

SUBMITTED BY SUBMITTED TO

NAMAN BANSAL DR. SUBODH KUMAR

24570074

24020570065

**ACKNOWLEDGEMENT**

I would like to take this opportunity to acknowledge everyone who has helped us in every stage of this project.

I am deeply indebted to my mathematics Professor, Dr. Subodh Kumar for his guidance and suggestions in completing this project. The completion of this project was possible under his guidance and support.

I am also very thankful to my parents and my friends who have boosted me up morally with their continuous support.

At last but not least, I am very thankful to God almighty for showering his blessings upon me.

1. Create a class SET. Create member functions to perform the following SET operations:

* Is member: check whether an element belongs to the set or not and return value as true/false.
* Powerset: list all the elements of the power set of a set .
* Subset: Check whether one set is a subset of the other or not.
* Union and Intersection of two Sets.
* Complement: Assume Universal Set as per the input elements from the user.
* Set Difference and Symmetric Difference between two sets.
* Cartesian Product of Sets.

Write a menu driven program to perform the above functions on an instance of the SET class.

2. Create a class RELATION, use Matrix notation to represent a relation. Include member functions to check if the relation is Reflexive, Symmetric, Anti-symmetric, Transitive. Using these functions check whether the given relation is: Equivalence or Partial Order relation or None

3. Write a Program that generates all the permutations of a given set of digits, with or 5 without repetition.

4. For any number n, write a program to list all the solutions of the equation x1 + x2 + x3 + ...+ xn = C, where C is a constant (C<=10) and x1 , x2 ,x3 ,...,xn are nonnegative integers, using brute force strategy.

5. Write a Program to evaluate a polynomial function. (For example store f(x) = 4n2 + 2n + 9 in an array and for a given value of n, say n = 5, compute the value of f(n)).

6. Write a Program to check if a given graph is a complete graph. Represent the graph using the Adjacency Matrix representation.

7. Write a Program to check if a given graph is a complete graph. Represent the graph using the Adjacency List representation.

8. Write a Program to accept a directed graph G and compute the in-degree and outdegree of each vertex.

**Practical 1**

CODE –

import itertools

class SET:

def \_\_init\_\_(self, elements):

self.set = list(set(elements)) # remove duplicates

def is\_member(self, element):

return element in self.set

def power\_set(self):

ps = []

for i in range(len(self.set)+1):

for combo in itertools.combinations(self.set, i):

ps.append(set(combo))

return ps

def is\_subset(self, other):

return set(self.set).issubset(set(other.set))

def union(self, other):

return list(set(self.set).union(set(other.set)))

def intersection(self, other):

return list(set(self.set).intersection(set(other.set)))

def complement(self, universal):

return list(set(universal.set) - set(self.set))

def difference(self, other):

return list(set(self.set) - set(other.set))

def symmetric\_difference(self, other):

return list(set(self.set).symmetric\_difference(set(other.set)))

def cartesian\_product(self, other):

return list(itertools.product(self.set, other.set))

# Menu-driven program

def main():

print("Enter elements of Set A (space-separated):")

A = SET(input().split())

print("Enter elements of Set B (space-separated):")

B = SET(input().split())

print("Enter elements of Universal Set (space-separated):")

U = SET(input().split())

while True:

print("\nMENU:")

print("1. Check Membership")

print("2. Power Set of A")

print("3. Check if A is subset of B")

print("4. Union of A and B")

print("5. Intersection of A and B")

print("6. Complement of A")

print("7. A - B (Set Difference)")

print("8. Symmetric Difference of A and B")

print("9. Cartesian Product of A and B")

print("0. Exit")

choice = input("Enter choice: ")

if choice == '1':

val = input("Enter element to check in A: ")

print("Present" if A.is\_member(val) else "Not Present")

elif choice == '2':

ps = A.power\_set()

print("Power Set of A:")

for s in ps:

print(s)

elif choice == '3':

print("A is subset of B" if A.is\_subset(B) else "A is not a subset of B")

elif choice == '4':

print("Union:", A.union(B))

elif choice == '5':

print("Intersection:", A.intersection(B))

elif choice == '6':

print("Complement of A:", A.complement(U))

elif choice == '7':

print("A - B:", A.difference(B))

elif choice == '8':

print("Symmetric Difference:", A.symmetric\_difference(B))

elif choice == '9':

print("Cartesian Product of A and B:")

cp = A.cartesian\_product(B)

for pair in cp:

print(pair)

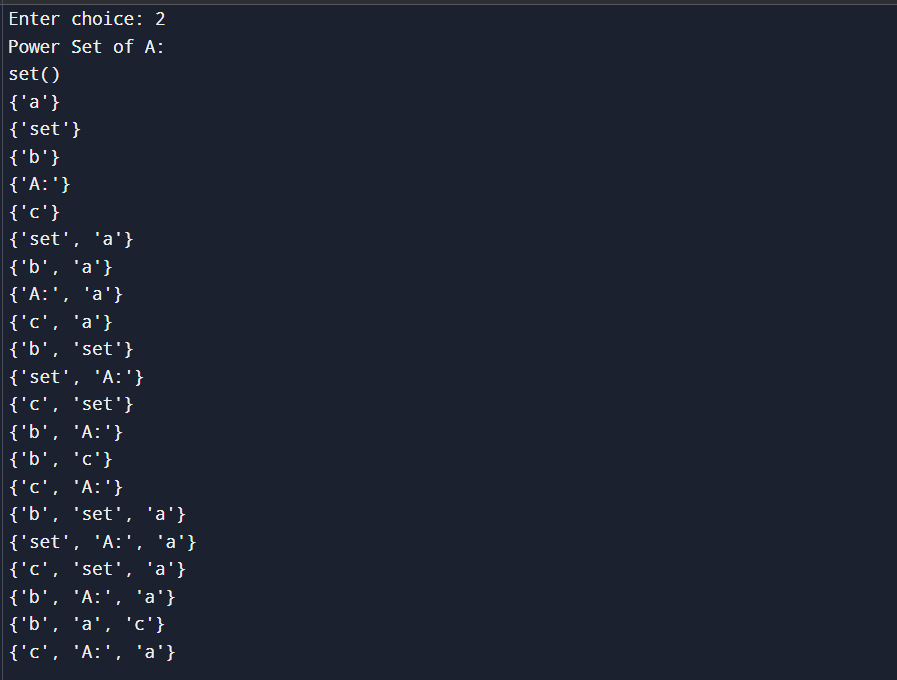
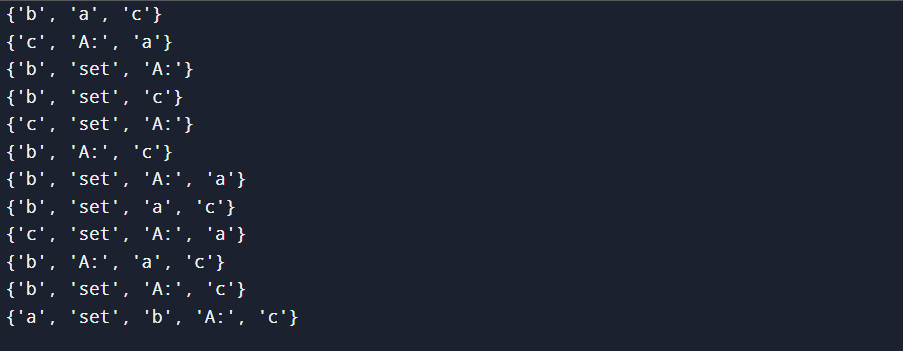
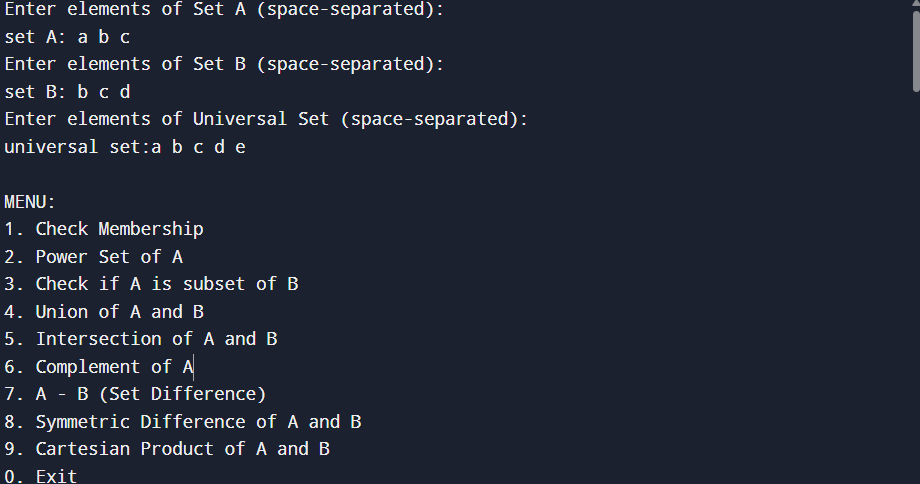
elif choice == '0':

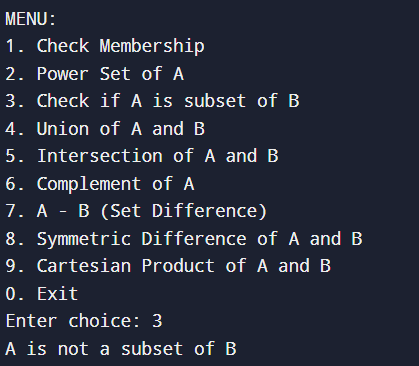
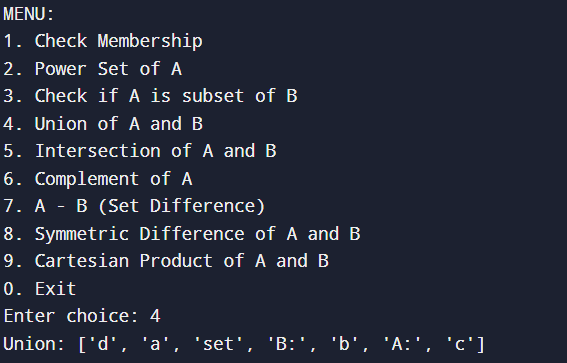
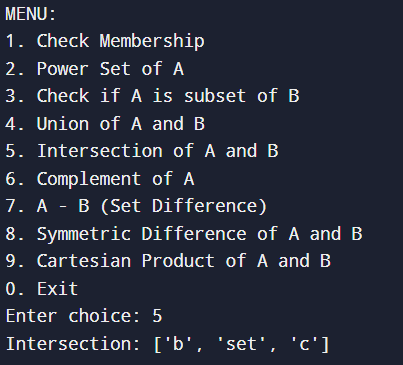
break

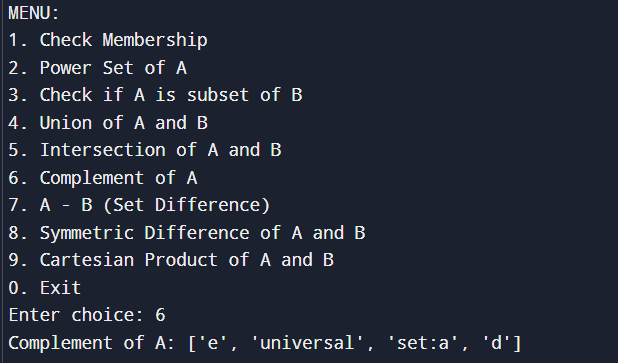
else:

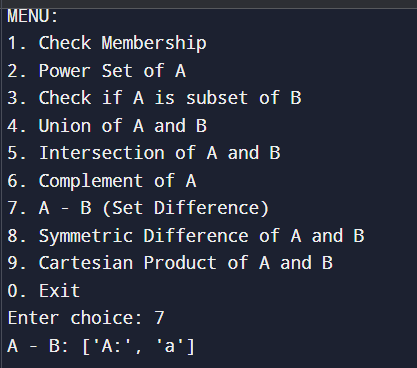
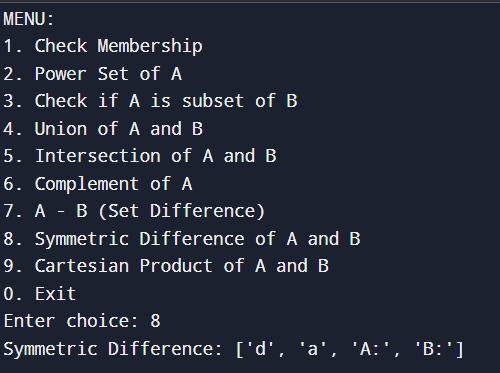
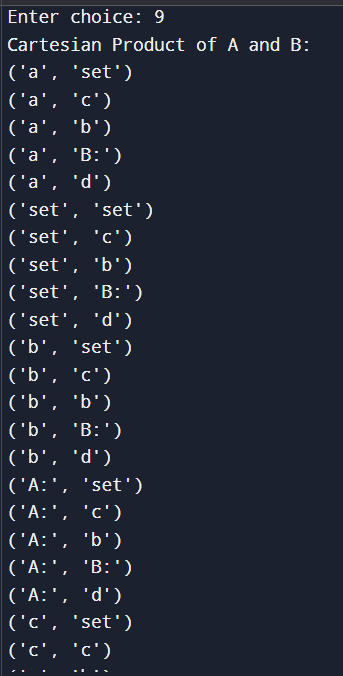
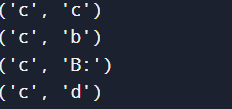
print("Invalid choice. Try again.")

main()

**OUTPUT**







**PRACTICAL 2**

CODE –

class RELATION:

def \_\_init\_\_(self, set\_elements, relation\_pairs):

self.set = list(set\_elements)

self.n = len(self.set)

self.index\_map = {self.set[i]: i for i in range(self.n)}

self.matrix = [[0]\*self.n for \_ in range(self.n)]

for a, b in relation\_pairs:

if a in self.index\_map and b in self.index\_map:

i = self.index\_map[a]

j = self.index\_map[b]

self.matrix[i][j] = 1

def print\_matrix(self):

print("Relation Matrix:")

for row in self.matrix:

print(row)

def is\_reflexive(self):

for i in range(self.n):

if self.matrix[i][i] != 1:

return False

return True

def is\_symmetric(self):

for i in range(self.n):

for j in range(self.n):

if self.matrix[i][j] != self.matrix[j][i]:

return False

return True

def is\_anti\_symmetric(self):

for i in range(self.n):

for j in range(self.n):

if i != j and self.matrix[i][j] == 1 and self.matrix[j][i] == 1:

return False

return True

def is\_transitive(self):

for i in range(self.n):

for j in range(self.n):

if self.matrix[i][j]:

for k in range(self.n):

if self.matrix[j][k] and not self.matrix[i][k]:

return False

return True

def relation\_type(self):

reflexive = self.is\_reflexive()

symmetric = self.is\_symmetric()

anti\_symmetric = self.is\_anti\_symmetric()

transitive = self.is\_transitive()

print("\nProperties:")

print("Reflexive:", reflexive)

print("Symmetric:", symmetric)

print("Anti-Symmetric:", anti\_symmetric)

print("Transitive:", transitive)

if reflexive and symmetric and transitive:

print("=> This is an Equivalence Relation.")

elif reflexive and anti\_symmetric and transitive:

print("=> This is a Partial Order Relation.")

else:

print("=> This is neither an Equivalence nor a Partial Order Relation.")

# Sample input for testing

def main():

print("Enter elements of the Set (space separated):")

elements = input().split()

print("Enter number of pairs in the relation:")

n = int(input())

print("Enter the relation pairs (space separated elements like a b):")

relation = []

for \_ in range(n):

a, b = input().split()

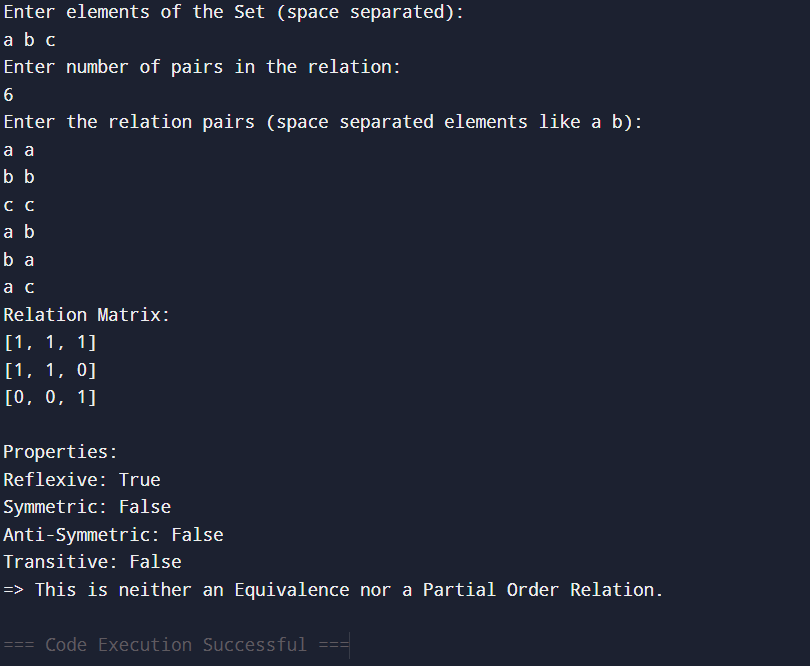
relation.append((a, b))

R = RELATION(elements, relation)

R.print\_matrix()

R.relation\_type()

main()

**OUTPUT**

**PRACTICAL 3**

CODE –

import itertools

def main():

print("Enter digits (without space, e.g., 123):")

digits = input().strip()

print("Enter length of permutation:")

r = int(input())

print("\nPermutations WITHOUT Repetition:")

perms\_without = list(itertools.permutations(digits, r))

for p in perms\_without:

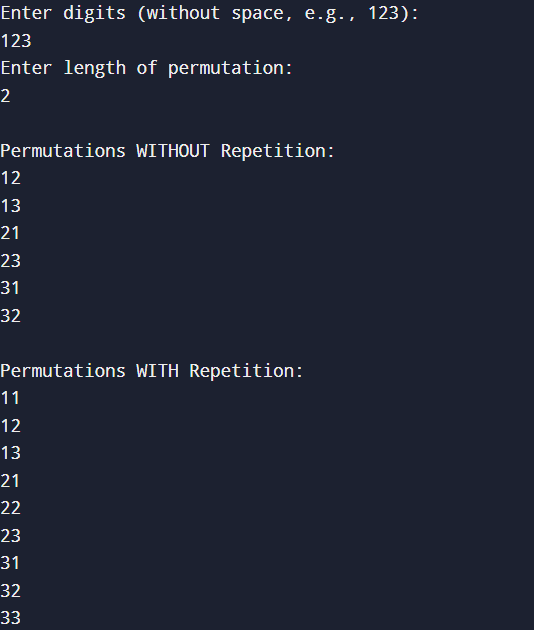
print(''.join(p))

print("\nPermutations WITH Repetition:")

perms\_with = list(itertools.product(digits, repeat=r))

for p in perms\_with:

print(''.join(p))

****main()

**OUTPUT**

**PRACTICAL 4**

CODE –

import itertools

def main():

print("Enter the number of variables (n):")

n = int(input())

print("Enter the constant value (C <= 10):")

C = int(input())

print(f"\nSolutions for x1 + x2 + ... + x{n} = {C} where all xi are non-negative integers:")

count = 0

for values in itertools.product(range(C + 1), repeat=n):

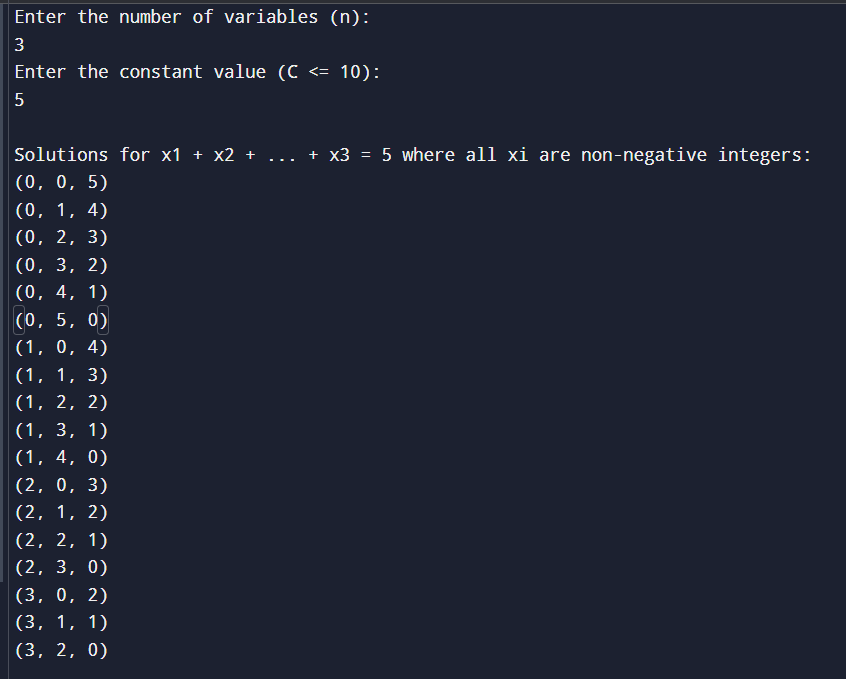
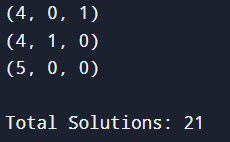
if sum(values) == C:

print(values)

count += 1

print(f"\nTotal Solutions: {count}")

main()

**OUTPUT**

**PRACTICAL 5**

CODE –

def evaluate\_polynomial(coeffs, n):

result = 0

degree = len(coeffs) - 1

for i in range(len(coeffs)):

result += coeffs[i] \* (n \*\* (degree - i))

return result

def main():

print("Enter degree of polynomial:")

d = int(input())

print(f"Enter {d + 1} coefficients (highest degree first, space separated):")

coeffs = list(map(int, input().split()))

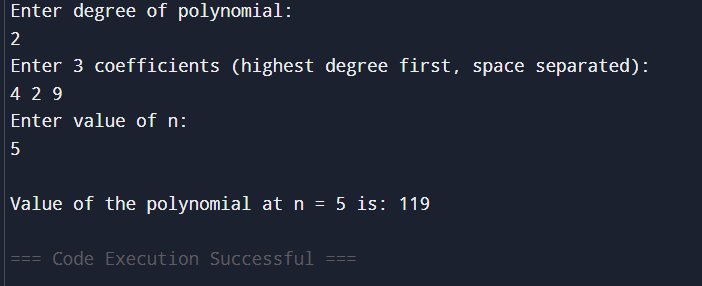
print("Enter value of n:")

n = int(input())

value = evaluate\_polynomial(coeffs, n)

print(f"\nValue of the polynomial at n = {n} is: {value}")

main()

**OUTPUT**

**PRACTICAL 6**

CODE –

def is\_complete\_graph(matrix):

n = len(matrix)

for i in range(n):

for j in range(n):

if i == j:

if matrix[i][j] != 0:

return False

else:

if matrix[i][j] != 1:

return False

return True

def main():

print("Enter number of vertices:")

n = int(input())

print("Enter the adjacency matrix row by row (space separated):")

matrix = []

for i in range(n):

row = list(map(int, input().split()))

matrix.append(row)

print("\nAdjacency Matrix:")

for row in matrix:

print(row)

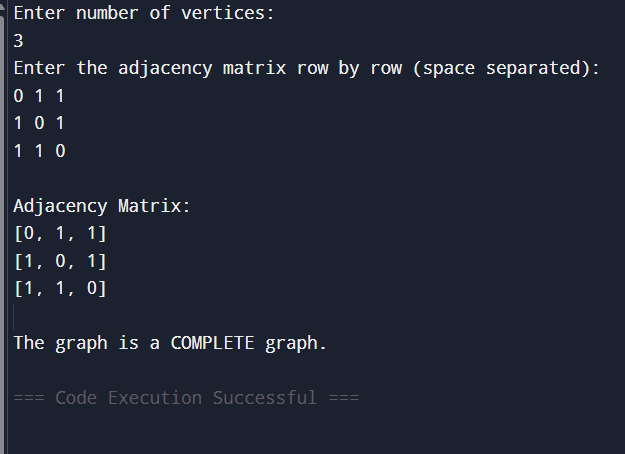
if is\_complete\_graph(matrix):

print("\nThe graph is a COMPLETE graph.")

else:

print("\nThe graph is NOT a complete graph.")

main()

**OUTPUT**

**PRACTICAL 7**

CODE –

def is\_complete\_graph(adj\_list, n):

for vertex in adj\_list:

neighbors = adj\_list[vertex]

# A complete graph must have (n - 1) neighbors (excluding itself)

if len(neighbors) != n - 1:

return False

return True

def main():

print("Enter number of vertices:")

n = int(input())

print("Enter the adjacency list:")

print("For each vertex, enter its neighbors separated by space.")

print("Enter one line per vertex, from 0 to", n-1)

adj\_list = {}

for i in range(n):

print(f"Neighbors of vertex {i}:")

neighbors = list(map(int, input().split()))

adj\_list[i] = neighbors

print("\nAdjacency List:")

for vertex in adj\_list:

print(f"{vertex} -> {adj\_list[vertex]}")

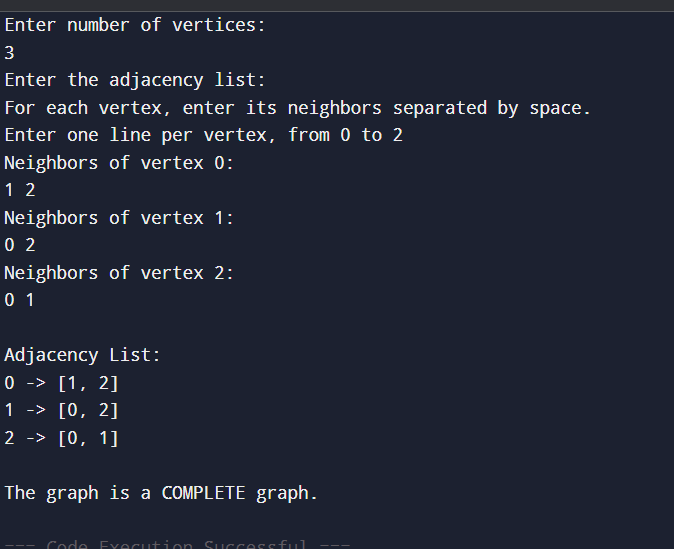
if is\_complete\_graph(adj\_list, n):

print("\nThe graph is a COMPLETE graph.")

else:

print("\nThe graph is NOT a complete graph.")

main()

**OUTPUT**

**PRACTICAL 8**

CODE –

def compute\_degrees(adj\_list, n):

in\_degree = [0] \* n

out\_degree = [0] \* n

for u in adj\_list:

out\_degree[u] = len(adj\_list[u])

for v in adj\_list[u]:

in\_degree[v] += 1

return in\_degree, out\_degree

def main():

print("Enter number of vertices:")

n = int(input())

print("Enter the adjacency list for each vertex:")

adj\_list = {}

for i in range(n):

print(f"Enter neighbors of vertex {i} (space separated):")

neighbors = list(map(int, input().split()))

adj\_list[i] = neighbors

print("\nAdjacency List:")

for v in adj\_list:

print(f"{v} -> {adj\_list[v]}")

in\_deg, out\_deg = compute\_degrees(adj\_list, n)

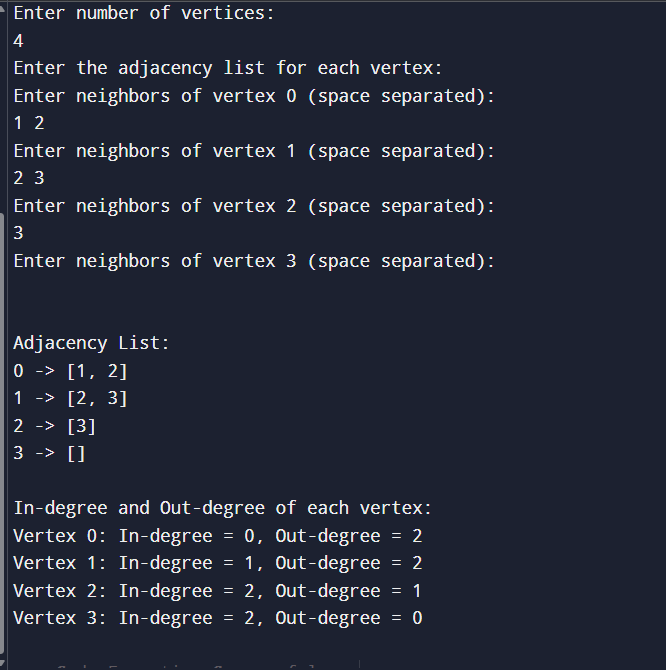
print("\nIn-degree and Out-degree of each vertex:")

for i in range(n):

print(f"Vertex {i}: In-degree = {in\_deg[i]}, Out-degree = {out\_deg[i]}")

main()

**OUTPUT**

****