1. Design a flowchart and write a Python program to calculate the Least Common Multiple (LCM) of two integers.

PROGRAM:

import math

def calculate\_lcm(a,b):

  gcd=math.gcd(a,b)

  lcm=abs(a\*b)//gcd

  return lcm

num1=int(input("Enter the first integer:"))

num2=int(input("Enter the second integer:"))

result=calculate\_lcm(num1,num2)

print(f"The Least Common Multiple(LCM) of {num1} and {num2} is:{result}")

2. Design a flowchart and write a Python program to compute the HCF of two integers.

PROGRAM:

def compute\_hcf(x, y):

  while(y):

    x, y = y, x % y

  return x

# Example usage

num1 = int(input("enter the first number"))

num2 = int(input("enter the second number"))

print("The HCF of", num1, "and", num2, "is", compute\_hcf(num1, num2))

3. Develop a program to demonstrate how to store and retrieve data using Python data structures: tuples, lists, sets, and dictionaries. Implement examples for:

• Adding and removing elements in lists.

• Accessing elements in a tuple.

• Performing set operations like union, intersection, and difference.

• Using a dictionary to store key-value pairs. Compare the advantages and limitations of each data structure.

PROGRAM:

print("1. Working with Lists") # 1. Working with Lists

fruits = ["apple", "banana", "cherry"] # Create a list

print("Initial list:", fruits)

fruits.append("orange") # Add elements to a list

print("After adding an element:", fruits)

fruits.remove("banana") # Remove elements from a list

print("After removing an element:", fruits)

print("\n")

print("2. Accessing Elements in a Tuple") # 2. Accessing Elements in a Tuple

coordinates = (10, 20, 30) # Create a tuple

print("Tuple:", coordinates)

print("First element:", coordinates[0]) # Access elements in a tuple

print("Second element:", coordinates[1])

print("Third element:", coordinates[2])

print("\n")

print("3. Performing Set Operations") # 3. Performing Set Operations

set\_a = {1, 2, 3, 4} # Create two sets

set\_b = {3, 4, 5, 6}

print("Set A:", set\_a)

print("Set B:", set\_b)

union\_set = set\_a | set\_b # Union

print("Union of A and B:", union\_set)

intersection\_set = set\_a & set\_b # Intersection

print("Intersection of A and B:", intersection\_set)

difference\_set = set\_a - set\_b # Difference

print("Difference of A and B (A - B):", difference\_set)

print("\n")

print("4. Using a Dictionary") # 4. Using a Dictionary to Store Key-Value Pairs

student\_info = {"name": "Alice", "age": 22, "major": "Computer Science"} # Create a dictionary

print("Initial Dictionary:", student\_info)

print("Name:", student\_info["name"]) # Access a value

student\_info["GPA"] = 3.8 # Add a new key-value pair

print("After adding a new key-value pair:", student\_info)

del student\_info["age"] # Remove a key-value pair

print("After removing a key-value pair:", student\_info)

print("\n")

print("Advantages and Limitations of Each Data Structure") # Advantages and Limitations of Each Data Structure

print("Lists:") # Lists

print("Advantages: Ordered, allows duplicates, dynamic sizing, and supports indexing and slicing.")

print("Limitations: Less efficient for membership checks, and no enforced uniqueness of elements.\n")

print("Tuples:") # Tuples

print("Advantages: Immutable (useful for fixed data), ordered, and supports indexing.")

print("Limitations: Cannot add or remove elements after creation, less flexible than lists.\n")

print("Sets:") # Sets

print("Advantages: Unordered, no duplicates, fast membership checks, and supports set operations (union,intersection, etc.).")

print("Limitations: Unordered (no indexing or slicing), and elements must be hashable.\n")

print("Dictionaries:") # Dictionaries

print("Advantages: Key-value storage, fast lookup by key, and flexible for a variety of data types.")

print("Limitations: Unordered (prior to Python 3.7), keys must be unique and hashable.\n")

4. Develop a program that uses set operations to demonstrate:

• Union (combining two sets of data).

• Intersection (finding common elements between sets).

• Difference (elements unique to one set). Include functionality for dynamically adding and removing items from the sets.

PROGRAM:

set\_A = {1, 2, 3, 4, 5} # Initialize two sets

set\_B = {4, 5, 6, 7, 8}

def display\_sets(): # Function to display sets

 print("Set A:", set\_A)

 print("Set B:", set\_B)

 print("\n")

def add\_item(target\_set, item): # Function to add items to a set

 target\_set.add(item)

 print(f"Item '{item}' added to the set.")

 display\_sets()

def remove\_item(target\_set, item): # Function to remove items from a set

 if item in target\_set:

 target\_set.remove(item)

 print(f"Item '{item}' removed from the set.")

 else:

 print(f"Item '{item}' not found in the set.")

 display\_sets()

def perform\_union(): # Function to perform union

 result = set\_A | set\_B

 print("Union of Set A and Set B:", result)

 print("\n")

def perform\_intersection(): # Function to perform intersection

 result = set\_A & set\_B

 print("Intersection of Set A and Set B:", result)

 print("\n")

def perform\_difference(): # Function to perform difference (Set A - Set B and Set B - Set A)

 diff\_A\_B = set\_A - set\_B

 diff\_B\_A = set\_B - set\_A

 print("Difference (Set A - Set B):", diff\_A\_B)

 print("Difference (Set B - Set A):", diff\_B\_A)

 print("\n")

print("Initial Sets:") # Main program

display\_sets()

print("Adding items to sets:") # Adding items to sets

add\_item(set\_A, 9)

add\_item(set\_B, 10)

print("Removing items from sets:") # Removing items from sets

remove\_item(set\_A, 3)

remove\_item(set\_B, 7)

print("Performing Set Operations:") # Performing set operations

perform\_union()

perform\_intersection()

perform\_difference()

5.Write a program using list comprehensions to generate lists based on certain conditions, such as:

• Generating a list of squares of even numbers between 1 and 20.

• Filtering a list of strings based on whether they start with a specific letter.

PROGRAM:

# 1. Generating a list of squares of even numbers between 1 and 20

squares\_of\_even\_numbers = [x\*\*2 for x in range(1, 21) if x % 2 == 0]

print("Squares of even numbers between 1 and 20:", squares\_of\_even\_numbers)

# 2. Filtering a list of strings based on whether they start with a specific letter

# Example list of strings

words = ["apple", "banana", "avocado", "cherry", "apricot", "blueberry", "almond"]

# Filter words that start with 'a'

filtered\_words = [word for word in words if word.startswith("a")]

print("Words starting with 'a':", filtered\_words)

filtered\_words1 = [word for word in words if word.startswith("b")]

print("Words starting with 'b':", filtered\_words1)

6. Create a program that defines several functions to perform tasks like:

• Calculating the area of a rectangle.

• Finding the maximum of a list of numbers.

• Printing personalized greetings. Demonstrate variable scope by modifying variables inside and outside functions.

PROGRAM:

global\_greeting = "Hello from the global scope!" # Global variable to demonstrate scope

def calculate\_area(length, width): # Function to calculate the area of a rectangle

 area = length \* width # Local variable inside the function

 print(f"Inside calculate\_area - Area: {area}")

 return area

def find\_max(numbers): # Function to find the maximum in a list of numbers

 max\_value = max(numbers) # Local variable inside the function

 print(f"Inside find\_max - Max Value: {max\_value}")

 return max\_value

def print\_greeting(name): # Function to print a personalized greeting

 global global\_greeting # Accessing a global variable inside a function

 print(global\_greeting) # Uses the global variable

 local\_greeting = f"Hello, {name}!" # Local variable

 print(f"Inside print\_greeting - Local Greeting: {local\_greeting}")

 global\_greeting = f"Hello, {name}! (updated globally)" # Modify the global variable

 print(f"Inside print\_greeting - Global Greeting Updated: {global\_greeting}")

# Main Program

# Outside any function, in global scope

print("Demonstration of Variable Scope\n")

# 1. Calculate the area of a rectangle

length = 5

width = 3

area\_result = calculate\_area(length, width)

print(f"Outside calculate\_area - Area Result: {area\_result}\n")

# 2. Find the maximum in a list of numbers

numbers = [10, 25, 37, 2, 89]

max\_result = find\_max(numbers)

print(f"Outside find\_max - Max Result: {max\_result}\n")

# 3. Print personalized greetings and modify global variable

name = "Siddhartha"

print\_greeting(name)

print(f"Outside print\_greeting - Global Greeting After Update: {global\_greeting}\n")

7. Develop a program using lambda functions to: Sort a list of tuples based on specific criteria. Filter a list of numbers to retain only those greater than a certain value.

PROGRAM:

# Program using lambda functions for sorting and filtering

# 1. Sort a list of tuples based on the second element in each tuple

tuples\_list = [(1, 3), (4, 1), (2, 5), (3, 2)]

print("Original list of tuples:", tuples\_list)

# Sorting tuples by the second element in each tuple

sorted\_tuples = sorted(tuples\_list, key=lambda x: x[1])

print("Tuples sorted by the second element:", sorted\_tuples)

# 2. Filter a list of numbers to retain only those greater than a certain value

numbers\_list = [10, 25, 37, 2, 89, 5, 12]

threshold = 20

print("\nOriginal list of numbers:", numbers\_list)

# Filtering numbers greater than the threshold

filtered\_numbers = list(filter(lambda x: x > threshold, numbers\_list))

print(f"Numbers greater than {threshold}:", filtered\_numbers)

8. Write a program that demonstrates variable lifetime by defining local and global variables within and outside of functions. Show how global variables retain their values across function calls.

PROGRAM:

# Global variable to demonstrate global scope and lifetime

global\_count = 0 # This variable will retain its value across function calls

# Function to demonstrate a local variable's lifetime

def increment\_local():

 # Local variable

 local\_count = 0 # This variable is recreated each time the function is called

 local\_count += 1

 print(f"Local count inside increment\_local function: {local\_count}")

# Function to demonstrate a global variable's lifetime

def increment\_global():

 global global\_count # Referencing the global variable

 global\_count += 1

 print(f"Global count inside increment\_global function: {global\_count}")

# Main program

print("Demonstrating Variable Lifetime\n")

# Calling the function with local variable multiple times

print("Calling increment\_local multiple times:")

increment\_local()

increment\_local()

increment\_local()

print("Notice that the local variable does not retain its value across calls.\n")

# Calling the function with global variable multiple times

print("Calling increment\_global multiple times:")

increment\_global()

increment\_global()

increment\_global()

print("Notice that the global variable retains its value across calls.\n")

# Checking the global variable value outside of functions

print(f"Global count outside functions: {global\_count}")

9. Develop a function for a math app that calculates the factorial of a number. Additionally, implement a recursive function to generate the nth Fibonacci number.

PROGRAM:

# Function to calculate the factorial of a number

def factorial(n):

 """Return the factorial of n."""

 if n < 0:

 return "Factorial is not defined for negative numbers."

 elif n == 0 or n == 1:

 return 1

 else:

 result = 1

 for i in range(2, n + 1):

 result \*= i

 return result

# Recursive function to calculate the nth Fibonacci number

def fibonacci(n):

 """Return the nth Fibonacci number using recursion."""

 if n <= 0:

 return "Fibonacci number is not defined for negative or zero index."

 elif n == 1:

 return 0

 elif n == 2:

 return 1

 else:

 return fibonacci(n - 1) + fibonacci(n - 2)

# Main Program to test the functions

number = 5

print(f"Factorial of {number}: {factorial(number)}")

n = 10

print(f"{n}th Fibonacci number: {fibonacci(n)}")

10. Implement a binary search algorithm to find a specific product in a sorted list of prices for a shopping app. Discuss the efficiency of this approach for large datasets.

def binary\_search(prices, target):

 """Return the index of target in prices list if found, otherwise return -1."""

 left, right = 0, len(prices) - 1

 while left <= right:

 mid = (left + right) // 2 # Calculate the middle index

 mid\_value = prices[mid]

 # Check if the mid value is equal to the target

 if mid\_value == target:

 return mid

 # If target is smaller, ignore the right half

 elif mid\_value > target:

 right = mid - 1

 # If target is larger, ignore the left half

 else:

 left = mid + 1

 return -1 # Target not found

# Main Program

prices = [5, 12, 18, 24, 32, 45, 56, 67, 78, 89, 99] # Sorted list of product prices

target\_price = 45

# Perform binary search

index = binary\_search(prices, target\_price)

# Display result

if index != -1:

 print(f"Price {target\_price} found at index {index}.")

else:

 print(f"Price {target\_price} not found in the list.")