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

**Code:**

import math

def brute\_force\_closest\_pair(points):

min\_distance = float('inf')

pair = None

for i in range(len(points)):

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

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

if distance < min\_distance:

min\_distance = distance

pair = (points[i], points[j])

return pair

points = [(1,2), (1, 1), (2, 2), (3, 3)]

closest\_pair = brute\_force\_closest\_pair(points)

print("Closest Pair:", closest\_pair)

**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?**

import math

def euclidean\_distance(point1, point2):

return math.sqrt((point1[0] - point2[0])\*\*2 + (point1[1] - point2[1])\*\*2)

def closest\_pair\_brute\_force(points):

min\_distance = float('inf')

closest\_pair = None

for i in range(len(points)):

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

distance = euclidean\_distance(points[i], points[j])

if distance < min\_distance:

min\_distance = distance

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

return closest\_pair

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

closest\_pair = closest\_pair\_brute\_force(sample\_points)

print("Closest Pair of Points:", closest\_pair)

**3.Write a program that finds the convex hull of a set of 2D points using the brute force approach.**

from itertools import combinations

def orientation(p, q, r):

val = (q[1] - p[1]) \* (r[0] - q[0]) - (q[0] - p[0]) \* (r[1] - q[1])

if val == 0:

return 0

return 1 if val > 0 else -1

def convex\_hull(points):

n = len(points)

if n < 3:

return points

hull = []

for p, q, r in combinations(points, 3):

if orientation(p, q, r) == -1:

hull.append(p)

hull.append(q)

hull.append(r)

return list(set(hull))

points = [(0, 3), (2, 2), (1, 1), (2, 1), (3, 0), (0, 0), (3, 3)]

convex\_points = convex\_hull(points)

print(convex\_points)

**4.You are given a list of cities represented by their coordinates. Develop a program that utilizes exhaustive search to solve the TSP.**

import itertools

def tsp(cities):

min\_distance = float('inf')

best\_path = None

for path in itertools.permutations(cities):

distance = 0

for i in range(len(path) - 1):

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

distance += calculate\_distance(path[-1], path[0])

if distance < min\_distance:

min\_distance = distance

best\_path = path

return best\_path, min\_distance

def calculate\_distance(city1, city2):

return ((city1[0] - city2[0])\*\*2 + (city1[1] - city2[1])\*\*2)\*\*0.5

cities = [(1, 2),(4,5),(7,1),(3,6)]

best\_path, min\_distance = tsp(cities)

print("Best Path:", best\_path)

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

**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).**

import itertools

def total\_cost(assignment, cost\_matrix):

total = 0

for i, j in enumerate(assignment):

total += cost\_matrix[i][j]

return total

def assignment\_problem(cost\_matrix):

num\_workers = len(cost\_matrix)

num\_tasks = len(cost\_matrix[0])

min\_cost = float('inf')

best\_assignment = None

for perm in itertools.permutations(range(num\_tasks), num\_workers):

cost = total\_cost(perm, cost\_matrix)

if cost < min\_cost:

min\_cost = cost

best\_assignment = perm

return best\_assignment

cost\_matrix :

[[3,10,7],

[8, 5,12],

[4,6,9]]

best\_assignment = assignment\_problem(cost\_matrix)

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

**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.**

def knapsack\_exhaustive\_search(values, weights, capacity):

n = len(values)

max\_value = 0

for i in range(2\*\*n):

bin\_str = bin(i)[2:].zfill(n)

current\_value = sum([values[j] for j in range(n) if bin\_str[j] == '1'])

current\_weight = sum([weights[j] for j in range(n) if bin\_str[j] == '1'])

if current\_weight <= capacity and current\_value > max\_value:

max\_value = current\_value

return max\_value

values = [60, 100, 120]

weights = [10, 20, 30]

capacity = 50

print(knapsack\_exhaustive\_search(values, weights, capacity)) # Output: 220