MAHARAJA SURAJMAL INSTITUTE OF TECHNOLOGY

STQA Lab

(ETCS – 453)

7th Semester

2017 - 2021

Submitted by:

Siddharth Agarwal

02696302717

CSE (Evening)

Submitted To:

Ms. Tripti Rathee

Assistant Professor

INDEX

EXPERIMENT 1

AIM: Test the program of Quadratic Equation using Boundary Value Analysis.

THEORY:

Boundary value analysis is a software testing technique in which tests are designed to include representatives of boundary values in a range. A boundary value analysis has a total of 4*n+1 distinct test cases, where n is the num ber of variables in a problem.

Here we have to consider all the three variables and design all the distinct possible test cases. We will have a total of 13 test cases as n = 3.

Quadratic equation will be of type: $ax^2+bx+c=0$

- Roots are real if $(b^2 - 4ac) > 0$
- Roots are imaginary if $(b^2 - 4ac) < 0$
- Roots are equal if $(b^2 - 4ac) = 0$
- Equation is not quadratic if a = 0

How do we design the test cases ?

For each variable we consider below 5 cases:

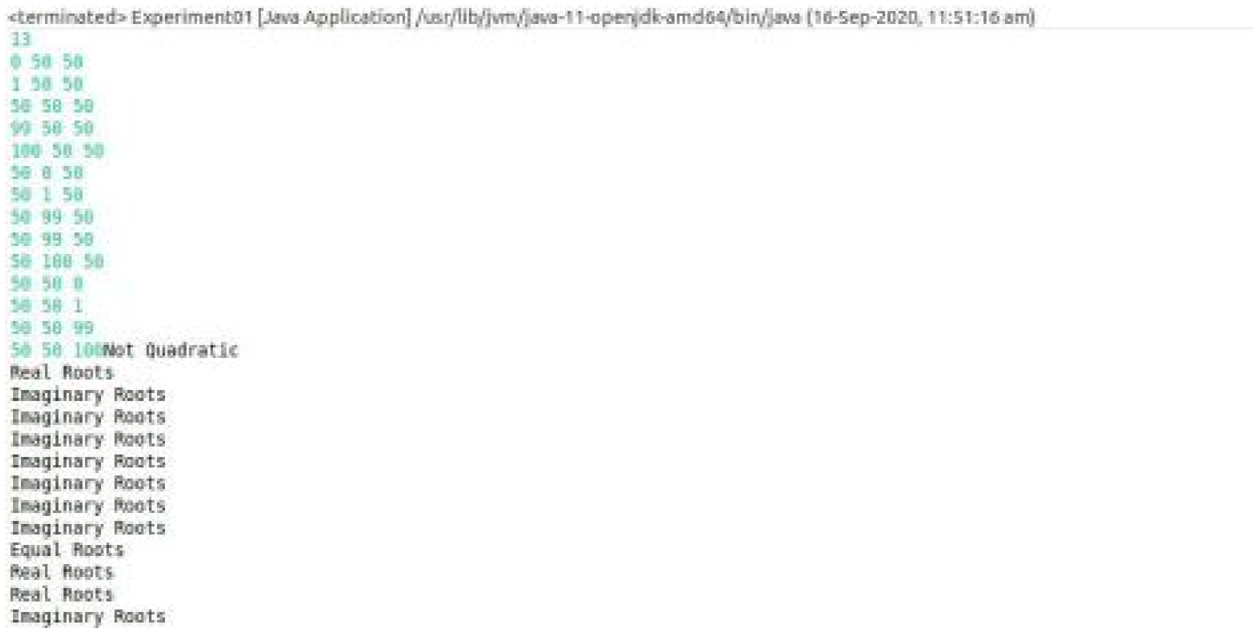
- $a_{min} = 0$
- $a_{min+1} = 1$
- $a_{nominal} = 50$
- $a_{max-1} = 99$
- $a_{max} = 100$

PROGRAM:

```
import java.util.Scanner;
// to detect nature of roots of quadratic equations
public class Main {
 public static void main(String[] args) {
 Scanner s = new Scanner(System.in);
 int t = s.nextInt();
```

```java
for (int i = 0; i < t; i++) {
int a = s.nextInt();
int b = s.nextInt();
int c = s.nextInt();
getRootNature(a, b, c);
}
s.close();
}
private static void getRootNature(int a, int b, int c) {
double D = Math.sqrt(((b * b) - (4 * a * c)));
if (a == 0) {
System.out.println("Not Quadratic");
} else if (D == 0) {
System.out.println("Equal Roots");
} else if (D > 0) {
System.out.println("Real Roots");
} else {
System.out.println("Imaginary Roots");
}
}
}
```

OUTPUT:



```
<terminated> Experiment01 [Java Application] /usr/lib/jvm/java-11-openjdk-amd64/bin/java (16-Sep-2020, 11:51:16 am)
13
0 50 50
1 50 50
50 50 50
99 50 50
100 50 50
50 0 50
50 1 50
50 99 50
50 99 50
50 100 50
50 50 0
50 50 1
50 50 99
50 50 100Not Quadratic
Real Roots
Imaginary Roots
Imaginary Roots
Imaginary Roots
Imaginary Roots
Imaginary Roots
Imaginary Roots
Imaginary Roots
Equal Roots
Real Roots
Real Roots
Imaginary Roots
```

EXPERIMENT 2

AIM: Test the program of Previous Date using Robust Value Analysis

THEORY:

Robustness testing can be seen as an extension of Boundary Value Analysis. The idea behind Robustness testing is to test for clean and dirty test cases. By clean I mean input variables that lie in the legitimate input range. By dirty I mean using input variables that fall just outside this input domain.

In addition to the 5 testing values (min, min+, nom, max -, max) we use two more values for each variable (min-, max+), which are designed to fall just outside of the input range.

If we adapt our function f to apply to Robustness testing we find the following equation:

f = 6n + 1

PROGRAM:

```
#include <iostream>
using namespace std;

void previousDate(int date, int month, int year) {
 if (date <= 0 || date >= 32 || month <= 0 || month >= 13 || year <= 1899 ||
year >= 2026) {
 cout << "Invalid date" << endl;
 return;
 }
 if (date == 31 && month == 6) {
 cout << "Invalid date" << endl;
 return;
 }
 if (date == 1) {
 if (month == 1) {
 date = 31;
 month = 12;
 year--;
 } else if (month == 2 || month == 4 || month == 6 || month == 8 || month
== 9 || month == 11) {
 date = 31;
 month--;
 } else {
 date = 30;
 month--;
 }
 } else
 date--;
 cout << "Prev. Date " << date << "-" << month << "-" << year << endl;
return;
}

void rvaTestCases() {
 cout << "\tTestcase" << "\tdate\tmonth\tyear\tResult" << endl;
int date, month, year;
 int testcase = 1;
 while (testcase <= 19) {
 if (testcase == 1) {
 date = 15;
 month = 6;
 year = 1899;
 } else if (testcase == 2) {
 date = 15;
 month = 6;
 year = 1900;
 } else if (testcase == 3) {
 date = 15;
 month = 6;
 year = 1901;
 } else if (testcase == 4) {
 date = 15;
 month = 6;
 year = 1962;
 } else if (testcase == 5) {
```

```
 date = 15;
 month = 6;
 year = 2024;
 } else if (testcase == 6) {  date
= 15;
 month = 6;
 year = 2025;
 } else if (testcase == 7) {  date
= 15;
 month = 6;
 year = 2026;
 } else if (testcase == 8) {  date
= 0;
 month = 6;
 year = 1962;
 } else if (testcase == 9) {  date
= 1;
 month = 6;
 year = 1962;
 } else if (testcase == 10) {  date
= 2;
 month = 6;
 year = 1962;
 } else if (testcase == 11) {  date
= 30;
 month = 6;
 year = 1962;
 } else if (testcase == 12) {  date
= 31;
 month = 6;
 year = 1962;
 } else if (testcase == 13) {  date
= 32;
 month = 6;
 year = 1962;
 } else if (testcase == 14) {  date
= 15;
 month = 0;
 year = 1962;
 } else if (testcase == 15) {  date
= 15;
 month = 1;
 year = 1962;
 } else if (testcase == 16) {
 date = 15;
 month = 2;
 year = 1962;
 } else if (testcase == 17) {
 date = 15;
 month = 11;
 year = 1962;
 } else if (testcase == 18) {
 date = 15;
 month = 12;
 year = 1962;
 } else if (testcase == 19) {
 date = 15;
 month = 13;
 year = 1962;
 }
 cout << "\t" << testcase << "\t\t" << date << "\t" << month << "\t" <<  year
<< "\t";
 previousDate(date, month, year);
 cout << endl;
 testcase++;
 }
}
int main() {
 rvaTestCases();
 return 0;
}
```

EXPERIMENT 3

AIM: Test the program of Quadratic Equation using Robust Value Analysis

THEORY:

Robustness testing can be seen as an extension of Boundary Value Analysis. The idea behind Robustness testing is to test for clean and dirty test cases. By clean I mean input variables that lie in the legitimate input range. By dirty I mean using input variables that fall just outside this input domain.

In addition to the 5 testing values (min, min+, nom, max -, max) we use two more values for each variable (min-, max+), which are designed to fall just outside of the input range.

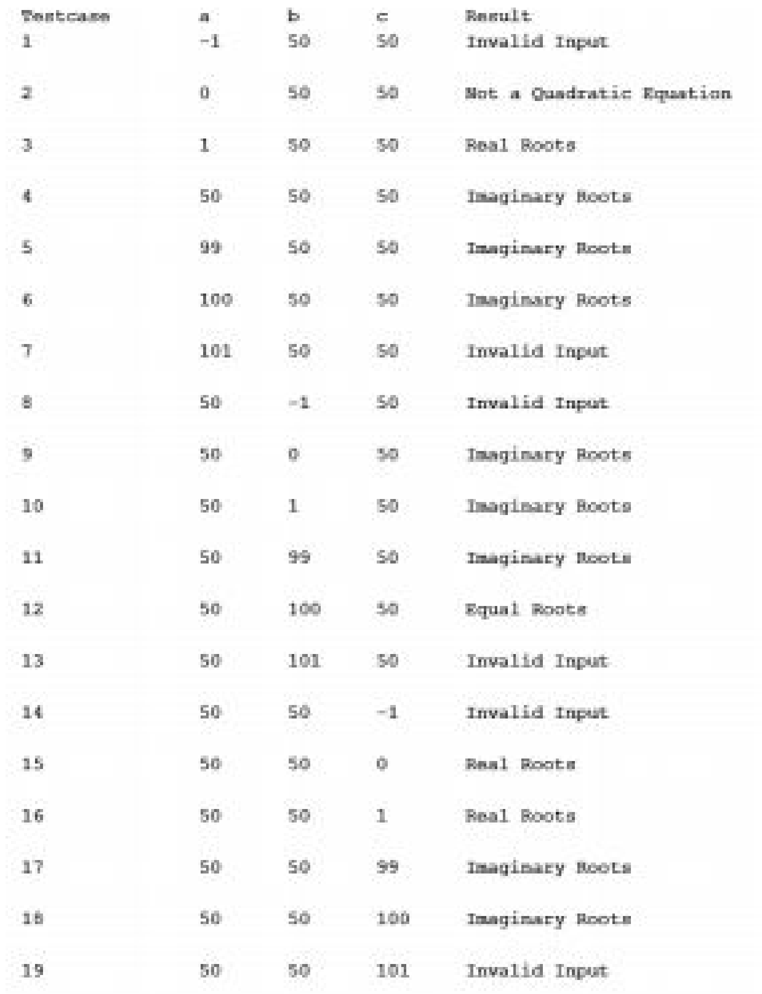
If we adapt our function f to apply to Robustness testing we find the following equation: **f =**

6n + 1

PROGRAM:

```
#include <iostream>
using namespace std;

void typeOfRoots(int a, int b, int c) {
 if (a < 0 || a > 100 || b < 0 || b > 100 || c < 0 || c > 100) {
cout << "Invalid Input" << endl;
 return;
 }
  if (a == 0) {
 cout << "Not a Quadratic Equation" << endl;
 return;
 }
 int D = b * b - 4 * a * c;
 if (D > 0) {
 cout << "Real Roots" << endl;
 } else if (D == 0) {
 cout << "Equal Roots" << endl;
 } else {
 cout << "Imaginary Roots" << endl;
 }
}

void rvaTestCases() {
 cout << "\tTestcase" << "\ta\tb\tc\tResult" << endl;
int a, b, c;
 int testcase = 1;
 while (testcase <= 19) {
 if (testcase == 1) {
 a = -1;
 b = 50;
 c = 50;
 } else if (testcase == 2) {
 a = 0;
 b = 50;
 c = 50;
 } else if (testcase == 3) {
 a = 1;
 b = 50;
 c = 50;
 } else if (testcase == 4) {
 a = 50;
 b = 50;
 c = 50;
 } else if (testcase == 5) {
 a = 99;
 b = 50;
 c = 50;
 } else if (testcase == 6) {
 a = 100;
 b = 50;
 c = 50;
 } else if (testcase == 7) {
 a = 101;
 b = 50;
 c = 50;
 } else if (testcase == 8) {  a =
50;
 b = -1;
 c = 50;
 } else if (testcase == 9) {  a =
50;
 b = 0;
 c = 50;
 } else if (testcase == 10) {  a =
50;
 b = 1;
 c = 50;
 } else if (testcase == 11) {  a =
50;
 b = 99;
 c = 50;
 } else if (testcase == 12) {  a =
50;
 b = 100;
 c = 50;
 } else if (testcase == 13) {  a =
50;
 b = 101;
 c = 50;
 } else if (testcase == 14) {  a =
50;
 b = 50;
 c = -1;
 } else if (testcase == 15) {  a =
50;
 b = 50;
 c = 0;
 } else if (testcase == 16) {  a =
50;
 b = 50;
 c = 1;
 } else if (testcase == 17) {  a =
50;
 b = 50;
 c = 99;
 } else if (testcase == 18) {
 a = 50;
 b = 50;
 c = 100;
 } else if (testcase == 19) {
 a = 50;
 b = 50;
 c = 101;
 }
 cout << "\t" << testcase << "\t\t" << a << "\t" << b << "\t" << c  <<
"\t";
 typeOfRoots(a, b, c);
 cout << endl;
 testcase++;
 }
}
int main() {
 rvaTestCases();
 return 0;
}
```

OUTPUT:



```
Testcase        a       b       c       Result
1               -1      50      50      Invalid Input

2               0       50      50      Not a Quadratic Equation

3               1       50      50      Real Roots

4               50      50      50      Imaginary Roots

5               99      50      50      Imaginary Roots

6               100     50      50      Imaginary Roots

7               101     50      50      Invalid Input

8               50      -1      50      Invalid Input

9               50      0       50      Imaginary Roots

10              50      1       50      Imaginary Roots

11              50      99      50      Imaginary Roots

12              50      100     50      Equal Roots

13              50      101     50      Invalid Input

14              50      50      -1      Invalid Input

15              50      50      0       Real Roots

16              50      50      1       Real Roots

17              50      50      99      Imaginary Roots

18              50      50      100     Imaginary Roots

19              50      50      101     Invalid Input
```

EXPERIMENT 4

AIM: Test the program of Triangle Problem using Boundary Value Analysis

THEORY:

One of the common problem for Test Case Design using BVA is the *Triangle Problem* that is discussed below –

Triangle Problem accepts three integers – a, b, c as three sides of the triangle .We also define a range for the sides of the triangle as [l, r] where l>0. It returns the type of triangle (Scalene, Isosceles, Equilateral, Not a Triangle) formed by a, b, c.

For a, b, c to form a triangle the following conditions should be satisfied – a <

b+c

b < a+c

c < a+b

If any of these conditions is violated output is Not a Triangle.

- For Equilateral Triangle all the sides are equal.
- For Isosceles Triangle exactly one pair of sides is equal.
- For Scalene Triangle all the sides are different.

The range value [l, r] is taken as [1, 100] and nominal value is taken as 50. The total test cases is,

4n+1 = 4*3+1 = 13

PROGRAM:

```
#include <iostream>
 using namespace std;

void natureOfTriangle(int a, int b, int c) {
```

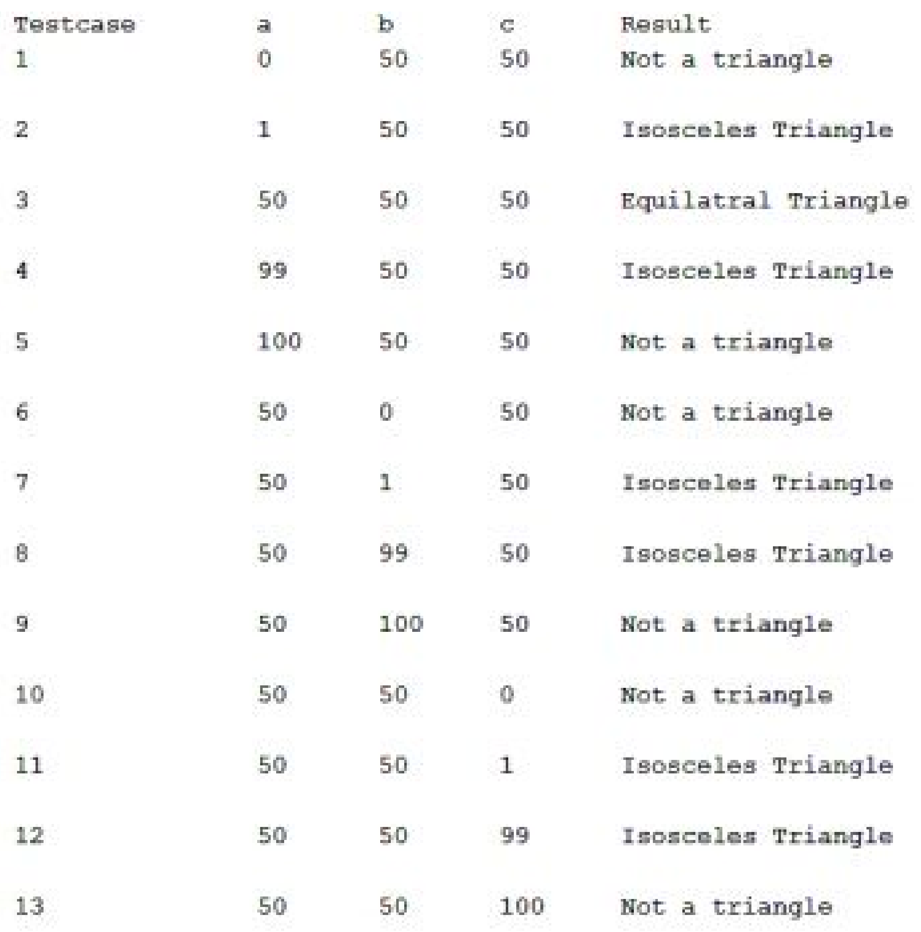
```
if (((a + b) > c) && ((b + c) > a) && ((c + a) > b)) {
if ((a == b) && (b == c)) {
 cout << "Equilatral Triangle" << endl;  } else
if ((a == b) || (b == c) || (c == a)) {  cout <<
"Isosceles Triangle" << endl;  } else
 cout << "Scalene Triangle" << endl;  } else
 cout << "Not a triangle" << endl;
}

void bvaTestCases() {
 cout << "\tTestcase" << "\ta\tb\tc\tResult" << endl;
int a, b, c;
 int testcase = 1;
 while (testcase <= 13) {
 if (testcase == 1) {
 a = 0;
 b = 50;
 c = 50;
 } else if (testcase == 2) {
 a = 1;
 b = 50;
 c = 50;
 } else if (testcase == 3) {
 a = 50;
 b = 50;
 c = 50;
```

```
} else if (testcase == 4) { a =
99;
b = 50;
c = 50;
} else if (testcase == 5) { a =
100;
b = 50;
c = 50;
} else if (testcase == 6) { a =
50;
b = 0;
c = 50;
} else if (testcase == 7) { a =
50;
b = 1;
c = 50;
} else if (testcase == 8) { a =
50;
b = 99;
c = 50;
} else if (testcase == 9) { a =
50;
b = 100;
c = 50;
} else if (testcase == 10) {
```

```
a = 50;
b = 50;
c = 0;
} else if (testcase == 11) {
a = 50;
b = 50;
c = 1;
} else if (testcase == 12) {
a = 50;
b = 50;
c = 99;
} else if (testcase == 13) {
a = 50;
b = 50;
c = 100;
}
cout << "\t" << testcase << "\t\t" << a << "\t" << b << "\t" << c <<
"\t";
natureOfTriangle(a, b, c);
cout << endl;
testcase++;
}
}
int main() {
bvaTestCases();
```

```
 return 0;
}
```

OUTPUT:



```
Testcase        a       b       c       Result
1               0       50      50      Not a triangle

2               1       50      50      Isosceles Triangle

3               50      50      50      Equilatral Triangle

4               99      50      50      Isosceles Triangle

5               100     50      50      Not a triangle

6               50      0       50      Not a triangle

7               50      1       50      Isosceles Triangle

8               50      99      50      Isosceles Triangle

9               50      100     50      Not a triangle

10              50      50      0       Not a triangle

11              50      50      1       Isosceles Triangle

12              50      50      99      Isosceles Triangle

13              50      50      100     Not a triangle
```

EXPERIMENT 5

AIM: Test the program of Triangle Problem using Robust Value Analysis

THEORY:

Triangle Problem accepts three integers – a, b, c as three sides of the triangle .We also define a range for the sides of the triangle as [l, r] where l>0. It returns the type of triangle (Scalene, Isosceles, Equilateral, Not a Triangle) formed by a, b, c.

For a, b, c to form a triangle the following conditions should be satisfied – a <

b+c

b < a+c

c < a+b

If any of these conditions is violated output is Not a Triangle.

- For Equilateral Triangle all the sides are equal.
- For Isosceles Triangle exactly one pair of sides is equal.
- For Scalene Triangle all the sides are different.

The range value [l, r] is taken as [1, 100] and nominal value is taken as 50. The total test cases for Robust Value Analysis is,

6n+1 = 6*3+1 = 19

PROGRAM:

```
#include <iostream>
using namespace std;

void natureOfTriangle(int a, int b, int c) {
 if (a < 0 || a > 100 || b < 0 || b > 100 || c < 0 || c > 100) {
 cout << "Invalid Input" << endl;
 return;
}
```

```
if (((a + b) > c) && ((b + c) > a) && ((c + a) > b)) {
if ((a == b)&& (b == c)) {
cout << "Equilatral Triangle" << endl; }
else if ((a == b) || (b == c) || (c == a)) {
cout << "Isosceles Triangle" << endl; } else
cout << "Scalene Triangle" << endl; }
else
cout << "Not a triangle" << endl;
}


void rvaTestCases() {
cout << "\tTestcase" << "\ta\tb\tc\tResult" << endl;
int a, b, c;
int testcase = 1;
while (testcase <= 19) {
if (testcase == 1) {
a = -1;
b = 50;
c = 50;
} else if (testcase == 2) {
a = 0;
b = 50;
c = 50;
} else if (testcase == 3) {
a = 1;
b = 50;
c = 50;
} else if (testcase == 4) {
a = 50;
b = 50;
c = 50;
} else if (testcase == 5) {
a = 99;
b = 50;
c = 50;
} else if (testcase == 6) {
```

```
a = 100;
b = 50;
c = 50;
} else if (testcase == 7) { a
= 101;
b = 50;
c = 50;
} else if (testcase == 8) { a
= 50;
b = -1;
c = 50;
} else if (testcase == 9) { a
= 50;
b = 0;
c = 50;
} else if (testcase == 10) { a
= 50;
b = 1;
c = 50;
} else if (testcase == 11) { a
= 50;
b = 99;
c = 50;
} else if (testcase == 12) { a
= 50;
b = 100;
c = 50;
} else if (testcase == 13) { a
= 50;
b = 101;
c = 50;
} else if (testcase == 14) { a
= 50;
b = 50;
c = -1;
} else if (testcase == 15) { a
= 50;
```

```
b = 50;
c = 0;
} else if (testcase == 16) {
a = 50;
b = 50;
c = 1;
} else if (testcase == 17) {
a = 50;
b = 50;
c = 99;
} else if (testcase == 18) {
a = 50;
b = 50;
c = 100;
} else if (testcase == 19) {
a = 50;
b = 50;
c = 101;
}
cout << "\t" << testcase << "\t\t" << a << "\t" << b << "\t" << c << "\t";
natureOfTriangle(a, b, c);
cout << endl;
testcase++;
}
}
int main() {
rvaTestCases();
return 0;
}
```

OUTPUT:

Testcase	a	b	c	Result
1	-1	50	50	Invalid Input
2	0	50	50	Not a triangle
3	1	50	50	Isosceles Triangle
4	50	50	50	Equilatral Triangle
5	99	50	50	Isosceles Triangle
6	100	50	50	Not a triangle
7	101	50	50	Invalid Input
8	50	-1	50	Invalid Input
9	50	0	50	Not a triangle
10	50	1	50	Isosceles Triangle
11	50	99	50	Isosceles Triangle
12	50	100	50	Not a triangle
13	50	101	50	Invalid Input
14	50	50	-1	Invalid Input
15	50	50	0	Not a triangle
16	50	50	1	Isosceles Triangle
17	50	50	99	Isosceles Triangle
18	50	50	100	Not a triangle
19	50	50	101	Invalid Input

<u>**EXPERIMENT 6**</u>

AIM: Test the program of Sum of Two Numbers using Worst Case Analysis

THEORY:

If we reject "single" fault assumption theory of reliability, and consider cases where more than 1 variable has extreme values, then it is known as worst case analysis.

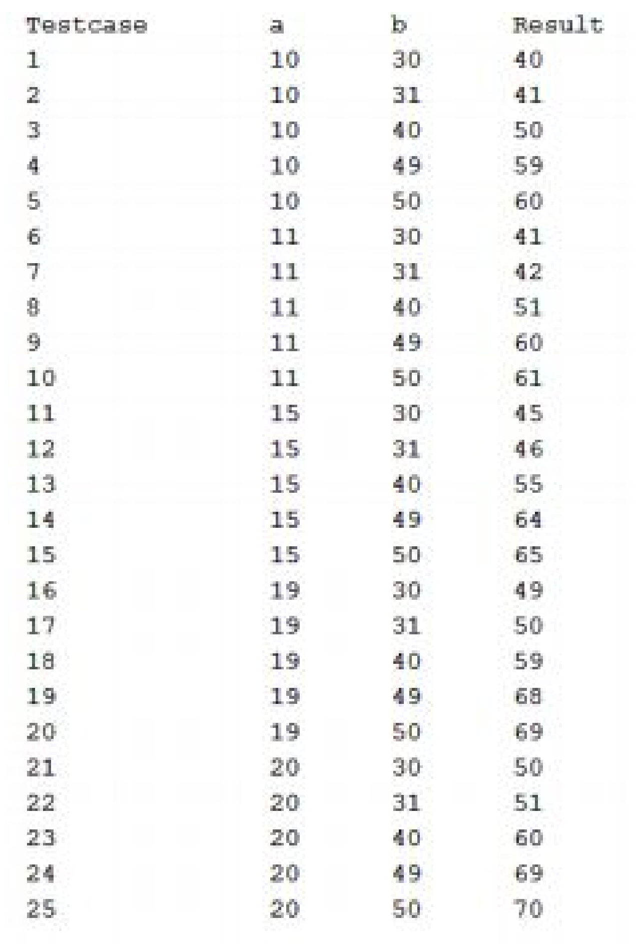
Total no. of test cases,

5^n = 5^2 = 25 cases

PROGRAM:

```
#include <iostream>
using namespace std;

void sumOfNos(int a, int b) {
 cout << a + b;
}

void wcaTestCases() {
 cout << "\tTestcase" << "\ta\tb\tResult" << endl;
 int a, b;
 //10<=a<=20;
 //30<=b<=50;
 int testcase = 1;
 while (testcase <= 25) {
 if (testcase == 1) {
 a = 10;
 b = 30;
 } else if (testcase == 2) {
 a = 10;
 b = 31;
 } else if (testcase == 3) {
 a = 10;
 b = 40;
 } else if (testcase == 4) {  a =
10;
 b = 49;
 } else if (testcase == 5) {  a =
10;
 b = 50;
 } else if (testcase == 6) {  a =
11;
 b = 30;
 } else if (testcase == 7) {  a =
11;
 b = 31;
 } else if (testcase == 8) {  a =
11;
 b = 40;
 } else if (testcase == 9) {  a =
11;
 b = 49;
 } else if (testcase == 10) {  a =
11;
 b = 50;
 } else if (testcase == 11) {  a =
15;
 b = 30;
 } else if (testcase == 12) {  a =
15;
 b = 31;
 } else if (testcase == 13) {  a =
15;
 b = 40;
 } else if (testcase == 14) {  a =
15;
 b = 49;
 } else if (testcase == 15) {  a =
15;
 b = 50;
 } else if (testcase == 16) {  a =
19;
 b = 30;
 } else if (testcase == 17) {
 a = 19;
 b = 31;
 } else if (testcase == 18) {
 a = 19;
 b = 40;
 } else if (testcase == 19) {
 a = 19;
 b = 49;
 } else if (testcase == 20) {
 a = 19;
 b = 50;
 } else if (testcase == 21) {
 a = 20;
 b = 30;
 } else if (testcase == 22) {
 a = 20;
 b = 31;
 } else if (testcase == 23) {
 a = 20;
 b = 40;
 } else if (testcase == 24) {
 a = 20;
 b = 49;
 } else if (testcase == 25) {
 a = 20;
 b = 50;
 }
 cout << "\t" << testcase << "\t\t" << a << "\t" << b << "\t";
sumOfNos(a, b);
 cout << endl;
 testcase++;
 }
}
int main() {
 wcaTestCases();
 return 0;
}
```

OUTPUT:



```
Testcase        a       b       Result
1               10      30      40
2               10      31      41
3               10      40      50
4               10      49      59
5               10      50      60
6               11      30      41
7               11      31      42
8               11      40      51
9               11      49      60
10              11      50      61
11              15      30      45
12              15      31      46
13              15      40      55
14              15      49      64
15              15      50      65
16              19      30      49
17              19      31      50
18              19      40      59
19              19      49      68
20              19      50      69
21              20      30      50
22              20      31      51
23              20      40      60
24              20      49      69
25              20      50      70
```

EXPERIMENT-07

AIM: To test program of Triangle using Decision Table Testing.

THEORY:

Decision Table

A Decision Table is a tabular representation of inputs versus rules/cases/test conditions. It is a very effective tool used for both complex software testing and requirements management. Decision table helps to check all possible combinations of conditions for testing and testers can also identify missed conditions easily. The conditions are indicated as True(T) and False(F) values.

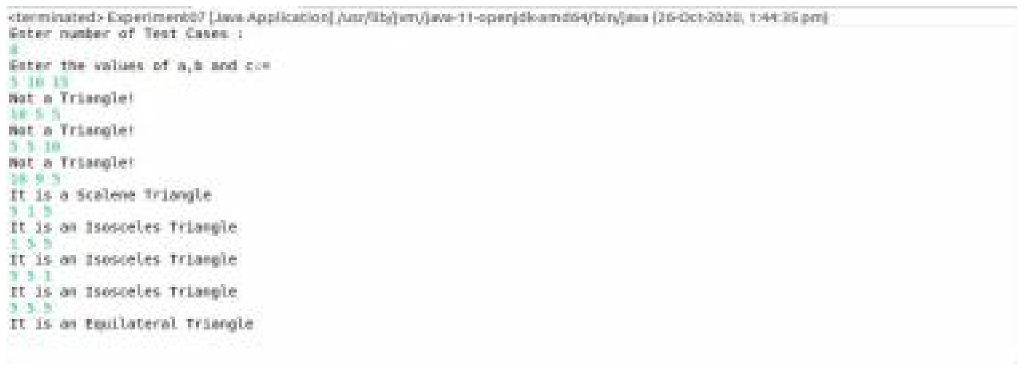
What is Decision Table Testing?

Decision table testing is a software testing technique used to test system behavior for different input combinations. This is a systematic approach where the different input combinations and their corresponding system behavior (Output) are captured in a tabular form. That is why it is also called as a **Cause-Effect** table where Cause and effects are captured for better test coverage.

CODE:

```
import java.util.Scanner;
public class Main {
 public static void main(String[] args) {
 Scanner s = new Scanner(System.in);
 System.out.println("Enter number of Test Cases :");
 int t = s.nextInt();
 System.out.println("Enter the values of a,b and c:= ");
 for (int i = 0; i < t; i++) {
 int a = s.nextInt();
 int b = s.nextInt();
 int c = s.nextInt();
 System.out.println(checkTriangle(a, b, c));
 }
 s.close();
}
private static String checkTriangle(int a, int b, int c) {
 if ((a >= 0 && a <= 10) && (b >= 0 && b <= 10) && (c >= 0 && c <= 10)) {
if ((a + b > c) && (b + c > a) && (c + a > b)) {
 if ((a == b) && (b == c)) {
 return "It is an Equilateral Triangle";
 } else if ((a == b) || (b == c) || (c == a)) {
 return "It is an Isosceles Triangle";  } else {
 return "It is a Scalene Triangle";
 }
 }
 }
 return "Not a Triangle!";
}
}
```

OUTPUT:



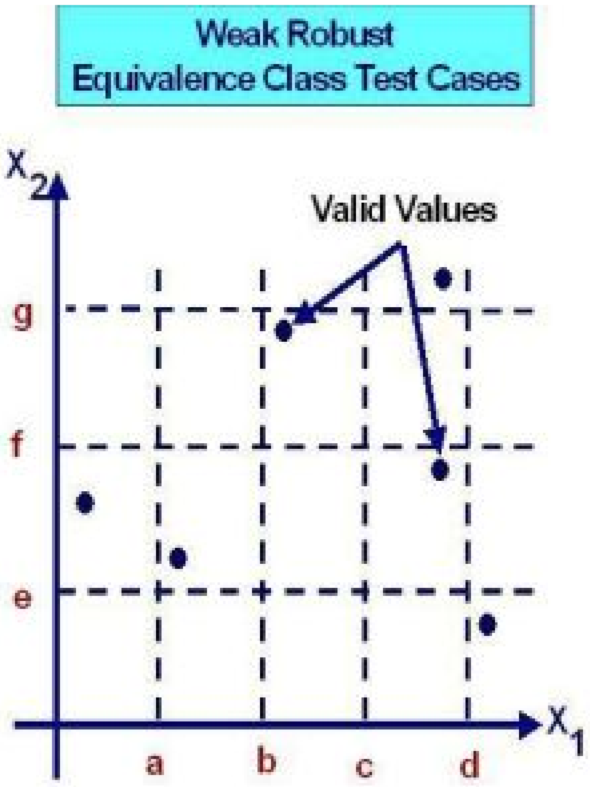
```
<terminated> Experiment07 [Java Application] /usr/lib/jvm/java-11-openjdk-amd64/bin/java (26-Oct-2020, 1:44:35 pm)
Enter number of Test Cases :
8
Enter the values of a,b and c:=
5 10 15
Not a Triangle!
10 5 5
Not a Triangle!
5 5 10
Not a Triangle!
10 9 5
It is a Scalene Triangle
5 1 5
It is an Isosceles Triangle
1 5 5
It is an Isosceles Triangle
5 5 1
It is an Isosceles Triangle
5 5 5
It is an Equilateral Triangle
```

EXPERIMENT 8

AIM: Test the program of Triangle Problem using Weak Robust Equivalence Class Testing

THEORY:

The name for this form of testing is counter intuitive and oxymoronic. The word' weak' means single fault assumption theory and the word 'Robust' refers to invalid values. The test cases resulting from this strategy are shown in the following figure.



Following two problems occur with robust equivalence testing:

a) Very often the specification does not define what the expected output for an invalid test case should be. Thus, testers spend a lot of time defining expected outputs for these cases.

b) Strongly typed languages like Pascal, Ada, eliminate the need for the consideration of invalid inputs. Traditional equivalence testing is a product of the time when languages such as

FORTRAN, C and COBOL were dominant. Thus this type of error was quite common.

PROGRAM:

```
#include <iostream>

using namespace std;

void natureOfTriangle(int a, int b, int c)

{

 if (a < 0 || a >100){

 cout<<"Invalid Input for a"<<endl;

 return;

 }

 else if (b < 0 || b >100){

 cout<<"Invalid Input for b"<<endl;

 return;

 }

 else if (c < 0 || c >100){

 cout<<"Invalid Input for c"<<endl;

 return;

 }

        if(((a+b)>c)&&((b+c)>a)&&((c+a)>b)){
         if((a==b)&&(b==c)){

         cout<<"Equilatral Triangle"<<endl;
```

```
            }
 else if((a==b)||(b==c)||(c==a)){
 cout<<"Isosceles Triangle"<<endl;
 }
 else
 cout<<"Scalene Triangle"<<endl;
 }else
 cout<<"Not a triangle"<<endl;
 }
void weakRobustEqClassTestCases()
{
        cout << "\tTestcase"<< "\ta\tb\tc\tResult\n"<< endl;
        int a, b, c;
        int testcase = 1;
        while (testcase <= 6) {
                if (testcase == 1) {
                        a = -1;
                        b = 50;
                        c = 50;
                }
```

```
}

else if (testcase == 2) {

        a = 50;

        b = -1;

        c = 50;

}

else if (testcase == 3) {

        a = 50;

        b = 50;

        c = -1;

}

else if (testcase == 4) {

        a = 101;

        b = 50;

        c = 50;

}

else if (testcase == 5) {

        a = 50;

        b = 101;

        c = 50;

}
else if (testcase == 6) {
```

```
                        a = 50;

                        b = 50;

                        c = 101;

                }

cout << "\t" << testcase << "\t\t"<< a << "\t" << b << "\t"<< c << "\t";

                        natureOfTriangle(a, b, c);

                cout << endl;

                testcase++;

        }

}

int main()

{

        weakRobustEqClassTestCases();

        return 0;

}
```
OUTPUT:

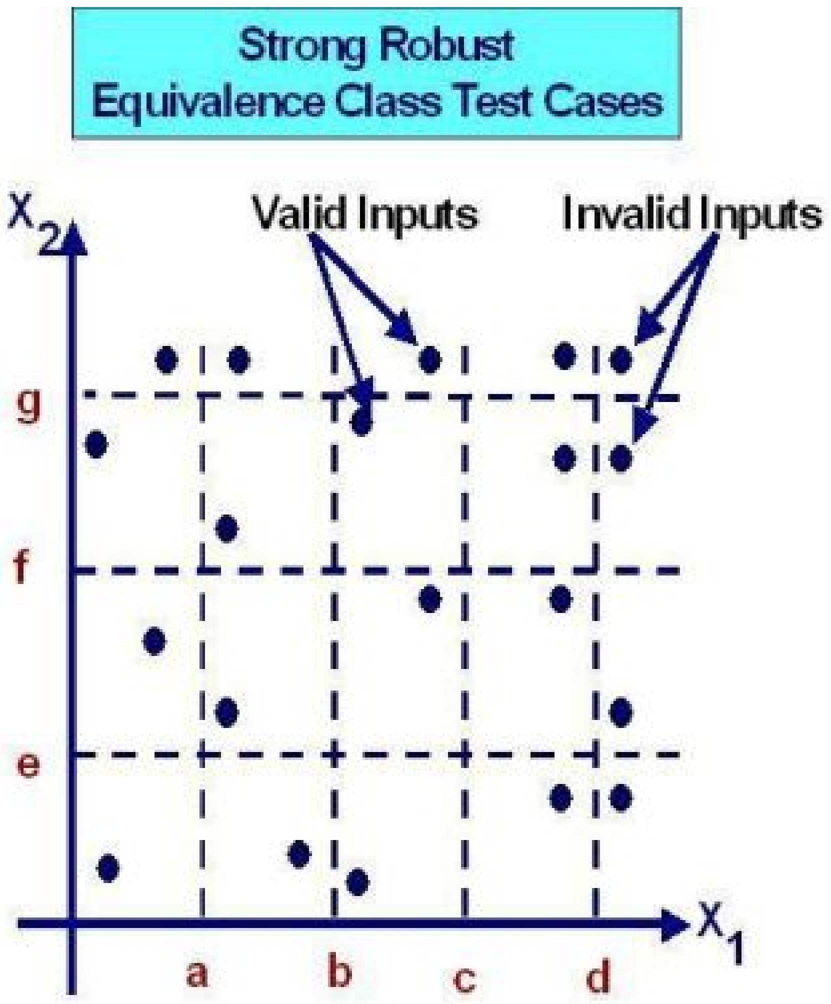
Testcase	a	b	c	Result
1	-1	50	50	Invalid Input for a
2	50	-1	50	Invalid Input for b
3	50	50	-1	Invalid Input for c
4	101	50	50	Invalid Input for a
5	50	101	50	Invalid Input for b
6	50	50	101	Invalid Input for c

EXPERIMENT 9

AIM: Test the program of Previous Date using Strong Robust Equivalence Class Testing

THEORY:

Strong Robust equivalence class testing is neither counter intuitive nor oxymoronic, but is just redundant. 'Robust' means consideration of invalid values and the 'strong' means multiple fault assumption. We obtain the test cases from each element of the Cartesian product of all the equivalence classes as shown in the following figure.

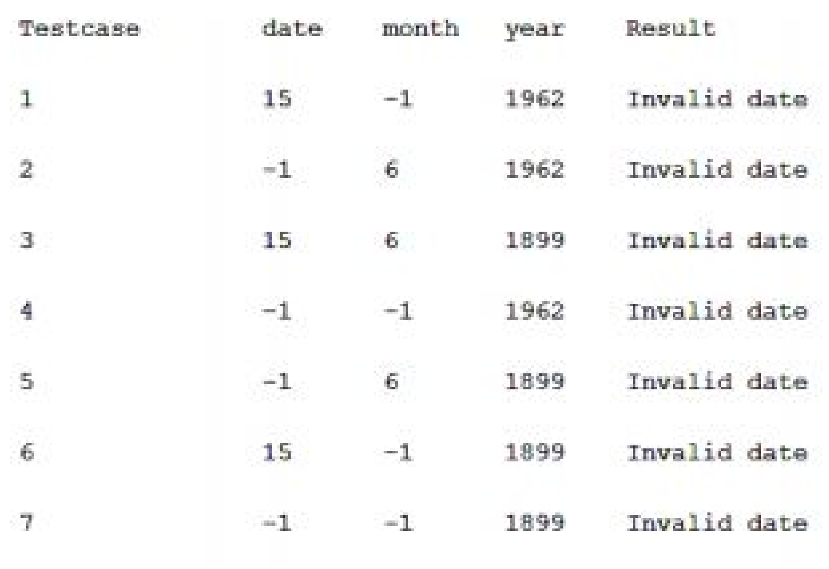


PROGRAM:

```
#include <iostream>
using namespace std;

void previousDate(int date, int month, int year) {
 if (date <= 0 || date >= 32 || month <= 0 || month >= 13 || year <= 1899 || year >=
2026) {
 cout << "Invalid date" << endl;
 return;
}
 if (date == 31 && month == 6) {
 cout << "Invalid date" << endl;
 return;
}
 if (date == 1) {
 if (month == 1) {
 date = 31;
 month = 12;
 year--;
 } else if (month == 2 || month == 4 || month == 6 || month == 8 || month == 9 ||
month == 11) {
 date = 31;
 month--;
 } else {
 date = 30;
 month--;
}
 } else
 date--;
 cout << "Prev. Date " << date << "-" << month << "-" << year << endl;
return;
}

void strongRobustEqClassTestCases() {
 cout << "\tTestcase" << "\tdate\tmonth\tyear\tResult\n" << endl;
int date, month, year;
 int testcase = 1;
 while (testcase <= 7) {
 if (testcase == 1) {
 date = 15;
 month = -1;
 year = 1962;
 } else if (testcase == 2) {
 date = -1;
 month = 6;
 year = 1962;
 } else if (testcase == 3) {
 date = 15;
 month = 6;
 year = 1899;
 } else if (testcase == 4) {
 date = -1;
 month = -1;
 year = 1962;
 } else if (testcase == 5) {
 date = -1;
 month = 6;
 year = 1899;
 } else if (testcase == 6) {
 date = 15;
 month = -1;
 year = 1899;
 } else if (testcase == 7) {
 date = -1;
 month = -1;
 year = 1899;
 }
 cout << "\t" << testcase << "\t\t" << date << "\t" <<month << "\t" << year << "\t";
previousDate(date, month, year);
 cout << endl;
 testcase++;
 }
```

```
}
int main() {
 strongRobustEqClassTestCases();
 return 0;
}
```

OUTPUT:



```
Testcase         date    month    year    Result

1                15      -1       1962    Invalid date

2                -1      6        1962    Invalid date

3                15      6        1899    Invalid date

4                -1      -1       1962    Invalid date

5                -1      6        1899    Invalid date

6                15      -1       1899    Invalid date

7                -1      -1       1899    Invalid date
```

EXPERIMENT 10

AIM: Test the program of matrix multiplication with DD path testing

THEORY:

Path testing is a structural testing method that involves using the source code of a program in order to find every possible executable path. It helps to determine all faults lying within a piece of code. This method is designed to execute all or selected path through a computer program. Decision to Decision path (D-D) - The CFG can be broken into various Decision to Decision paths and then collapsed into individual nodes.

Properties

· Every node on the flow graph of a program belongs to one DD path.

· It is used to find independent path for testing.

· Every statement has been extended at least once.

Note: - For designing a DD path a flow graph is made first and then combining all sequential steps into one step. The program of finding the larger of the two num bers consists of 11 lines.

Step 1: Generating flow graph

```
        1->2->3->4->5->6->7->8
                          |
               12<-11<-10<-9->13->14->15
                          |
                          17
               |
               18
               |
               19
```

1-8: Sequential Steps

9: Decision Step

Step 2: Generating DD path

```
                        A
                        |
                        B
                    / \
                    C D
                    \ /
                    E
                        |
                        F
```

No of Edges: 6

Next step is to find out independent path using cyclometric complexity:
There are many methods:-

1. e – n + 2p
   = 6 – 6 + 2
   = 2
2. No of Regions = 2

So no of independent paths are 2

1. ABCEF
2. ABDEF

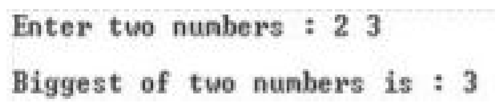
Test Cases:

Test Case Id	a	b	Expected Output
1	4	3	A is greater
2	3	4	B is greater

PROGRAM:

```
import java.util.Scanner;
public class Main {
 public static void main(String args[]) {
 Scanner scn = new Scanner(System.in);
 System.out.println("Enter two numbers : ");
 int a = scn.nextInt();
 int b = scn.nextInt();
 System.out.println("Biggest of two numbers is : "+Math.max(a,b));
scn.close();
 }
}
```

OUTPUT:



```
Enter two numbers : 2 3
Biggest of two numbers is : 3
```

EXPERIMENT 11

AIM: Introduction of Rational Robot. Test the GUI of 'Classics Online Application' with Rational Robot

THEORY:

Rational Robot is a complete set of components for automating the testing of Microsoft Windows client/server and Internet applications running under Windows NT 4.0, Windows 2000, Windows 98, and Windows 95.

The main component of Robot lets you start recording tests in as few as two mouse clicks. After recording, Robot plays back the tests in a fraction of the time it would take to repeat the actions manually.

Rational Robot is an automated functional regression testing tool.

· Functional Test: Functional Tests are designed to make sure that the application performs as it was intended

· Regression Test: A regression test is a test where an application is subjected to a suite of functional tests at each build to ensure that everything that worked continues to work.

Other components of Robot are:

· Rational Administrator Use to create and manage Rational projects, which store your testing information.

· Rational TestManager Log Use to review and analyze test results.

· Object Properties, Text, Grid, and Image Comparators Use to view and analyze the results of verification point playback.

· Rational SiteCheck Use to manage Internet and intranet Web sites.

How does rational robot works?

Rational robot let's all members of your development and testing teams implement a complete and effective testing methodology, Robot replaces the repetitive, often error-prone process of manual testing with the software tools to automate your testing effort.

With Robot's automated functional testing, you save time and ensure that your testing process produces predictable and accurate results. With Robot, you can start recording tests in as few as two mouse click. After recording, Robot plays back the tests in a fraction of the time it would take to repeat the steps manually.

Robot's object-Oriented Recording technology lets you generate scripts quickly & simple by running and using the application-under test.

Object-Oriented Recording identifies objects by their internal object names, not by screen coordinates.

Analyzing the results in the log viewer and comparators

The Rational Log Viewer lets you view logs that are created when you play back scripts in Robot. Reviewing the playback results in the Log Viewer reveals whether each script and its components passed or failed.

To analyse each failure and then remedy it, you can use one of the Log Viewer's four comparators Grid, Object Properties, Text and Images. Each Comparator graphically displays the 'before and after' results of playback. If there is no failure on playback, only a baseline file displaying the recorded data or image is displayed. If a failure occurs on playba ck, an actual file is displayed next to the baseline file.

Tracking applications with rational test factory

· Automatically create and maintain a detailed map of the application-under-test. · Automatically generates scripts that provide extensive product coverage and scripts that encounter defects without recording.

· Track executed and unexecuted source code, and report detailed findings. · Shorten the product testing cycle by minimizing the time invested in writing navigation code.

· Automated distributed functional testing with Accelerator- an application that drives and manages the execution of scripts on remote machines.