DESIGN AND ANALYSIS OF ALGORITHMS

NAME : MALREDDY.P

REGISTER NO: 192372015

DAY-4 PROGRAMMES

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 calculate_distance(point1, point2):
   return math.sqrt((point1[0] - point2[0])**2 + (point1[1] - point2[1])**2)
def closest_pair_of_points(points):
   min_distance = float('inf')
   closest_pair = None
   for i in range(len(points)):
      for j in range(i + 1, len(points)):
         distance = calculate_distance(points[i], points[j])
         if distance < min_distance:
            min_distance = distance
            closest_pair = (points[i], points[j])
   return closest_pair, min_distance
points = []
n = int(input("Enter the number of points: "))
for _ in range(n):
   x, y = map(int, input("Enter the coordinates (x y): ").split())
   points.append((x, y))
closest_pair, min_distance = closest_pair_of_points(points)
print(f"Closest pair: {closest_pair[0]} - {closest_pair[1]} Minimum distance:
{min_distance}")
```

sample output:

```
Enter the number of points: 4
Enter the coordinates (x y): 1 2
Enter the coordinates (x y): 4 5
Enter the coordinates (x y): 7 8
Enter the coordinates (x y): 3 1
Closest pair: (1, 2) - (3, 1) Minimum distance: 2.23606797749979
```

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:

```
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)
```

sample output:

```
Closest Pair of Points: ((9, 3.5), (7.5, 4.5))
```

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:

```
import itertools
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 i in range(n):
      for j in range(i+1, n):
         if all(orientation(points[i], points[j], points[k]) >= 0 for k in range(n) if k != i
and k != j):
            hull.append(points[i])
            hull.append(points[j])

   return list(set(hull))
points = [(1, 1), (4, 6), (8, 1), (0, 0), (3, 3)]
convex_hull_points = convex_hull(points)
print("Convex Hull Points:", convex_hull_points)
```

sample output:

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

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.**

- **Return the minimum distance and the shortest path (including the starting city at the beginning and end).**

- **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. Simple Case: Four cities with basic coordinates (e.g., [(1, 2), (4, 5), (7, 1), (3, 6)])
  2. More Complex Case: Five cities with more intricate coordinates (e.g., [(2, 4), (8, 1), (1, 7), (6, 3), (5, 9)])
  Output: Test Case 1: Shortest Distance: 7.0710678118654755 Shortest Path: [(1, 2), (4, 5), (7, 1), (3, 6), (1, 2)]
  Test Case 2: Shortest Distance: 14.142135623730951 Shortest Path: [(2, 4), (1, 7), (6, 3), (5, 9), (8, 1), (2, 4)]

Code:

```
import itertools
def distance(city1, city2):
    return ((city1[0] - city2[0])**2 + (city1[1] - city2[1])**2) ** 0.5
def tsp(cities):
    min_distance = float('inf')
    shortest_path = None
    for perm in itertools.permutations(cities[1:]):
        total_distance = 0
        path = [cities[0]] + list(perm) + [cities[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
cities = [(0, 0), (1, 2), (3, 1), (5, 3)]
min_dist, shortest_route = tsp(cities)
print("Minimum Distance:", min_dist)
print("Shortest Route:", shortest_route)
```

sample output:

```
Minimum Distance: 12.349878388032021
Shortest Route: [(0, 0), (1, 2), (5, 3), (3, 1), (0, 0)]
```

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):
    total = 0
    for worker, task in assignment:
        total += cost_matrix[worker][task]
    return total
def assignment_problem(cost_matrix):
    workers = range(len(cost_matrix))
    min_cost = float('inf')
    optimal_assignment = None
    for perm in itertools.permutations(workers):
        assignment = list(zip(perm, range(len(cost_matrix))))
        cost = total_cost(assignment, cost_matrix)
        if cost < min_cost:
            min_cost = cost
            optimal_assignment = assignment
    return optimal_assignment, min_cost
cost_matrix = [[3, 10, 7],
               [8, 5, 12],
               [4, 6, 9]]
optimal_assignment, total_cost = assignment_problem(cost_matrix)
print("Optimal Assignment:", [(f"worker {worker + 1}", f"task {task + 1}") for worker, task in optimal_assignment])
print("Total Cost:", total_cost)
```

sample output:

```
Optimal Assignment: [('worker 3', 'task 1'), ('worker 2', 'task 2'), ('worker 1', 'task 3')]
Total Cost: 16
```

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:1

Code:

```
from itertools import combinations

def total_value(selected_items, values):

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

def is_feasible(selected_items, weights, capacity):

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

def knapsack_exhaustive_search(weights, values, capacity):

   n = len(weights)

   max_value = 0

   optimal_selection = []

   for r in range(n + 1):

      for selected_items in combinations(range(n), r):

         if is_feasible(selected_items, weights, capacity):

            current_value = total_value(selected_items, values)

            if current_value > max_value:

               max_value = current_value

               optimal_selection = selected_items

   return optimal_selection, max_value

weights = list(map(int, input("Enter the weights of items: ").split()))

values = list(map(int, input("Enter the values of items: ").split()))

capacity = int(input("Enter the capacity of the knapsack: "))
```

```
optimal_selection, max_value = knapsack_exhaustive_search(weights, values, capacity)

print(f"Optimal Selection: {list(optimal_selection)}")

print(f"Total Value: {max_value}")
```

sample output:

```
Enter the weights of items: 2 3 1

Enter the values of items: 4 5 3

Enter the capacity of the knapsack: 4

Optimal Selection: [1, 2]

Total Value: 8
```