SOFTWARE TESTING JOURNAL

**EXPERIMENT NO:1**

**THEORY:**

Test Data : Enter the 3 Integer Value( a , b And c )

Pre-condition : a < b + c , b < a + c and c < a + b

Brief Description : Check whether given value for a equilateral, isosceles , Scalene triangle or can't form a triangle

**ALGORITHM:**

1. Start

2. Print a message to prompt the user to enter the lengths of the three sides of the triangle.

3. Read the input values for side1, side2, and side3 as floating-point numbers.

4. heck if the sum of any two sides is greater than the third side (the Triangle Inequality Theorem). If all three conditions are true, proceed to the next step; otherwise, go to step 8.

Print "Triangle is possible to construct."

Check the type of triangle based on the side lengths:

5. If side1 is equal to side2 and side2 is equal to side3, then it's an equilateral triangle. Print "Equilateral triangle."

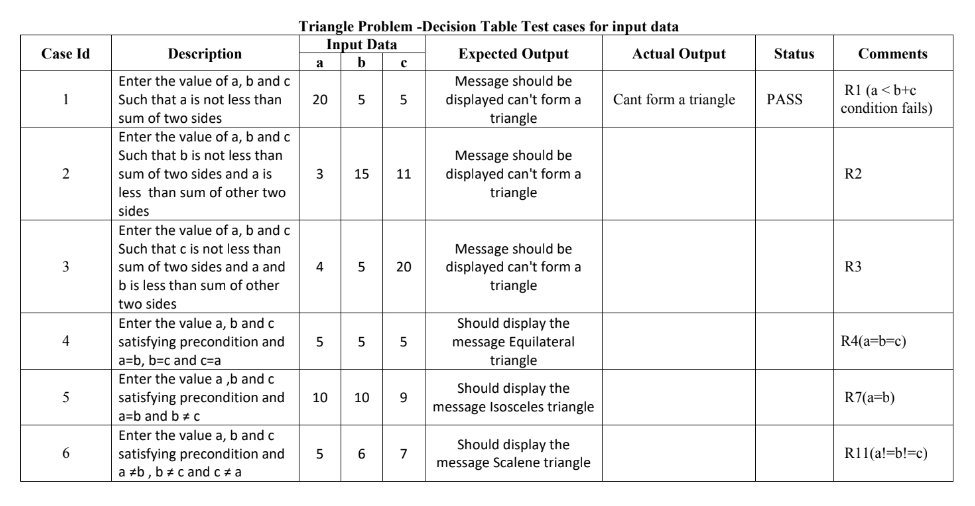
Else if side1 is equal to side2 or side2 is equal to side3 or side3 is equal to side1, then it's an isosceles triangle. Print "Isosceles triangle."

Otherwise, it's a scalene triangle. Print "Scalene triangle."

End the program.

6. If any of the conditions in step 4 is false, print "Triangle is not possible to construct."

7. End the program.

**OUTPUT:**

**EXPERIMENT NO:02**

**THEORY:**

Test Data : Enter the 3 Integer Value( a , b And c )

Pre-condition : 1 ≤ a ≤ 10 , 1 ≤ b ≤ 10 and 1 ≤ c ≤ 10 and a < b + c , b < a + c and c < a + b

Brief Description : Check whether given value for a Equilateral, Isosceles , Scalene triangle or can't form a triangle.

**ALGORITHM:**

1. Start

2. Print a message to prompt the user to enter the lengths of the three sides of the triangle.

3. Read the input values for side1, side2, and side3 as floating-point numbers.

4. heck if the sum of any two sides is greater than the third side (the Triangle Inequality Theorem). If all three conditions are true, proceed to the next step; otherwise, go to step 8.

Print "Triangle is possible to construct."

Check the type of triangle based on the side lengths:

5. If side1 is equal to side2 and side2 is equal to side3, then it's an equilateral triangle. Print "Equilateral triangle."

Else if side1 is equal to side2 or side2 is equal to side3 or side3 is equal to side1, then it's an isosceles triangle. Print "Isosceles triangle."

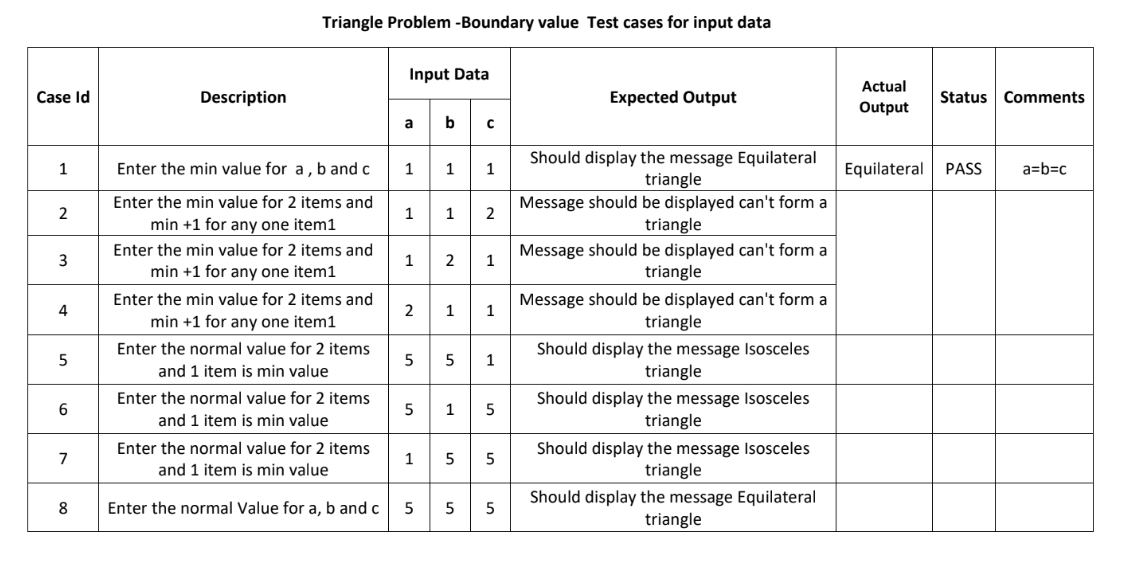
Otherwise, it's a scalene triangle. Print "Scalene triangle."

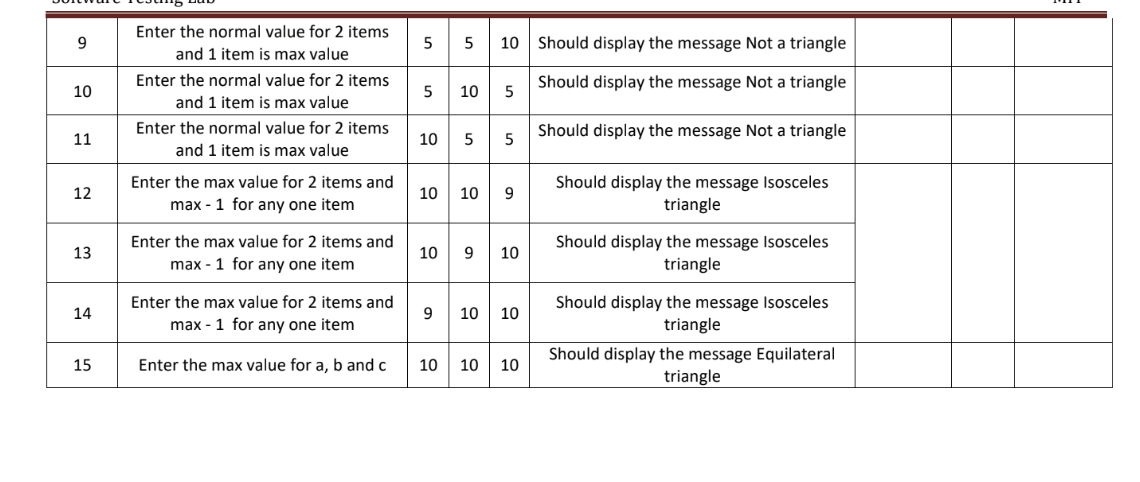
End the program.

6. If any of the conditions in step 4 is false, print "Triangle is not possible to construct."

7. End the program.

**OUTPUT:**

****

****

**EXPERIMENT NO: 03**

**THEORY:**

Test Data : Enter the 3 Integer Value( a , b And c )

Classes: { (1,3),(3,5),(5,7)(7,10) }

Pre-condition : 1 ≤ a ≤ 10 , 1 ≤ b ≤ 10 and 1 ≤ c ≤ 10 and a < b + c , b < a + c and c < a + b

Brief Description : Check whether given value for a Equilateral, Isosceles , Scalene triangle or can't form a triangle

**ALGORITHM:**

1. Start

2. Print a message to prompt the user to enter the lengths of the three sides of the triangle.

3. Read the input values for side1, side2, and side3 as floating-point numbers.

4. heck if the sum of any two sides is greater than the third side (the Triangle Inequality Theorem). If all three conditions are true, proceed to the next step; otherwise, go to step 8.

Print "Triangle is possible to construct."

Check the type of triangle based on the side lengths:

5. If side1 is equal to side2 and side2 is equal to side3, then it's an equilateral triangle. Print "Equilateral triangle."

Else if side1 is equal to side2 or side2 is equal to side3 or side3 is equal to side1, then it's an isosceles triangle. Print "Isosceles triangle."

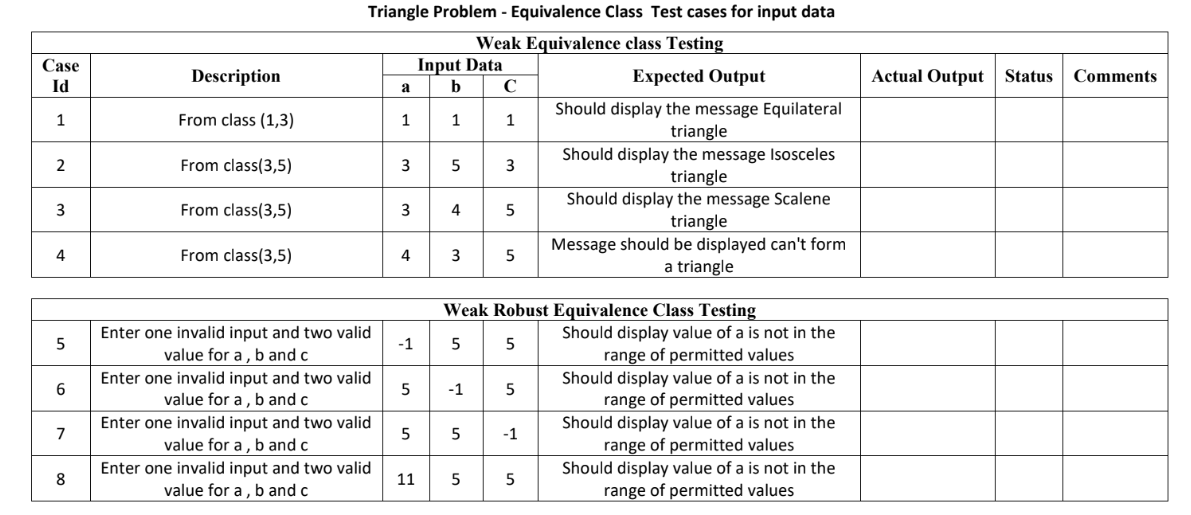
Otherwise, it's a scalene triangle. Print "Scalene triangle."

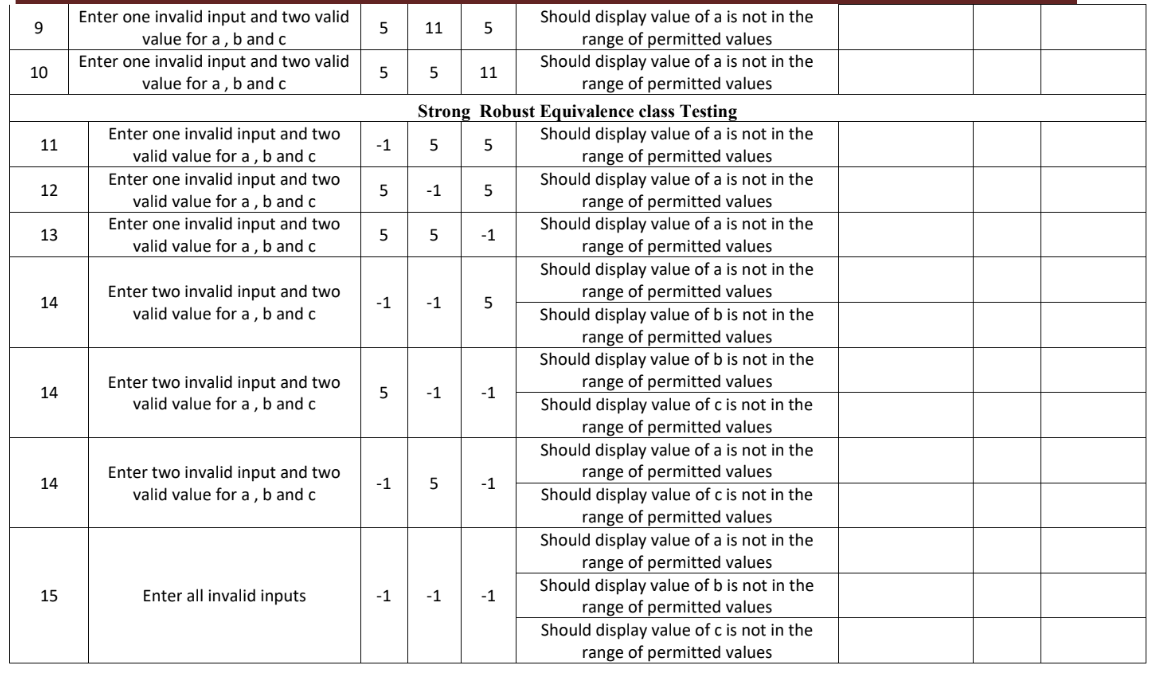
End the program.

6. If any of the conditions in step 4 is false, print "Triangle is not possible to construct."

7. End the program.

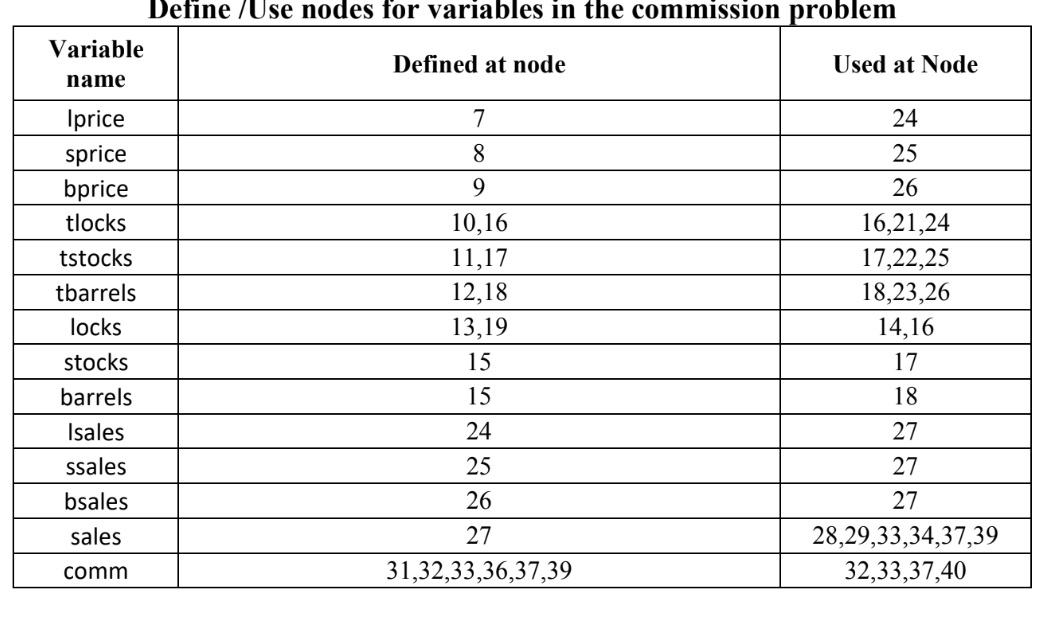
OUTPUT:

****

****

**EXPERIMENT NO: 04**

**THEORY:**

****

**…….**

Precondition : Enter -1 for locks to exit from input loop

Brief Description : Enter the locks, stocks and barrels > 0

**ALGORITHM:**

1. Start

2. Initialize lock\_price to 45, stock\_price to 30, and barrel\_price to 25 (prices of lock, stock, and barrel, respectively).

3. Initialize lock\_limit to 70, stock\_limit to 80, and barrel\_limit to 90 (production limits for lock, stock, and barrel, respectively).

4. nitialize commission\_level\_1 to 0.10, commission\_level\_2 to 0.15, and commission\_level\_3 to 0.20 (commission levels for different sales amounts).

5. Prompt the user to enter the number of locks sold and store it in the variable locks\_sold.

6. Prompt the user to enter the number of stocks sold and store it in the variable stocks\_sold.

7. Prompt the user to enter the number of barrels sold and store it in the variable barrels\_sold.

8. Calculate the total sales by multiplying the number of locks sold by lock\_price, the number of stocks sold by stock\_price, and the number of barrels sold by barrel\_price. Sum these values and store the result in the variable total\_sales.

9. Print "The total sales for the salesperson is: $" followed by the value of total\_sales.

10. Check the total sales amount to determine the commission:

11. If total\_sales is less than or equal to 1000, calculate the commission as total\_sales multiplied by commission\_level\_1.

Else, if total\_sales is less than or equal to 1800, calculate the commission as (1000 \* commission\_level\_1) + ((total\_sales - 1000) \* commission\_level\_2).

Else, calculate the commission as (1000 \* commission\_level\_1) + (800 \* commission\_level\_2) + ((total\_sales - 1800) \* commission\_level\_3).

Store the calculated commission in the variable commission.

12. Print "The total commission for the salesperson is: $" followed by the value of commission.

13. End

**OUTPUT:**

**EXPERIMENT NO: 05**

**THEORY:**

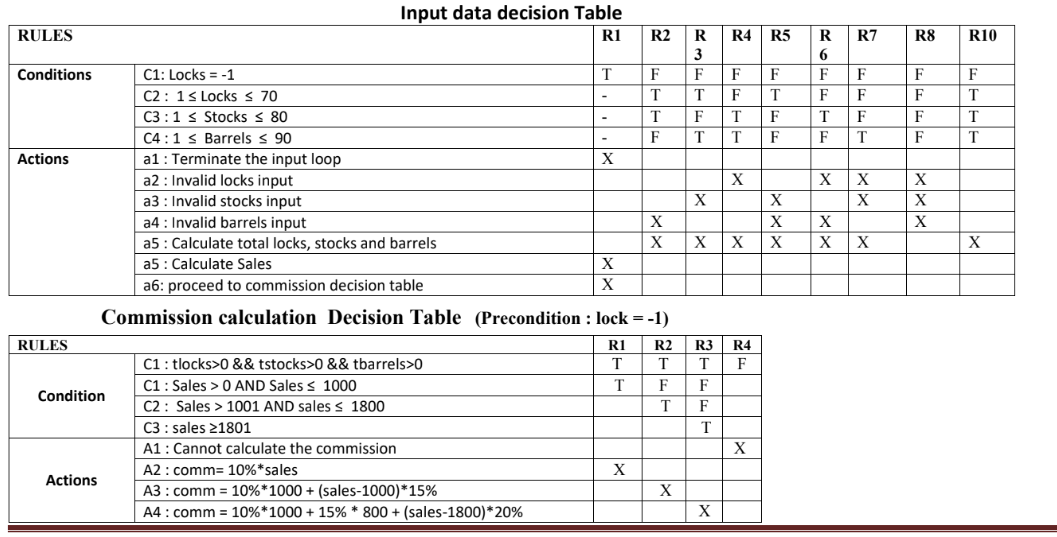
Test data : price Rs for lock - 45.0 , stock - 30.0 and barrel - 25.0

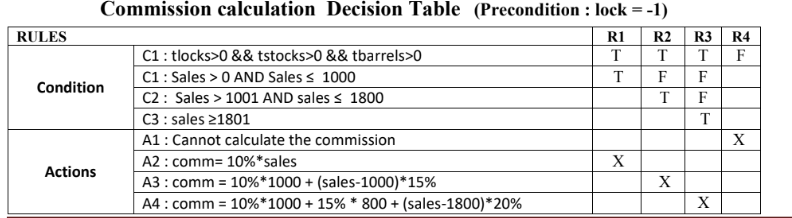
sales = total lock \* lock price + total stock \* stock price + total barrel \* barrel price

commission : 10% up to sales Rs 1000 , 15 % of the next Rs 800 and 20 % on any sales in excess of 1800

Pre-condition : lock = -1 to exit and 1< =lock < = 70 , 1<=stock <=80 and 1<=barrel<=90

Brief Description : The salesperson had to sell at least one complete rifle per month.





**ALGORITHM:**

1. Start

2. Initialize lock\_price to 45, stock\_price to 30, and barrel\_price to 25 (prices of lock, stock, and barrel, respectively).

3. Initialize lock\_limit to 70, stock\_limit to 80, and barrel\_limit to 90 (production limits for lock, stock, and barrel, respectively).

4. nitialize commission\_level\_1 to 0.10, commission\_level\_2 to 0.15, and commission\_level\_3 to 0.20 (commission levels for different sales amounts).

5. Prompt the user to enter the number of locks sold and store it in the variable locks\_sold.

6. Prompt the user to enter the number of stocks sold and store it in the variable stocks\_sold.

7. Prompt the user to enter the number of barrels sold and store it in the variable barrels\_sold.

8. Calculate the total sales by multiplying the number of locks sold by lock\_price, the number of stocks sold by stock\_price, and the number of barrels sold by barrel\_price. Sum these values and store the result in the variable total\_sales.

9. Print "The total sales for the salesperson is: $" followed by the value of total\_sales.

10. Check the total sales amount to determine the commission:

11. If total\_sales is less than or equal to 1000, calculate the commission as total\_sales multiplied by commission\_level\_1.

Else, if total\_sales is less than or equal to 1800, calculate the commission as (1000 \* commission\_level\_1) + ((total\_sales - 1000) \* commission\_level\_2).

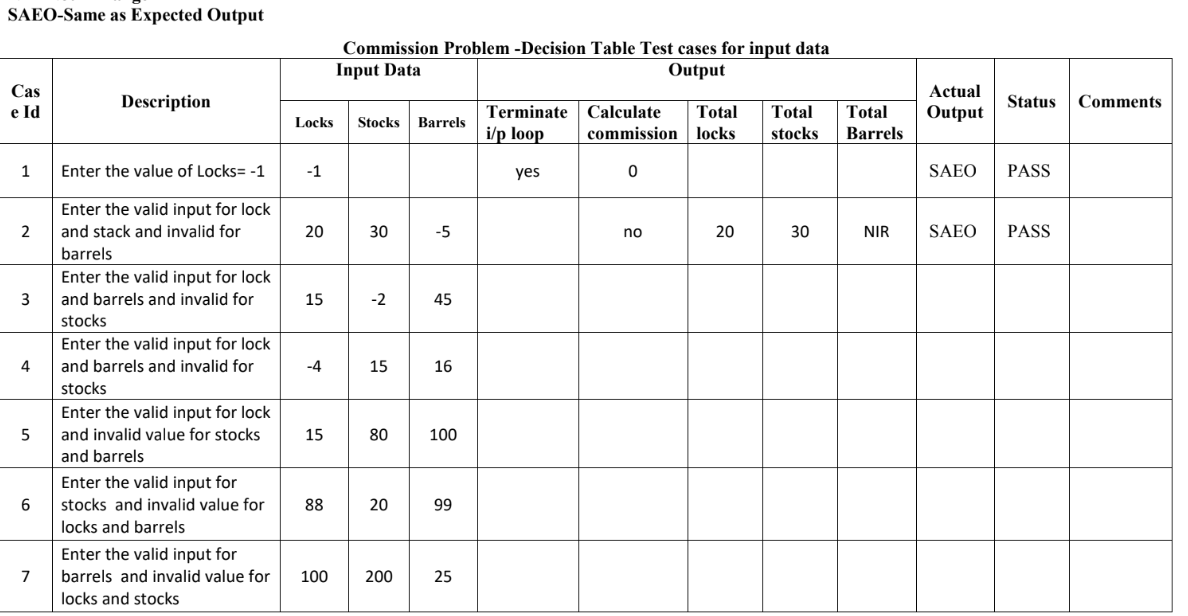
Else, calculate the commission as (1000 \* commission\_level\_1) + (800 \* commission\_level\_2) + ((total\_sales - 1800) \* commission\_level\_3).

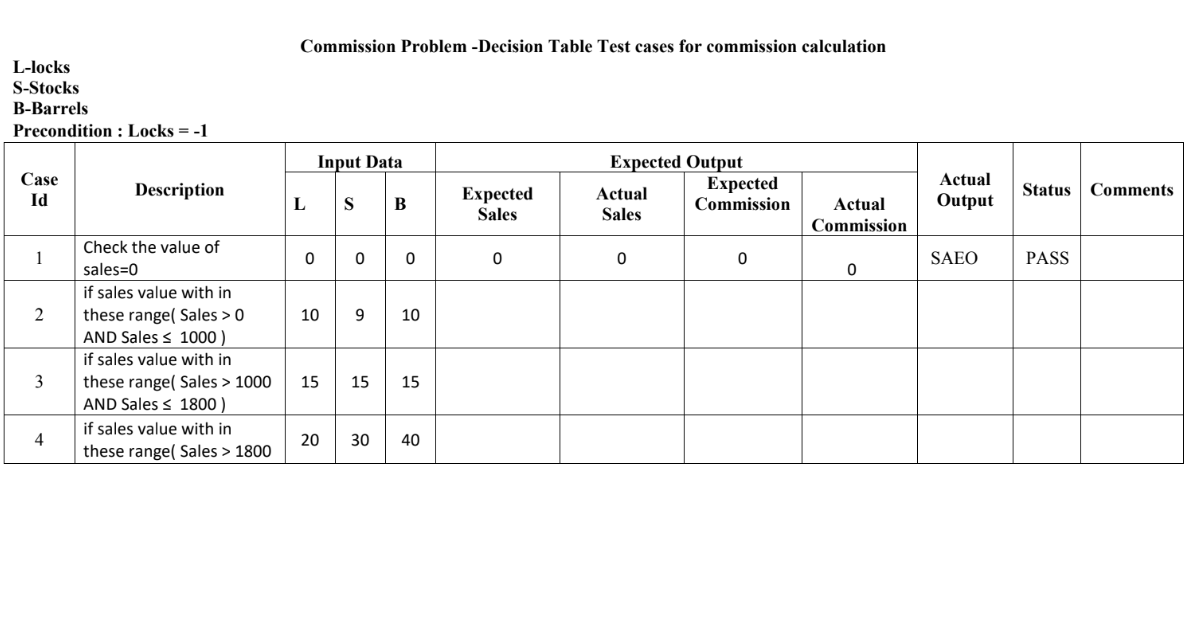
Store the calculated commission in the variable commission.

12. Print "The total commission for the salesperson is: $" followed by the value of commission.

13. End

**OUTPUT:**

****

****

**EXPERIMENT NO: 06**

**THEORY:** Binary search is a search algorithm that efficiently finds a target value within a sorted array. The basic idea is to repeatedly divide the search range in half. Here's a high-level overview of the algorithm:

Initialize: Set two pointers, low and high, to the start and end of the array, respectively.

Midpoint Calculation: Calculate the midpoint as (low + high) // 2.

Comparison: Compare the element at the midpoint with the target.

If they are equal, the target is found, and the index is returned.

If the element at the midpoint is less than the target, narrow the search to the right half.

If the element at the midpoint is greater than the target, narrow the search to the left half.

Repeat: Repeat steps 2-3 until the target is found or the search range is empty.

Binary search has a time complexity of O(log n), making it highly efficient for large datasets.

**Basis Paths in Testing:**

Basis path testing is a white-box testing method that focuses on testing the fundamental execution paths of a program. The goal is to ensure that each path through the code is tested at least once. Basis path testing is often used to design test cases for control flow structures such as loops and conditional statements.

For a program with control flow structures, the basis set is the set of all possible paths through the program. The number of basis paths can be determined by analyzing the cyclomatic complexity, a metric that measures the complexity of a program's control flow.

In the context of the binary search algorithm, the basis paths would include the different paths through the loop and conditional statements. The test cases derived from these basis paths ensure that various scenarios are covered, leading to more comprehensive testing.

**OUTPUT:**

|  |
| --- |
| Test Case Input (arr, target) Expected Output Actual output |
| 1 ([], 5) -1 (Target not found)  2 ([1, 2, 3, 4, 5, 6], 4) 3 (Index where the target is found)  3 ([1, 2, 3, 4, 5], 3) 2  4 ([1, 2, 3, 4, 5], 4) 3  5 ([1, 2, 3, 5, 6], 4) -1 (Target not found) |

**EXPERIMENT NO:07**

Algorithm:

the quicksort algorithm:

list, the pivot element, and the sorted

list.

lists using the quicksort algorithm.

and

Recursively sort the Concatenate the sorted list, and return the result.

5.

6.

element is less than or equal to the pivot, append it to the list. If it is greater than the pivot, append it to the

Iterate over the elements in the input array (excluding the pivot). If the

and **greater**.

Create two empty lists called

If the length of the input array is less than 2, return the array. Select the first element of the array as the pivot.

1.

2.

3.

4.

**less**

**less**

**greater**

**greater**

**greater**

**less**

**less**

Line numbered Code:

1. def quicksort(array):
2. if len(array) < 2:
3. return array
4. else:
5. pivot = array[0]
6. less = [i for i in array[1:] if i <= pivot]
7. greater = [i for i in array[1:] if i > pivot]
8. return quicksort(less) + [pivot] + quicksort(greater) 9.
9. print(quicksort([10, 5, 2, 3]))

**Output:**

Test cases for the quicksort algorithm using basis path testing:

|  |  |  |
| --- | --- | --- |
| **Test Case** | **Input (array)** | **Output (sorted array)** |
| **1** | [10, 5, 2, 3] | [2, 3, 5, 10] |
| **2** | [1] | [1] |
| **3** | [] | [] |
| **4** | [2, 3, 1] | [1, 2, 3] |
| **5** | [5, 5, 5] | [5, 5, 5] |

Basis path testing is a white box testing technique that involves testing all possible paths through a program's code. In the case of the quicksort algorithm, we can test the following paths:

* 1. The base case where the length of the input array is less than 2 (line 2).
  2. The case where the length of the input array is greater than or equal to 2 (line 4).

**EXPERIMENT NO: 08**

#include<stdio.h>

int main()

{

float per;

char grade;

scanf("%f",&per);

if(per>=90)

grade= 'A';

else if(per>=80 && per<90)

grade ='B';

else if(per>=70 && per<80)

grade ='C';

else if(per>=60 && per<70)

grade='D';

else grade='E';

switch(grade)

{

case 'A': printf("\nEXCELLENT"); break;

case 'B':printf("\nVery Good"); break;

case 'C' : printf("\nGood"); break;

case 'D': printf("\nAbove Average"); break;

case 'E': printf("\n Satisfactory"); break;

}

printf("\t The percentage = %f and grade is %c ",per,grade);

return 0;

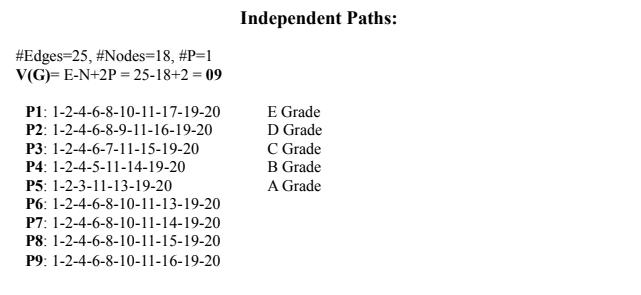
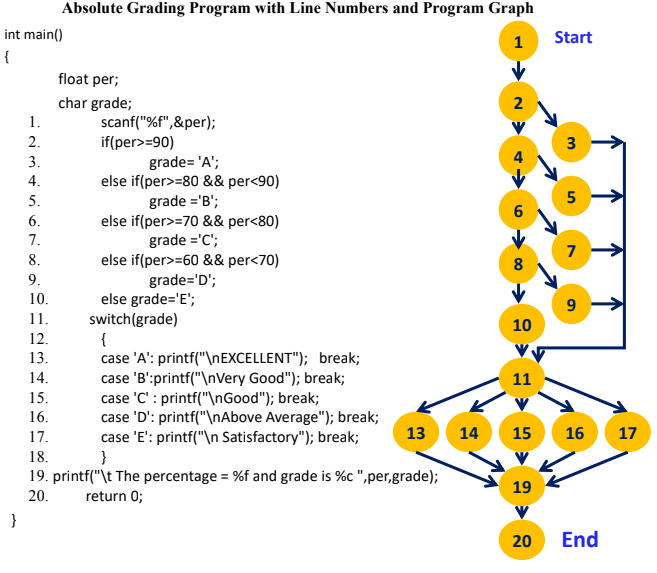
}

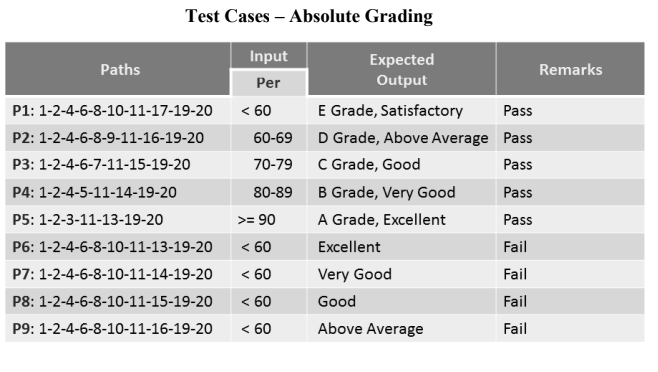
Absolute Grading Program with Line Numbers and Program Graph

Independent Paths:

#Edges=25, #Nodes=18, #P=1

V(G)= E-N+2P = 25-18+2 = 09





| **Path** | **Outcome** |
| --- | --- |
| 1 | **per >= 90** -> **Above Average** |
| 2 | **per >= 80** -> **Very Good** |
| 3 | **per >= 70** -> **Good** |
| 4 | **per >= 60** -> **Above Average** |
| 5 | **per < 0** -> `Invalid percentage... |
| 6 | **per > 100** -> `Invalid percentage... |
| 7 | **per < 60** -> **Satisfactory** |
| 8 | **0 <= per < 60** -> **Satisfactory** |

basis path test cases for this program, along with the expected outcomes:

| **Test Case** | **Percentage** | **Expected Output** |
| --- | --- | --- |
| 1 | 95 | **EXCELLENT**<br>**The percentage = 95.0 and grade is A** |
| 2 | 85 | **Very Good**<br>**The percentage = 85.0 and grade is B** |
| 3 | 75 | **Good**<br>**The percentage = 75.0 and grade is C** |
| 4 | 65 | **Above Average**<br>**The percentage = 65.0 and grade is D** |
| 5 | 55 | **Satisfactory**<br>**The percentage = 55.0 and grade is E** |
| 6 | -5 | **Invalid percentage, enter again** |
| 7 | 105 | **Invalid percentage, enter again** |
| 8 | 0 | **Satisfactory**<br>**The percentage = 0.0 and grade is E** |

These test cases cover all of the linearly independent paths through the program, and can be used to test the correctness of the program.

**EXPERIMENT NO: 09**

Pre-condition : Month 1 to 12 , DAY 1 TO 31 AND YEAR 1812 TO 2013

Valid Cases

M1 = { month ; 1 ≤ month ≤ 12 }

D1 = { day : 1 ≤ day ≤ 31 }

Y1 = { year : 1812 ≤ year ≤ 2013 }

Invalid cases

M2 = {month : month < 1}

M3 = {month : month > 12}

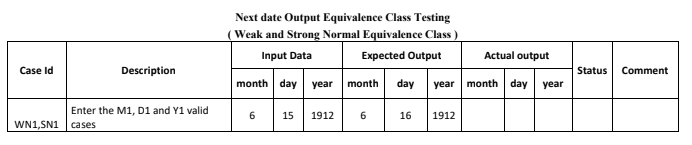
D2 = {day : day < 1}

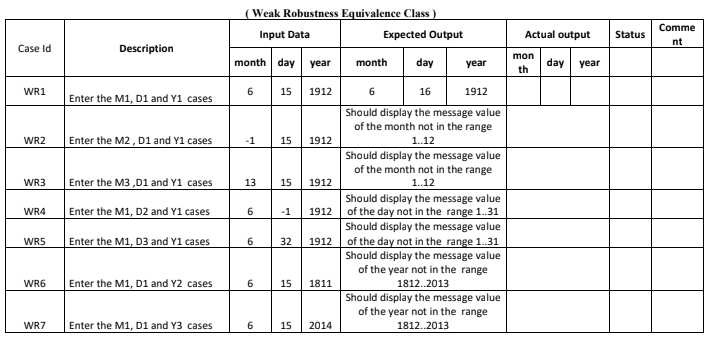
D3 = {day : day > 31}

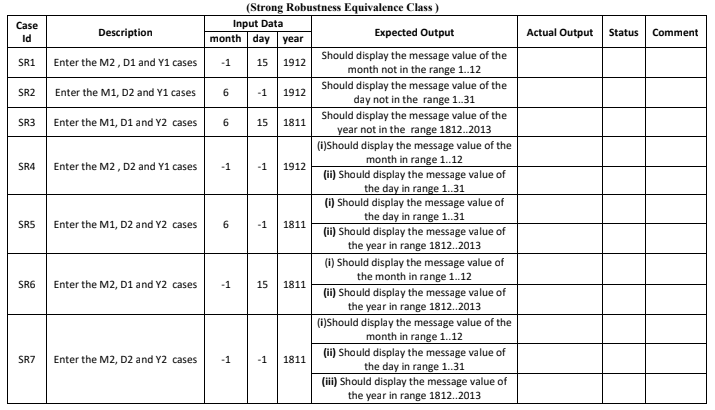
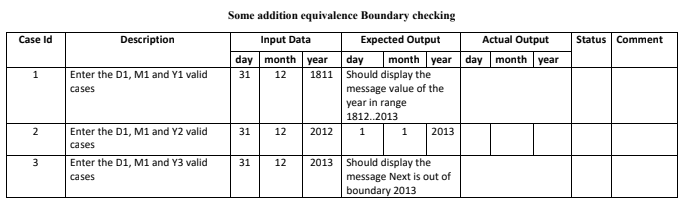
Y2 = {year : year < 1812}

Y3 = {year : year > 2013}

**OUTPUT:**





| **Test Case** | **Month** | **Date** | **Year** | **Expected Output** |
| --- | --- | --- | --- | --- |
| 1 | 1 | 1 | 2000 | (1, 2, 2000) |
| 2 | 1 | 32 | 2000 | Invalid Input |
| 3 | 13 | 31 | 2000 | Invalid Input |
| 4 | 12 | 31 | 2000 | (1, 1, 2001) |
| 5 | 12 | 31 | 1811 | Invalid Input |
| 6 | 2 | 29 | 2000 | (3, 1, 2000) |
| 7 | 2 | 29 | 1900 | (3, 1, 1900) |
| 8 | 2 | 28 | 2000 | (2, 29, 2000) |
| 9 | 2 | 30 | 2000 | Invalid Input |
| 10 | 4 | 30 | 2000 | (5, 1, 2000) |
| 11 | 6 | 30 | 2000 | (7, 1, 2000) |
| 12 | 9 | 30 | 2000 | (10, 1, 2000) |
| 13 | 11 | 30 | 2000 | (12, 1, 2000) |
| 14 | 1 | 31 | 2000 | (2, 1, 2000) |
| 15 | 3 | 31 | 2000 | (4, 1, 2000) |
| 16 | 5 | 31 | 2000 | (6, 1, 2000) |
| 17 | 7 | 31 | 2000 | (8, 1, 2000) |
| 18 | 8 | 31 | 2000 | (9, 1, 2000) |
| 19 | 10 | 31 | 2000 | (11, 1, 2000) |
| 20 | 12 | 31 | 2000 | (1, 1, 2001) |
| 21 | 4 | 31 | 2000 | Invalid Input |
| 22 | 6 | 31 | 2000 | Invalid Input |
| 23 | 9 | 31 | 2000 | Invalid Input |
| 24 | 11 | 31 | 2000 | Invalid Input |
| 25 | 2 | 0 | 2000 | Invalid Input |
| 26 | 2 | 32 | 2000 | Invalid Input |
| 27 | 0 | 29 | 2000 | Invalid Input |
| 28 | 13 | 29 | 2000 | Invalid Input |
| 29 | 2 | 29 | 1812 | (3, 1, 1812) |
| 30 | 2 | 29 | 2012 | (3, 1, 2012) |
| 31 | 2 | 29 | 2013 | Invalid Input |
| 32 | 2 | 29 | 1811 | Invalid Input |
| 33 | 2 | 29 | 2011 | (3, 1, 2011) |