme	For A	For B
Max	10	5

Min	1	1
Max-1	9	4
Min+1	2	2

Nominal value for A is = (max+min)/2
N.V =(10+1)/2
N.v = 11/2=5.5=5

So, n.v for A is 5

Nominal value for B is = (max+min)/2
N.V =(5+1)/2
N.v = 6/2=3=3

So, n.v for B is 3

Test cases to design =4n+1
4(2)+1=9
So,here we have to design 9 cases for bvc

TC	A	B	a^b
001	5	5	3125
002	5	1	5
003	5	4	625
004	5	2	25
005	10	3	1000
006	1	3	1
007	9	3	27
008	2	3	28
009	5	3	125

Example 2: A program reads an integer number within the range [1,100] and determines whether it is a prime number or not. Design test cases for this program using BVC.

Solution

(a) Test cases using BVC Since there is one variable, the total number of test cases will be 4n + 1 = 5.

In our example, the set of minimum and maximum values is shown below:
Min value = 1
Min+ 1 = 2
Max value = 100

Max– 1 = 99
Nominal value = 50–55
Using these values, test cases can be designed as shown below:
Test Case ID Integer Variable Expected Output

Test Case ID	Integer Variable	Expected Output
1	1	Not a prime number
2	2	Prime number
3	100	Not a prime number
4	99	Not a prime number
5	53	Prime number

Example 3 :A program reads three numbers, A, B, and C, within the range [1, 50] and prints the largest number. Design test cases for this program using BVC

there are three variables, A, B, and C,the total number of test cases will be 4n + 1 = 13. The set of boundary
values is shown below:

Min value = 1 Min+ 1 = 2 Max value = 50 Max– 1 = 49 Nominal value = 25–30

Using these values, test cases can be designed as shown below:

Test Case ID	A	B	C	Expected Output
1	1	25	27	C is largest
2	2	25	28	C is largest
3	49	25	25	B and C are Equal
4	50	25	29	A is largest
5	25	1	30	C is largest
6	25	2	26	C is largest
7	25	49	27	B is largest
8	25	50	28	B is largest
9	25	28	1	B is largest
10	25	27	2	B is largest
11	25	26	49	C is largest
12	25	26	50	C is largest
13	25	25	25	Three are equal

Example 4 : A program determines the next date in the calendar. Its input is entered in the form of <ddmmyyyy> with the following range:

1≤ mm ≤ 12
1 ≤ dd ≤ 31
1900 ≤ yyyy ≤ 2025
Its output would be the next date or it will display 'invalid date.' Design test cases for this program using BVC, robust testing, and worst-case testing methods

Solution

there are three variables, month, day,and year, the total number of test cases will be 4n + 1 = 13. The set ofboundary values is shown below:

	Month	**Day**	**Year**
Min value	1	1	1900
Max value	12	31	2025
Max– value	11	30	2024
Min+ value	2	2	1901
Nominal value	6	16	1962

Using these values, test cases can be designed as shown below:

Test Case ID	**Month**	**Day**	**Year**	**Expected Output**
1	1	16	1962	16-1-1962
2	12	16	1962	16-2-1962
3	11	16	1962	16-11-1962
4	2	16	1962	16-12-1962
5	6	1	1962	2/6/1962
6	6	2	1962	3/6/1962
7	6	30	1962	1/7/1962
8	6	31	1962	Invalid
9	6	16	1900	16-6-1900
10	6	16	1901	16-6-1901
11	6	16	2024	16-6-2024
12	6	16	2025	16-6-2025
13	6	16	1962	16-6-1962

ROBUST TESTING METHOD

The variable at its extreme value can be selected at:

1. Maximum value (Max)

2. Minimum value (Min)
3. Value just below the maximum value (Max−)
4. Value just above the minimum value (Min+)
5. Value just above the maximum value (Max+)
6. Value just below the minimum value (Min-)

Example 1: A program reads an integer number within the range [1,100] and determines whether it is a prime number or not. Design test cases for this program using Robust testing methods.
Solution

there is one variable, the total
number of test cases will be 6*n* + 1 = 7. The set of boundary values is shown below:

Min value = 1 Min– 1 = 0 Min+ 1 = 2 Max value = 100 Max– 1 = 99 Max+ 1 = 101 Nominal value = 50–55

Using these values, test cases can be designed as shown below:

Test Case ID	Integer Variable	Expected Output
1	0	Invalid input
2	1	Not a number
3	2	Prime number
4	100	Not a number
5	99	Not a number
6	101	Invalid input
7	53	Prime number

Example 2: A program computes a^b where a lies in the range [1,10] and b within [1,5].Design test cases for this program using robust testing method.

Solution

There are two variables, a and b, the total number of test cases will be 6n + 1 = 13. The set of boundary values is shown below:

	a	b
Min value	1	1
Min– value	0	0
Min+ value	2	2
Max value	10	5
Max+ value	11	6
Max– value	9	4
Nominal value	5	3

Using these values, test cases can be designed as shown below:

Test Case ID	a	b	Expected Output
1	0	3	Invalid input
2	1	3	1
3	2	3	8
4	10	3	1000
5	11	3	Invalid input
6	9	3	729
7	5	0	Invalid input
8	5	1	5
9	5	2	25
10	5	4	625
11	5	5	3125
12	5	6	Invalid input
13	5	3	125

Example 3: A program reads three numbers, A, B, and C, within the range [1, 50] and prints the largest number. Design test cases for this program using robust testing methods.

Min value = 1
Min– value = 0
Min+ value = 2
Max value = 50

Max+ value = 51
Max– value = 49
Nominal value = 25–30

Test Case ID	A	B	C	Expected Output
1	0	25	27	Invalid input
2	1	25	27	C is largest
3	2	25	28	C is largest
4	49	25	25	B and C are Largest largest
5	50	25	29	A is largest
6	51	27	25	Invalid input
7	25	0	26	Invalid input
8	25	1	30	C is largest
9	25	2	26	C is largest
10	25	49	27	B is Largest
11	25	50	28	B is Largest
12	26	51	25	Invalid input
13	25	25	0	Invalid input
14	25	28	1	B is Largest
15	25	27	2	B is Largest
16	25	26	49	C is largest
17	25	26	50	C is largest
18	25	29	51	Invalid input
19	25	25	25	Three are equal

WORST-CASE TESTING METHOD

We can again extend the concept of BVC by assuming more than one variable on the boundary.

It is called worst – case testing method .

It can be generalized that for n input variables in a module , 5^n test cases can be designed with worst case testing.

Example 1:

A Program computes a^b a lies in range [1,10] and b lies in range [1,5] deign test cases for ROBUST TEST METHOD.

Extremevalues for A and B are as follow

Extreme	For A	For B
Max	10	5
Min	1	1

Max-1	9	4
Min+1	2	2
Nominal Value	5	3

Test cases to design $=5^n$

$5^2=25$

So,here we have to design 25 cases for WORST-CASE

Test Case ID	**a**	**b**	**Expected Output**
1	1	1	1
2	1	2	1
3	1	3	3
4	1	4	1
5	1	5	1
6	2	1	2
7	2	2	4
8	2	3	8
9	2	4	16
10	2	5	32
11	5	1	5
12	5	2	25
13	5	3	125
14	5	4	625
15	5	5	3125
16	9	1	9
17	9	2	81
18	9	3	729
19	9	4	6561
20	9	5	59049
21	10	1	10
22	10	2	100
23	10	3	1000
24	10	4	10000
25	10	5	100000

Example 2 : A program reads an integer number within the range [1,100] and determines whether it is a prime number or not. Design test cases for this program usingworst-case testing methods.

Solution :

Test cases using worst-case Since there is one variable, the total number of test cases will be $5^1 = 5$

In our example, the set of minimum and maximum values is shown below:
Min value = 1
Min+ 1 = 2
Max value = 100
Max– 1 = 99
Nominal value = 50–55
Using these values, test cases can be designed as shown below:
Test Case ID Integer Variable Expected Output

Test Case ID	**Integer Variable**	**Expected Output**
1	1	Not a prime number
2	2	Prime number
3	100	Not a prime number
4	99	Not a prime number
5	53	Prime number

Example 3: A program reads three numbers, A, B, and C, within the range [1, 50] and prints the largest number. Design test cases for this program using worst-case testing methods.

Solution : there are three variables, A,B, and C, the total number of test cases will be 5n = 125.

The set of boundary values is shown below:

Min value = 1 Min+ value = 2 Max value = 50 Max– value = 49 Nominal value = 25–30

Test Case ID	A	B	C	Expected Output
1	1	1	1	All three are equal
2	1	1	2	C is greatest
3	1	1	25	C is greatest
4	1	1	49	C is greatest
5	1	1	50	C is greatest
6	1	2	1	B is greatest
7	1	2	2	B and C
8	1	2	25	C is greatest
9	1	2	49	C is greatest
10	1	2	50	C is greatest
11	1	25	1	B is greatest
12	1	27	2	B is greatest
13	1	26	25	B is greatest
14	1	25	49	B is greatest
15	1	27	50	C is greatest
16	1	49	1	B is greatest
17	1	49	2	B is greatest
18	1	49	25	B is greatest
19	1	49	49	B and C
20	1	49	50	C is greatest
21	1	50	1	B is greatest
22	1	50	2	B is greatest
23	1	50	25	B is greatest
24	1	50	49	B is greatest
25	1	50	50	B and C
26	2	1	1	A is largest
27	2	1	2	A and C
28	2	1	25	C is greatest
29	2	1	49	C is greatest
30	2	1	50	C is greatest
31	2	2	1	A and B
32	2	2	2	All three are equal
33	2	2	25	C is greatest
34	2	2	49	C is greatest

35	2	2	50	C is greatest
36	2	25	1	B is greatest
37	2	27	2	B is greatest
38	2	28	25	B is greatest
39	2	26	49	C is greatest
40	2	28	50	C is greatest
41	2	49	1	B is greatest
42	2	49	2	B is greatest
43	2	49	25	B is greatest
44	2	49	49	B and C
45	2	49	50	C is greatest
46	2	50	1	B is greatest
47	2	50	2	B is greatest
48	2	50	25	B is greatest
49	2	50	49	B is greatest
50	2	50	50	B and C
51	25	1	1	A is greatest
52	25	1	2	A is greatest
53	25	1	25	A and C
54	25	1	49	C is greatest
55	25	1	50	C is greatest
56	25	2	1	A is greatest
57	25	2	2	A is greatest
58	25	2	25	A and C
59	25	2	49	C is greatest
60	25	2	50	C is greatest
61	25	27	1	B is greatest
62	25	26	2	B is greatest
63	25	25	25	All three are equal
64	25	28	49	C is greatest
65	25	29	50	C is greatest
66	25	49	1	B is greatest
67	25	49	2	B is greatest
68	25	49	25	B is greatest
69	25	49	49	B is greatest
70	25	49	50	C is greatest

71	25	50	1	B is greatest
72	25	50	2	B is greatest
73	25	50	25	B is greatest
74	25	50	49	B is greatest
75	25	50	50	B is greatest
76	49	1	1	A is greatest
77	49	1	2	A is greatest
78	49	1	25	A is greatest
79	49	1	49	A and C
80	49	1	50	C is greatest
81	49	2	1	A is greatest
82	49	2	2	A is greatest
83	49	2	25	A is greatest
84	49	2	49	A and C
85	49	2	50	C is greatest
86	49	25	1	A is greatest
87	49	29	2	A is greatest
88	49	25	25	A is greatest
89	49	27	49	A and C
90	49	28	50	C is greatest
91	49	49	1	A and B
92	49	49	2	A and B
93	49	49	25	A and B
94	49	49	49	All three are equal
95	49	49	50	C is greatest
96	49	50	1	B is greatest
97	49	50	2	B is greatest
98	49	50	25	B is greatest
99	49	50	49	B is greatest
100	49	50	50	B and C
101	50	1	1	A is greatest
102	50	1	2	A is greatest
103	50	1	25	A is greatest
104	50	1	49	A is greatest
105	50	1	50	A and C
106	50	2	1	A is greatest

107	50	2	2	A is greatest
108	50	2	25	A is greatest
109	50	2	49	A is greatest
110	50	2	50	A and C
111	50	26	1	A is greatest
112	50	25	2	A is greatest
113	50	27	25	A is greatest
114	50	29	49	A is greatest
115	50	30	50	A and C
116	50	49	1	A is greatest
117	50	49	2	A is greatest
118	50	49	26	A is greatest
119	50	49	49	A is greatest
120	50	49	50	A and C
121	50	50	1	A and B
122	50	50	2	A and B
123	50	50	26	A and B
124	50	50	49	A and B
125	50	50	50	All three are equal

Example 4 : A program determines the next date in the calendar. Its input is entered in the form of <ddmmyyyy> with the following range:
1 ≤ mm ≤ 12
1 ≤ dd ≤ 31
1900 ≤ yyyy ≤ 2025
Its output would be the next date or it will display 'invalid date.' Design test cases for this program using worst-case testingmethods.
Solution

The total number of test cases will be 5n = 125. The set of boundary values is shown below:

	Month	Day	Year
Min Value	1	1	1900
Min+ 1	2	2	1901
Max value	12	31	2025
Max− 1	11	30	2024
Nominal value	6	15	1962

Test Case ID	Month	Day	Year	Expected Output
1	1	1	1900	2/1/1900
2	1	1	1901	2/1/1901
3	1	1	1962	2/1/1962
4	1	1	2024	2/1/2024
5	1	1	2025	2/1/2025
6	1	2	1900	3/1/1900
7	1	2	1901	3/1/1901
8	1	2	1962	3/1/1962
9	1	2	2024	3/1/2024
10	1	2	2025	3/1/2025
11	1	15	1900	16-1-1900
12	1	15	1901	16-1-1901
13	1	15	1962	16-1-1962
14	1	15	2024	16-1-2024
15	1	15	2025	16-1-2025
16	1	30	1900	31-1-1900
17	1	30	1901	31-1-1901
18	1	30	1962	31-1-1962
19	1	30	2024	31-1-2024
20	1	30	2025	31-1-2025
21	1	31	1900	1/2/1900
22	1	31	1901	1/2/1901
23	1	31	1962	1/2/1962
24	1	31	2024	1/2/2024
25	1	31	2025	1/2/2025
26	2	1	1900	2/2/1900
27	2	1	1901	2/2/1901
28	2	1	1962	2/2/1962
29	2	1	2024	2/1/2024
30	2	1	2025	2/2/2025
31	2	2	1900	3/2/1900
32	2	2	1901	3/2/1901
33	2	2	1962	3/2/1962
34	2	2	2024	3/2/2024

35	2	2	2025	3/2/2025
36	2	15	1900	16-2-1900
37	2	15	1901	16-2-1901
38	2	15	1962	16-2-1962
39	2	15	2024	16-2-2024
40	2	15	2025	16-2-2025
41	2	30	1900	Invalid date
42	2	30	1901	Invalid date
43	2	30	1962	Invalid date
44	2	30	2024	Invalid date
45	2	30	2025	Invalid date
46	2	31	1900	Invalid date
47	2	31	1901	Invalid date
48	2	31	1962	Invalid date
49	2	31	2024	Invalid date
50	2	31	2025	Invalid date
51	6	1	1900	2/6/1900
52	6	1	1901	2/6/1901
53	6	1	1962	2/6/1962
54	6	1	2024	2/6/2024
55	6	1	2025	2/6/2025
56	6	2	1900	3/6/1900
57	6	2	1901	3/6/1901
58	6	2	1962	3/6/1962
59	6	2	2024	3/6/2024
60	6	2	2025	3/6/2025
61	6	15	1900	16-6-1900
62	6	15	1901	16-6-1901
63	6	15	1962	16-6-1962
64	6	15	2024	16-6-2024
65	6	15	2025	16-6-2025
66	6	30	1900	1/7/1900
67	6	30	1901	1/7/1901
68	6	30	1962	1/7/1962
69	6	30	2024	1/7/2024
70	6	30	2025	1/7/2025
71	6	31	1900	Invalid

72	6	31	1901	Invalid
73	6	31	1962	Invalid
74	6	31	2024	Invalid
75	6	31	2025	Invalid
76	11	1	1900	2/11/1900
77	11	1	1901	2/11/1901
78	11	1	1962	2/11/1962
79	11	1	2024	2/11/2024
80	11	1	2025	2/11/2025
81	11	2	1900	3/11/1900
82	11	2	1901	3/11/1901
83	11	2	1962	3/11/1962
84	11	2	2024	3/11/2024
85	11	2	2025	3/11/2025
86	11	15	1900	16-11-1900
87	11	15	1901	16-11-1901
88	11	15	1962	16-11-1962
89	11	15	2024	16-11-2024
90	11	15	2025	16-11-2025
91	11	30	1900	1/12/1900
92	11	30	1901	1/12/1901
93	11	30	1962	1/12/1962
94	11	30	2024	1/12/2024
95	11	30	2025	1/12/2025
96	11	31	1900	Invalid
97	11	31	1901	Invalid
98	11	31	1962	Invalid
99	11	31	2024	Invalid
100	11	31	2025	Invalid
101	12	1	1900	2/12/1900
102	12	1	1901	2/12/1901
103	12	1	1962	2/12/1962
104	12	1	2024	2/12/2024
105	12	1	2025	2/12/2025
106	12	2	1900	3/12/1900
107	12	2	1901	3/12/1901
108	12	2	1962	3/12/1962

109	12	2	2024	3/12/2024
110	12	2	2025	3/12/2025
111	12	15	1900	16-12-1900
112	12	15	1901	16-12-1901
113	12	15	1962	16-12-1962
114	12	15	2024	16-12-2024
115	12	15	2025	16-12-2025
116	12	30	1900	31-12-1900
117	12	30	1901	31-12-1901
118	12	30	1962	31-12-1962
119	12	30	2024	31-12-2024
120	12	30	2025	31-12-2025
121	12	31	1900	1/1/1901
122	12	31	1901	1/1/1902
123	12	31	1962	1/1/1963
124	12	31	2024	1/1/2025
125	12	31	2025	1/1/2026

Que 4 : Equivalence Class Testing

- Equivalence partitioning is a technique of software testing in which input data is divided into partitions of valid and invalid values, and it is mandatory that all partitions must show the same behaviour.
- If a condition of one partition is true, then the condition of another equal partition must also be true, and if a condition of one partition is false, then the condition of another equal partition must also be false.
- The principle of equivalence partitioning is, test cases should be designed to cover each partition at least once.
- Each value of every equal partition must display the same behaviour as other.
- The equivalence partitions are derived from requirements and specifications of the software.
- The advantage of this approach is, it helps to reduce the time of testing due to a smaller number of test cases from infinite to finite.
- It is applicable at all levels of the testing process

Equivalence partitioning method for designing test cases has the following goals:

Completeness

- Without executing all the test cases, we try to trace the completeness of testing domain.

Non-redundancy

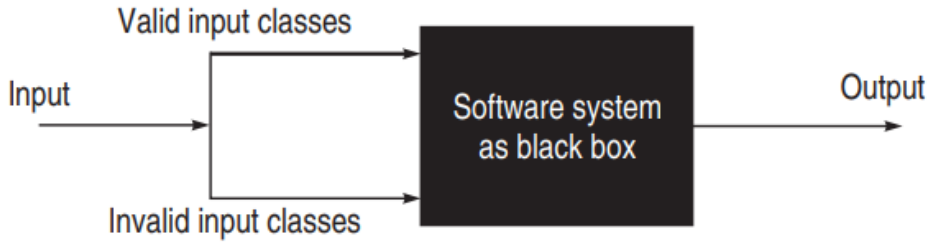
- When the test cases are executed having inputs from the same class, then there is redundancy in executing the test cases.
- Time and resources are wasted in executing these redundant test cases, as they explore the same type of bug.
- Thus, the goal of equivalence partitioning method is to reduce these redundant test cases.

To use equivalence partitioning, one needs to perform two steps:

1. **Identify equivalence classes**
    - Different equivalence classes are designed by grouping inputs for which the behaviour pattern of the module is similar.
    - The basis of creating equivalence classes like this is the assumption that if the specifications require exactly the same behaviour for each element in a class of values, then the program is likely to be

constructed such that it either succeeds or fails for each value in that class.

- For example, the specifications of a module that determines the absolute value for integers specify different behaviour patterns for positive and negative integers.
- In this case, we will form two classes:
- one consisting of positive integers and another consisting of negative integers
- Two types of classes can always be identified as discussed below: Valid equivalence classes These classes consider valid inputs to the program.
- Invalid equivalence classes One must not be restricted to valid inputs only.
- We should also consider invalid inputs that will generate error conditions or unexpected behaviour of the program, as shown in Figure.



2. Identify test cases

- A few guidelines are given below to identify test cases through generated equivalence classes:
  - Assign a unique identification number to each equivalence class.
  - Write a new test case covering as many of the uncovered valid equivalence classes as possible, until all valid equivalence classes have been covered by test cases.
  - Write a test case that covers one uncovered invalid equivalence classes, until all invalid equivalence classes have been covered by test cases.

Example :

- A program reads three numbers, A, B, and C, with a range [1, 50] and prints the largest number. Design test cases for this program using equivalence class testing technique.

Solution

First we partition the domain of input as valid input values and invalid values, getting the following classes:

I1 = {<A,B,C> : $1 \le A \le 50$}
I2 = {<A,B,C> : $1 \le B \le 50$}
I3 = {<A,B,C> : $1 \le C \le 50$}
I4 = {<A,B,C> : $A < 1$}
I5 = {<A,B,C> : $A > 50$}
I6 = {<A,B,C> : $B < 1$}
I7 = {<A,B,C> : $B > 50$}
I8 = {<A,B,C> : $C < 1$}
I9 = {<A,B,C> : $C > 50$}

Now the test cases can be designed from the above derived classes, taking one test case from each class such that the test case covers maximum valid input classes, and separate test cases for each invalid class.

The test cases are shown below:

Test case ID	A	B	C	Expected result	Classes covered by the test case
1	13	25	36	C is greatest	I_1, I_2, I_3
2	0	13	45	Invalid input	I_4
3	51	34	17	Invalid input	I_5
4	29	0	18	Invalid input	I_6
5	36	53	32	Invalid input	I_7
6	27	42	0	Invalid input	I_8
7	33	21	51	Invalid input	I_9

We can derive another set of equivalence classes based on some possibilities for three integers, A, B, and C. These are given below:

I1 = {<A,B,C> : $A > B, A > C$}
I2 = {<A,B,C> : $B > A, B > C$}
I3 = {<A,B,C> : $C > A, C > B$}
I4 = {<A,B,C> : $A = B, A \ne C$}
I5 = {<A,B,C> : $B = C, A \ne B$}
I6 = {<A,B,C> : $A = C, C \ne B$}
I7 = {<A,B,C> : $A = B = C$}

Test case ID	A	B	C	Expected Result	Classes Covered by the test case
1	25	13	13	A is greatest	I_1, I_5
2	25	40	25	B is greatest	I_2, I_6
3	24	24	37	C is greatest	I_3, I_4
4	25	25	25	All three are equal	I_7

Example 2 :

A program determines the next date in the calendar. Its input is entered in the form of <ddmmyyyy> with the following range:
1 ≤ mm ≤ 12
1 ≤ dd ≤ 31
1900 ≤ yyyy ≤ 2025
Its output would be the next date or an error message 'invalid date.' Design test cases using equivalence class partitioning method.

Solution

First we partition the domain of input in terms of valid input values and invalid values, getting the following classes:
I1 = {<m, d, y> : 1 ≤ m ≤ 12}
I2 = {<m, d, y> : 1 ≤ d ≤ 31}
I3 = {<m, d, y> : 1900 ≤ y ≤ 2025}
I4 = {<m, d, y> : m < 1}
I5 = {<m, d, y> : m > 12}
I6 = {<m, d, y> : d < 1}
I7 = {<m, d, y> : d > 31}
I8 = {<m, d, y> : y < 1900}
I9 = {<m, d, y> : y > 2025}

- The test cases can be designed from the above derived classes, taking one test case from each class such that the test case covers maximum valid input classes, and separate test cases for each invalid class.
- The test cases are shown below:

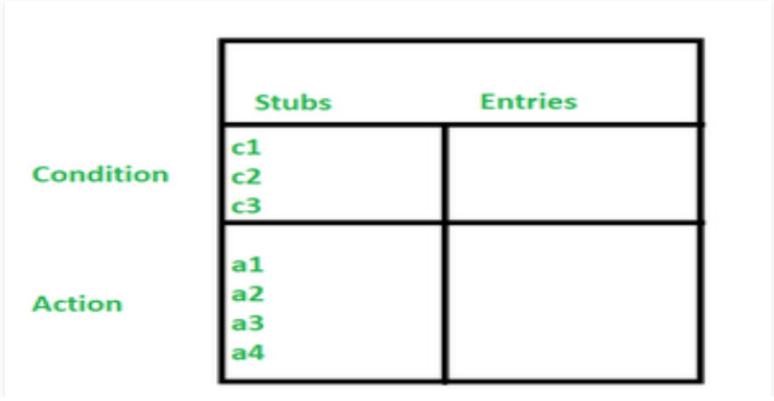
Test case ID	mm	dd	yyyy	Expected result	Classes covered by the test case
1	5	20	1996	21-5-1996	I_1, I_2, I_3
2	0	13	2000	Invalid input	I_4
3	13	13	1950	Invalid input	I_5
4	12	0	2007	Invalid input	I_6
5	6	32	1956	Invalid input	I_7
6	11	15	1899	Invalid input	I_8
7	10	19	2026	Invalid input	I_9

Que 5 : DECISION TABLE-BASED TESTING

- This testing is a very effective tool in testing the software and its requirements management.
- The output may be dependent on many input conditions and decision tables give a tabular view of various combinations of input conditions and these conditions are in the form of True(T) and False(F).
- Also, it provides a set of conditions and its corresponding actions required in the testing.

Parts of Decision Tables :

The decision table has 4 parts which are divided into portions and are given below :



Condition stub The conditions are listed in this first upper left part of the decision table that is used to determine a particular action or set of actions.

Action Stubs : All the possible actions are given in the first lower left portion (i.e, below condition stub) of the decision table.
Condition Entries : In the condition entry, the values are inputted in the upper right portion of the decision table. In the condition entries part of the table, there are multiple rows and columns which are known as **Rule.**
Action Entries : In the action entry, every entry has some associated action or set of actions in the lower right portion of the decision table and these values are called outputs.

Types of Decision Tables :

The decision tables are categorized into two types and these are given below:

1. **Limited Entry :** In the limited entry decision tables, the condition entries are restricted to binary values.
2. **Extended Entry :** In the extended entry decision table, the condition entries have more than two values. The decision tables use multiple conditions where a condition may have many possibilities instead of only 'true' and 'false' are known as extended entry decision tables.

Advantages of Decision Table Testing

- When the system behaviour is different for different input and not same for a range of inputs, both equivalent partitioning, and boundary value analysis won't help, but decision table can be used.
- The representation is simple so that it can be easily interpreted and is used for development and business as well.
- This table will help to make effective combinations and can ensure a better coverage for testing
- Any complex business conditions can be easily turned into decision tables
- In a case we are going for 100% coverage typically when the input combinations are low, this technique can ensure the coverage.

Disadvantages of Decision Table Testing

The main disadvantage is that when the number of input increases the table will become more complex

EXAMPLE

A university is admitting students in a professional course subject to the following

conditions:
(a) Marks in Java ≥ 70
(b) Marks in C++ ≥ 60
(c)Marks in OOAD ≥ 60
(d) Total in all three subjects ≥ 220 OR Total in Java and C++ ≥ 150
If the aggregate mark of an eligible candidate is more than 240, he will be eligible for scholarship course, otherwise he will be eligible for normal course. The program reads the marks in the three subjects and generates the following
outputs:
(i) Not eligible
(ii) Eligible for scholarship course
(iii) Eligible for normal course
Design test cases for this program using decision table testing.

Solution

	ENTRY									
	R1	**R2**	**R3**	**R4**	**R5**	**R6**	**R7**	**R8**	**R9**	**R10**
C1: marks in Java ≥ 70	T	T	T	T	F	I	I	I	T	T
C2: marks in C++ ≥ 60	T	T	T	T	I	F	I	I	T	T
C3: marks in OOAD ≥ 60	T	T	T	T	I	I	F	I	T	T
C4:Total in three subjects ≥ 220	T	F	T	T	I	I	I	F	T	T
C5:Total in Java & C++ ≥ 150	F	T	F	T	I	I	I	F	T	T
C6:Aggregate marks > 240	F	F	T	T	I	I	I	I	F	T
A1: Eligible for normal course	X	X							X	
A2: Eligible for scholarship course			X	X						X
A3: Not eligible					X	X	X	X		

Test Case ID	Java	C++	OOAD	Aggregate Marks	Expected Output
1	70	75	60	224	Eligible for normal course
2	75	75	70	220	Eligible for normal course
3	75	74	91	242	Eligible for scholarship course
4	76	77	89	242	Eligible for scholarship course
5	68	78	80	226	Not eligible
6	78	45	78	201	Not eligible
7	80	80	50	210	Not eligible
8	70	72	70	212	Not eligible
9	75	75	70	220	Eligible for normal course
10	76	80	85	241	Eligible for scholarship course

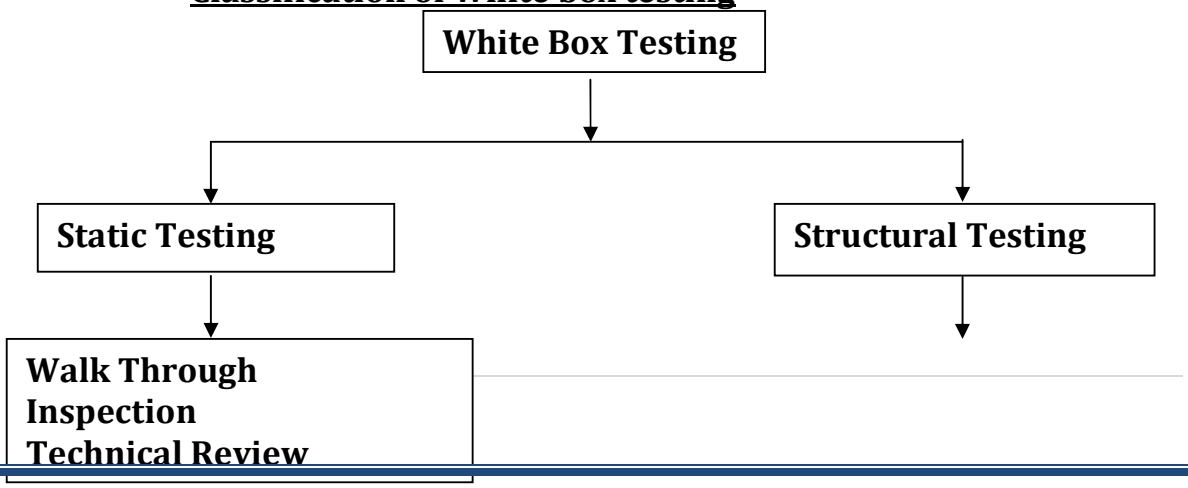
Que 6 : What is white box Testing?

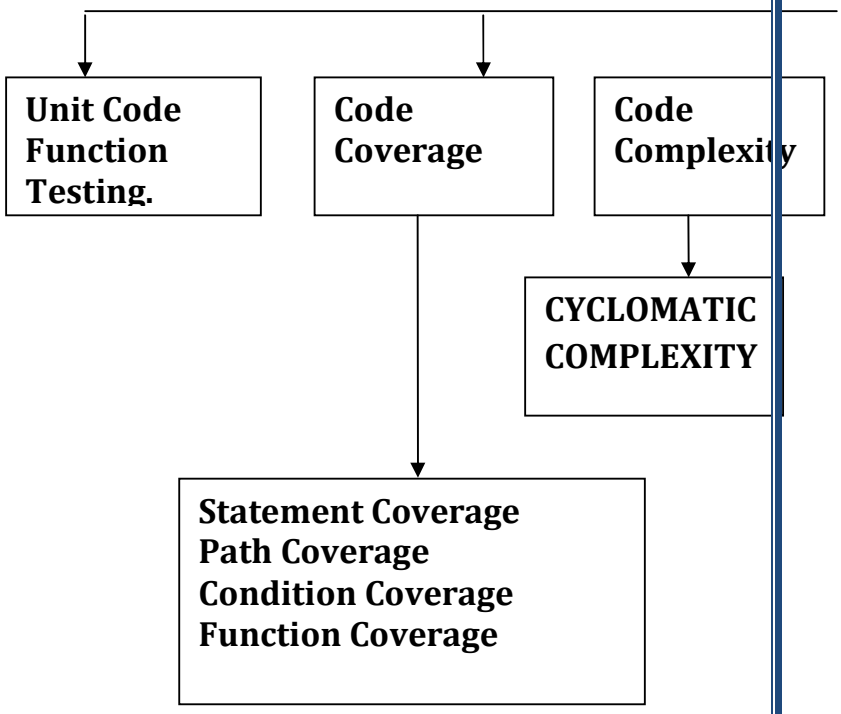
- **White Box Testing** is software testing technique in which internal structure, design and coding of software are tested to verify flow of input-output and to improve design, usability and security.
- In white box testing, code is visible to testers so it is also called Clear box testing, Open box testing, Transparent box testing, Code-based testing and Glass box testing.

Que 7 : What are NEED OF WHITE-BOX TESTING.

- The supporting reasons for white-box testing are given below:
  - In fact, white-box testing techniques are used for testing the module for initial stage testing.
  - Black-box testing is the second stage for testing the software.
  - Though test cases for black box can be designed earlier than white-box test cases, they cannot be executed until the code is produced and checked using white-box testing techniques. Thus, white-box testing is not an alternative but an essential stage.
  - Since white-box testing is opposite to black-box testing, there are categories of bugs which can be open by white-box testing, but not through black-box testing.
  - There may be portions in the code which are not checked when executing functional test cases, but these will be executed and tested by white-box testing.
  - Errors which have come from the design phase will also be reflected in the code, therefore we must execute white-box test cases for verification of code (unit verification).
  - We often believe that a logical path is not likely to be executed when, infect, it may be executed on a regular basis.
  - White-box testing explores these paths too.
  - Some typographical errors are not observed and go undetected and are not covered by black-box testing techniques.
  - White-box testing techniqueshelp detect these errors.

Classification of White box testing





Que 8: Explain logical coverage criteria with its various forms

- Structural testing considers the program code, and test cases are designed based on the logic of the program such that every element of the logic is covered.
- Therefore the intention in white-box testing is to cover the whole logic.
- Discussed below are the basic forms of logic coverage.

Statement Coverage

- The first kind of logic coverage can be identified in the form of statements.
- It is assumed that if all the statements of the module are executed once, every bug will be notified.

```
scanf ("%d", &x);
scanf ("%d", &y);
while (x != y)
{
      if (x > y)
            x = x - y;
      else
            y = y - x;
 }
 printf ("x = ", x);
 printf ("y = ", y);
```

Figure 5.1 Sample code

- If we want to cover every statement in the above code, then the following test cases must be designed:

- Test case 1: $x = y = n$, where n is any number
- Test case 2: $x = n, y = n'$, where n and n' are different numbers.

- Test case 1 just skips the while loop and all loop statements are not executed.
- Considering test case 2, the loop is also executed.
- However, every statement inside the loop is not executed.
- So two more cases are designed:
  - Test case 3: $x > y$
  - Test case 4: $x < y$
- These test cases will cover every statement in the code segment, however statement coverage is a poor criteria for logic coverage.
- We can see that test case 3 and 4 are sufficient to execute all the statements in the code.
- But, if we execute only test case 3 and 4, then conditions and paths in test case 1 will never be tested and errors will go undetected.
- Thus, statement coverage is a necessary but not sufficient criteria for logic coverage.

Decision or Branch Coverage

- Branch coverage states that each decision takes on all possible outcomes (True or False) at least once. In other words, each branch direction must be traversed at least once.
- In the previous sample code shown in Figure *while* and *if*

Statements have two outcomes: True and False. So test cases must be designed such that both outcomes for *while* and *if* statements are tested.
The test cases are designed as:
Test case 1: $x = y$
Test case 2: $x != y$
Test case 3: $x < y$
Test case 4: $x > y$

Condition Coverage

- Condition coverage states that each condition in a decision takes on all possible outcomes at least once.
  For example, consider the following statement:
  while ((I ≤5) && (J <COUNT))
- In this loop statement, two conditions are there. So test cases should be designed such that both the conditions are tested for True and False outcomes.

The following test cases are designed:
Test case 1: *I* ≤5, *J* <COUNT
Test case 2: *I* <5, *J* >COUNT

Decision/condition Coverage

- Condition coverage in a decision does not mean that the decision has been covered.
- If the decision *if (A && B)*is being tested, the condition coverage would allow one to write two test cases:

Test case 1: *A* is True, *B* is False.
Test case 2: *A* is False, *B* is True.

- But these test cases would not cause the THEN clause of the IF to execute(i.e. execution of decision).
- The obvious way out of this dilemma is a criterion called decision/condition coverage.
- It requires sufficient test cases such that each condition in a decision takes on all possible outcomes at least once, each decision takes on all possible outcomes at least once, and each point of entry is invoked at least once .

Multiple condition coverage

- In case of multiple conditions, even decision/condition coverage fails to exercise all outcomes of all conditions.
- The reason is that we have considered all possible outcomes of each condition in the decision, but we have not taken all combinations of different multiple conditions.
- Certain conditions cover other conditions. For example, if an AND condition is False, none of the subsequent conditions in the expression will be evaluated.
- Similarly, if an OR condition is True, none of the subsequent conditions will be evaluated.
- Thus, condition coverage and decision/condition coverage need not necessarily uncover all the errors.
- Therefore, multiple condition coverage requires that we should write sufficient test cases such that all possible combinations of condition outcomes.

- in each decision and all points of entry are invoked at least once. Thus, as indecision/condition coverage, all possible combinations of multiple conditions should be considered.
  The following test cases can be there:

Test case 1: *A* = True, *B* = True
Test case 2: *A* = True, *B* = False
Test case 3: *A* = False, *B* = True
Test case 4: *A* = False, *B* = False

Que 9 : Explain PATH TESTING

- **Basis Path Testing** in software engineering is a White Box Testing method in which test cases are defined based on flows or logical paths that can be taken through the program.
- The objective of basis path testing is to define the number of independent paths, so the number of test cases needed can be defined explicitly to maximize test coverage.
- Basis path testing involves execution of all possible blocks in a program and achieves maximum path coverage with the least number of test cases. It is a hybrid method of branch testing and path testing methods.

The guidelines for effectiveness of path testing are discussed below:
1. Path testing is based on control structure of the program for which flow graph is prepared.
2. Path testing requires complete knowledge of the program's structure.
3. Path testing is closer to the developer and used by him to test his module.
4. The effectiveness of path testing gets reduced with the increase in size of software under test [9].
5. Choose enough paths in a program such that maximum logic coverage is achieved.

Guidelines for Basis Path Testing

Draw the fl ow graph using the code provided for which we have to write test cases.

- Determine the cyclomatic complexity of the fl ow graph.
- Cyclomatic complexity provides the number of independent paths.

Determine a basis set of independent paths through the program control structure.

- The basis set is in fact the base for designing the test cases. Based on every independent path, choose the data such that this path is executed.

Que 10 : CONTROL FLOW GRAPH

- The control flow graph is a graphical representation of control structure of a program.
- Flow graphs can be prepared as a directed graph.
- A directed graph (V, E) consists of a set of vertices V and a set of edges.

following notations are used for a flow graph:

Node

- It represents one or more procedural statements.
- The nodes are denoted by a circle.
- These are numbered or labeled.

Edges or links

- They represent the flow of control in a program.
- This is denoted by an arrow on the edge.
- An edge must terminate at a node.

Decision node

- A node with more than one arrow leaving it is called a decision node.

Junction node

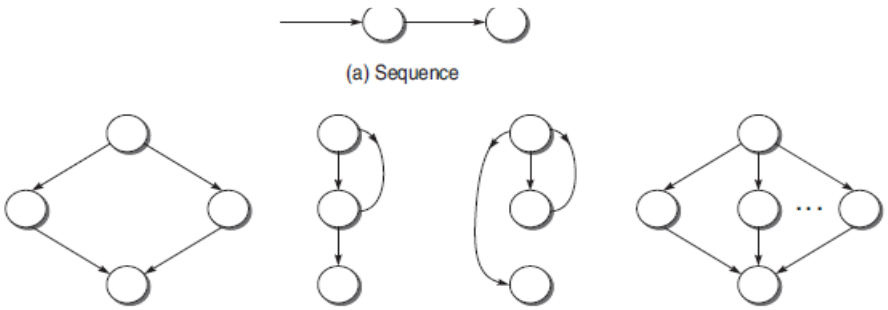
- A node with more than one arrow entering it is called a junction.

Regions

- Areas bounded by edges and nodes are called regions.
- When counting the regions, the area outside the graph is also considered a region.

Que 11 : FLOW GRAPH NOTATIONS FOR DIFFERENT PROGRAMMING CONSTRUCTS

- Since a flow graph is prepared on the basis of control structure of a program.
- some fundamental graphical notations are shown in figure.



- Using the **CONTROL FLOW GRAPH** notations, a flow graph can be created.

- Sequential statements having no conditions or loops can be merged in a single node.
- That is why, the flow graph is also known as **decision-to-decision-graph or DD graph**

Que 12: PATH TESTING TERMINOLOGY.

Path

- A path through a program is a sequence of instructions or statements that starts at an entry, junction, or decision and ends at another, or possibly the same, junction, decision, or exit.
- A path may go through several junctions, processes, or decisions, one or more times.

Segment

- Paths consist of segments.

- The smallest segment is a link, that is, a single process that lies between two nodes .
- A direct connection between two nodes, as in an unconditional GOTO, is also called a process by principle, even though no actual processing takes place.

Path segment

- A path segment is a series of repeated links that belongs to some path.

Length of a path

- The length of a path is measured by the number of links in it and not by the number of instructions or statements executed along the path.

Independent path

- An independent path is one that represents a path in the program that traces a new set of procedural statements or conditions.
- If we take it in the situation of a flow graph, the independent path traces the edges in the flow graph that are not traversed before the path is defined

Que 13: Explain CYCLOMATIC COMPLEXITY

- Cyclomatic complexity is a source code complexity measurement that is being correlated to a number of coding errors.
- It is calculated by developing a Control Flow Graph of the code that measures the number of linearly-independent paths through a program module.
- It can be represented using the below formula:

Cyclomatic complexity = E - N + 2*P

where,
E = number of edges in the flow graph.
N = number of nodes in the flow graph.
P = number of nodes that have exit points

Example

```
main()
{
int number, index;
1. printf("Enter a number");
2. scanf("%d, &number);
3. index = 2;
4. while(index <= number – 1)
5. {
6. if (number % index == 0)
7. {
8. printf("Not a prime number");
9. break;
10. }
11. index++;
12. }
13. if(index == number)
14. printf("Prime number");
15. }
```

Solution :

1. Draw the Control Flow Graph –

Step-1:

- Start numbering the statements after declaration of the variables
- For the given program, this is how numbering will be done

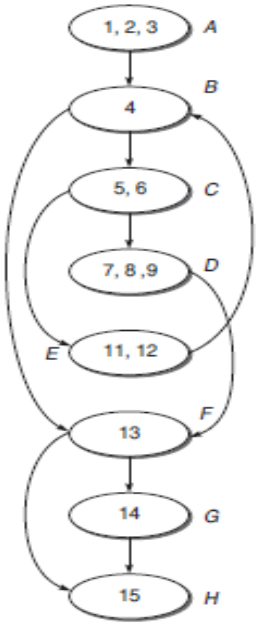


Figure 5.4 DD graph for Example 5.1

Step-2:

Put the sequential statements into one single node. For example, statements 1, 2 and 3 are all sequential statements and hence should be combined into a single node.

Note –

Use alphabetical numbering on nodes for simplicity.

The graph obtained will be as follows :

(b) Cyclomatic complexity

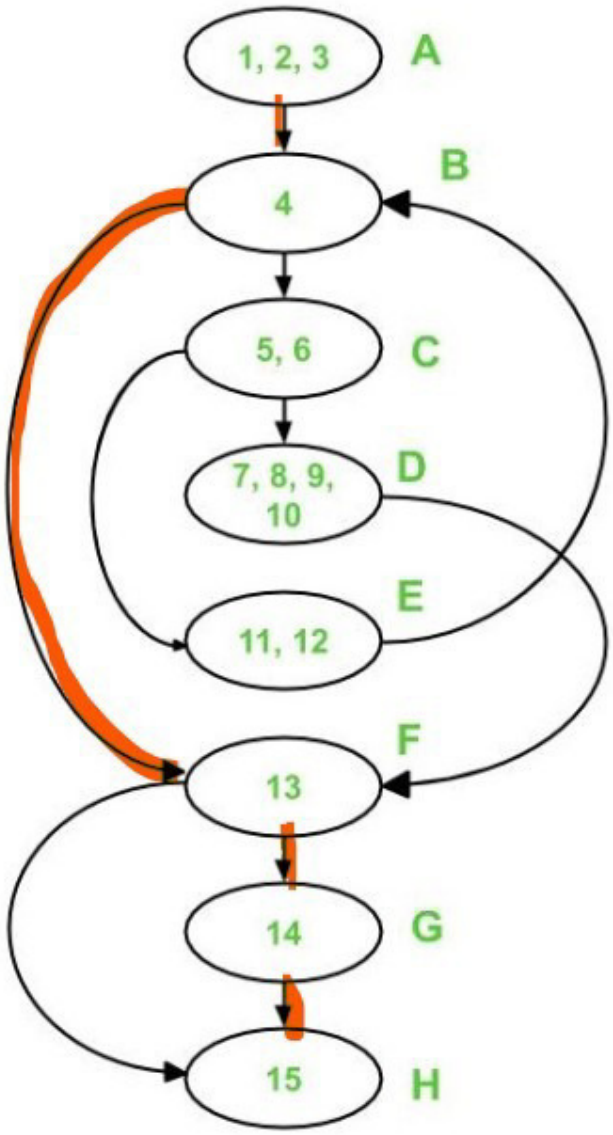
V(G) = e - n + 2*p

= 10 – 8 + 2

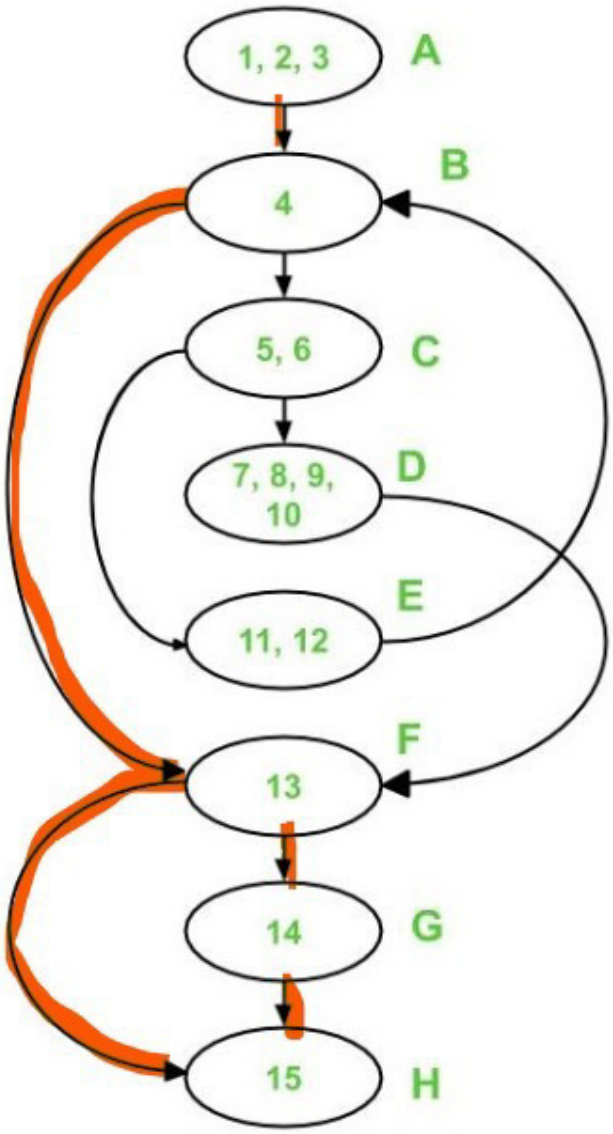
= 4

Independent Paths :

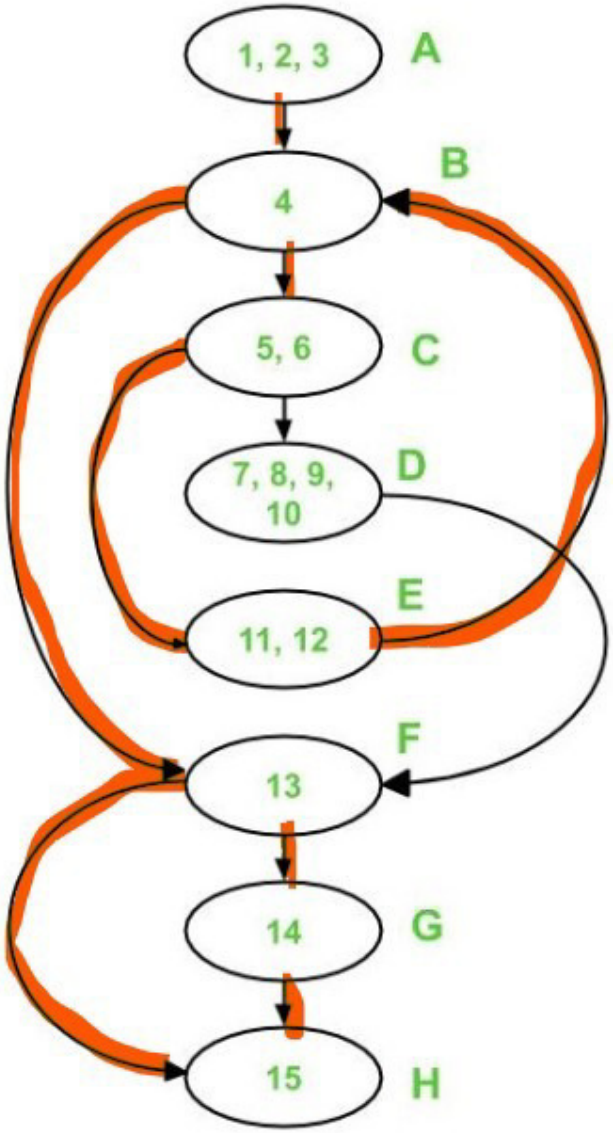
Path 1 : A - B - F - G - H



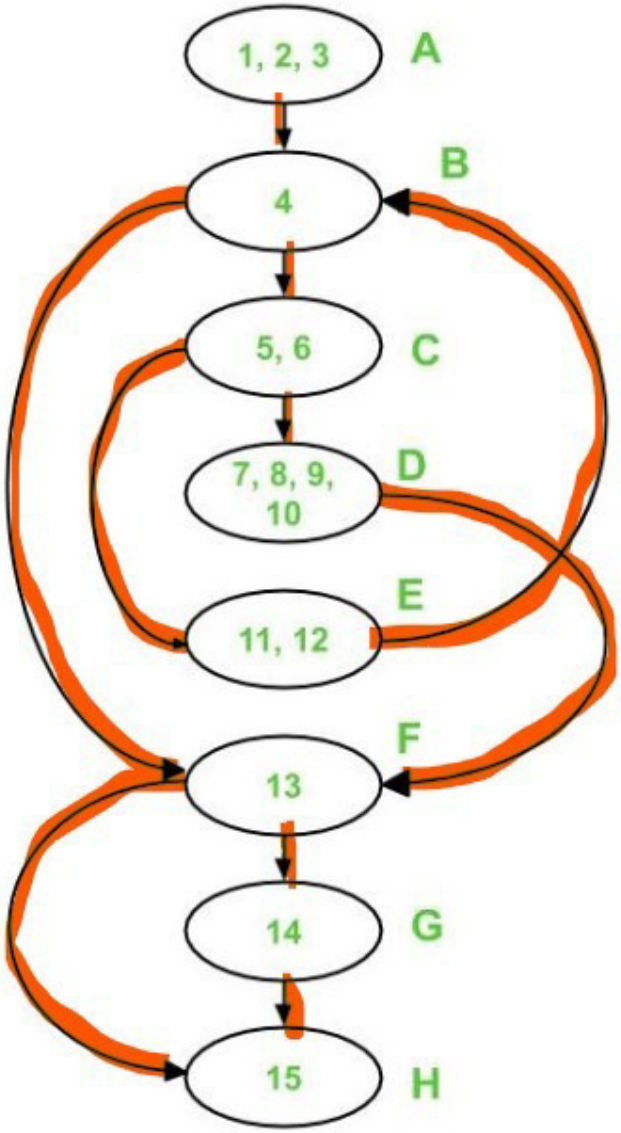
Path 2 : A - B - F - H



Path 3 : A - B - C - E - B - F - G - H



Path 4 : A - B - C - D - F - H



4. Test cases :

- To derive test cases, we have to use the independent paths obtained previously.
- To design a test case, provide input to the program such that each independent path is executed.

For the given program, the following test cases will be obtained:

Test case ID	Input Number	Output	Independent Path covered
1	1	No output	A-B-F-H
2	2	It is a prime number	A-B-F-G-H
3	3	It is a prime number	A-B-C-E-B-F-G-H
4	4	It is not a prime number	A-B-C-D-F-H

Que 13:Explain Inspection.

For the inspection process, a minimum of the following four team members are required.

Author/Owner/Producer

- A programmer or designer responsible for producing the program or document.
- He is also responsible for fixing defects discovered during the inspection process.

Inspector

- A peer member of the team, i.e. he is not a manager or supervisor.
- He is not directly related to the product under inspection and may be concerned with some other product.
- He finds errors, omission, and inconsistencies in programs and documents.

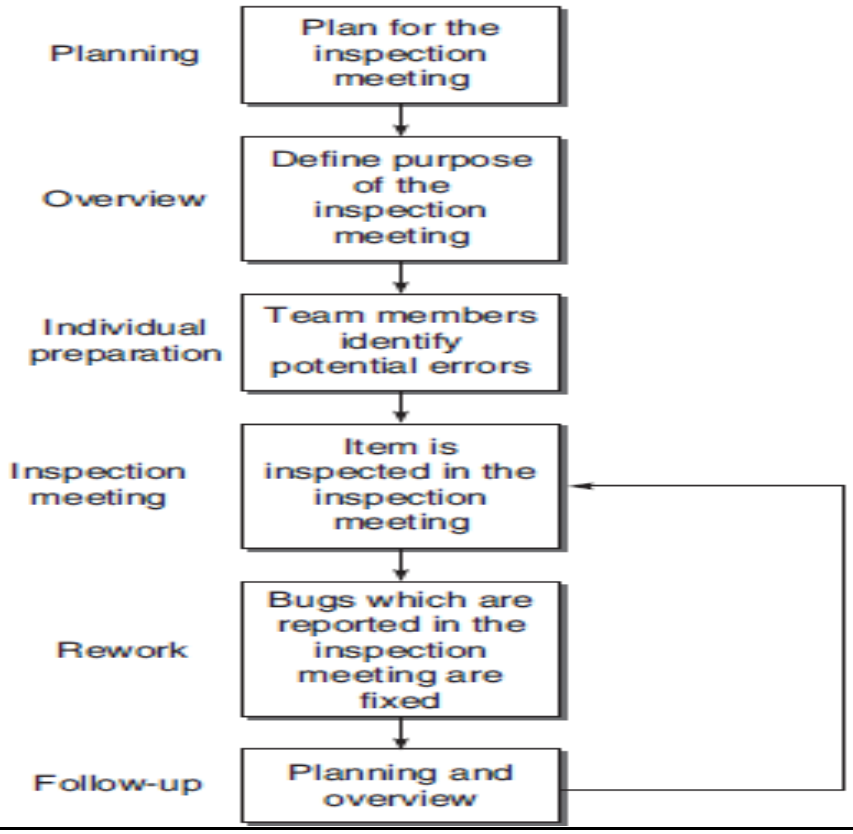
Moderator

- A team member who manages the whole inspection process.
- He schedules, leads, and controls the inspection session.
- He is the key person with the responsibility of planning and successful execution of the inspection.

Recorder

- One who records all the results of the inspection meeting.

INSPECTION PROCESS



Planning :

- Inspection is planned by moderator.

Overview meeting :

- Author describes background of work product.

Preparation :

- Each inspector examines work product to identify possible defects.

Inspection meeting :

- During this meeting, reader reads through work product, part by part and inspectors points out the defects for every part.

Rework :

- Author makes changes to work product according to action plans from the inspection meeting.

Follow-up :

- Changes made by author are checked to make sure that everything is correct.

BENEFITS OF INSPECTION PROCESS

- Bug reduction
- Bug Prevention
- Productivity
- Real-time feedback to software engineers
- Reduction in Development Resources
- Quality improvement
- Project management
- Checking Coupling and cohesion
- Learning through inspection
- Process Improvement

Que 14. Explain STRUCTURED WALKTHROUGHS in detail

- It is a less formal and less rigorous technique as compared to inspection.
- The common term used for static testing is inspection but it is a very formal process.
- If you want to go for a less formal process having no bars of organized meeting, then
- walkthroughs are a good option.
- A typical structured walkthrough team consists of the following members:
  - **Coordinator** Organizes, moderates, and follows up the walkthrough activities.
  - **Presenter/Developer** Introduces the item to be inspected. This member is optional.
  - **Scribe/Recorder** Notes down the defects found and suggestion proposed by the members.
  - **Reviewer/Tester** Finds the defects in the item. Maintenance Oracle Focuses on long-term implications and future maintenance of the project.
  - **Standards Bearer** Assesses adherence to standards.
  - **User Representative/Accreditation Agent** Reflects the needs and concerns of the user.