

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

```
main.py  Run  Output  Clear
10 def compareX(a, b):
11     p1 = a
12     p2 = b
13     return (p1.x - p2.x)
14
15
16 def compareY(a, b):
17     p1 = a
18     p2 = b
19     return (p1.y - p2.y)
20
21
22 def dist(p1, p2):
23     return math.sqrt((p1.x - p2.x)*(p1.x - p2.x) + (p1.y - p2.y)*(p1.y - p2.y))
24
25
26 def bruteForce(P, n):
27     min_dist = float("inf")
28     for i in range(n):
29         for j in range(i+1, n):
30             if dist(P[i], P[j]) < min_dist:
31                 min_dist = dist(P[i], P[j])
32     return min_dist
33
34
35 def min(x, y):
36     return x if x < y else y
37
```

```
The smallest distance is 1.4142135623730951
=== Code Execution Successful ===
```

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)]

```
main.py  Run  Output  Clear
156 left_hull = divide(left)
157 right_hull = divide(right)
158 # merging the convex hulls
159 return merger(left_hull, right_hull)
160
161
162 # Driver Code
163 if __name__ == '__main__':
164     a = []
165     a.append([0, 0])
166     a.append([1, -4])
167     a.append([-1, -5])
168     a.append([-5, -3])
169     a.append([-3, -1])
170     a.append([-1, -3])
171     a.append([-2, -2])
172     a.append([-1, -1])
173     a.append([-2, -1])
174     a.append([-1, 1])
175     n = len(a)
176     # sorting the set of the points according
177     # to x-coordinate
178     a.sort()
179     ans = divide(a)
180     print('Convex Hull:')
181     for x in ans:
182         print(int(x[0]), int(x[1]))
183
```

```
Convex Hull:
-5 -3
-1 -5
1 -4
0 0
-1 1
=== Code Execution Successful ===
```

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)

```

main.py
3 # dist[i][j] represents shortest distance to go from i to j
4 # this matrix can be calculated for any given graph using
5 # all-pair shortest path algorithms
6 dist = [[0, 0, 0, 0, 0], [0, 0, 10, 15, 20], [
7     0, 10, 0, 25, 25], [0, 15, 25, 0, 30], [0, 20, 25, 30, 0]]
8
9 # memoization for top down recursion
10 memo = [[-1]*(1 << (n+1)) for _ in range(n+1)]
11
12
13 def fun(i, mask):
14     # base case
15     # if only ith bit and 1st bit is set in our mask,
16     # it implies we have visited all other nodes already
17     if mask == ((1 << i) | 3):
18         return dist[1][i]
19
20     # memoization
21     if memo[i][mask] != -1:
22         return memo[i][mask]
23
24     res = 10**9 # result of this sub-problem
25
26     # we have to travel all nodes j in mask and end the path at ith node
27     # so for every node j in mask, recursively calculate cost of
28     # travelling all nodes in mask
29     # except i and then travel back from node j to node i taking
30     # the shortest path take the minimum of all possible j nodes
31     for j in range(1, n+1):

```

Output

The cost of most efficient tour = 80

=== Code Execution Successful ===

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  [Icons] Share Run Output Clear

1 import heapq
2 import copy
3
4 N = 4
5
6 # State space tree node
7 class Node:
8     def __init__(self, x, y, assigned, parent):
9         self.parent = parent
10        self.pathCost = 0
11        self.cost = 0
12        self.workerID = x
13        self.jobID = y
14        self.assigned = copy.deepcopy(assigned)
15        if y != -1:
16            self.assigned[y] = True
17
18 # Custom heap class with push and pop functions
19 class CustomHeap:
20     def __init__(self):
21         self.heap = []
22
23     def push(self, node):
24         heapq.heappush(self.heap, (node.cost, node))
25
26     def pop(self):
27         if self.heap:
28             _, node = heapq.heappop(self.heap)
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
```

```
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 ===
```