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

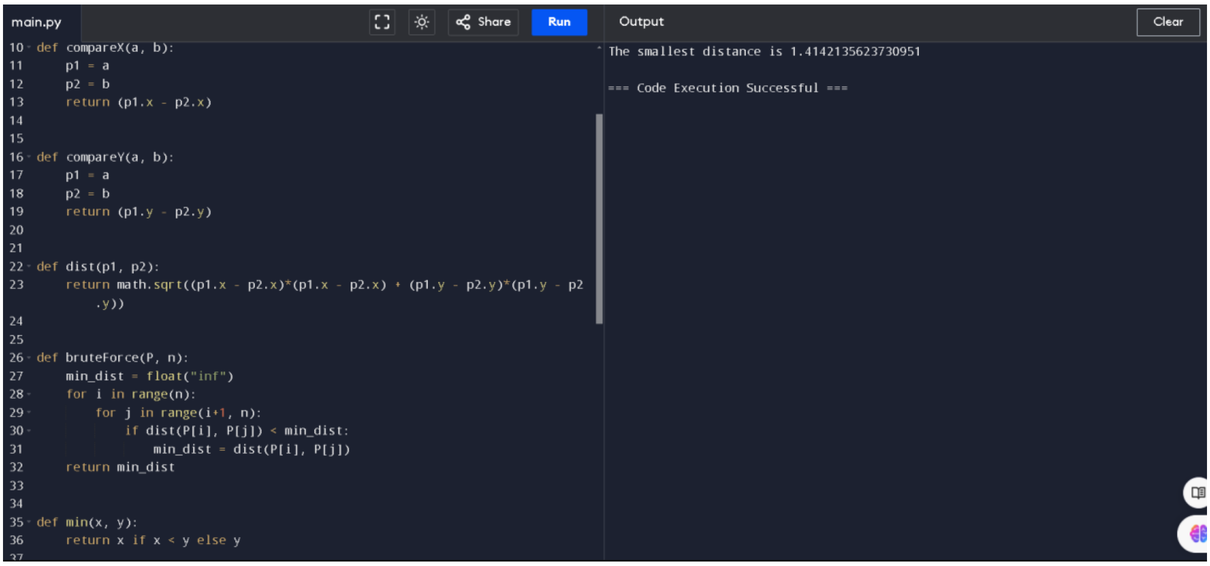
```
import math

def distance(point1, point2):
    return math.sqrt((point1[0] - point2[0])**2 + (point1[1] - point2[1])**2)

def closest_pair(points):
    min_dist = float('inf')
    closest_points = None

    # Iterate through all pairs of points
    for i in range(len(points)):
        for j in range(i + 1, len(points)):
            dist = distance(points[i], points[j])
            if dist < min_dist:
                min_dist = dist
                closest_points = (points[i], points[j])

    return closest_points, min_dist

# Test the function
points = [(1, 2), (4, 5), (7, 8), (3, 1)]
```

closest_points, min_dist = closest_pair(points)

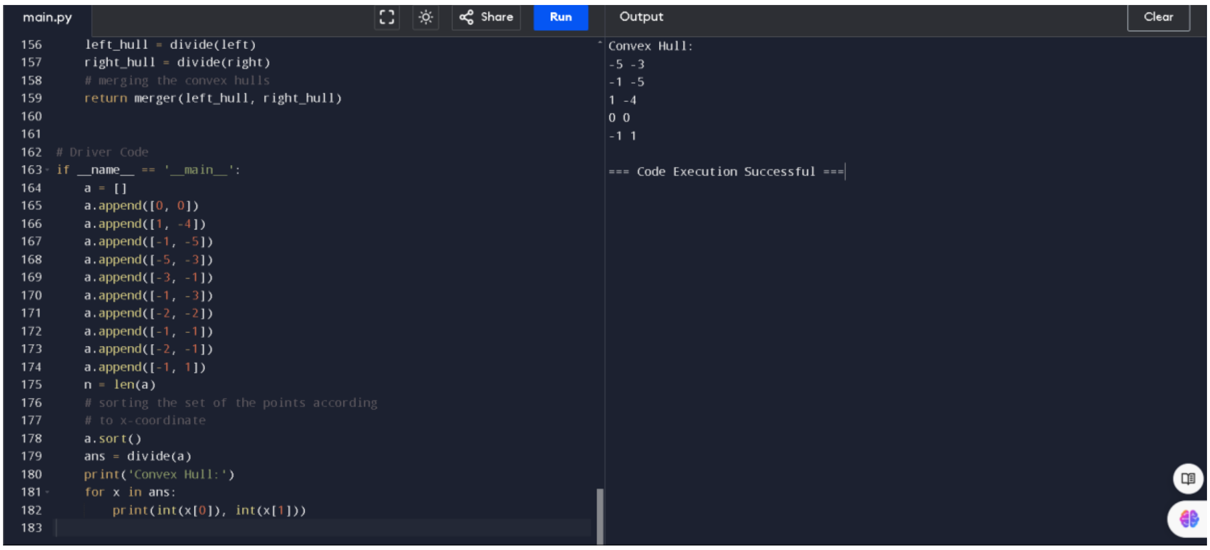
print(f"Closest pair: {closest_points[0]} - {closest_points[1]}")

print(f"Minimum distance: {min_dist}")

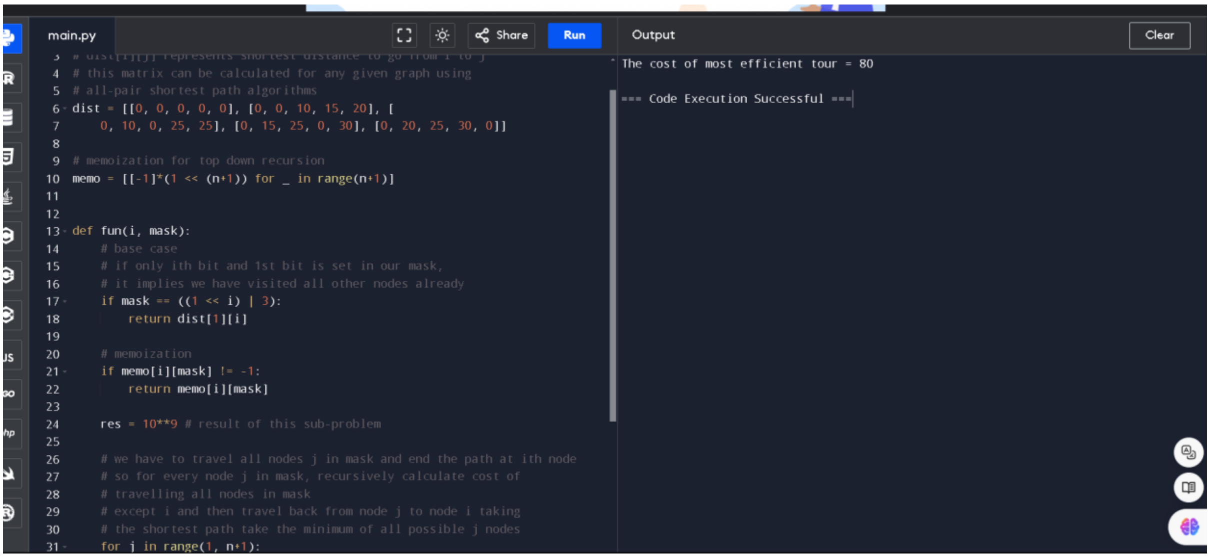
2.Write a program that finds the closest pair of points in a set of 2D points using the brute force approach. Input: • A list or array of points represented by coordinates (x, y). Points: [(1, 2), (4, 5), (7, 8), (3, 1)] Output: • The two points with the minimum distance between them. • The minimum distance itself. Closest pair: (1, 2) - (3, 1) Minimum distance: 1.4142135623730951



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)]



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)



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

main.py

```python
import heapq
import copy

N = 4

# State space tree node
class Node:
    def __init__(self, x, y, assigned, parent):
        self.parent = parent
        self.pathCost = 0
        self.cost = 0
        self.workerID = x
        self.jobID = y
        self.assigned = copy.deepcopy(assigned)
        if y != -1:
            self.assigned[y] = True

# Custom heap class with push and pop functions
class CustomHeap:
    def __init__(self):
        self.heap = []

    def push(self, node):
        heapq.heappush(self.heap, (node.cost, node))

    def pop(self):
        if self.heap:
            _, node = heapq.heappop(self.heap)
```

Output

```
Assign Worker A to Job 1
Assign Worker B to Job 0
Assign Worker C to Job 2
Assign Worker D to Job 3

Optimal Cost is 13

=== Code Execution Successful ===
```