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

CODE:

import math

def distance(p1, p2):

return math.sqrt((p1[0] - p2[0]) \*\* 2 + (p1[1] - p2[1]) \*\* 2)

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

min\_distance = float('inf')

closest\_points = None

for i in range(len(points)):

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

dist = distance(points[i], points[j])

if dist < min\_distance:

min\_distance = dist

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

return closest\_points, min\_distance

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

closest\_points, min\_dist = closest\_pair\_brute\_force(points)

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

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

OUTPUT:

Closest pair: (1, 2) - (3, 1)

Minimum distance: 1.4142135623730951

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

CODE:

def distance(p1, p2):

return math.sqrt((p1[0] - p2[0]) \*\* 2 + (p1[1] - p2[1]) \*\* 2)

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

min\_distance = float('inf')

closest\_points = None

for i in range(len(points)):

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

dist = distance(points[i], points[j])

if dist < min\_distance:

min\_distance = dist

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

return closest\_points, min\_distance

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

closest\_points, min\_dist = closest\_pair\_brute\_force(points)

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

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

OUTPUT:

Closest pair: (5, 3) - (7.5, 4.5)

Minimum distance: 2.5

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)]

CODE:

def cross\_product(o, a, b):

return (a[0] - o[0]) \* (b[1] - o[1]) - (a[1] - o[1]) \* (b[0] - o[0])

def convex\_hull(points):

points = sorted(points)

if len(points) <= 1:

return points

lower, upper = [], []

for p in points:

while len(lower) >= 2 and cross\_product(lower[-2], lower[-1], p) <= 0:

lower.pop()

lower.append(p)

for p in reversed(points):

while len(upper) >= 2 and cross\_product(upper[-2], upper[-1], p) <= 0:

upper.pop()

upper.append(p)

return lower[:-1] + upper[:-1]

points = [(1, 1), (4, 6), (8, 1), (0, 0), (3, 3)]

hull = convex\_hull(points)

print(f"Convex Hull: {hull}")

OUTPUT:

Convex Hull: [(0, 0), (1, 1), (8, 1), (4, 6)]

4) 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. Shortest Distance: 7.0710678118654755

Shortest Path: [(1, 2), (4, 5), (7, 1), (3, 6), (1, 2)]

CODE:

import itertools

def distance(city1, city2):

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

def tsp(cities):

n = len(cities)

min\_path = None

min\_dist = float('inf')

start = cities[0]

for perm in itertools.permutations(cities[1:]):

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

dist = sum(distance(path[i], path[i + 1]) for i in range(n))

if dist < min\_dist:

min\_dist = dist

min\_path = path

return min\_dist, min\_path

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

min\_dist, min\_path = tsp(cities)

print(f"Shortest Distance: {min\_dist}")

print(f"Shortest Path: {min\_path}")

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).

Test Cases:

Input

1. Simple Case: Cost Matrix:

[[3, 10, 7],

[8, 5, 12],

[4, 6, 9]]

2. More Complex Case: Cost Matrix:

[[15, 9, 4],

[8, 7, 18],

[6, 12, 11]]

Output:

Test Case 1:

Optimal Assignment: [(worker 1, task 2), (worker 2, task 1), (worker 3, task 3)]

Total Cost: 19

Test Case 2:

Optimal Assignment: [(worker 1, task 3), (worker 2, task 1), (worker 3, task 2)]

Total Cost: 24

CODE:

import itertools

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

return sum(cost\_matrix[i][assignment[i]] for i in range(len(assignment)))

def assignment\_problem(cost\_matrix):

num\_workers = len(cost\_matrix)

workers = range(num\_workers) # Indices of workers

min\_cost = float('inf')

best\_assignment = None

for perm in itertools.permutations(workers):

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

if current\_cost < min\_cost:

min\_cost = current\_cost

best\_assignment = perm

optimal\_assignment = [(f"worker {i+1}", f"task {best\_assignment[i]+1}") for i in range(num\_workers)]

return optimal\_assignment, min\_cost

cost\_matrix1 = [

[3, 10, 7],

[8, 5, 12],

[4, 6, 9]

]

cost\_matrix2 = [

[15, 9, 4],

[8, 7, 18],

[6, 12, 11]

]

assignment1, cost1 = assignment\_problem(cost\_matrix1)

print(f"Test Case 1:\nOptimal Assignment: {assignment1}\nTotal Cost: {cost1}")

assignment2, cost2 = assignment\_problem(cost\_matrix2)

print(f"Test Case 2:\nOptimal Assignment: {assignment2}\nTotal Cost: {cost2}")

OUTPUT:

Test Case 1:

Optimal Assignment: [('worker 1', 'task 3'), ('worker 2', 'task 2'), ('worker 3', 'task 1')]

Total Cost: 19

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.

Test Cases:

1. Simple Case:

 Items: 3 (represented by indices 0, 1, 2)

 Weights: [2, 3, 1]

 Values: [4, 5, 3]

 Capacity: 4

2. More Complex Case:

 Items: 4 (represented by indices 0, 1, 2, 3)

 Weights: [1, 2, 3, 4]

 Values: [2, 4, 6, 3]

 Capacity: 6

Output:

Test Case 1:

Optimal Selection: [0, 2] (Items with indices 0 and 2)

Total Value: 7

Test Case 2:

Optimal Selection: [0, 1, 2] (Items with indices 0, 1, and 2)

Total Value: 10

CODE:

import itertools

def total\_value(items, values):

return sum(values[i] for i in items)

def is\_feasible(items, weights, capacity):

return sum(weights[i] for i in items) <= capacity

def knapsack\_problem(weights, values, capacity):

num\_items = len(weights)

best\_value = 0

best\_selection = []

for r in range(num\_items + 1):

for combination in itertools.combinations(range(num\_items), r):

if is\_feasible(combination, weights, capacity):

current\_value = total\_value(combination, values)

if current\_value > best\_value:

best\_value = current\_value

best\_selection = combination

return list(best\_selection), best\_value

weights1 = [2, 3, 1]

values1 = [4, 5, 3]

capacity1 = 4

weights2 = [1, 2, 3, 4]

values2 = [2, 4, 6, 3]

capacity2 = 6

selection1, value1 = knapsack\_problem(weights1, values1, capacity1)

print(f"Test Case 1:\nOptimal Selection: {selection1} (Items with indices {selection1})\nTotal Value: {value1}")

selection2, value2 = knapsack\_problem(weights2, values2, capacity2)

print(f"Test Case 2:\nOptimal Selection: {selection2} (Items with indices {selection2})\nTotal Value: {value2}")

OUTPUT:

Test Case 1:

Optimal Selection: [0, 2] (Items with indices [0, 2])

Total Value: 7