Day-04

PRACTICE QUESTIONS

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 closest_pair_brute_force(points):
   return min(((p1, p2) for i, p1 in enumerate(points) for p2 in points[i+1:]),
           key=lambda pair: math.dist(*pair))
points = [(1, 2), (4, 5), (7, 8), (3, 1)]
pair = closest_pair_brute_force(points)
print(f"Closest pair: {pair[0]} - {pair[1]}")
print(f"Minimum distance: {math.dist(*pair)}")
```

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?

Code:

```
import math
def euclidean_distance(p1, p2):
   return math.sqrt((p1[0] - p2[0]) ** 2 + (p1[1] - p2[1]) ** 2)
def closest_pair(points):
   min_distance = float('inf')
   pair = ()
   for i in range(len(points)):
      for j in range(i + 1, len(points)):
         dist = euclidean_distance(points[i], points[j])
         if dist < min_distance:
            min_distance = dist
            pair = (points[i], points[j])
   return pair, min_distance
def convex_hull(points):
   hull = []
```

```
    for p in points:
        while len(hull) >= 2 and cross_product(hull[-2], hull[-1], p) <= 0:
            hull.pop()
        hull.append(p)
    return hull
def cross_product(o, a, b):
    return (a[0] - o[0]) * (b[1] - o[1]) - (a[1] - o[1]) * (b[0] - o[0])
points = [(10, 0), (11, 5), (5, 3), (9, 3.5), (15, 3), (12.5, 7), (6, 6.5), (7.5, 4.5)]
closest = closest_pair(points)
hull = convex_hull(points)
print("Closest Pair:", closest)
print("Convex Hull:", hull)
```

Output: Closest Pair: (((9, 3.5), (7.5, 4.5)), 1.8027756377319946)

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

Code:

```
def is_left_turn(p1, p2, p3):
    return (p2[0] - p1[0]) * (p3[1] - p1[1]) - (p3[0] - p1[0]) * (p2[1] - p1[1]) > 0

def convex_hull(points):
    hull = []
    for i, p1 in enumerate(points):
        for j, p2 in enumerate(points):
            if i == j:
                continue
            if all(is_left_turn(p1, p2, p3) for k, p3 in enumerate(points) if k != i and k != j):
                hull.extend([p1, p2])
    return sorted(set(hull), key=lambda p: (p[0], p[1]))
points = [(1, 1), (4, 6), (8, 1), (0, 0), (3, 3)]
hull_points = convex_hull(points)

print("Convex Hull:", hull_points)
```

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

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.

Code:

```
import itertools, math
 def distance(city1, city2):
   return math.dist(city1, city2)
def tsp(cities):
   start, other_cities = cities[0], cities[1:]
   min_route = min(itertools.permutations(other_cities),
              key=lambda perm: sum(distance(a, b) for a, b in zip([start] + list(perm), list(perm)
+ [start])))
   return sum(distance(a, b) for a, b in zip([start] + list(min_route), list(min_route) + [start])),
[start] + list(min_route) + [start]
test_cases = [
   [(1, 2), (4, 5), (7, 1), (3, 6)],
   [(2, 4), (8, 1), (1, 7), (6, 3), (5, 9)]
]
 for i, cities in enumerate(test_cases):
   min_distance, best_route = tsp(cities)
   print(f"Test Case {i+1}:\nShortest Distance: {min_distance}\nBest Route: {best_route}\n")
```

Output: Test Case 1:
Shortest Distance: 18.48528137423857
Best Route: [(1, 2), (7, 1), (4, 5), (3, 6), (1, 2)]

Test Case 2:
Shortest Distance: 22.511064607415383
Best Route: [(2, 4), (6, 3), (8, 1), (1, 7), (5, 9), (2, 4)]

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).

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):
   return min((total_cost(perm, cost_matrix), perm) for perm in
itertools.permutations(range(len(cost_matrix))))
cost_matrix = [
   [9, 2, 7, 8],
   [6, 4, 3, 7],
   [5, 8, 1, 8],
   [7, 6, 9, 4]
]
min_cost, best_assignment = assignment_problem(cost_matrix)
print(f"Minimum Cost: {min_cost}, Best Assignment: {best_assignment}")
```

Output: Minimum Cost: 13, Best Assignment: (1, 2, 0, 3)

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

Code:

```
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(weights, values, capacity):
```

```
    from itertools import combinations
    n = len(weights)
    best_value = 0
    for r in range(n + 1):
        for items in combinations(range(n), r):
            if is_feasible(items, weights, capacity):
                best_value = max(best_value, total_value(items, values))
    return best_value
weights = [2, 3, 1]
values = [4, 5, 3]
capacity = 4
print(knapsack(weights, values, capacity))
```

Output: 7