UNIT-2

Structural Testing

CONTROL FLOW TESTING

This technique is very popular due to its simplicity and effectiveness. We identify paths of the program and write test cases to execute those paths. As we all know, path is a sequence of statements that begins at an entry and ends at an exit. As shown in chapter 1, there may be too many paths in a program and it may not be feasible to execute all of them. As the number of decisions increase in the program, the number of paths also increase accordingly.

Every path covers a portion of the program. We define 'coverage' as a 'percentage of sourcecode that has been tested with respect to the total source code available for testing'. We may like to achieve a reasonable level of coverage using control flow testing. The most reasonable level may be to test every statement of a program at least once before the completion of testing. Hence, we may write test cases that ensure the execution of every statement. If we do so, we have some satisfaction about reasonable level of coverage. If we stop testing without achievingthis level (every statement execution), we do unacceptable and intolerable activity which maylead to dangerous results in future. Testing techniques based on program coverage criterion may provide an insight about the effectiveness of test cases. Some of such techniques are discussed which are part of control flow testing.

- ## Statement Coverage

We want to execute every statement of the program in order to achieve 100% statement coverage. Consider the following portion of a source code along with its program graph given in Figure 4.1.

```
        #include<stdio.h>
        #include<conio.h>

1.      void main()
2.     {
3.      int a,b,c,x=0,y=0;
4.      clrscr();
5.      printf("Enter three numbers:");
6.      scanf("%d %d %d",&a,&b,&c);
7.      if((a>b)&&(a>c)){
8.                  x=a*a+b*b;
9.     }
10.     if(b>c){
11.                 y=a*a-b*b;
12.    }
13.     printf("x= %d y= %d",x,y);
14.     getch();
15.    }
```

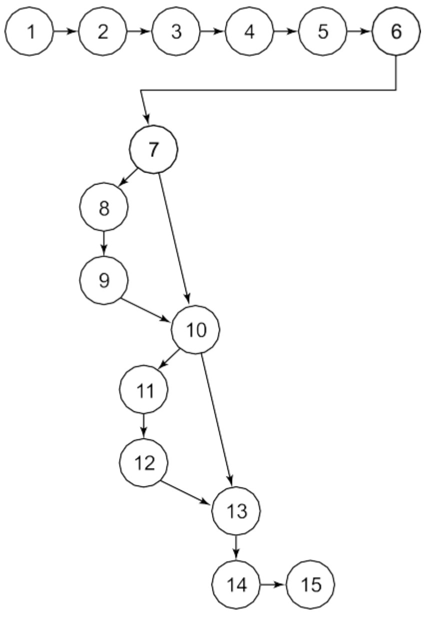


Figure 4.1. Source code with program graph

If, we select inputs like:

a=9, b=8, c=7, all statements are executed and we have achieved 100% statement coverage by only one test case. The total paths of this program graph are given as:

(i) 1–7, 10, 13–15
(ii) 1–7, 10–15
(iii) 1–10, 13–15
(iv) 1–15

The cyclomatic complexity of this graph is:

$$V(G) = e - n + 2P = 16 - 15 + 2 = 3$$
$$V(G) = \text{no. of regions} = 3$$
$$V(G) = \Pi + 1 = 2 + 1 = 3$$

Hence, independent paths are three and are given as:

(i) 1–7, 10, 13–15
(ii) 1–7, 10–15
(iii) 1–10, 13–15

Only one test case may cover all statements but will not execute all possible four paths and not even cover all independent paths (three in this case).

The objective of achieving 100% statement coverage is difficult in practice. A portion of the program may execute in exceptional circumstances and some conditions are rarely possible, and the affected portion of the program due to such conditions may not execute at all.

- **Branch Coverage**

We want to test every branch of the program. Hence, we wish to test every ‘True’ and ‘False’ condition of the program. We consider the program given in Figure 4.1. If we select a = 9, b = 8, c = 7, we achieve 100% statement coverage and the path followed is given as (all true conditions):

Path = 1–15

We also want to select all false conditions with the following inputs:

a = 7, b = 8, c = 9, the path followed is

Path = 1–7, 10, 13–15

These two test cases out of four are sufficient to guarantee 100% branch coverage. The branch coverage does not guarantee 100% path coverage but it does guarantee 100% statement coverage.

- **Condition Coverage**

Condition coverage is better than branch coverage because we want to test every condition at least once. However, branch coverage can be achieved without testing every condition.

Consider the seventh statement of the program given in Figure 4.1. The statement number 7 has two conditions (a>b) and (a>c). There are four possibilities namely:

(i) Both are true
(ii) First is true, second is false
(iii) First is false, second is true
(iv) Both are false

If a > b and a > c, then the statement number 7 will be true (first possibility). However, if a < b, then second condition (a > c) would not be tested and statement number 7 will be false (third and fourth possibilities). If a > b and a < c, statement number 7 will be false (second possibility). Hence, we should write test cases for every true and false condition. Selected inputs may be given as:

(i) a = 9, b = 8, c = 7 (first possibility when both are true)
(ii) a = 9, b = 8, c = 10 (second possibility – first is true, second is false)
(iii) a = 7, b = 8, c = 9 (third and fourth possibilities- first is false, statement number 7 is

false)

Hence, these three test cases out of four are sufficient to ensure the execution of every condition of the program.

- **Path Coverage**

In this coverage criteria, we want to test every path of the program. There are too many paths in any program due to loops and feedback connections. It may not be possible to achieve this goal of executing all paths in many programs. If we do so, we may be confident about the correctness of the program. If it is unachievable, at least all independent paths should be executed. The program given in Figure 4.1 has four paths as given as:

(i) 1–7, 10, 13–15
(ii) 1–7, 10–15
(iii) 1–10, 13–15
(iv) 1–15

Execution of all these paths increases confidence about the correctness of the program. Inputs for test cases are given as:

S. No.	Paths Id.	Paths	Inputs			Expected Output
			a	b	c	
1.	Path-1	1-7,10, 13-15	7	8	9	x=0 y=0
2.	Path-2	1-7, 10-15	7	8	6	x=0 y=-15
3.	Path-3	1-10, 13-15	9	7	8	x=130 y=0
4.	Path-4	1-15	9	8	7	x=145 y=17

Some paths are possible from the program graph, but become impossible when we give inputs as per logic of the program. Hence, some combinations may be found to be impossible to create.

Path testing guarantee statement coverage, branch coverage and condition coverage. However, there are many paths in any program and it may not be possible to execute all the paths. We should do enough testing to achieve a reasonable level of coverage. We should execute at least (minimum level) all independent paths which are also referred to as basis paths to achieve reasonable coverage. These paths can be found using any method of cyclomatic complexity.

We have to decide our own coverage level before starting control flow testing. As we go up (statement coverage to path coverage) in the ladder, more resources and time may be required.

Example 4.1: Consider the program for the determination of the division of a student. The program and its program graph are given in Figure 3.15 and 3.16 of chapter 3 respectively. Derive test cases so that 100% path coverage is achieved.

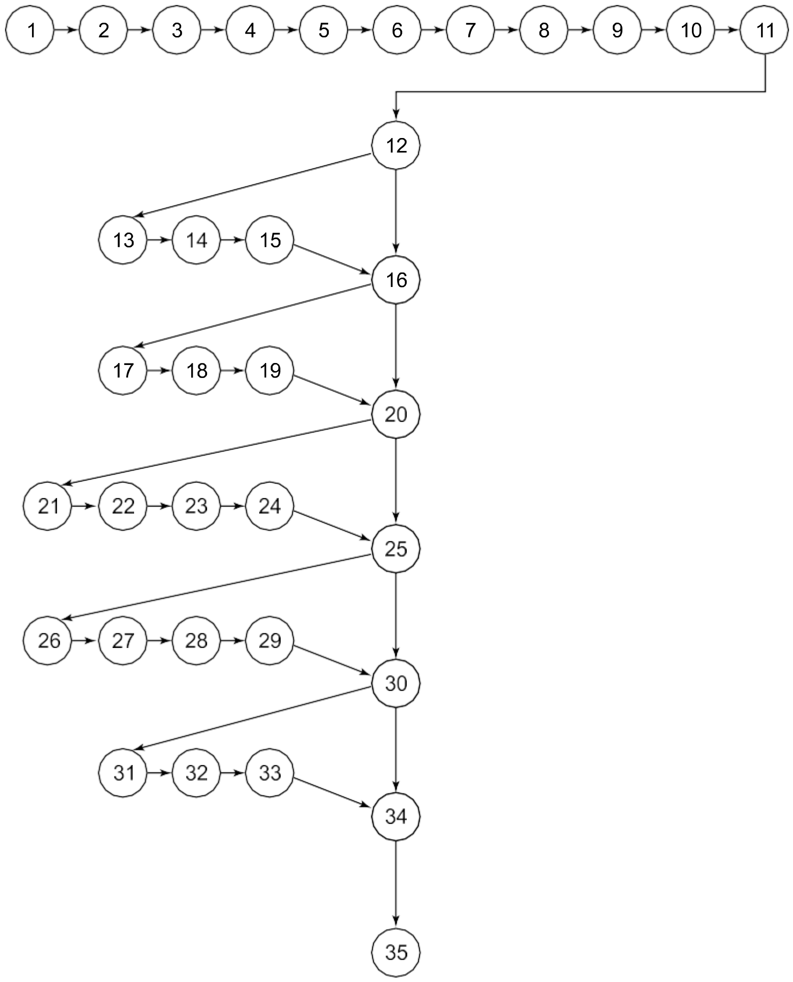
Solution:
The test cases are given in Table 4.1.

Table 4.1. Test cases

S. No.	mark1	mark2	mark3	Expected output	Paths
1.	30	-1	20	Invalid marks	1-14, 33, 34
2.	40	20	45	Fail	1-12, 15-19, 32, 33,34
3.	45	47	50	Third division	1-13, 15-17, 20-22, 32-34
4.	55	60	57	Second division	1-12, 15-17, 20, 23, 26-28, 32-34
5.	65	70	75	First division	1-12, 15-17, 20, 23, 26-28,32-34
6.	80	85	90	First division with distinction	1-12, 15-17, 20, 23, 26, 29-34

Example 4.2: Consider the program and program graph given below. Derive test cases so that 100% statement coverage and path coverage is achieved.

```
        /*Program to validate input data*/
        #include<stdio.h>
        #include<string.h>
        #include<conio.h>
1.      void main()
2.      {
3.      char fname[30],address[100],Email[100];
4.      int valid=1,flag=1;
5.      clrscr();
6.      printf("Enter first name:");
7.      scanf("%s",fname);
8.      printf("\nEnter address:");
9.      scanf("%s",address);
10.     printf("\nEnter  Email:");
11.     scanf("%s",Email);
12.     if(strlen(fname)<4||strlen(fname)>30){
13.             printf("\nInvalid first name");
14.             valid=0;
15.     }
16.     if(strlen(address)<4||strlen(address)>100){
17.             printf("\nInvalid address length");
18.             valid=0;
19.     }
20.     if(strlen(Email)<8||strlen(Email)>100){
21.             printf("\nInvalid Email length");
22.             flag=0;
23.             valid=0;
24.     }
25.     if(flag==1){
26.             if(strchr(Email,'.')==0||strchr(Email,'@')==0){
27.                     printf("\nEmail must contain . and @ characters");
28.                     valid=0;
29.             }
30.     }
31.     if(valid) {
32.     printf("\nFirst name: %s \t Address: %s \t Email:
        %s",fname,address,Email);
33.     }
34.     getch();
35.     }
```



Solution:
The test cases to guarantee 100% statement and branch coverage are given in Table 4.2.

Table 4.2. Test cases for statement coverage

S. No.	First name	Address	Email	Expected output	Paths
1.	ashok	E-29, east-ofkailash	abc@yahoo.com	First name: ashok Address: E-29, east-ofkailash Email: abc@ yahoo.com	1-12, 16, 20, 25, 31-35
2.	ruc	E29	abc	Invalid first name Invalid address length Invalid email length	1-25, 30, 31, 34, 35
3.	ruc	E-29	abc@yahoocom	Invalid first name Invalid address length Email must contain . and @ character	1-20, 25-31, 34, 35

(*Contd.*)

Total paths of the program graph are given in Table 4.3.

Table 4.3. Test cases for path coverage

S. No.	First name	Address	Email	Expected output	Paths
1.	-	-	-	-	1–35
2.	-	-	-	-	1–30, 34,35
3.	-	-	-	-	1–25, 30–35
4.	ruc	E29	abc	Invalid first name Invalid address length Invalid email length	1-25, 30, 31, 34, 35
5.	-	-	-	-	1–20, 25–35
6.	ruc	E-29	abc@yahoocom	Invalid first name Invalid address length Email must contain . and @ character	1-20, 25-31, 34, 35
7.	-	-	-	-	1-20, 25, 30-35
8.	ruc	E-29	Abs@yahoo.com	Invalid first name Invalid address length	1-20, 25, 30, 31, 34, 35
9.	-	-	-	-	1–16, 20–35
10.	-	-	-	-	1-16, 20-31, 34, 35
11.	-	-	-	-	1-16, 20-25, 30–35
12.	ruc	E-29, east-ofkailash	Abs	Invalid first name Invalid email length	1-16, 20-25, 30, 31, 34, 35
13.	-	-	-	-	1-16, 20, 25-35
14.	ruc	E-29, east-ofkailash	abc@yahoocom	Invalid first name Email must contain . and @ character	1-16, 20, 25-31, 34, 35
15.	-	-	-	-	1-16, 20, 25, 31–35
16.	ruc	E-29, east-ofkailash	abc@yahoo.com	Invalid first name	1-16, 20, 25, 30, 31, 34, 35
17.	-	-	-	-	1–12, 16–35
18.	-	-	-	-	1-12, 16-31, 34,35
19.	-	-	-	-	1-12, 16-25, 30–35
20.	ashok	E29	Abc	Invalid address length Invalid email length	1-12, 16-25, 30, 31, 34, 35
21.	-	-	-	-	1-12, 16-20, 25–35

(*Contd.*)

(*Contd.*)

S. No.	First name	Address	Email	Expected output	Paths
22.	ashok	E29	abc@yahoocom	Invalid address length Email must contain . and @ character	1-12, 16-20, 25-31, 34, 35
23.	-	-	-	-	1-12, 16-20, 25, 30-35
24.	ashok	E29	abc@yahoo.com	Invalid address length	1-12, 16-20, 25, 30, 31, 34, 35
25.	-	-	-	-	1-12, 16, 20-35
26.	-	-	-	-	1-12, 16, 20-31, 34, 35
27.	-	-	-	-	1-12, 16, 20-25, 30-35
28.	ashok	E-29, east-ofkailash	Abs	Invalid email length	1-12, 16, 20-25, 30, 31, 34, 35
29.	-	-	-	-	1-12, 16, 20, 25-35
30.	ashok	E-29, east-ofkailash	Abcyahoo.com	Email must contain . and @ character	1-12, 16, 20, 25-31, 34, 35
31.	ashok	E-29, east-ofkailash	abc@yahoo.com	First name: ashok Address: E-29, east-ofkailash Email: abc@ yahoo.com	1-12, 16, 20, 25, 31-35
32.	-	-	-	-	1-12, 16, 20, 25, 30, 31, 34, 35

Example 4.3: Consider the program for classification of a triangle given in Figure 3.10. Derive test cases so that 100% statement coverage and path coverage is achieved.

Solution:
The test cases to guarantee 100% statement and branch coverage are given in Table 4.4.

Table 4.4. Test cases for statement coverage

S. No.	a	b	c	Expected output	Paths
1.	30	20	40	Obtuse angled triangle	1-16,20-27,34,41,42
2.	30	40	50	Right angled triangle	1-16,20-25,28-30,34,41,42
3.	40	50	60	Acute angled triangle	1-6,20-25,28,31-34,41,42
4.	30	10	15	Invalid triangle	1-14,17-21,35-37,41,42
5.	102	50	60	Input values out of range	1-13,21,35,38,39,40-42

Total paths of the program graph are given in Table 4.5.

Table 4.5. Test cases for path coverage

S. No.	a	b	c	Expected output	Paths
1.	102	−1	6	Input values out of range	1-13,21,35,38,39,40-42
2.	-	-	-	-	1-14,17-19,20,21,35,38,39,40-42
3.	-	-	-	-	1-16,20,21,35,38,39,40-42
4.	-	-	-	-	1-13,21,35,36,37,41,42
5.	30	10	15	Invalid triangle	1-14,17-21,35-37,41,42
6.	-	-	-	-	1-16,20,21,35-37,41,42
7.	-	-	-	-	1-13,21-25,28,31-34,41,42
8.	-	-	-	-	1-14,17-25,28,31-34,41,42
9.	40	50	60	Acute angled triangle	1-16,20-25,28,31-34,41,42
10.	-	-	-	-	1-13,21-25,28-30,34,41,42
11.	-	-	-	-	1-14,17-25,28-30,34,41,42
12.	30	40	50	Right angled triangle	1-16,20-25,28-30,34,41,42
13.	-	-	-	-	1-13,21-27,34,41,42
14.	-	-	-	-	1-14,17-27,34,41,42
15.	30	20	40	Obtuse angled triangle	1-16,20-27,34,41,42

Thus, there are 15 paths, out of which 10 paths are not possible to be executed as per the logic of the program.

DATA FLOW TESTING

In control flow testing, we find various paths of a program and design test cases to execute those paths. We may like to execute every statement of the program at least once before the completion of testing. Consider the following program:

```
1. # include < stdio.h>
2. void main ()
3. {
4. int a, b, c;
5. a = b + c;
6. printf (“%d”, a);
7. }
```

What will be the output? The value of ‘a’ may be the previous value stored in the memory location assigned to variable ‘a’ or a garbage value. If we execute the program, we may get an unexpected value (garbage value). The mistake is in the usage (reference) of this variable without first assigning a value to it. We may assume that all variables are automatically assigned to zero initially. This does not happen always. If we define at line number 4, ‘static int a, b, c’, then all variables are given zero value initially. However, this is a language and compiler dependent feature and may not be generalized.

Data flow testing may help us to minimize such mistakes. It has nothing to do with data-flow diagrams. It is based on variables, their usage and their definition(s) (assignment) in the program. The main points of concern are:

(i) Statements where variables receive values (definition).
(ii) Statements where these values are used (referenced).

Data flow testing focuses on variable definition and variable usage. In line number 5 of the above program, variable 'a' is defined and variables 'b' and 'c' are used. The variables are defined and used (referenced) throughout the program. Hence, this technique concentrates on how a variable is defined and used at different places of the program.

• Define/Reference Anomalies

Some of the define / reference anomalies are given as:

(i) A variable is defined but never used / referenced.
(ii) A variable is used but never defined.
(iii) A variable is defined twice before it is used.
(iv) A variable is used before even first-definition.

We may define a variable, use a variable and redefine a variable. So, a variable must be first defined before any type of its usage. Define / reference anomalies may be identified by static analysis of the program i.e. analyzing program without executing it. This technique uses the program graphs to understand the 'define / use' conditions of all variables. Some terms are used frequently in data flow testing and such terms are discussed in the next sub-section.

• Definitions

A program is first converted into a program graph. As we all know, every statement of a program is replaced by a node and flow of control by an edge to prepare a program graph. There may be many paths in the program graph.

(i) Defining node
A node of a program graph is a defining node for a variable υ if and only if, the value of the variable υ is defined in the statement corresponding to that node. It is represented as DEF (υ, n) where υ is the variable and n is the node corresponding to the statement in which υ is defined.

(ii) Usage node
A node of a program graph is a usage node for a variable υ, if and only if, the value of the variable υ is used in the statement corresponding to that node. It is represented as USE (υ, n), where 'υ' is the variable and 'n' in the node corresponding to the statement in which 'υ' is used.
A usage node USE (υ, n) is a predicate use node (denoted as P-use), if and only if, the statement corresponding to node 'n' is a predicate statement otherwise USE (υ, n) is a computation use node (denoted as C-use).

(iii) Definition use Path
A definition use path (denoted as du-path) for a variable 'υ' is a path between two nodes 'm' and 'n' where 'm' is the initial node in the path but the defining node for variable 'υ' (denoted as DEF (υ, m)) and 'n' is the final node in the path but usage node for variable 'υ' (denoted as USE (υ, n)).

(iv) Definition clear path
A definition clear path (denoted as dc-path) for a variable 'υ' is a definition use path with initial and final nodes DEF (υ, m) and USE (υ, n) such that no other node in the path is a defining node of variable 'υ'.

The du-paths and dc-paths describe the flow of data across program statements from statements where values are defined to statements where the values are used. A du-path for a variable 'υ' may have many redefinitions of variable 'υ' between initial node (DEF (υ, m)) and final node (USE (υ, n)). A dc-path for a variable 'υ' will not have any definition of variable 'υ' between initial node (DEF (υ, m)) and final node (USE (υ, n)). The du-paths that are not definition clear paths are potential troublesome paths. They should be identified and tested on topmost priority.

• Identification of du and dc Paths

The various steps for the identification of du and dc paths are given as:

(i) Draw the program graph of the program.

(ii) Find all variables of the program and prepare a table for define / use status of all variables using the following format:

S. No.	Variable(s)	Defined at node	Used at node

(iii) Generate all du-paths from define/use variable table of step (iii) using the following format:

S. No.	Variable	du-path(begin, end)

(iv) Identify those du-paths which are not dc-paths.

• Testing Strategies Using du-Paths

We want to generate test cases which trace every definition to each of its use and every use is traced to each of its definition. Some of the testing strategies are given as:

(i) Test all du-paths
All du-paths generated for all variables are tested. This is the strongest data flow testing strategy covering all possible du-paths.

(ii) Test all uses
Find at least one path from every definition of every variable to every use of that variable which can be reached by that definition.
For every use of a variable, there is a path from the definition of that variable to the use of that variable.

(iii) Test all definitions
Find paths from every definition of every variable to at least one use of that variable; we may choose any strategy for testing. As we go from 'test all du-paths' (no. (i)) to 'test all definitions' (no.(iii)), the number of paths are reduced. However, it is best to test all du-paths (no. (i)) and give priority to those du-paths which are not definition clear paths. The first requires that each definition reaches all possible uses through all possible du-paths, the second requires that each definition reaches all possible uses, and the third requires that each definition reaches at least one use.

• Generation of Test Cases

After finding paths, test cases are generated by giving values to the input parameter. We get different test suites for each variable.

Consider the program given in Figure 3.11 to find the largest number amongst three numbers. Its program graph is given in Figure 3.12. There are three variables in the program namely A, B and C. Define /use nodes for all these variables are given below:

S. No.	Variable	Defined at node	Used at node
1.	A	6	11, 12, 13
2.	B	8	11, 20, 24
3.	C	10	12, 16, 20, 21

The du-paths with beginning node and end node are given as:

Variable	du-path (Begin, end)
A	6, 11
	6, 12
	6, 13
B	8, 11
	8, 20
	8, 24
C	10, 12
	10, 16
	10, 20
	10, 21

The first strategy (best) is to test all du-paths, the second is to test all uses and the third is to test all definitions. The du-paths as per these three strategies are given as:

	Paths	Definition clear?	
All du paths and all uses (Both are same in this example)	6–11	Yes	
	6–12	Yes	
	6–13	Yes	
	8–11	Yes	
	8-11, 19, 20	Yes	
	8-11, 19, 20, 23, 24	Yes	
	10–12	Yes	
	10-12, 15, 16	Yes	
	10, 11, 19, 20	Yes	
	10, 11, 19-21	Yes	
All definitions	6–11	Yes	
	8–11	Yes	
	10–12	Yes	

Here all du-paths and all-uses paths are the same (10 du-paths). But in the 3rd case, for all definitions, there are three paths.
Test cases are given below:

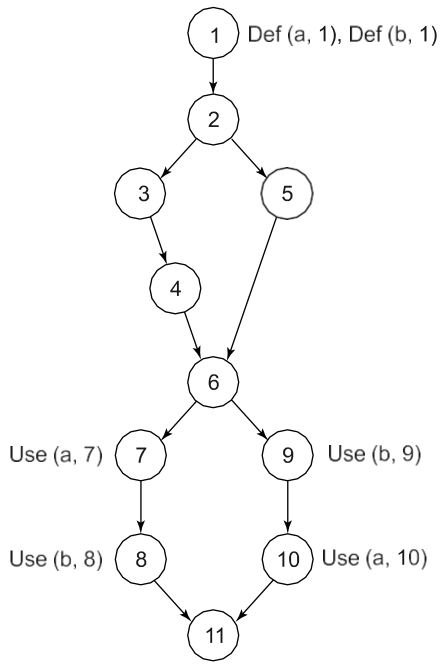
Test all du-paths

S. No.	Inputs			Expected Output	Remarks
	A	B	C		
1.	9	8	7	9	6–11
2.	9	8	7	9	6–12
3.	9	8	7	9	6–13
4.	7	9	8	9	8–11
5.	7	9	8	9	8–11, 19, 20
6.	7	9	8	9	8–11, 19, 20, 23, 24
7.	8	7	9	9	10-12
8.	8	7	9	9	10-12, ,15, 16
9.	7	8	9	9	10, 11, 19, 20
10.	7	8	9	9	10, 11, 19-21

Test All definitions

S. No.	Inputs			Expected Output	Remarks
	A	B	C		
1.	9	8	7	9	6–11
2.	7	9	8	9	8–11
3.	8	7	9	9	10–12

In this example all du-paths and all uses yield the same number of paths. This may not always be true. If we consider the following graph and find du paths with all three strategies, we will get a different number of all-du paths and all-uses paths.



Def/Use nodes table

S. No.	Variables	Defined at node	Used at node
1.	a	1	7, 10
2.	b	1	8, 9

The du paths are identified as:

S. No.	Variables	du-paths (Begin, end)
1.	a	1, 7
		1, 10
2.	b	1, 8
		1, 9

The du-paths are identified as per three testing strategies:

	Paths	Definition clear?
All du paths	1-4, 6, 7	Yes
(8 paths)	1, 2, 5-7	Yes
	1-4, 6, 9, 10	Yes
	1, 2, 5, 6, 9, 10	Yes
	1-4, 6, 7, 8	Yes
	1, 2, 5-8	Yes
	1-4, 6, 9	Yes
	1, 2, 5, 6, 9	Yes

(*Contd.*)

(*Contd.*)

	Paths	Definition clear?
All uses	1-4, 6, 7	Yes
(4 paths)	1-4, 6, 9, 10	Yes
	1-4, 6-8	Yes
	1-4, 6, 9	Yes
All definitions	1-4, 6, 7	Yes
(2 paths)	1-4, 6-8	Yes

Hence the number of paths is different in all testing strategies. When we find all du-paths, some paths may become impossible paths. We show them in order to show all combinations.

Example 4.4: Consider the program for the determination of the division problem. Its input is a triple of positive integers (mark1, mark2, mark3) and values for each of these may be from interval [0, 100]. The program is given in Figure 3.15. The output may have one of the options given below:

(i) Fail
(ii) Third division
(iii) Second division
(iv) First division
(v) First division with distinction
(vi) Invalid marks

Find all du-paths and identify those du-paths that are definition clear. Also find all du-paths, all-uses and all-definitions and generate test cases for these paths.

Solution:

(i) The program graph is given in Figure 3.16. The variables used in the program are mark1, mark2, mark3, avg.
(ii) The define/ use nodes for all variables are given below:

S. No.	Variable	Defined at node	Used at node
1.	mark1	7	12, 16
2.	mark2	9	12, 16
3.	mark3	11	12, 16
4.	avg	16	17, 20, 23, 26

(iii) The du-paths with beginning and ending nodes are given as:

S. No.	Variable	Du-path (begin, end)
1.	mark1	7, 12
		7, 16
2.	mark2	9, 12
		9, 16
3.	mark3	11, 12
		11, 16

(*Contd.*)

(*Contd.*)

S. No.	Variable	Du-path (begin, end)
4.	Avg	16, 17
		16, 20
		16, 23
		16, 26

(iv) All du-paths, all-uses and all-definitions are given below:

	Paths	Definition clear?
All du-paths and all-uses	7–12	Yes
	7–12, 15, 16	Yes
	9–12	Yes
	9–12, 15, 16	Yes
	11, 12	Yes
	11, 12, 15, 16	Yes
	16, 17	Yes
	16, 17, 20	Yes
	16, 17, 20, 23	Yes
	16, 17, 20, 23, 26	Yes
All definitions	7–12	Yes
	9–12	Yes
	11, 12	Yes
	16, 17	Yes

Test cases for all du-paths and all-uses are given in Table 4.6 and test cases for all definitions are given in Table 4.7.

Table 4.6. Test cases for all du-paths and all-uses

S. No.	mark1	mark2	mark3	Expected Output	Remarks
1.	101	50	50	Invalid marks	7–12
2.	60	50	40	Second division	7–12, 15, 16
3.	50	101	50	Invalid marks	9–12
4.	60	70	80	First division	9–12, 15, 16
5.	50	50	101	Invalid marks	11, 12
6.	60	75	80	First division	11, 12, 15, 16
7.	30	40	30	Fail	16, 17
8.	45	50	50	Third division	16, 17, 20
9.	55	60	50	Second division	16, 17, 20, 23
10.	65	70	70	First division	16, 17, 20, 23, 26

Table 4.7.

S. No.	mark1	mark2	mark3	Expected Output	Remarks
1.	101	50	50	Invalid marks	7–12
2.	50	101	50	Invalid marks	9–12
3.	50	50	101	Invalid marks	11, 12
4.	30	40	30	Fail	16, 17

Example 4.5: Consider the program of classification of a triangle. Its input is a triple of positive integers (a, b and c) and values for each of these may be from interval [0, 100]. The program is given in Figure 3.18. The output may have one of the options given below:

(i) Obtuse angled triangle
(ii) Acute angled triangle
(iii) Right angled triangle
(iv) Invalid triangle
(v) Input values out of range

Find all du-paths and identify those du-paths that are definition clear. Also find all du-paths, all-uses and all definitions and generate test cases from them.

Solution:

(i) The program graph is given in Figure 3.19. The variables used are a, b, c, a1, a2, a3, valid.
(ii) Define / use nodes for all variables are given below:

S. No.	Variable	Defined at node	Used at node
1.	a	8	13, 14, 22, 23, 24
2.	b	10	13, 14, 22, 23, 24
3.	c	12	13, 14, 22-24
4.	a1	22	25. 28
5.	a2	23	25, 28
6.	a3	24	25, 28
7.	valid	5, 15, 18	21, 35

(iii) The du-paths with beginning and ending nodes are given as:

S. No.	Variable	du-path (Begin, end)
1.	a	8, 13
		8, 14
		8, 22
		8, 23
		8, 24
2.	b	10, 13
		10, 14
		10, 22
		10, 23
		10, 24
3.	c	12, 13
		12, 14
		12, 22
		12, 23
		12, 24
4.	a1	22. 25
		22, 28

(*Contd.*)

(*Contd.*)

S. No.	Variable	du-path (Begin, end)
5.	a2	23, 25
		23, 28
6.	a3	24, 25
		24, 28
7.	Valid	5, 21
		5, 35
		15, 21
		15, 35
		18, 21
		18, 35

All du-paths are given in Table 4.8 and the test cases for all du-paths are given in Table 4.9.

Table 4.8. All du-paths

All du-paths	Definition clear?	All du paths	Definition clear?
8-13	Yes	12-14, 17-22	Yes
8-14	Yes	12, 13, 21, 22	Yes
8-16, 20-22	Yes	12-16, 20-23	Yes
8-14, 17-22	Yes	12-14, 17-23	Yes
8-13, 21,22	Yes	12, 13, 21-23	Yes
8-16, 20-23	Yes	12-16, 20-24	Yes
8-14, 17-23	Yes	12-14, 17-24	Yes
8-13, 21-23	Yes	12, 13, 21-24	Yes
8-16, 20-24	Yes	22-25	Yes
8-14, 17-24	Yes	22-25, 28	Yes
8-13, 21-24	Yes	23-25	Yes
10-13	Yes	23-25, 28	Yes
10-14	Yes	24, 25	Yes
10-16, 20-22	Yes	24, 25, 28	Yes
10-14, 17-22	Yes	5-16, 20, 21	No
10-13, 21,22	Yes	5-14, 17-21	No
10-16, 20-23	Yes	5-13, 21	Yes
10-14, 17-23	Yes	5-16, 20, 21, 35	No
10-13, 21-23	Yes	5-14, 17-21, 35	No
10-16, 20-24	Yes	5-13, 21, 35	Yes
10-14, 17-24	Yes	15, 16, 20, 21	Yes
10-13, 21-24	Yes	15, 16, 20, 21, 35	Yes
12, 13	Yes	18-21	Yes
12-14	Yes	18-21, 35	Yes
12-16, 20-22	Yes		

We consider all combinations for the design of du-paths. In this process, test cases corresponding to some paths are not possible, but these paths are shown in the list of 'all du-paths'. They may be considered only for completion purpose.

Table 4.9. Test cases for all du-paths

S. No.	A	b	c	Expected output	Remarks
1.	30	20	40	Obtuse angled triangle	8-13
2.	30	20	40	Obtuse angled triangle	8-14
3.	30	20	40	Obtuse angled triangle	8-16, 20-22
4.	-	-	-	-	8-14, 17-22
5.	-	-	-	-	8-13, 21,22
6.	30	20	40	Obtuse angled triangle	8-16, 20-23
7.	-	-	-	-	8-14, 17-23
8.	-	-	-	-	8-13, 21-23
9.	30	20	40	Obtuse angled triangle	8-16, 20-24
10.	-	-	-	-	8-14, 17-24
11.	-	-	-	-	8-13, 21-24
12.	30	20	40	Obtuse angled triangle	10-13
13.	30	20	40	Obtuse angled triangle	10-14
14.	30	20	40	Obtuse angled triangle	10-16, 20-22
15.	-	-	-	-	10-14, 17-22
16.	-	-	-	-	10-13, 21,22
17.	30	20	40	Obtuse angled triangle	10-16, 20-23
18.	-	-	-	-	10-14, 17-23
19.	-	-	-	-	10-13, 21-23
20.	30	20	40	Obtuse angled triangle	10-16, 20-24
21.	-	-	-	-	10-14, 17-24
22.	-	-	-	-	10-13, 21-24
23.	30	20	40	Obtuse angled triangle	12, 13
24.	30	20	40	Obtuse angled triangle	12-14
25.	30	20	40	Obtuse angled triangle	12-16, 20-22
26.	-	-	-	-	12-14, 17-22
27.	-	-	-	-	12, 13, 21, 22
28.	30	20	40	Obtuse angled triangle	12-16, 20-23
29.	-	-	-	-	12-14, 17-23
30.	-	-	-	-	12, 13, 21-23
31.	30	20	40	Obtuse angled triangle	12-16, 20-24
32.	-	-	-	-	12-14, 17-24
33.	-	-	-	-	12, 13, 21-24
34.	30	20	40	Obtuse angled triangle	22-25

(*Contd.*)

(*Contd.*)

S. No.	A	b	c	Expected output	Remarks
35.	30	40	50	Right angled triangle	22-25, 28
36.	30	20	40	Obtuse angled triangle	23-25
37.	30	40	50	Right angled triangle	23-25, 28
38.	30	20	40	Obtuse angled triangle	24, 25
39.	30	40	50	Right angled triangle	24, 25, 28
40.	30	20	40	Obtuse angled triangle	5-16, 20, 21
41.	30	10	15	Invalid triangle	5-14, 17-21
42.	102	−1	6	Input values out of range	5-13, 21
43.	-	-	-	-	5-16, 20, 21, 35
44.	30	10	15	Invalid triangle	5-14, 17-21, 35
45.	102	-1	6	Input values out of range	5-13, 21, 35
46.	30	20	40	Obtuse angled triangle	15, 16, 20, 21
47.	-	-	-	-	15, 16, 20, 21, 35
48.	30	10	15	Invalid triangle	18-21
49.	30	10	15	Invalid triangle	18-21, 35

The ‘all-uses’ paths are given in Table 4.10 and the test cases for all du-paths are given in Table 4.11. The ‘all-definitions’ paths and the test cases are given in Tables 4.12 and 4.13 respectively.

Table 4.10. All uses paths for triangle classification problem

All uses	Definition clear?	All uses	Definition clear?
8-13	Yes	12-16, 20-24	Yes
8-14	Yes	22-25	Yes
8-16, 20-22	Yes	22-25, 28	Yes
8-16, 20-23	Yes	23-25	Yes
8-16, 20-24	Yes	23-25, 28	Yes
10-13	Yes	24, 25	Yes
10-14	Yes	24, 25, 28	Yes
10-16, 20-22	Yes	5-16, 20, 21	No
10-13, 21-23	Yes	5-14, 17-21, 35	No
10-16, 20-24	Yes	15, 16, 20, 21	Yes
12,13	Yes	15, 16, 20, 21, 35	Yes
12-14	Yes	18-21	Yes
12-16, 20, 21, 22	Yes	18-21, 35	Yes
12-16, 20-23	Yes		

Table 4.11. Test cases for all uses paths

S. No.	a	b	c	Expected output	Remarks
1.	30	20	40	Obtuse angled triangle	8–13
2.	30	20	40	Obtuse angled triangle	8–14
3.	30	20	40	Obtuse angled triangle	8–16, 20–22
4.	30	20	40	Obtuse angled triangle	8–16, 20–23
5.	30	20	40	Obtuse angled triangle	8–16, 20–24
6.	30	20	40	Obtuse angled triangle	10–13
7.	30	20	40	Obtuse angled triangle	10–14
8.	30	20	40	Obtuse angled triangle	10–16, 20–22
9.	30	20	40	Obtuse angled triangle	10–13, 21–23
10.	30	20	40	Obtuse angled triangle	10–16, 20–24
11.	30	20	40	Obtuse angled triangle	12,13
12.	30	20	40	Obtuse angled triangle	12–14
13.	30	20	40	Obtuse angled triangle	12–16, 20, 21, 22
14.	30	20	40	Obtuse angled triangle	12–16, 20–23
15.	30	20	40	Obtuse angled triangle	12–16, 20–24
16.	30	20	40	Obtuse angled triangle	22–25
17.	30	40	50	Right angled triangle	22-25, 28
18.	30	20	40	Obtuse angled triangle	23–25
19.	30	40	50	Right angled triangle	23-25, 28
20.	30	20	40	Obtuse angled triangle	24, 25
21.	30	40	50	Right angled triangle	24, 25, 28
22.	30	20	40	Obtuse angled triangle	5–16, 20, 21
23.	30	10	15	Invalid triangle	5–14, 17–21, 35
24.	30	20	40	Obtuse angled triangle	15, 16, 20, 21
25.	-	-	-	-	15, 16, 20, 21, 35
26.	30	10	15	Invalid triangle	18–21
27.	30	10	15	Invalid triangle	18-21, 35

Table 4.12. All definitions paths for triangle classification problem

All definitions	Definition clear?
8–13	Yes
10–13	Yes
12, 13	Yes
22–25	Yes
23–25	Yes
24,25	Yes
5–16, 20, 21	No
15, 16, 20, 21	Yes
18–21	Yes

Table 4.13.

S. No.	a	b	c	Expected output	Remarks
1.	30	20	40	Obtuse angled triangle	8–13
2.	30	20	40	Obtuse angled triangle	10–13
3.	30	20	40	Obtuse angled triangle	12, 13
4.	30	20	40	Obtuse angled triangle	22–25
5.	30	20	40	Obtuse angled triangle	23–25
6.	30	20	40	Obtuse angled triangle	24,25
7.	30	20	40	Obtuse angled triangle	5–16, 20, 21
8.	30	20	40	Obtuse angled triangle	15, 16, 20, 21
9.	30	10	15	Invalid triangle	18–21

Example 4.6: Consider the program given in Figure 3.21 for the determination of day of the week. Its input is at triple of positive integers (day, month, year) from the interval

$1 \leq \text{day} \leq 31$

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

$1900 \leq \text{year} \leq 2058$

The output may be:

[Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday]

Find all du-paths and identify those du-paths that are definition clear. Also find all du-paths, all-uses and all-definitions and generate test cases for these paths.

Solution:

(i) The program graph is given in Figure 3.22. The variables used in the program are day, month, year, century, Y, Y1, M, date, validDate, leap.

(ii) Define / use nodes for all variables are given below:

S. No.	Variable	Defined at node	Used at node
1.	Day	6	19, 27, 30, 37, 91 93, 96, 99, 102 105, 108, 111, 115
2.	Month	8	18, 26, 37, 54 62, 70, 73, 76, 79 82, 85, 93, 96, 99 102, 105, 108, 111, 115
3.	Year	10	11, 12, 14, 45, 47 51, 93, 96, 99, 102 105, 108, 111, 115
4.	Century	46, 50	91
5.	Y	53	91
6.	Y1	47, 51	53

(*Contd.*)

(*Contd.*)

S. No.	Variable	Defined at node	Used at node
7.	M	56, 59, 64 67, 71, 74 77, 80, 83 86, 89	91
8.	Date	91	92, 95, 98, 101, 104, 107
9.	ValidDate	3, 20, 23 28, 31, 34, 38, 41	44
10.	Leap	3, 13, 15	27, 55, 63

(iii) The du-paths with beginning and ending nodes are given as:

S. No.	Variable	du-path (begin, end)
1.	Day	6, 19 6, 27 6, 30 6, 37 6, 91 6, 93 6, 96 6, 99 6, 102 6, 105 6, 108 6, 111 6, 115
2.	Month	8, 18 8, 26 8, 37 8, 54 8, 62 8, 70 8, 73 8, 76 8, 79 8, 82 8, 85 8, 93 8, 96 8, 99 8, 102 8, 105 8, 108 8, 111 8, 115

(*Contd.*)

(*Contd.*)

S. No.	Variable	du-path (begin, end)
3.	Year	10, 11
		10, 12
		10, 14
		10, 45
		10, 47
		10, 51
		10, 93
		10, 96
		10, 99
		10, 102
		10, 105
		10, 108
		10, 111
		10, 115
4.	Century	46, 91
		50, 91
5.	Y	53, 91
6.	Y1	47, 53
		51, 53
7.	M	56, 91
		59, 91
		64, 91
		67, 91
		71, 91
		74, 91
		77, 91
		80, 91
		83, 91
		86, 91
		89, 91
8.	Date	91, 92
		91, 95
		91, 98
		91, 101
		91, 104
		91, 107
9.	ValidDate	3, 44
		20, 44
		23, 44
		28, 44
		31, 44
		34, 44
		38, 44
		41, 44

(*Contd.*)

(*Contd.*)

S. No.	Variable	du-path (begin, end)
10.	Leap	3, 27 3, 55 3, 63 13, 27 13, 55 13, 63 15, 27 15, 55 15, 63

There are more than 10,000 du-paths and it is neither possible nor desirable to show all of them. The all uses paths and their respective test cases are shown in Table 4.14 and Table 4.15 respectively. The 'all definitions' paths are shown in Table 4.16 and their corresponding test cases are given in Table 4.17.

Table 4.14. All uses paths for determination of the day of week problem

All uses	Definition clear?
6–19	Yes
6-18, 26, 27	Yes
6-18, 26, 27, 30	Yes
6-18, 26, 37	Yes
6-21, 25, 43-48, 53, 54, 62, 70, 73, 76, 79-81, 91	Yes
6-21, 25, 43-48, 53, 54, 62, 70, 73, 76, 79-81, 91-93	Yes
6-21, 25, 43-48, 53, 54, 62, 70, 73, 76, 79-81, 91, 92, 95, 96	Yes
6-21, 25, 43-48, 53, 54, 62, 70, 73, 76, 79-81, 91, 92, 95, 98, 99	Yes
6-21, 25, 43-48, 53, 54, 62, 70, 73, 76, 79-81, 91, 92, 95, 98, 101, 102	Yes
6-21, 43-48, 53, 54, 62, 70, 73, 76, 79-81, 91, 92, 95, 98, 101, 104, 105	Yes
6-21, 25, 43-48, 53, 54, 62, 70, 73, 76, 79-81, 91, 92, 95, 98, 101,104, 107, 108	Yes
6-21, 25, 43-48, 53, 54, 62, 70, 73, 76, 79-81, 91, 92, 95, 98, 101, 104, 107, 110, 111	Yes
6-11, 44, 114, 115	Yes
8–18	Yes
8-18, 26	Yes
8-18, 26, 37	Yes
8-21, 25, 43-48, 53, 54	Yes
8-21, 25, 43-48, 53, 54, 62	Yes
8-25, 43-48, 53, 54, 62, 70	Yes
8-21, 25, 43-48, 53, 54, 62, 70, 73	Yes
8-21, 25, 43-48, 53, 54, 62, 70, 73, 76	Yes

(*Contd.*)

(*Contd.*)

All uses

	Definition clear?
8-21, 25, 43-48, 53, 54, 62, 70, 73, 76, 79	Yes
8-21, 25, 43-48, 53, 54, 62, 70, 73, 76, 79, 82	Yes
8-21, 25, 43-48, 53, 54, 62, 70, 73, 76, 79, 82, 85	Yes
8-21, 25, 43-48, 53, 54, 62, 70, 73, 76, 79-81, 91, 92, 93	Yes
8-21, 25, 43-48, 53, 54, 62, 70, 73, 76, 79-81, 91, 92, 95, 96	Yes
8-21, 25, 43-48, 53, 54, 62, 70, 73, 76, 79-81, 91, 92, 95, 98, 99	Yes
8-21, 25, 43-48, 53, 54, 62, 70, 73, 76, 79-81, 91, 92, 95, 98, 101, 102	Yes
8-21, 25, 43-48, 53, 54, 62, 70, 73, 76, 79-81, 91, 92, 95, 98, 101, 104, 105	Yes
8-21, 25, 43-48, 53, 54, 62, 70, 73, 76, 79-81, 91, 92, 95, 98, 101, 104, 107, 108	Yes
8-21, 25, 43-48, 53, 54, 62, 70, 73, 76, 79-81, 91, 92, 95, 98, 101, 104, 107, 110, 111	Yes
8-11, 44, 114, 115	Yes
10, 11	Yes
10-12	Yes
10-14	Yes
10-21, 25, 43-45	Yes
10-21, 25, 43-47	Yes
10-21, 25, 43-45, 49-51	Yes
10-21, 25, 43-48, 53, 54, 62, 70, 73, 76, 79-81, 91-93	Yes
10-21, 25, 43-48, 53, 54, 62, 70, 73, 76, 79-81, 91, 92, 95, 96	Yes
10-21, 25, 43-48, 53, 54, 62, 70, 73, 76, 79-81, 91, 92, 95, 98, 99	Yes
10-21, 25, 43-48, 53, 54, 62, 70, 73, 76, 79-81, 91, 92, 95, 98, 101, 102	Yes
10-21, 25, 43-48, 53, 54, 62, 70, 73, 76, 79-81, 91, 92, 95, 98, 101, 104, 105	Yes
10-21, 25, 43-48, 53, 54, 62, 70, 73, 76, 79-81, 91, 92, 95, 98, 101, 104, 107, 108	Yes
10-21, 25, 43-48, 53, 54, 62, 70, 73, 76, 79-81, 91, 92, 95, 98, 101, 104, 107, 110, 111	Yes
10, 11, 44, 114, 115	Yes
46-48, 53-57, 61, 91	Yes
50-57, 61, 91	Yes
53-61, 91	Yes
47, 48, 53	Yes
51-53	Yes
56, 57, 61, 91	Yes
59-61, 91	Yes

(*Contd.*)

(*Contd.*)

All uses	Definition clear?
64, 65, 69, 91	Yes
67–69, 91	Yes
71, 72, 91	Yes
74, 75, 91	Yes
77, 78, 91	Yes
80, 81, 91	Yes
83, 84, 91	Yes
86, 87, 91	Yes
89, 90, 91	Yes
91, 92	Yes
91, 92, 95	Yes
91, 92, 95, 98	Yes
91, 92, 95, 98, 101	Yes
91, 92, 95, 98, 101, 104	Yes
91, 92, 95, 98, 101, 104, 107	Yes
3–11, 44	No
20, 21, 25, 43, 44	Yes
23–25, 43, 44	Yes
28, 29, 36, 43, 44	Yes
31, 32, 36, 43, 44	Yes
34–36, 43, 44	Yes
38, 39, 43, 44	Yes
41–44	Yes
3–18, 26, 27	No
3–18, 26, 37–39, 43–48, 53–55	No
3–18, 26, 27, 30–32, 36, 43–48, 53, 54, 62, 63	No
13–18, 26, 27	No
13–18, 26, 37–39, 43–48, 53–55	No
13–18, 26, 27, 30–32, 36, 43–48, 53, 54, 62, 63	No
15–18, 26, 27	Yes
15–18, 26, 37–39, 43–48, 53–55	Yes
15–18, 26, 27, 30–32, 36, 43–48, 53, 54, 62, 63	Yes

Table 4.15. Test cases for all uses

S. No.	Month	Day	Year	Expected output	Remarks
1.	6	15	1900	Friday	6–19
2.	2	15	1900	Thursday	6-18, 26, 27
3.	2	15	1900	Thursday	6-18, 26, 27, 30
4.	7	15	1900	Sunday	6-18, 26, 37
5.	6	15	1900	Friday	6-21, 25, 43-48, 53, 54, 62, 70, 73, 76, 79–81, 91
6.	6	10	1900	Sunday	6-21, 25, 43-48, 53, 54, 62, 70, 73, 76, 79–81, 91–93
7.	6	11	1900	Monday	6-21, 25, 43-48, 53, 54, 62, 70, 73, 76, 79-81, 91, 92, 95, 96
8.	6	12	1900	Tuesday	6-21, 25, 43-48, 53, 54, 62, 70, 73, 76, 79-81, 91, 92, 95, 98, 99
9.	6	13	1900	Wednesday	6-21, 25, 43-48, 53, 54, 62, 70, 73, 76, 79-81, 91, 92, 95, 98, 101, 102
10.	6	14	1900	Thursday	6-21, 43-48, 53, 54, 62, 70, 73, 76, 79-81, 91, 92, 95, 98, 101, 104, 105
11.	6	15	1900	Friday	6-21, 25, 43-48, 53, 54, 62, 70, 73, 76, 79-81, 91, 92, 95, 98, 101,104, 107, 108
12.	6	16	1900	Saturday	6-21, 25, 43-48, 53, 54, 62, 70, 73, 76, 79-81, 91, 92, 95, 98, 101, 104, 107, 110, 111
13.	6	15	2059	Invalid Date	6-11, 44, 114, 115
14.	6	15	1900	Friday	8–18
15.	2	15	1900	Thursday	8-18, 26
16.	1	15	1900	Monday	8-18, 26, 37
17.	6	15	1900	Friday	8-21, 25, 43-48, 53, 54
18.	6	15	1900	Friday	8-21, 25, 43-48, 53, 54, 62
19.	6	15	1900	Friday	8-25, 43-48, 53, 54, 62, 70
20.	4	15	1900	Sunday	8-21, 25, 43-48, 53, 54, 62, 70, 73
21.	6	15	1900	Friday	8-21, 25, 43-48, 53, 54, 62, 70, 73, 76
22.	6	15	1900	Friday	8-21, 25, 43-48, 53, 54, 62, 70, 73, 76, 79
23.	9	15	1900	Saturday	8-21, 25, 43-48, 53, 54, 62, 70, 73, 76, 79, 82

(*Contd.*)

(*Contd.*)

S. No.	Month	Day	Year	Expected output	Remarks
24.	9	15	1900	Saturday	8-21, 25, 43-48, 53, 54, 62, 70, 73, 76, 79, 82, 85
25.	6	10	1900	Sunday	8-21, 25, 43-48, 53, 54, 62, 70, 73, 76, 79-81, 91, 92, 93
26.	6	11	1900	Monday	8-21, 25, 43-48, 53, 54, 62, 70, 73, 76, 79-81, 91, 92, 95, 96
27.	6	12	1900	Tuesday	8-21, 25, 43-48, 53, 54, 62, 70, 73, 76, 79-81, 91, 92, 95, 98, 99
28.	6	13	1900	Wednesday	8-21, 25, 43-48, 53, 54, 62, 70, 73, 76, 79,80, 81, 91, 92, 95, 98, 101, 102
29.	6	14	1900	Thursday	8-21, 25, 43-48, 53, 54, 62, 70, 73, 76, 79,80, 81, 91, 92, 95, 98, 101, 104, 105
30.	6	15	1900	Friday	8-21, 25, 43-48, 53, 54, 62, 70, 73, 76, 79, 80, 81, 91, 92, 95, 98, 101, 104, 107, 108
31.	6	16	1900	Saturday	8-21, 25, 43-48, 53, 54, 62, 70, 73, 76, 79, 80, 81, 91, 92, 95, 98, 101, 104, 107, 110, 111
32.	6	15	2059	Invalid Date	8-11, 44, 114, 115
33.	6	15	1900	Friday	10, 11
34.	6	15	1900	Friday	10–12
35.	6	15	1900	Friday	10–14
36.	6	15	1900	Friday	10-21, 25, 43-45
37.	6	15	1900	Friday	10-21, 25, 43-47
38.	6	15	2009	Monday	10-21, 25, 43-45, 49-51
39.	6	10	1900	Sunday	10-21, 25, 43-48, 53, 54, 62, 70, 73, 76, 79-81, 91–93
40.	6	11	1900	Monday	10-21, 25, 43-48, 53, 54, 62, 70, 73, 76, 79-81, 91, 92, 95, 96
41.	6	12	1900	Tuesday	10-21, 25, 43-48, 53, 54, 62, 70, 73, 76, 79-81, 91, 92, 95, 98, 99
42.	6	13	1900	Wednesday	10-21, 25, 43-48, 53, 54, 62, 70, 73, 76, 79-81, 91, 92, 95, 98, 101, 102
43.	6	14	1900	Thursday	10-21, 25, 43-48, 53, 54, 62, 70, 73, 76, 79-81, 91, 92, 95, 98, 101, 104, 105
44.	6	15	1900	Friday	10-21, 25, 43-48, 53, 54, 62, 70, 73, 76, 79-81, 91, 92, 95, 98, 101, 104, 107, 108

(*Contd.*)

(*Contd.*)

S. No.	Month	Day	Year	Expected output	Remarks
45.	6	16	1900	Saturday	10–21, 25, 43–48, 53, 54, 62, 70, 73, 76, 79–81, 91, 92, 95, 98, 101, 104, 107, 110, 111
46.	6	15	2059	Invalid Date	10, 11, 44, 114, 115
47.	1	15	1900	Monday	46–48, 53–57, 61, 91
48.	1	15	2009	Thursday	50–57, 61, 91
49.	1	15	2009	Thursday	53–61, 91
50.	6	15	1900	Friday	47, 48, 53
51.	6	15	2009	Monday	51–53
52.	1	15	2009	Thursday	56, 57, 61, 91
53.	1	15	2000	Saturday	59–61, 91
54.	1	15	2009	Thursday	64, 65, 69, 91
55.	2	15	2000	Tuesday	67–69, 91
56.	3	15	2009	Sunday	71, 72, 91
57.	4	15	2009	Wednesday	74, 75, 91
58.	5	15	2009	Friday	77, 78, 91
59.	6	15	2009	Monday	80, 81, 91
60.	8	15	2009	Saturday	83, 84, 91
61.	9	15	2009	Tuesday	86, 87, 91
62.	7	15	2009	Wednesday	89, 90, 91
63.	5	7	2009	Sunday	91, 92
64.	6	7	2009	Monday	91, 92, 95
65.	7	7	2009	Tuesday	91, 92, 95, 98
66.	8	7	2009	Wednesday	91, 92, 95, 98, 101
67.	9	7	2009	Thursday	91, 92, 95, 98, 101, 104
68.	10	7	2009	Friday	91, 92, 95, 98, 101, 104, 107
69.	6	15	1900	Friday	3–11, 44
70.	6	15	1900	Friday	20, 21, 25, 43, 44
71.	6	31	2009	Invalid Date	23–25, 43, 44

(*Contd.*)

(*Contd.*)

S. No.	Month	Day	Year	Expected output	Remarks
72.	2	15	2000	Tuesday	28, 29, 36, 43, 44
73.	2	15	2009	Sunday	31, 32, 36, 43, 44
74.	2	30	2009	Invalid Date	34-36, 43, 44
75.	8	15	2009	Saturday	38,39, 43, 44
76.	13	1	2009	Invalid Date	41–44
77.	2	15	1900	Thursday	3-18, 26, 27
78.	1	15	1900	Monday	3-18, 26, 37–39, 43–48, 53–55
79.	2	15	1900	Thursday	3-18, 26, 27, 30-32, 36, 43–48, 53, 54, 62, 63
80.	2	15	1900	Thursday	13–18, 26, 27
81.	1	15	1900	Monday	13–18, 26, 37–39, 43–48, 53–55
82.	2	15	1900	Thursday	13–18, 26, 27, 30-32, 36, 43–48, 53, 54, 62, 63
83.	2	15	1900	Thursday	15–18, 26, 27
84.	1	15	1900	Monday	15–18, 26, 37–39, 43–48, 53–55
85.	2	15	1900	Thursday	15–18, 26, 27, 30-32, 36, 43–48, 53, 54, 62, 63

Table 4.16. All definitions paths for determination of the day of week problem

All definitions	Definition clear?
6–19	Yes
8–18	Yes
10, 11	Yes
46–48, 53–57, 61, 91	Yes
50–57, 61, 91	Yes
53–57, 61, 91	Yes
47, 48, 53	Yes
51–53	Yes
56, 57, 61, 91	Yes
59, 60, 61, 91	Yes
64, 65, 69, 91	Yes
67–69, 91	Yes
71, 72, 91	Yes
74, 75, 91	Yes

(*Contd.*)

(*Contd.*)

All definitions	Definition clear?
77, 78, 91	Yes
80, 81, 91	Yes
83, 84, 91	Yes
86, 87, 91	Yes
89–91	Yes
91, 92	Yes
3–11, 44	No
20, 21, 25, 43, 44	Yes
23–25, 43, 44	Yes
28, 29, 36, 43, 44	Yes
31, 32, 36, 43, 44	Yes
34–36, 43, 44	Yes
38, 39, 43, 44	Yes
41–44	Yes
3–18, 26, 27	No
13–18, 26, 27	No
15–18, 26, 27	Yes

Table 4.17. definitions

S. No.	Month	Day	Year	Expected output	Remarks
1.	6	15	1900	Friday	6–19
2.	6	15	1900	Friday	8–18
3.	6	15	1900	Friday	10, 11
4.	1	15	1900	Monday	46–48, 53–57, 61, 91
5.	1	15	2009	Thursday	50–57, 61, 91
6.	1	15	2009	Thursday	53–57, 61, 91
7.	6	15	1900	Friday	47, 48, 53
8.	6	15	2009	Monday	51–53
9.	1	15	2009	Thursday	56, 57, 61, 91
10.	1	15	2000	Saturday	59, 60, 61, 91
11.	1	15	2009	Thursday	64, 65, 69, 91
12.	2	15	2000	Tuesday	67–69, 91
13.	3	15	2009	Sunday	71, 72, 91
14.	4	15	2009	Wednesday	74, 75, 91
15.	5	15	2009	Friday	77, 78, 91
16.	6	15	2009	Monday	80, 81, 91
17.	8	15	2009	Saturday	83, 84, 91

(*Contd.*)

(*Contd.*)

S. No.	Month	Day	Year	Expected output	Remarks
18.	9	15	2009	Tuesday	86, 87, 91
19.	7	15	2009	Wednesday	89-91
20.	6	15	2009	Monday	91, 92
21.	6	15	2059	Invalid Date	3-11, 44
22.	6	15	1900	Friday	20, 21, 25, 43, 44
23.	6	31	2009	Invalid Date	23–25, 43, 44
24.	2	15	2000	Tuesday	28, 29, 36, 43, 44
25.	2	15	2009	Sunday	31, 32, 36, 43, 44
26.	2	30	2009	Invalid Date	34–36, 43, 44
27.	8	15	2009	Saturday	38, 39, 43, 44
28.	13	1	2009	Invalid Date	41–44
29.	2	15	1900	Thursday	3–18, 26, 27
30.	2	15	1900	Thursday	13–18, 26, 27
31.	2	15	1900	Thursday	15–18, 26, 27

SLICE BASED TESTING

Program slicing was introduced by Mark Weiser [WEIS84] where we prepare various subsets (called slices) of a program with respect to its variables and their selected locations in the program. Each variable with one of its location will give us a program slice. A large program may have many smaller programs (its slices), each constructed for different variable subsets. The slices are typically simpler than the original program, thereby simplifying the process of testing of the program. Keith and James [KEIT91] have explained this concept as:

> "Program slicing is a technique for restricting the behaviour of a program to some specified subset of interest. A slice S(v, n) of program P on variable v, or set of variables, at statement n yields the portions of the program that contributed to the value of v just before statement n is executed. S (v, n) is called a slicing criteria. Slices can be computed automatically on source programs by analyzing data flow. A program slice has the added advantage of being an executable program."

Hence, slices are smaller than the original program and may be executed independently. Only two things are important here, variable and its selected location in the program.

• Guidelines for Slicing

There are many variables in the program but their usage may be different in different statements. The following guidelines may be used for the creation of program slices.

1. All statements where variables are defined and redefined should be considered. Consider the program for classification of a triangle (given in Figure 3.18) where variable 'valid' is defined at line number 5 and redefined at line number 15 and line number 18.

5 int valid = 0
15 valid = 1
18 valid = –1
Hence, we may create S(valid, 5), S(valid, 15) and S(valid, 18) slices for variable 'valid' of the program.

2. All statements where variables receive values externally should be considered. Consider the triangle problem (given in Figure 3.18) where variables 'a', 'b' and 'c' receive values externally at line number 8, line number 10 and line number 12 respectively as shown below:
   8 scanf ("%lf", &a);
   10 scanf ("%lf", &b);
   12 scanf ("%lf", &c);
   Hence, we may create S(a, 8), S(b, 10) and S(c, 12) slices for these variables.
3. All statements where output of a variable is printed should be considered. Consider the program to find the largest amongst three numbers (given in Figure 3.11) where variable 'C' is printed at line number 16 and 21 as given below:
   16 printf ("The largest number is: % f \n", C);
   21 printf ("The largest number is: % f \n", C)
   Hence, we may create S(C, 16) and S(C, 21) as slices for 'C' variable
4. All statements where some relevant output is printed should be considered. Consider the triangle classification program where line number 26, 29, 32, 36 and 39 are used for printing the classification of the triangle (given in Figure 3.18) which is very relevant as per logic of the program. The statements are given as:
   26 printf ("Obtuse angled triangle");
   29 printf ("Right angled triangle");
   32 printf ("Acute angled triangle");
   36 printf ("\nInvalid triangle");
   39 printf ("\nInput Values out of Range");
   We may create S(a1, 26), S(a1, 29), S(a1, 32), S(valid, 36) and S(valid, 39) as slices. These are important slices for the purpose of testing.
5. The status of all variables may be considered at the last statement of the program. We consider the triangle classification program (given in figure 3.18) where line number 42 is the last statement of the program. We may create S(a1, 42), S(a2, 42), S(a3, 42), S(valid, 42), S(a, 42), S(b,42) and S(c, 42) as slices.

• Creation of Program Slices

Consider the portion of a program given in Figure 4.2 for the identification of its slices.

1. a = 3;
2. b = 6;
3. c = b^2;
4. d = $a^2 + b^2$;
5. c = a + b;

Figure 4.2. Portion of a program

We identify two slices for variable ‘c’ at statement number 3 and statement number 5 as given in Figure 4.3.

1.	a	=	3;
2.	b	=	6;
5.	c	=	a + b;
S(c, 5)			

Variable ‘c’ at statement 5

2.	b	=	6;
3.	c	=	b^2;
S(c, 3)			

Variable ‘c’ at statement 5

Figure 4.3. Two slices for variable ‘c’

Consider the program given in Figure 4.4.

```
1. void main ( )
2. {
3.   int a, b, c, d, e;
4.   printf (“Enter the values of a, b and c \ n”);
5.  scanf (“%d %d %d”, & a, &b, &c);
6.   d = a+b;
7.   e = b+c:
8.   printf (“%d”, d);
9.   printf (“%d”, e);
10. }
```

Figure 4.4. Example program

Many slices may be created as per criterion (mentioned in section 4.3.1) of the program given in the Figure 4.4. Some of these slices are shown below:

```
1. main ( )
2. {
3.   int a, b, c, d, e;
4.   printf (“Enter the values of a, b and c \ n”);
5.  scanf (“%d %d %d”, &a, &b, &c);
7.  e = b + c;
9.  printf (“%d”, e);
10. }
```

Slice on criterion S (e, 10) = (1, 2, 3, 4, 5, 7, 9, 10)

```
1. main ( )
2. {
3.   int a, b, c, d, e;
4.   printf ("Enter the values of a, b and c \ n");
5.  scanf ("%d %d %d", &a, &b, &c);
6.  d = a + b;
8.  printf ("%d", d);
10. }
```

Slice on criterion S (d,10) = (1, 2, 3, 4, 5, 6, 8, 10)

```
1. main ( )
2. {
3. int a, b, c, d, e;
4. printf ("Enter the values of a, b and c \ n");
5. scanf ("%d %d %d", &a, &b, &c);
7. e = b + c;
10. }
```

Slice on criterion S (e,7) = (1, 2, 3, 4, 5, 7,10)

```
1. main ( )
2. {
3.   int a, b, c, d, e;
4.   printf ("Enter the values of a, b and c \ n");
5.  scanf ("%d %d %d", &a, &b, &c);
6.  d = a + b;
10. }
```

Slice on criterion S (d,6) = (1, 2, 3, 4, 5, 6, 10)

```
1. main ( )
2. {
3.   int a, b, c, d, e;
4.   printf ("Enter the values of a, b and c \ n");
5.  scanf ("%d %d %d", &a, &b, &c);
10. }
```

Slice on criterion S (a, 5) = (1, 2, 3, 4, 5, 10)

We also consider the program to find the largest number amongst three numbers as given in Figure 3.11. There are three variables A, B and C in the program. We may create many slices like S (A, 28), S (B, 28), S (C, 28) which are given in Figure 4.8.

Some other slices and the portions of the program covered by these slices are given as:

S (A, 6) = {1– 6, 28}
S (A, 13) = {1–14, 18, 27, 28}
S (B, 8) = {1– 4, 7, 8, 28}
S (B, 24) = {1–11, 18–20, 22–28}
S (C, 10) = {1– 4, 9, 10, 28}
S (C, 16) = {1–12, 14–18, 27, 28}
S (C, 21) = {1–11, 18–22, 26–28}

It is a good programming practice to create a block even for a single statement. If we consider C++/C/Java programming languages, every single statement should be covered with curly braces { }. However, if we do not do so, the compiler will not show any warning / error message. In the process of generating slices we delete many statements (which are not required in the slice). It is essential to keep the starting and ending brackets of the block of the deleted statements. It is also advisable to give a comment 'do nothing' in order to improve the readability of the source code.

```
     #include<stdio.h>
     #include<conio.h>
1.   void main()
2.   {
3.   float A,B,C;
4.   clrscr();
5.   printf("Enter number 1:\n");
6.   scanf("%f", &A);
7.   printf("Enter number 2:\n");
8.   scanf("%f", &B);
9.   printf("Enter number 3:\n");
10.  scanf("%f", &C);
11.  if(A>B) {
12.  if(A>C) {
13.  printf("The largest number is: %f\n",A);
14.            }
18.  }
27.  getch();
28.  }
```

(a) S(A, 28) ={1-14, 18, 27, 28}

```
     #include<stdio.h>
     #include<conio.h>
1.   void main()
2.   {
3.   float A,B,C;
4.   clrscr();
5.   printf("Enter number 1:\n");
6.   scanf("%f", &A);
7.   printf("Enter number 2:\n");
8.   scanf("%f", &B);
9.   printf("Enter number 3:\n");
10.  scanf("%f", &C);
11.  if(A>B) { /*do nothing*/
18.  }
19.  else {
20.  if(C>B) { /*do nothing*/
22.            }
23.  else {
24.  printf("The largest number is: %f\n",B);
25.     }
26.  }
27.  getch();
28.  }
```

(b) S(B, 28) ={1-11, 18-20, 22-28}

(*Contd.*)

(*Contd.*)

```
        #include<stdio.h>
        #include<conio.h>
 1.     void main()
 2.     {
 3.     float A,B,C;
 4.     clrscr();
 5.     printf("Enter number 1:\n");
 6.     scanf("%f", &A);
 7.     printf("Enter number 2:\n");
 8.     scanf("%f", &B);
 9.     printf("Enter number 3:\n");
10.     scanf("%f", &C);
11.     if(A>B) { /*do nothing*/
18.     }
19.     else {
20.     if(C>B) {
21.             printf("The largest number is: %f\n",C);
22.             }
26.     }
27.     getch();
28.     }
```

(c) S(C, 28)={1-11, 18-22, 26-28}

Figure 4.5. Some slices of program in Figure 3.11

A statement may have many variables. However, only one variable should be used to generate a slice at a time. Different variables in the same statement will generate a different program slice. Hence, there may be a number of slices of a program depending upon the slicing criteria. Every slice is smaller than the original program and can be executed independently. Each slice may have one or more test cases and may help us to focus on the definition, redefinition, last statement of the program, and printing/reading of a variable in the slice. Program slicing has many applications in testing, debugging, program comprehension and software measurement. A statement may have many variables. We should use only one variable of a statement for generating a slice.

- **Generation of Test Cases**

Every slice should be independently executable and may cover some lines of source code of the program as shown in previous examples. The test cases for the slices of the program given in Figure 3.3 (to find the largest number amongst three numbers) are shown in Table 4.18. The generated slices are S(A, 6), S(A, 13), S(A, 28), S(B, 8), S(B, 24), S(B, 28), S(C, 10), S(C, 16), S(C, 21), S(C, 28) as discussed in previous section 4.3.1.

Table 4.18. Test cases using program slices of program to find the largest among three numbers

S. No.	Slice	Lines covered	A	B	C	Expected output
1.	S(A, 6)	1-6, 28	9			No output
2.	S(A, 13)	1-14, 18, 27, 28	9	8	7	9
3.	S(A, 28)	1-14, 18, 27, 28	8	8	7	9
4.	S(B, 8)	1-4, 7, 8, 28		9		No output
5.	S(B, 24)	1-11, 18-20, 22-28	7	9	8	9
6.	S(B, 28)	1-11, 19, 20, 23-28	7	9	8	9
7.	S(C, 10)	1-4, 9, 10, 28			9	No output
8.	S(C, 16)	1-12, 14-18, 27, 28	8	7	9	9
9.	S(C, 21)	1-11, 18-22, 26-28	7	8	9	9
10.	S(C, 28)	1-11, 18-22, 26-28	7	8	9	9

Slice based testing is a popular structural testing technique and focuses on a portion of the program with respect to a variable location in any statement of the program. Hence slicing simplifies the way of testing a program's behaviour with respect to a particular subset of its variables. But slicing cannot test a behaviour which is not represented by a set of variables or a variable of the program.

Example 4.7: Consider the program for determination of division of a student. Consider all variables and generate possible program slices. Design at least one test case from every slice.

Solution:
There are four variables – mark1, mark2, mark3 and avg in the program. We may create many slices as given below:

S (mark1, 7) = {1–7, 34}

S (mark1, 13) = {1–14, 33, 34}

S (mark2, 9) = {1–5, 8, 9, 34}

S (mark2, 13) = {1–14, 33, 34}

S (mark3, 11) = {1–5, 10, 11, 34}

S (mark3, 13) = {1–14, 33, 34}

S (avg, 16) = {1–12, 14–16, 32, 34}

S (avg, 18) = {1–12, 14–19, 32–34}

S (avg, 21) = {1–12, 14–17, 19–22, 29, 31–34}

S (avg, 24) = {1–12, 14–17, 19, 20, 22–25, 29, 31–34}

S (avg, 27) = {1–12, 14–17, 19, 20, 22, 23, 25–29, 31–34}

S (avg, 30) = {1–12, 14–17, 19, 20, 22, 23, 25, 26, 28–34}

The program slices are given in Figure 4.6 and their corresponding test cases are given in Table 4.19.

```
      #include<stdio.h>
      #include<conio.h>
1.    void main()
2.    {
3.    int mark1, mark2,mark3,avg;
4.    clrscr();
5.    printf("Enter marks of 3 subjects
      (between 0-100)\n");
6.    printf("Enter marks of first
      subject:");
7.    scanf("%d", &mark1);
34.   }
```

(a) S(mark1,7)/S(mark1,34)

```
      #include<stdio.h>
      #include<conio.h>
1.    void main()
2.    {
3.    int mark1, mark2,mark3,avg;
4.    clrscr();
5.    printf("Enter marks of 3 subjects (between
      0-100)\n");
8.    printf("Enter marks of second subject:");
9.    scanf("%d", &mark2);
34.   }
```

(b) S(mark2,9)/S(mark2,34)

```
      #include<stdio.h>
      #include<conio.h>
1.    void main()
2.    {
3.    int mark1, mark2,mark3,avg;
4.    clrscr();
5.    printf("Enter marks of 3 subjects
      (between 0-100)\n");
10.   printf("Enter marks of third subject:");
11.   scanf("%d",&mark3);
34.   }
```

(c) S(mark3,11)/S(mark3,34)

```
      #include<stdio.h>
      #include<conio.h>
1.    void main()
2.    {
3.    int mark1, mark2,mark3,avg;
4.    clrscr();
5.    printf("Enter marks of 3 subjects (between
      0-100)\n");
6.    printf("Enter marks of first subject:");
7.    scanf("%d", &mark1);
8.    printf("Enter marks of second subject:");
9.    scanf("%d", &mark2);
10.   printf("Enter marks of third subject:");
11.   scanf("%d",&mark3);
12.   if(mark1>100||mark1<0||mark2>100||mark2<0||
      mark3>100||mark3<0){
13.   printf("Invalid Marks! Please try again");
14.   }
33.   getch();
34.   }
```

(d) S(mark1,13)/S(mark2,13)/S(mark3,13)

(*Contd.*)

(*Contd.*)

```
     #include<stdio.h>
     #include<conio.h>
1.   void main()
2.   {
3.   int mark1, mark2,mark3,avg;
4.   clrscr();
5.   printf("Enter marks of 3 subjects (between
     0-100)\n");
6.   printf("Enter marks of first subject:");
7.   scanf("%d", &mark1);
8.   printf("Enter marks of second subject:");
9.   scanf("%d", &mark2);
10.  printf("Enter marks of third subject:");
11.  scanf("%d",&mark3);
12.  if(mark1>100||mark1<0||mark2>100||mark2
     <0||mark3>100||mark3<0){ /* do nothing*/
14.                 }
15.  else {
16.  avg=(mark1+mark2+mark3)/3;
17.  if(avg<40){
18.                 printf("Fail");
19.            }
32.  }
33.  getch();
34.  }
```

(e) S(avg,18)

```
     #include<stdio.h>
     #include<conio.h>
1.   void main()
2.   {
3.   int mark1, mark2,mark3,avg;
4.   clrscr();
5.   printf("Enter marks of 3 subjects
     (between 0-100)\n");
6.   printf("Enter marks of first subject:");
7.   scanf("%d", &mark1);
8.   printf("Enter marks of second subject:");
9.   scanf("%d", &mark2);
10.  printf("Enter marks of third subject:");
11.  scanf("%d",&mark3);
12.  if(mark1>100||mark1<0||mark2>100||mark2
     <0||mark3>100||mark3<0){
14.  } /* do nothing*/
15.  else {
16.  avg=(mark1+mark2+mark3)/3;
17.  if(avg<40){ /* do nothing*/
19.                 }
20.  else if(avg>=40&&avg<50) {
21.                 printf("Third Division");
22.          }
29.  else { /* do nothing*/
31.                 }
32.  }
33.  getch();
34.  }
```

(f) S(avg,21)

```
     #include<stdio.h>
     #include<conio.h>
1.   void main()
2.   {
3.   int mark1, mark2,mark3,avg;
4.   clrscr();
5.   printf("Enter marks of 3 subjects (between
     0-100)\n");
```

```
     #include<stdio.h>
     #include<conio.h>
1.   void main()
2.   {
3.   int mark1, mark2,mark3,avg;
4.   clrscr();
5.   printf("Enter marks of 3 subjects
     (between 0-100)\n");
```

(*Contd.*)

(*Contd.*)

```
6.    printf("Enter marks of first subject:");
7.    scanf("%d", &mark1);
8.    printf("Enter marks of second subject:");
9.    scanf("%d", &mark2);
10.   printf("Enter marks of third subject:");
11.   scanf("%d",&mark3);
12.   if(mark1>100||mark1<0||mark2>100||mark2
      <0||mark3>100||mark3<0)  {
      /* do nothing*/
14.             }
15.   else {
16.   avg=(mark1+mark2+mark3)/3;
17.   if(avg<40) { /* do nothing*/
19.   }         19.
20.   else if(avg>=40&&avg<50) {
      /* do nothing*/
22.             }
23.   else if(avg>=50&&avg<60) {
24.             printf("Second  Division");
25.             }
29.   else      { /* do nothing*/
31.             }
32.   }
33.   getch();
34.   }
```

(g) S(avg,24)

```
6.    printf("Enter marks of first subject:");
7.    scanf("%d", &mark1);
8.    printf("Enter marks of second subject:");
9.    scanf("%d", &mark2);
10.   printf("Enter marks of third subject:");
11.   scanf("%d",&mark3);
12.   if(mark1>100||mark1<0||mark2>100||mar
      k2<0||mark3>100||mark3<0) {
      /* do nothing*/
14.             }
15.   else {
16.   avg=(mark1+mark2+mark3)/3;
17.   if(avg<40) { /* do nothing*/
      }
20.   else if(avg>=40&&avg<50) {
      /* do nothing*/
22.             }
23.   else if(avg>=50&&avg<60) {
25.             }
26.   else if(avg>=60&&avg<75) {
27.             printf("First Division");
28.             }
29.   else      { /* do nothing*/
31.             }
32.   }
33.   getch();
34.   }
```

(h) S(avg,27)

```
      #include<stdio.h>
      #include<conio.h>
1.    void main()
2.    {
3.    int mark1, mark2,mark3,avg;
4.    clrscr();
5.    printf("Enter marks of 3 subjects (between 0-100)\n");
6.    printf("Enter marks of first subject:");
7.    scanf("%d", &mark1);
8.    printf("Enter marks of second subject:");
```

(*Contd.*)

(*Contd.*)

```
9.    scanf("%d", &mark2);
10.   printf("Enter marks of third subject:");
11.   scanf("%d",&mark3);
12.   if(mark1>100||mark1<0||mark2>100||mark2<0||mark3>100||mark3<0) { /* do nothing*/
14.            }
15.   else {
16.   avg=(mark1+mark2+mark3)/3;
17.   if(avg<40) { /* do nothing*/
19.            }
20.   else if(avg>=40&&avg<50) {/* do nothing*/
22.            }
23.   else if(avg>=50&&avg<60) {/* do nothing*/
25.            }
26.   else if(avg>=60&&avg<75) {/* do nothing*/
28.            }
29.   else     {
30.            printf("First Division with Distinction");
31.            }
32.   }
33.   getch();
34.   }
```

(i) S(avg,30)/S(avg,34)

Figure 4.6. Slices of program for determination of division of a student

Table 4.19. Test cases using program slices

S. No.	Slice	Line covered	mark1	mark2	mark3	Expected output
1.	S(mark1, 7)	1-7, 34	65			No output
2.	S(mark1, 13)	1-14, 33, 34	101	40	50	Invalid marks
3.	S(mark1, 34)	1-7, 34	65			No output
4.	S(mark2, 9)	1-5, 8, 9, 34		65		No output
5.	S(mark2, 13)	1-14, 33, 34	40	101	50	Invalid marks
6.	S(mark2, 34)	1-5, 8, 9, 34		65		No output
7.	S(mark3, 11)	1-5, 10, 11, 34			65	No output
8.	S(mark3, 13)	1-14, 33, 34	40	50	101	Invalid marks

(*Contd.*)

(*Contd.*)

S. No.	Slice	Line covered	mark1	mark2	mark3	Expected output
9.	S(mark3, 34)	1-5, 10, 11, 34			65	No output
10.	S(avg, 16)	1-12, 14-16, 32, 34	45	50	45	No output
11.	S(avg, 18)	1-12, 14-19, 32-34	40	30	20	Fail
12.	S(avg, 21)	1-12, 14-17, 19-22, 29, 32-34	45	50	45	Third division
13.	S(avg, 24)	1-12, 14-17, 19, 20, 22-25, 29, 31-34	55	60	57	Second division
14.	S(avg, 27)	1-12, 14-17, 19, 20, 22, 23, 25-29, 31-34	65	67	65	First division
15.	S(avg, 30)	1-12, 14-17, 19, 20, 22, 23, 25, 26, 28-34	79	80	85	First division with distinction
16.	S(avg, 34)	1-12, 14-17, 19, 20, 22, 23, 25, 26, 28-34	79	80	85	First division with distinction
17.	S(avg, 16)	1-12, 14-16, 32, 34	45	50	45	No output

Example 4.8: Consider the program for classification of a triangle. Consider all variables and generate possible program slices. Design at least one test case from every slice.

Solution:
There are seven variables ‘a’, ‘b’, ‘c’, ‘a1’, ‘a2’, ‘a3’ and ‘valid’ in the program. We may create many slices as given below:

i. S (a, 8) = {1–8, 42}
ii. S (b, 10) = {1–6, 9, 10, 42}
iii. S (c, 12) = {1–6, 11, 12, 42}
iv. S (a1, 22) = {1–16, 20–22, 34, 42}
v. S (a1, 26) = {1–16, 20–22, 25–27, 34, 41, 42}
vi. S (a1, 29) = {1–16, 20–22, 25, 27–31, 33, 34, 41, 42}
vii. S (a1, 32) = {1–16, 20–22, 25, 27, 28, 30–34, 41, 42}
viii. S (a2, 23) = {1–16, 20, 21,23, 34, 42}
ix. S (a2, 26) = {1–16, 20, 21, 23, 25–27, 34, 41, 42)
x. S (a2, 29) = {1–16, 20, 21, 23, 25, 27–31, 33, 34, 41, 42}
xi. S (a2, 32) = {1–16, 20, 21, 23, 25, 27, 28, 30–34, 41, 42}
xii. S (a3, 26) = {1–16, 20, 21, 24–27, 34, 41, 42}
xiii. S (a3, 29) = {1–16, 20, 21, 24, 25, 27–31, 33, 34, 41,42}
xiv. S (a3, 32) = {1–16, 20, 21, 24, 25, 27, 28, 30–34, 41, 42}
xv. S (valid, 5) = {1–5, 42}
xvi. S (valid, 15) = {1–16, 20, 42}
xvii. S (valid, 18) = {1–14, 16–20, 42}
xviii. S (valid, 36) = {1–14, 16–20, 21, 34–38, 40–42}
xix. S (valid, 39) = {1–13, 20, 21, 34, 35, 37–42}

The test cases of the above slices are given in Table 4.20.

Table 4.20. Test cases using program slices

S. No.	**Slice**	**Path**	**a**	**b**	**c**	**Expected output**
1.	S(a, 8)/S(a,42)	1-8, 42	20			No output
2.	S(b, 10)/S(b,42)	1-6, 9, 10, 42		20		No output
3.	S(c, 12)/S(c,42)	1-6, 11, 12, 42			20	No output
4.	S(a1, 22)	1-16, 20-22, 34, 42	30	20	40	No output
5.	S(a1, 26)	1-16, 20-22, 25-27, 34, 41, 42	30	20	40	Obtuse angled triangle
6.	S(a1, 29)	1-16, 20-22, 25, 27-31, 33, 34, 41, 42	30	40	50	Right angled triangle
7.	S(a1, 32)	1-16, 20-22, 25, 27, 28, 30-34, 41, 42	50	60	40	Acute angled triangle
8.	S(a1, 42)	1-16, 20-22, 34, 42	30	20	40	No output
9.	S(a2, 23)	1-16, 20, 21, 23, 34, 42	30	20	40	No output
10.	S(a2, 26)	1-16, 20, 21, 23, 25-27, 34, 41, 42	40	30	20	Obtuse angled triangle
11.	S(a2, 29)	1-16, 20, 21, 23, 25, 27-31, 33, 34, 41, 42	50	40	30	Right angled triangle
12.	S(a2, 32)	1-16, 20, 21, 23, 25, 27, 28, 30-34, 41, 42	40	50	60	Acute angled triangle
13.	S(a2, 42)	1-16, 20, 21, 23, 34, 42	30	20	40	No output
14.	S(a3, 24)	1-16, 20, 21, 24, 34, 42	30	20	40	No output
15.	S(a3, 26)	1-16, 20, 21, 24-27, 34, 41, 42	20	40	30	Obtuse angled triangle
16.	S(a3, 29)	1-16, 20, 21, 24, 25, 27-31, 33, 34, 41, 42	40	50	30	Right angled triangle
17.	S(a3, 32)	1-16, 20, 21, 24, 25, 27, 28, 30-34, 41, 42	50	40	60	Acute angled triangle
18.	S(a3, 42)	1-16, 20, 21, 24, 34, 42	30	20	40	No output
19.	S(valid,5)	1-2, 5, 42				No output
20.	S(valid,15)	1-16, 20, 42	20	40	30	No output
21.	S(valid,18)	1-14, 16-20, 42	30	10	15	No output
22.	S(valid,36)	1-14, 16-20, 21, 34-38, 40-42	30	10	15	Invalid triangle
23.	S(valid,39)	1-13, 20, 21, 34, 35, 37-42	102	-1	6	Input values out of range
24.	S(valid,42)	1-14, 16-20, 42	30	10	15	No output

Example 4.9. Consider the program for determination of day of the week given in Figure 3.13. Consider variables day, validDate, leap and generate possible program slices. Design at least one test case from each slice.

Solution:
There are ten variables – day, month, year, century Y, Y1, M, date, valid date, and leap. We may create many slices for variables day, validDate and leap as given below:

1.	S(day, 6)	=	{1-6, 118}
2.	S(day, 93)	=	{1-11, 18-21, 25, 43-48, 53, 54, 61, 62, 69, 70, 72, 73, 75, 76, 78-81, 88, 90-94, 113, 117, 118}
3.	S(day, 96)	=	{1-11, 18-21, 25, 43-48, 53, 54, 61, 62, 69, 70, 72, 73, 75, 76, 78-81, 88, 90-92, 94-97, 110, 112, 113, 117, 118}
4.	S(day, 99)	=	{1-11, 18-21, 25, 43-48, 53, 54, 61, 62, 69, 70, 72, 73, 75, 76, 78-81, 88, 90-92, 94, 95, 97-100, 110, 112, 113, 117, 118}
5.	S(day, 102)	=	{1-11, 18-21, 25, 43-48, 53, 54, 61, 62, 69, 70, 72, 73, 75, 76, 78-81, 88, 90-92, 94, 95, 97, 98, 100-103, 110, 112, 113, 117, 118}
6.	S(day, 105)	=	{1-11, 18-21, 25, 43-48, 53, 54, 61, 62, 69, 70, 72, 73, 75, 76, 78-81, 88, 90-92, 94, 95, 97, 98, 100, 101, 103-106, 110, 112, 113, 117, 118}
7.	S(day, 108)	=	{1-11, 18-21, 25, 43-48, 53, 54, 61, 62, 69, 70, 72, 73, 75, 76, 78-81, 88, 90-92, 94, 95, 97, 98, 100, 101, 103, 104, 106-110, 112, 113, 117, 118}
8.	S(day, 111)	=	{1-11, 18-21, 25, 43-48, 53, 54, 61, 62, 69, 70, 72, 73, 75, 76, 78-81, 88, 90-92, 94, 95, 97, 98, 100, 101, 103, 104, 106, 107, 109-113, 117, 118}
9.	S(day, 115)	=	{1-11, 43, 44, 113-118}
10.	S(day, 118)	=	{1-6, 118}
11.	S(validDate,3)	=	{1-3, 118}
12.	S(validDate,20)	=	{1-11, 18-21, 25, 43, 118}
13.	S(validDate,23)	=	{1-11, 18, 19, 21-25, 43, 118}
14.	S(validDate,28)	=	{1-13, 17, 18, 25, 26-29, 36, 40, 42, 43, 118}
15.	S(validDate,31)	=	{1-11, 18, 25, 26, 27, 29-33, 35, 36, 40, 42, 43, 118}
16.	S(validDate,34)	=	{1-11, 18, 25, 26, 27, 29, 30, 32-36, 40, 42, 43, 118}
17..	S(validDate,38)	=	{1-11, 18, 25, 26, 36-40, 42, 43, 118}
18.	S(validDate,41)	=	{1-11, 18, 25, 26, 36, 37, 39-43, 118}
19.	S(validDate,118)	=	{1-11, 18, 25, 26, 36, 37, 39-43, 118}
20.	S(leap,3)	=	{1-3, 118}
21.	S(leap,13)	=	{1-13, 17, 43, 118}
22.	S(leap,15)	=	{1-17, 43, 118}
23.	S(leap,118)	=	{1-17, 43, 118}

The test cases for the above slices are given in Table 4.21.

Table 4.21. Test cases using program slices

S. No.	Slice	Lines covered	Month	Day	Year	Expected output
1.	S(day, 6)	1-6, 118	6	-	-	No output
2.	S(day, 93)	1-11, 18-21, 25, 43-48, 53, 54, 61, 62, 69, 70, 72, 73, 75, 76, 78-81, 88, 90-94, 113, 117, 118	6	13	1999	Sunday
3.	S(day, 96)	1-11, 18-21, 25, 43-48, 53, 54, 61, 62, 69, 70, 72, 73, 75, 76, 78-81, 88, 90-92, 94-97, 110, 112, 113, 117, 118	6	14	1999	Monday
4.	S(day, 99)	1-11, 18-21, 25, 43-48, 53, 54, 61, 62, 69, 70, 72, 73, 75, 76, 78-81, 88, 90-92, 94, 95, 97-100, 110, 112, 113, 117, 118	6	15	1999	Tuesday
5.	S(day, 102)	1-11, 18-21, 25, 43-48, 53, 54, 61, 62, 69, 70, 72, 73, 75, 76, 78-81, 88, 90-92, 94, 95, 97, 98, 100-103, 110, 112, 113, 117, 118	6	16	1999	Wednesday
6.	S(day, 105)	1-11, 18-21, 25, 43-48, 53, 54, 61, 62, 69, 70, 72, 73, 75, 76, 78-81, 88, 90-92, 94, 95, 97, 98, 100, 101, 103-106, 110, 112, 113, 117, 118	6	17	1999	Thursday
7.	S(day, 108)	1-11, 18-21, 25, 43-48, 53, 54, 61, 62, 69, 70, 72, 73, 75, 76, 78-81, 88, 90-92, 94, 95, 97, 98, 100, 101, 103, 104, 106-110, 112, 113, 117, 118	6	18	1999	Friday
8.	S(day, 111)	1-11, 18-21, 25, 43-48, 53, 54, 61, 62, 69, 70, 72, 73, 75, 76, 78-81, 88, 90-92, 94, 95, 97, 98, 100, 101, 103, 104, 106, 107, 109-113, 117, 118	6	19	1999	Saturday
9.	S(day, 115)	1-11, 43, 44, 113-118	6	31	2059	Invalid Date
10.	S(day, 118)	1-6, 118	6	19	1999	Saturday
11.	S(validDate,3)	1-3, 118	-	-	-	No output
12.	S(validDate,20)	1-11, 18-21, 25, 43, 118	6	15	2009	No output
13.	S(validDate,23)	1-11, 18, 19, 21-25, 43, 118	6	31	2009	No output
14.	S(validDate,28)	1-13, 17, 18, 25, 26-29, 36, 40, 42, 43, 118	2	15	2000	No output

(*Contd.*)

(*Contd.*)

S. No.	Slice	Lines covered	Month	Day	Year	Expected output
15.	S(validDate,31)	1-11, 18, 25, 26, 27, 29-33, 35, 36, 40, 42, 43, 118	2	15	2009	No output
16.	S(validDate,34)	1-11, 18, 25, 26, 27, 29, 30, 32-36, 40, 42, 43, 118	2	29	2009	No output
17.	S(validDate,38)	1-11, 18, 25, 26, 36-40, 42, 43, 118	8	15	2009	No output
18.	S(validDate,41)	1-11, 18, 25, 26, 36, 37, 39-43, 118	13	15	2009	No output
19.	S(validDate,118)	1-11, 18, 25, 26, 36, 37, 39-43, 118	13	15	2009	No output
20.	S(leap,3)	1-3, 118	-	-	-	No output
21.	S(leap,13)	1-13, 17, 43, 118	8	15	2000	No output
22.	S(leap,15)	1-17, 43, 118	8	15	1900	No output
23.	S(leap,118)	1-17, 43, 118	8	15	1900	No output

MUTATION TESTING

It is a popular technique to assess the effectiveness of a test suite. We may have a large number of test cases for any program. We neither have time nor resources to execute all of them. We may select a few test cases using any testing technique and prepare a test suite. How do we assess the effectiveness of a selected test suite? Is this test suite adequate for the program? If the test suite is not able to make the program fail, there may be one of the following reasons:

(i) The test suite is effective but hardly any errors are there in the program. How will a test suite detect errors when they are not there?

(ii) The test suite is not effective and could not find any errors. Although there may be errors, they could not be detected due to poor selection of test suite. How will errors be detected when the test suite is not effective?

In both the cases, we are not able to find errors, but the reasons are different. In the first case, the program quality is good and the test suite is effective and in the second case, the program quality is not that good and the test suite is also not that effective. When the test suite is not able to detect errors, how do we know whether the test suite is not effective or the program quality is good? Hence, assessing the effectiveness and quality of a test suite is very important. Mutation testing may help us to assess the effectiveness of a test suite and may also enhance the test suite, if it is not adequate for a program.

• Mutation and Mutants

The process of changing a program is known as mutation. This change may be limited to one, two or very few changes in the program. We prepare a copy of the program under test and make a change in a statement of the program. This changed version of the program is known as a mutant of the original program. The behaviour of the mutant may be different from the original program due to the introduction of a change. However, the original program and mutant are syntactically correct and should compile correctly. To mutate a program means to change a program. We generally make only one or two changes in order to assess the effectiveness of the selected test suite. We may make many mutants of a program by making small changes in the program. Every mutant will have a different change in a program. Consider a program to find the largest amongst three numbers as given in Figure 3.11 and its two mutants are given in Figure 4.7 and Figure 4.8. Every change of a program may give a different output as compared to the original program.

Many changes can be made in the program given in Figure 3.11 till it is syntactically correct. Mutant M_1 is obtained by replacing the operator '>' of line number 11 by the operator ' ='. Mutant M_2 is obtained by changing the operator '>' of line number 20 to operator '<'. These changes are simple changes. Only one change has been made in the original program to obtain mutant M_1 and mutant M_2.

```
            #include<stdio.h>
            #include<conio.h>
1.          void main()
2.          {
3.          float A,B,C;
4.          clrscr();
5.          printf("Enter number 1:\n");
6.          scanf("%f", &A);
7.          printf("Enter number 2:\n");
8.          scanf("%f", &B);
9.          printf("Enter number 3:\n");
10.         scanf("%f", &C);
            /*Check for greatest of three numbers*/
11.         if(A>B){  ← if(A=B) { mutated statement ('>' is replaced by '=')
12.         if(A>C) {
13.                 printf("The largest number is: %f\n",A);
14.                 }
15.         else  {
16.                 printf("The largest number is: %f\n",C);
17.                 }
18.         }
19.         else {
20.         if(C>B) {
21.                 printf("The largest number is: %f\n",C);
22.                 }
23.         else {
24.                 printf("The largest number is: %f\n",B);
25.                 }
26.         }
```

(*Contd.*)

(*Contd.*)

```
27.         getch();
28.         }
```

M_1 : First order mutant

Figure 4.7. Mutant$_1$ (M_1) of program to find the largest among three numbers

```
            #include<stdio.h>
            #include<conio.h>
1.          void main()
2.          {
3.          float A,B,C;
4.          clrscr();
5.          printf("Enter number 1:\n");
6.          scanf("%f", &A);
7.          printf("Enter number 2:\n");
8.          scanf("%f", &B);
9.          printf("Enter number 3:\n");
10.         scanf("%f", &C);
            /*Check for greatest of three numbers*/
11.         if(A>B) {
12.         if(A>C) {
13.                 printf("The largest number is: %f\n",A);
14.                 }
15.         else {
16.                 printf("The largest number is: %f\n",C);
17.                 }
18.         }
19.         else {
20.         if(C>B) {    ← if(C<B) { mutated statement ('>' is replaced by '<')
21.                 printf("The largest number is: %f\n",C);
22.                 }
23.         else  {
24.                 printf("The largest number is: %f\n",B);
25.                 }
26.         }
27.         getch();
28.         }
```

M_2 : First order mutant

Figure 4.8. Mutant$_2$ (M_2) of program to find the largest among three numbers

The mutants generated by making only one change are known as first order mutants. We may obtain second order mutants by making two simple changes in the program and third order mutants by making three simple changes, and so on. The second order mutant (M_3) of the program given in Figure 3.11 is obtained by making two changes in the program and thus changing operator '>' of line number 11 to operator '<' and operator '>' of line number 20 to '≥' as given in Figure 4.9. The second order mutants and above are called higher order mutants. Generally, in practice, we prefer to use only first order mutants in order to simplify the process of mutation.

```
        #include<stdio.h>
        #include<conio.h>
1.      void main()
2.      {
3.      float A,B,C;
4.      clrscr();
5.      printf("Enter number 1:\n");
6.      scanf("%f", &A);
7.      printf("Enter number 2:\n");
8.      scanf("%f", &B);
9.      printf("Enter number 3:\n");
10.     scanf("%f", &C);
        /*Check for greatest of three numbers*/
11.     if(A>B) {  ← if(A<B) { mutated statement (replacing '>' by '<')
12.     if(A>C) {
13.             printf("The largest number is: %f\n",A);
14.             }
15.     else  {
16.             printf("The largest number is: %f\n",C);
17.             }
18.     }
19.     else {
20.     if(C>B) {  ← if(C≥B) { mutated statement (replacing '>'by ≥')
21.             printf("The largest number is: %f\n",C);
22.             }
23.     else  {
24.             printf("The largest number is: %f\n",B);
25.             }
26.     }
27.     getch();
28.     }
```

M_3 : Second order mutant

Figure 4.9. Mutant$_3$ (M_3) of program to find the largest among three numbers

• Mutation Operators

Mutants are produced by applying mutant operators. An operator is essentially a grammatical rule that changes a single expression to another expression. The changed expression should be grammatically correct as per the used language. If one or more mutant operators are applied to all expressions of a program, we may be able to generate a large set of mutants. We should measure the degree to which the program is changed. If the original expression is x + 1, and the mutant for that expression is x + 2, that is considered as a lesser change as compared to a mutant where the changed expression is (y * 2) by changing both operands and the operator. We may have a ranking scheme, where a first order mutant is a single change to an expression, a second order mutant is a mutation to a first order mutant, and so on. Higher order mutants become difficult to manage, control and trace. They are not popular in practice and first order mutants are recommended to be used. To kill a mutant, we should be able to execute the changed statement of the program. If we are not able to do so, the fault will not be detected. If x – y is changed to x – 5 to make a mutant, then we should not use the value of y to be equal to 5. If we do so, the fault will not be revealed. Some of the mutant operators for object oriented languages like Java, C++ are given as:

(i) Changing the access modifier, like public to private.
(ii) Static modifier change
(iii) Argument order change
(iv) Super Keyword change
(v) Operator change
(vi) Any operand change by a numeric value.

• Mutation Score

When we execute a mutant using a test suite, we may have any of the following outcomes:

(i) The results of the program are affected by the change and any test case of the test suite detects it. If this happens, then the mutant is called a killed mutant.
(ii) The results of the program are not affected by the change and any test case of the test suite does not detect the mutation. The mutant is called a live mutant.
The mutation score associated with a test suite and its mutants is calculated as:

$$\text{Mutation Score} = \frac{\text{Number of mutants killed}}{\text{Total number of mutants}}$$

The total number of mutants is equal to the number of killed mutants plus the number of live mutants. The mutation score measures how sensitive the program is to the changes and how accurate the test suite is. A mutation score is always between 0 and 1. A higher value of mutation score indicates the effectiveness of the test suite although effectiveness also depends on the types of faults that the mutation operators are designed to represent.

The live mutants are important for us and should be analyzed thoroughly. Why is it that any test case of the test suite not able to detect the changed behaviour of the program? One of the reasons may be that the changed statement was not executed by these test cases. If executed, then also it has no effect on the behaviour of the program. We should write new test cases for live mutants and kill all these mutants. The test cases that identify the changed behaviour should be preserved and transferred to the original test suite in order to enhance the capability of the test suite. Hence, the purpose of mutation testing is not only to assess the capability of a test suite but also to enhance the test suite. Some mutation testing tools are also available in the market like Insure++, Jester for Java (open source) and Nester for C++ (open source).

Example 4.10: Consider the program to find the largest of three numbers as given in figure 3.11. The test suite selected by a testing technique is given as:

S. No.	A	B	C	Expected Output
1.	6	10	2	10
2.	10	6	2	10
3.	6	2	10	10
4.	6	10	20	20

Generate five mutants (M_1 to M_5) and calculate the mutation score of this test suite.

Solution:
The mutated line numbers and changed lines are shown in Table 4.22.

Table 4.22. Mutated statements

Mutant No.	Line no.	Original line	Modified Line
M_1	11	if(A>B)	if (A<B)
M_2	11	if(A>B)	if(A>(B+C))
M_3	12	if(A>C)	if(A<C)
M_4	20	if(C>B)	if(C=B)
M_5	16	printf("The Largest number is:%f\n",C);	printf("The Largest number is:%f\n",B);

The actual output obtained by executing the mutants M_1-M_5 is shown in Tables 4.23-4.27.

Table 4.23. Actual output of mutant M_1

Test case	A	B	C	Expected output	Actual output
1.	6	10	2	10	6
2.	10	6	2	10	6
3.	6	2	10	10	10
4.	6	10	20	20	20

Table 4.24. Actual output of mutant M_2

Test case	A	B	C	Expected output	Actual output
1.	6	10	2	10	10
2.	10	6	2	10	10
3.	6	2	10	10	10
4.	6	10	20	20	20

Table 4.25. Actual output of mutant M_3

Test case	A	B	C	Expected output	Actual output
1.	6	10	2	10	10
2.	10	6	2	10	2
3.	6	2	10	10	6
4.	6	10	20	20	20

Table 4.26. Actual output of mutant M_4

Test case	A	B	C	Expected output	Actual output
1.	6	10	2	10	10
2.	10	6	2	10	10
3.	6	2	10	10	10
4.	6	10	20	20	10

Table 4.27. Actual output of mutant M_5

Test case	A	B	C	Expected output	Actual output
1.	6	10	2	10	10
2.	10	6	2	10	10
3.	6	2	10	10	2
4.	6	10	20	20	20

$$\text{Mutation Score} = \frac{\text{Number of mutants killed}}{\text{Total number of mutants}}$$

$$= \frac{4}{5}$$

$$= 0.8$$

Higher the mutant score, better is the effectiveness of the test suite. The mutant M_2 is live in the example. We may have to write a specific test case to kill this mutant. The additional test case is given in Table 4.28.

Table 4.28. Additional test case

Test case	A	B	C	Expected output
5.	10	5	6	10

Now when we execute the test case 5, the actual output will be different from the expected output (see Table 4.29), hence the mutant will be killed.

Table 4.29. Output of added test case

Test case	A	B	C	Expected output	Actual output
5.	10	5	6	10	6

This test case is very important and should be added to the given test suite. Therefore, the revised test suite is given in Table 4.30.

Table 4.30. Revised test suite

Test case	A	B	C	Expected output
1.	6	10	2	10
2.	10	6	2	10
3.	6	2	10	10
4.	6	10	20	20
5.	10	5	6	10

Example 4.11: Consider the program for classification of triangle given in Figure 3.18. The test suite A and B are selected by two different testing techniques and are given in Table 4.31 and Table 4.32, respectively. The five first order mutants and the modified lines are given in Table 4.33. Calculate the mutation score of each test suite and compare their effectiveness. Also, add any additional test case, if required.

Table 4.31. Test suite A

Test case	a	b	c	Expected output
1.	30	40	90	Invalid triangle
2.	30	20	40	Obtuse angled triangle
3.	50	40	60	Acute angled triangle
4.	30	40	50	Right angled triangle
5.	–1	50	40	Input values are out of range
6.	50	150	90	Input values are out of range
7.	50	40	–1	Input values are out of range

Table 4.32.

Test case	a	b	c	Expected output
1.	40	90	20	Invalid triangle
2.	40	30	60	Obtuse angled triangle
3.	40	50	60	Acute angled triangle
4.	30	40	50	Right angled triangle
5.	–1	50	40	Input values are out of range
6.	30	101	90	Input values are out of range
7.	30	90	0	Input values are out of range

Table 4.33. Mutated lines

Mutant No.	Line no.	Original line	Modified Line
M_1	13	if(a>0&&a<=100&&b>0&&b<=100&&c>0&&c<=100) {	if(a>0||a<=100&&b>0&&b<=100&&c>0&&c<=100) {
M_2	14	if((a+b)>c&&(b+c)>a&&(c+a)>b) {	if((a+b)>c&&(b+c)>a&&(b+a)>b) {
M_3	21	if(valid==1) {	if(valid>1) {
M_4	23	a2=(b*b+c*c)/(a*a);	a2=(b*b+c*c)*(a*a);
M_5	25	if(a1<1||a2<1||a3<1) {	if(a1>1||a2<1||a3<1) {

Solution:

(a) Test cases for Test Suite A

The actual outputs of mutants M_1-M_5 on test suite A are shown in Tables 4.34-4.38.

Table 4.34. Actual output of M_1(A)

Test case	a	b	c	Expected output	Actual output
1.	30	40	90	Invalid triangle	Invalid triangle
2.	30	20	40	Obtuse angled triangle	Obtuse angled triangle
3.	50	40	60	Acute angled triangle	Acute angled triangle
4.	30	40	50	Right angled triangle	Right angled triangle
5.	–1	50	40	Input values are out of range	Invalid triangle
6.	50	150	90	Input values are out of range	Invalid triangle
7.	50	40	–1	Input values are out of range	Invalid triangle

Table 4.35. Actual output of M_2(A)

Test case	a	b	c	Expected output	Actual output
1.	30	40	90	Invalid triangle	Invalid triangle
2.	30	20	40	Obtuse angled triangle	Obtuse angled triangle
3.	50	40	60	Acute angled triangle	Acute angled triangle
4.	30	40	50	Right angled triangle	Right angled triangle
5.	–1	50	40	Input values are out of range	Input values are out of range
6.	50	150	90	Input values are out of range	Input values are out of range
7.	50	40	–1	Input values are out of range	Input values are out of range

Table 4.36. Actual output of M_3(A)

Test case	a	b	c	Expected output	Actual output
1.	30	40	90	Invalid triangle	Invalid triangle
2.	30	20	40	Obtuse angled triangle	Input values are out of range
3.	50	40	60	Acute angled triangle	Input values are out of range
4.	30	40	50	Right angled triangle	Input values are out of range
5.	–1	50	40	Input values are out of range	Input values are out of range
6.	50	150	90	Input values are out of range	Input values are out of range
7.	50	40	–1	Input values are out of range	Input values are out of range

Table 4.37. Actual output of M_4(A)

Test case	a	b	c	Expected output	Actual output
1.	30	40	90	Invalid triangle	Invalid triangle
2.	30	20	40	Obtuse angled triangle	Obtuse angled triangle
3.	50	40	60	Acute angled triangle	Acute angled triangle
4.	30	40	50	Right angled triangle	Right angled triangle
5.	–1	50	40	Input values are out of range	Input values are out of range
6.	50	150	90	Input values are out of range	Input values are out of range
7.	50	40	–1	Input values are out of range	Input values are out of range

Table 4.38. Actual output of M_5(A)

Test case	a	b	c	Expected output	Actual output
1.	30	40	90	Invalid triangle	Invalid triangle
2.	30	20	40	Obtuse angled triangle	Acute angled triangle
3.	50	40	60	Acute angled triangle	Obtuse angled triangle
4.	30	40	50	Right angled triangle	Right angled triangle
5.	-1	50	40	Input values are out of range	Input values are out of range
6.	50	150	90	Input values are out of range	Input values are out of range
7.	50	40	-1	Input values are out of range	Input values are out of range

Two mutants are M_2 and M_4 are live. Thus, the mutation score using test suite A is 0.6.

$$\text{Mutation Score} = \frac{\text{Number of mutants killed}}{\text{Total number of mutants}}$$
$$= \frac{3}{5}$$
$$= 0.6$$

(b) Test cases for Test Suite B

The actual outputs of mutants M_1-M_5 on test suite B are shown in Tables 4.39-4.43.

Table 4.39. Actual output of M_1(B)

Test case	a	b	c	Expected output	Actual output
1.	40	90	20	Invalid triangle	Invalid triangle
2.	40	30	60	Obtuse angled triangle	Obtuse angled triangle
3.	40	50	60	Acute angled triangle	Acute angled triangle
4.	30	40	50	Right angled triangle	Right angled triangle
5.	-1	50	40	Input values are out of range	Invalid triangle
6.	30	101	90	Input values are out of range	Obtuse angled triangle
7.	30	90	0	Input values are out of range	Invalid triangle

Table 4.40. Actual output of M_2

Test case	a	b	c	Expected output	Actual output
1.	40	90	20	Invalid triangle	Obtuse angled triangle
2.	40	30	60	Obtuse angled triangle	Obtuse angled triangle
3.	40	50	60	Acute angled triangle	Acute angled triangle
4.	30	40	50	Right angled triangle	Right angled triangle
5.	-1	50	40	Input values are out of range	Input values are out of range
6.	30	101	90	Input values are out of range	Input values are out of range
7.	30	90	0	Input values are out of range	Input values are out of range

Table 4.41. Actual output of M_3 (B)

Test case	a	b	c	Expected output	Actual output
1.	40	90	20	Invalid triangle	Invalid triangle
2.	40	30	60	Obtuse angled triangle	Input values are out of range
3.	40	50	60	Acute angled triangle	Input values are out of range
4.	30	40	50	Right angled triangle	Input values are out of range
5.	–1	50	40	Input values are out of range	Input values are out of range
6.	30	101	90	Input values are out of range	Input values are out of range
7.	30	90	0	Input values are out of range	Input values are out of range

Table 4.42. Actual output of M_4 (B)

Test case	a	b	c	Expected output	Actual output
1.	40	90	20	Invalid triangle	Invalid triangle
2.	40	30	60	Obtuse angled triangle	Obtuse angled triangle
3.	40	50	60	Acute angled triangle	Acute angled triangle
4.	30	40	50	Right angled triangle	Right angled triangle
5.	–1	50	40	Input values are out of range	Input values are out of range
6.	30	101	90	Input values are out of range	Input values are out of range
7.	30	90	0	Input values are out of range	Input values are out of range

Table 4.43. Actual output of M_5(B)

Test case	a	b	c	Expected output	Actual output
1.	40	90	20	Invalid triangle	Invalid triangle
2.	40	30	60	Obtuse angled triangle	Acute angled triangle
3.	40	50	60	Acute angled triangle	Obtuse angled triangle
4.	30	40	50	Right angled triangle	Right angled triangle
5.	–1	50	40	Input values are out of range	Input values are out of range
6.	30	101	90	Input values are out of range	Input values are out of range
7.	30	90	0	Input values are out of range	Input values are out of range

$$\text{Mutation Score} = \frac{\text{Number of mutants killed}}{\text{Total number of mutants}}$$
$$= \frac{4}{5}$$
$$= 0.8$$

The mutation score of Test suite B is higher as compared to the mutation score of test suite A, hence test suite B is more effective in comparison to test suite A. In order to kill the live mutant (M_4), an additional test case should be added to test suite B as shown in Table 4.44.

Table 4.44. Additional test case

Test case	a	b	c	Expected output
8.	40	30	20	Obtuse angled triangle

The revised test suite B is given in Table 4.45.

Table 4.45. Revised test suite B

Test case	a	b	c	Expected output
1.	40	90	20	Invalid triangle
2.	40	30	60	Obtuse angled triangle
3.	40	50	60	Acute angled triangle
4.	30	40	50	Right angled triangle
5.	−1	50	40	Input values are out of range
6.	30	101	90	Input values are out of range
7.	30	90	0	Input values are out of range
8.	40	30	20	Obtuse angled triangle