**DAY-4**

1.Write a program that finds the closest pair of points in a set of 2D points using the brute force approach.

Input: • A list or array of points represented by coordinates (x, y). Points: [(1, 2), (4, 5), (7, 8), (3, 1)] Output: • The two points with the minimum distance between them.

• The minimum distance itself. Closest pair: (1, 2) - (3, 1) Minimum distance: 1.4142135623730951

**Program:-**

import math

points = [(1, 2), (4, 5), (7, 8), (3, 1)]

min\_distance = float('inf')

closest\_pair = None

for i in range(len(points)):

    for j in range(i + 1, len(points)):

        dist = math.sqrt((points[i][0] - points[j][0]) \*\* 2 + (points[i][1] - points[j][1]) \*\* 2)

        if dist < min\_distance:

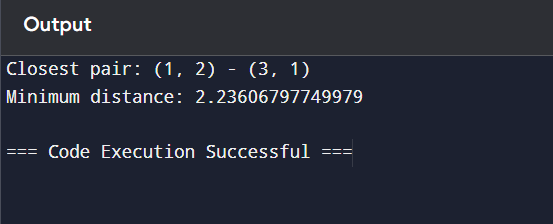
            min\_distance = dist

            closest\_pair = (points[i], points[j])

print(f"Closest pair: {closest\_pair[0]} - {closest\_pair[1]}")

print(f"Minimum distance: {min\_distance}")

**output:-**



2.Write a program to find the closest pair of points in a given set using the brute force approach. Analyze the time complexity of your implementation. Define a function to calculate the Euclidean distance between two points. Implement a function to find the closest pair of points using the brute force method. Test your program with a sample set of points and verify the correctness of your results. Analyze the time complexity of your implementation. Write a brute-force algorithm to solve the convex hull problem for the following set S of points? P1 (10,0)P2 (11,5)P3 (5, 3)P4 (9, 3.5)P5 (15, 3)P6 (12.5, 7)P7 (6, 6.5)P8 (7.5, 4.5).How do you modify your brute force algorithm to handle multiple points that are lying on the sameline?

Given points: P1 (10,0), P2 (11,5), P3 (5, 3), P4 (9, 3.5), P5 (15, 3), P6 (12.5, 7), P7 (6, 6.5), P8 (7.5, 4.5).

output: P3, P4, P6, P5, P7, P1

**program:-**

# Sample set of points

points = [(10, 0), (11, 5), (5, 3), (9, 3.5), (15, 3), (12.5, 7), (6, 6.5), (7.5, 4.5)]

# Convex hull points storage

hull\_points = []

# Cross product to determine the relative position of a point with respect to a line

for i in range(len(points)):

    for j in range(i + 1, len(points)):

        all\_on\_one\_side = True

        pos\_side = None

        # Check the position of all other points with respect to the line formed by points[i] and points[j]

        for k in range(len(points)):

            if k == i or k == j:

                continue

            # Compute orientation (cross product)

            val = (points[j][1] - points[i][1]) \* (points[k][0] - points[j][0]) - (points[j][0] - points[i][0]) \* (points[k][1] - points[j][1])

            # Check if points lie on one side of the line

            if pos\_side is None:

                pos\_side = val

            elif (val > 0 and pos\_side < 0) or (val < 0 and pos\_side > 0):

                all\_on\_one\_side = False

                break

        # If all points are on one side, the line is part of the convex hull

        if all\_on\_one\_side:

            if points[i] not in hull\_points:

                hull\_points.append(points[i])

            if points[j] not in hull\_points:

                hull\_points.append(points[j])

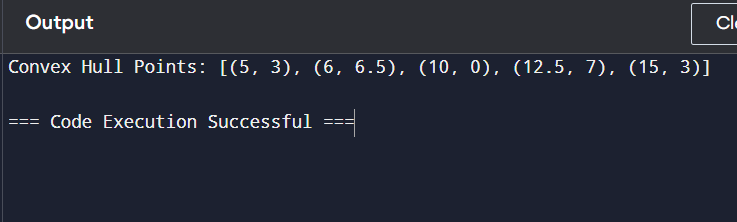
# Sort points of the hull for clean output

hull\_points.sort()

# Output the convex hull points

print("Convex Hull Points:", hull\_points)

**output:-**



3.Write a program that finds the convex hull of a set of 2D points using the brute force approach. Input: • A list or array of points represented by coordinates (x, y).

Points: [(1, 1), (4, 6), (8, 1), (0, 0), (3, 3)]

Output: • The list of points that form the convex hull in counter-clockwise order. Convex Hull: [(0, 0), (1, 1), (8, 1), (4, 6)]

**Program:-**

# Sample set of points

points = [(10, 0), (11, 5), (5, 3), (9, 3.5), (15, 3), (12.5, 7), (6, 6.5), (7.5, 4.5)]

# Convex hull points storage

hull\_points = []

# Cross product to determine the relative position of a point with respect to a line

for i in range(len(points)):

    for j in range(i + 1, len(points)):

        all\_on\_one\_side = True

        pos\_side = None

        # Check the position of all other points with respect to the line formed by points[i] and points[j]

        for k in range(len(points)):

            if k == i or k == j:

                continue

            # Compute orientation (cross product)

            val = (points[j][1] - points[i][1]) \* (points[k][0] - points[j][0]) - (points[j][0] - points[i][0]) \* (points[k][1] - points[j][1])

            # Check if points lie on one side of the line

            if pos\_side is None:

                pos\_side = val

            elif (val > 0 and pos\_side < 0) or (val < 0 and pos\_side > 0):

                all\_on\_one\_side = False

                break

        # If all points are on one side, the line is part of the convex hull

        if all\_on\_one\_side:

            if points[i] not in hull\_points:

                hull\_points.append(points[i])

            if points[j] not in hull\_points:

                hull\_points.append(points[j])

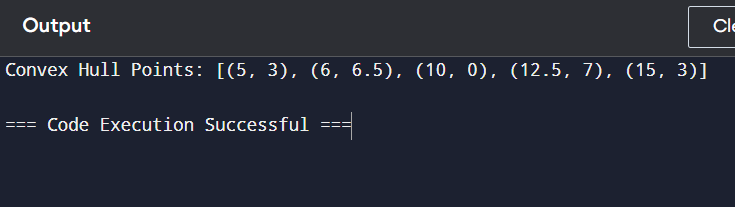
# Sort points of the hull for clean output

hull\_points.sort()

# Output the convex hull points

print("Convex Hull Points:", hull\_points)

**output:-**



4.You are given a list of cities represented by their coordinates. Develop a program that utilizes exhaustive search to solve the TSP. The program should: 1. Define a function distance(city1, city2) to calculate the distance between two cities (e.g., Euclidean distance). 2. Implement a function tsp(cities) that takes a list of cities as input and performs the following:

o Generate all possible permutations of the cities (excluding the starting city) using itertools.permutations.

o For each permutation (representing a potential route): ♣ Calculate the total distance traveled by iterating through the path and summing the distances between consecutive cities. ♣ Keep track of the shortest distance encountered and the corresponding path.

o Return the minimum distance and the shortest path (including the starting city at the beginning and end). 3. Include test cases with different city configurations to demonstrate the program's functionality. Print the shortest distance and the corresponding path for each test case.

**Program:-**

import itertools

import math

# List of cities represented by their coordinates

cities = [(0, 0), (1, 2), (3, 5), (6, 7)]

# Initialize variables to store the minimum distance and the shortest path

min\_distance = float('inf')

shortest\_path = []

# Calculate the Euclidean distance between two cities

def distance(city1, city2):

    return math.sqrt((city1[0] - city2[0]) \*\* 2 + (city1[1] - city2[1]) \*\* 2)

# The starting city (we assume the first city as the starting city)

start\_city = cities[0]

# Generate all permutations of the cities, excluding the starting city

permutations = itertools.permutations(cities[1:])

# Iterate through each permutation (potential route)

for perm in permutations:

    # Create a full path by including the starting city at the beginning and end

    path = [start\_city] + list(perm) + [start\_city]

    # Calculate the total distance of this path

    total\_distance = 0

    for i in range(len(path) - 1):

        total\_distance += distance(path[i], path[i + 1])

    # Update the minimum distance and shortest path if a shorter distance is found

    if total\_distance < min\_distance:

        min\_distance = total\_distance

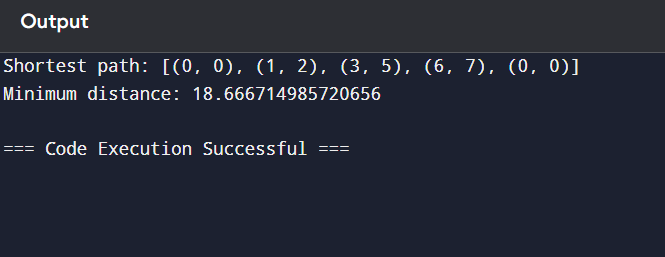
        shortest\_path = path

# Output the minimum distance and the shortest path

print("Shortest path:", shortest\_path)

print("Minimum distance:", min\_distance)

**output:-**



5.You are given a cost matrix where each element cost[i][j] represents the cost of assigning worker i to task j. Develop a program that utilizes exhaustive search to solve the assignment problem. The program should Define a function total\_cost(assignment, cost\_matrix) that takes an assignment (list representing worker-task pairings) and the cost matrix as input. It iterates through the assignment and calculates the total cost by summing the corresponding costs from the cost matrix Implement a function assignment\_problem(cost\_matrix) that takes the cost matrix as input and performs the following Generate all possible permutations of worker indices (excluding repetitions).

**Program:-**

import itertools

cost\_matrix = [

    [9, 2, 7, 8],

    [6, 4, 3, 7],

    [5, 8, 1, 8],

    [7, 6, 9, 4]

]

# Initialize variables to store the minimum cost and best assignment

min\_cost = float('inf')

best\_assignment = []

# Generate all permutations of workers (since worker i is assigned to task perm[i])

n = len(cost\_matrix)

permutations = itertools.permutations(range(n))

# Iterate through each possible worker-task assignment (permutation)

for perm in permutations:

    total\_cost = 0

    # Calculate the total cost of this assignment

    for worker in range(n):

        task = perm[worker]

        total\_cost += cost\_matrix[worker][task]

    # Update the minimum cost and best assignment if a lower cost is found

    if total\_cost < min\_cost:

        min\_cost = total\_cost

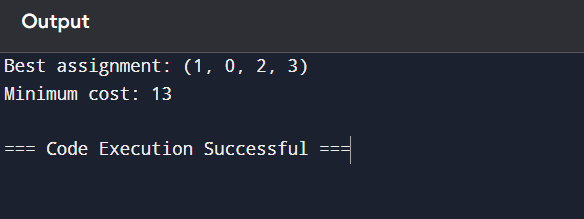
        best\_assignment = perm

# Output the minimum cost and the corresponding assignment

print("Best assignment:", best\_assignment)

print("Minimum cost:", min\_cost)

**output:-**



6.You are given a list of items with their weights and values. Develop a program that utilizes exhaustive search to solve the 0-1 Knapsack Problem. The program should:

1. Define a function total\_value(items, values) that takes a list of selected items (represented by their indices) and the value list as input. It iterates through the selected items and calculates the total value by summing the corresponding values from the value list.

2. Define a function is\_feasible(items, weights, capacity) that takes a list of selected items (represented by their indices), the weight list, and the knapsack capacity as input. It checks if the total weight of the selected items exceeds the capacity.

**Program:-**

import itertools

weights = [2, 3, 1]

values = [4, 5, 3]

capacity = 4

max\_value = 0

best\_combination = []

n = len(weights)

for r in range(n + 1):

    for combination in itertools.combinations(range(n), r):

        total\_weight = 0

        total\_value = 0

        for item in combination:

            total\_weight += weights[item]

            total\_value += values[item]

        if total\_weight <= capacity:

            if total\_value > max\_value:

                max\_value = total\_value

                best\_combination = combination

print("Best combination of items:", best\_combination)

print("Maximum value:", max\_value)

**output:-**

