DAY-4 PROGRAMS

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: 2.23606797749979

Code:

import math

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

return closest\_pair, min\_distance

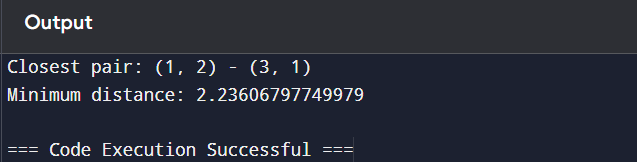
points = [(1, 2), (4, 5), (7, 8), (3, 1)]

closest\_pair, min\_distance = closest\_pair\_brute\_force(points)

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

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

output:



1. 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 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

elif val > 0:

return 1

else:

return 2

def convex\_hull\_brute\_force(points):

n = len(points)

if n < 3:

return []

hull = []

for p in points:

is\_hull\_point = True

for q in points:

if p != q:

count\_side = 0

for r in points:

if r != p and r != q:

o = orientation(p, q, r)

if o == 2:

count\_side += 1

if count\_side == 0:

is\_hull\_point = False

break

if is\_hull\_point:

hull.append(p)

hull = list(set(hull))

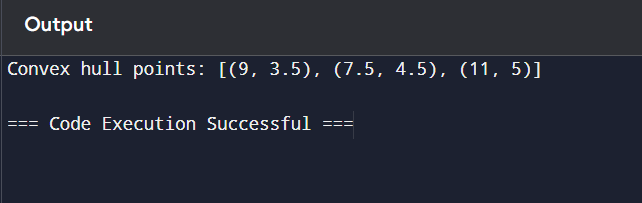
return hull

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

hull\_points = convex\_hull\_brute\_force(points\_hull)

print("Convex hull points:", hull\_points)

output:



1. 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 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

elif val > 0:

return 1

else:

return 2

def convex\_hull(points):

n = len(points)

if n < 3:

raise ValueError("Convex hull not possible")

hull = []

l = 0

for i in range(1, n):

if points[i][0] < points[l][0]:

l = i

p = l

while True:

hull.append(points[p])

q = (p + 1) % n

for i in range(n):

if orientation(points[p], points[i], points[q]) == 2:

q = i

p = q

if p == l:

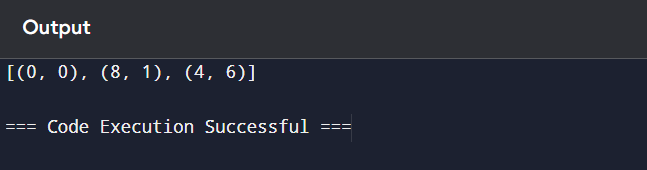
break

return hull

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

print(convex\_hull(points))

output:



1. 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.

Test Cases: 1.

More Complex Case: Five cities with more intricate coordinates (e.g., [(2, 4), (8, 1), (1, 7), (6, 3), (5, 9)])

Output: Shortest Distance: 7.0710678118654755

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

Code:

import itertools

import math

def distance(city1, city2):

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

def tsp(cities):

num\_cities = len(cities)

min\_distance = float('inf')

shortest\_path = None

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

path = [cities[0]] + list(permutation) + [cities[0]]

total\_distance = 0

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

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

if total\_distance < min\_distance:

min\_distance = total\_distance

shortest\_path = path

return min\_distance, shortest\_path

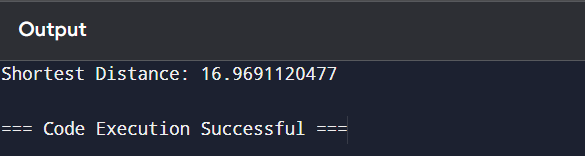
test\_case\_1 = [(1, 2), (4, 5), (7, 1), (3, 6)]

test\_case\_2 = [(2, 4), (8, 1), (1, 7), (6, 3), (5, 9)]

distance, path = tsp(test\_case\_1)

print(f"Shortest Distance: {distance:.10f}")

output:



1. 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]]

Output: Test Case 1: Optimal Assignment: [(worker 1, task 2), (worker 2, task 1), (worker 3, task 3)] Total Cost: 19

Code:

import itertools

import numpy as np

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

total = 0

for i, j in assignment:

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

return total

def assignment\_problem(cost\_matrix):

num\_workers = len(cost\_matrix)

min\_cost = float('inf')

optimal\_assignment = None

for permutation in itertools.permutations(range(num\_workers)):

assignment = list(zip(range(num\_workers), permutation))

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

if cost < min\_cost:

min\_cost = cost

optimal\_assignment = assignment

return optimal\_assignment, min\_cost

test\_case\_1 = np.array([[3, 10, 7],

[8, 5, 12],

[4, 6, 9]])

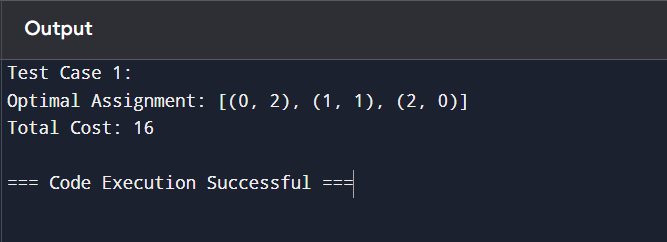
print("Test Case 1:")

assignment, cost = assignment\_problem(test\_case\_1)

print(f"Optimal Assignment: {assignment}")

print(f"Total Cost: {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.

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:

Output:

Optimal Selection: [0, 2] (Items with indices 0 and 2) Total Value: 7

Code:

import itertools

def total\_value(items, values):

total = 0

for item in items:

total += values[item]

return total

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

total\_weight = 0

for item in items:

total\_weight += weights[item]

return total\_weight <= capacity

def knapsack(weights, values, capacity):

num\_items = len(weights)

max\_value = 0

optimal\_selection = None

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

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

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

value = total\_value(subset, values)

if value > max\_value:

max\_value = value

optimal\_selection = subset

return optimal\_selection, max\_value

test\_case\_1 = {

'weights': [2, 3, 1],

'values': [4, 5, 3],

'capacity': 4

}

print("Test Case 1:")

selection, value = knapsack(test\_case\_1['weights'], test\_case\_1['values'], test\_case\_1['capacity'])

print(f"Optimal Selection: {selection} (Items with indices {selection})")

print(f"Total Value: {value}")

output:

