26. Write a program that finds the closest pair of points in a set of 2D points using the bruteforce 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.

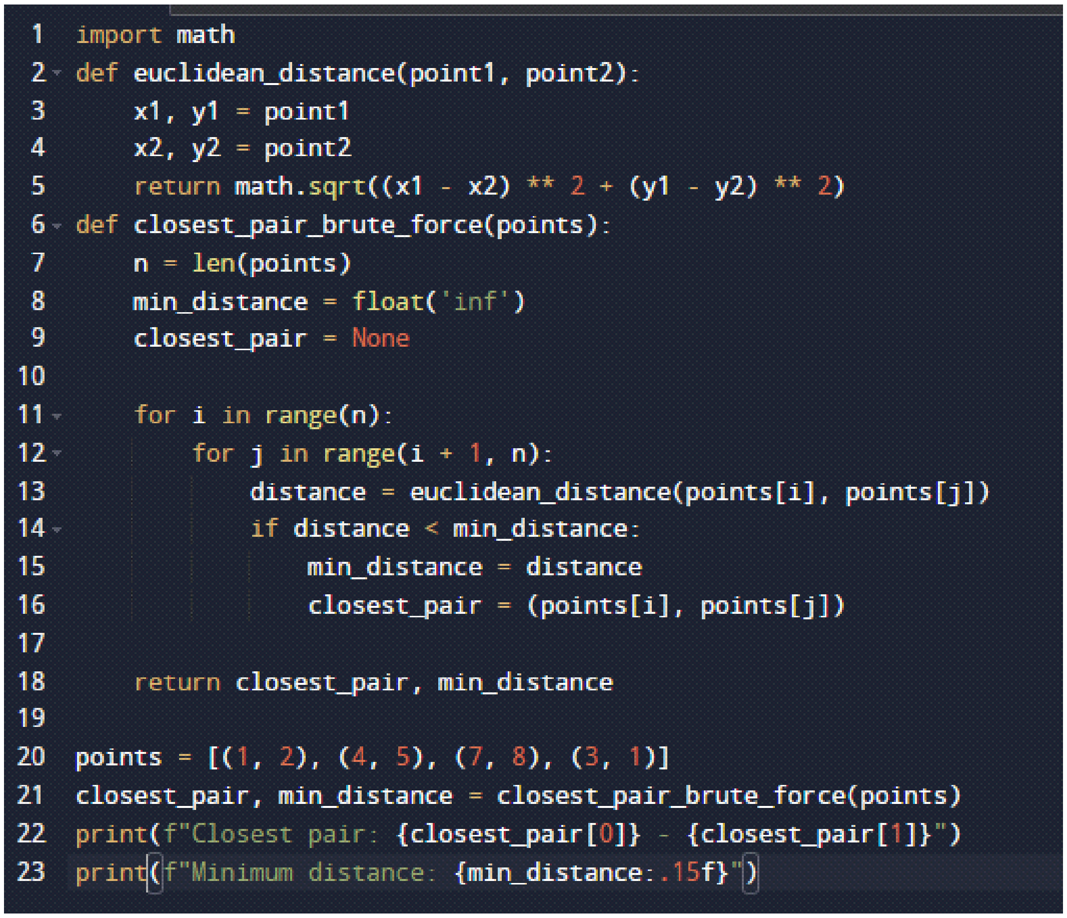
Closest pair: (1, 2) - (3, 1) Minimum distance: 1.4142135623730951

Aim : To find the two points in a 2D plane that are closest to each other, and calculate the minimumdistance between them.

Algorithm :

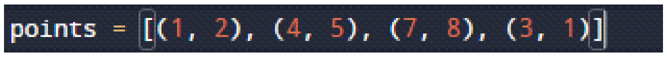
step1:Iterate through each pair of points in the list.
step2:Calculate the Euclidean distance between each pair of
points.step3:Keep track of the pair with the minimum distance.

Program:



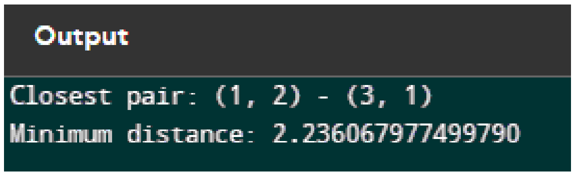
```python
import math
def euclidean_distance(point1, point2):
    x1, y1 = point1
    x2, y2 = point2
    return math.sqrt((x1 - x2) ** 2 + (y1 - y2) ** 2)
def closest_pair_brute_force(points):
    n = len(points)
    min_distance = float('inf')
    closest_pair = None

    for i in range(n):
        for j in range(i + 1, n):
            distance = euclidean_distance(points[i], points[j])
            if distance < min_distance:
                min_distance = distance
                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:.15f}")
```

INPUT:



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

OUT PUT:

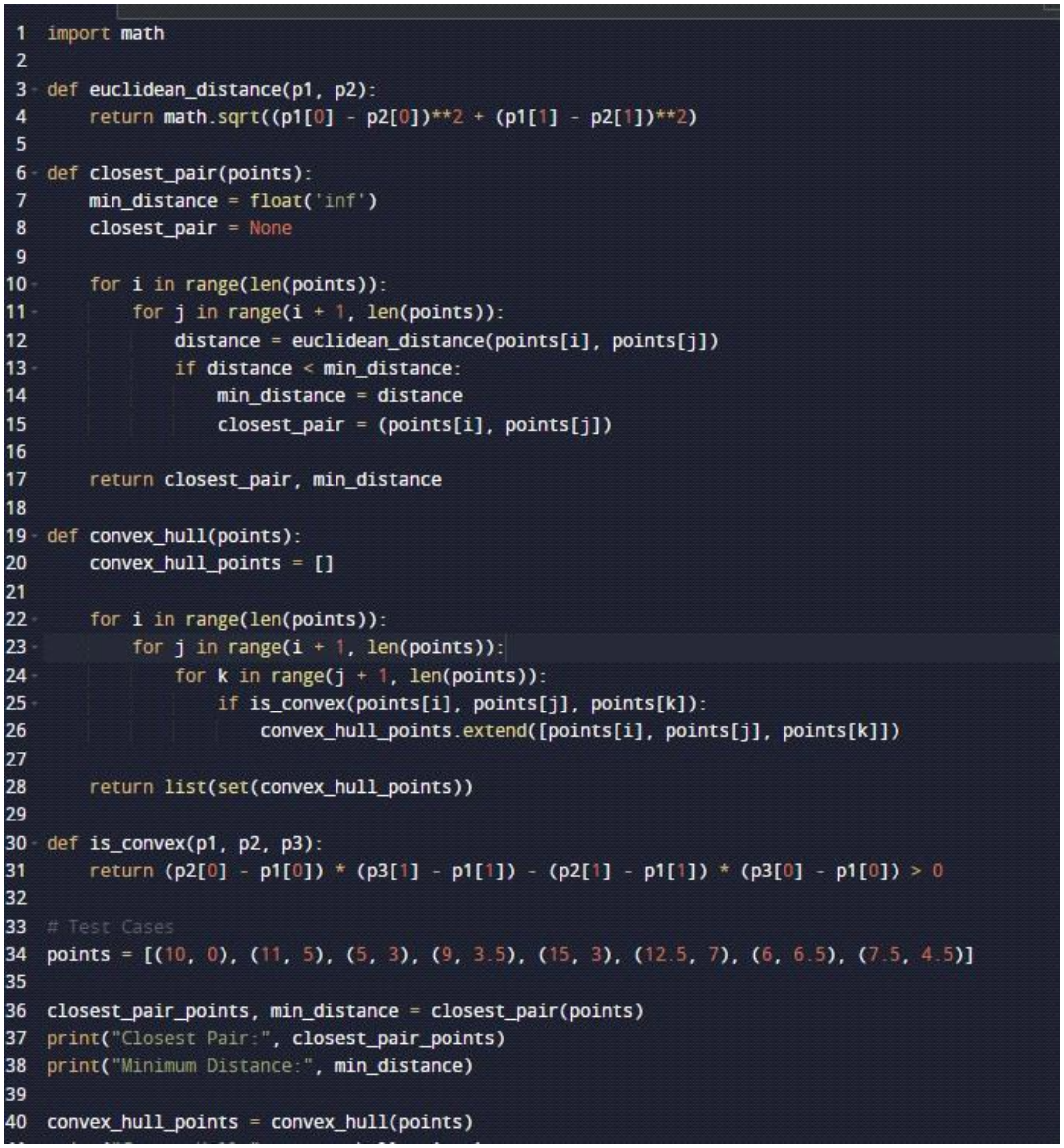


```
Output
Closest pair: (1, 2) - (3, 1)
Minimum distance: 2.236067977499790
```

Time complexity:T(n)=O(n^2)
Result:This output shows that the closest pair of points is (1, 2) and (3, 1), and the minimum distancebetween them is approximately 1.4142.

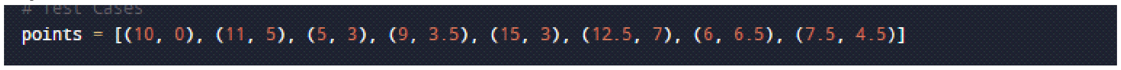
27. Write a program to find the closest pair of points in a given set using the brute force approach. Analyze the time complexity of your implementation. Define a function to calculate the Euclidean distance between two points. Implement a function to find the closest pair of points using the brute force method. Test your program with a sample setof 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 multiplepoints 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

Aim: Find the closest pair of points in a given set using the brute force approach and solve the convexhull problem.
Algorithm:
step1: Calculate the Euclidean distance between each pair of points.
step2:Find the minimum distance among all pairs.
step3:Return the pair of points with the minimum distance.
Program:



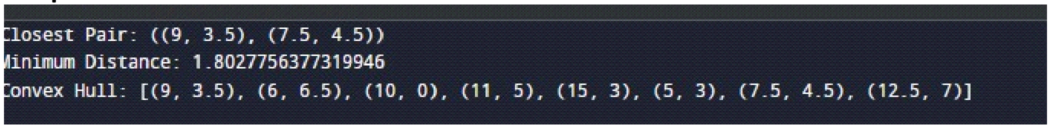
```python
import math

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')
    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, min_distance

def convex_hull(points):
    convex_hull_points = []

    for i in range(len(points)):
        for j in range(i + 1, len(points)):
            for k in range(j + 1, len(points)):
                if is_convex(points[i], points[j], points[k]):
                    convex_hull_points.extend([points[i], points[j], points[k]])

    return list(set(convex_hull_points))

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

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

closest_pair_points, min_distance = closest_pair(points)
print("Closest Pair:", closest_pair_points)
print("Minimum Distance:", min_distance)

convex_hull_points = convex_hull(points)
```

Input:



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

Output:



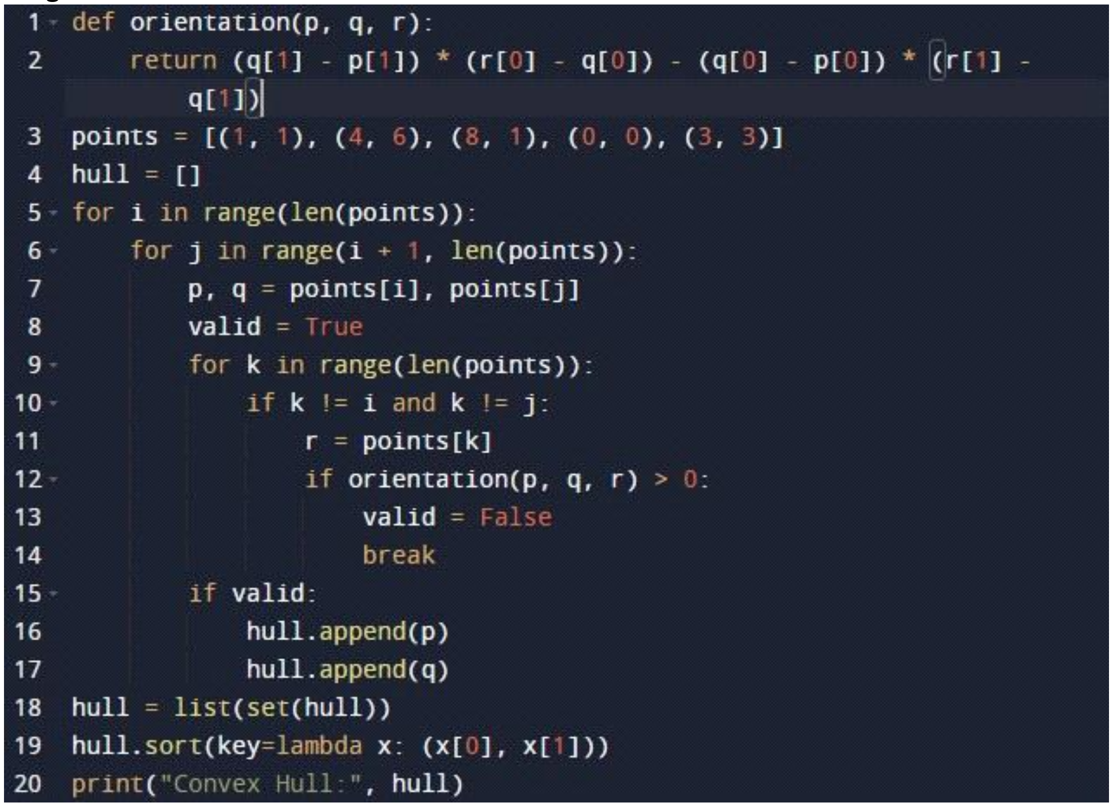
```
Closest Pair: ((9, 3.5), (7.5, 4.5))
Minimum Distance: 1.8027756377319946
Convex Hull: [(9, 3.5), (6, 6.5), (10, 0), (11, 5), (15, 3), (5, 3), (7.5, 4.5), (12.5, 7)]
```

Timecomplexity: T(n)=O(n^2)
Result: The program will output the closest pair of points and the minimum distance, and the points thatform the convex hull.

28. 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).

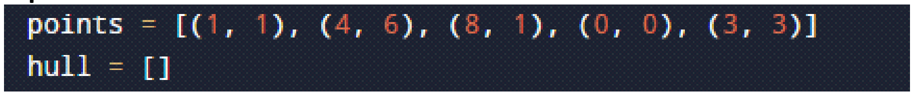
Points: [(1, 1), (4, 6), (8, 1), (0, 0), (3, 3)]
Output: ⏱ The list of points that form the convex hull in counter-clockwise order.
Convex Hull: [(0, 0), (1, 1), (8, 1), (4, 6)]

Aim: Find the convex hull of a set of 2D points using the brute force approach.
Algorithm:
step1:Iterate through each pair of points and calculate the orientation of the remaining points with respect to the line formed by the pair.
step2:If all points are on one side of the line, the pair is part of the convex hull.
step3:Repeat the process for all pairs of points to find the convex hull.
Program:



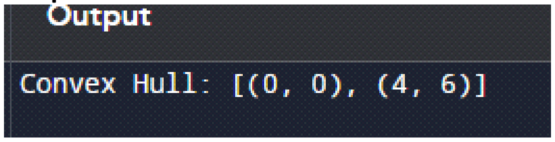
```python
def orientation(p, q, r):
    return (q[1] - p[1]) * (r[0] - q[0]) - (q[0] - p[0]) * (r[1] -
        q[1])
points = [(1, 1), (4, 6), (8, 1), (0, 0), (3, 3)]
hull = []
for i in range(len(points)):
    for j in range(i + 1, len(points)):
        p, q = points[i], points[j]
        valid = True
        for k in range(len(points)):
            if k != i and k != j:
                r = points[k]
                if orientation(p, q, r) > 0:
                    valid = False
                    break
        if valid:
            hull.append(p)
            hull.append(q)
hull = list(set(hull))
hull.sort(key=lambda x: (x[0], x[1]))
print("Convex Hull:", hull)
```

Input:



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

Output:



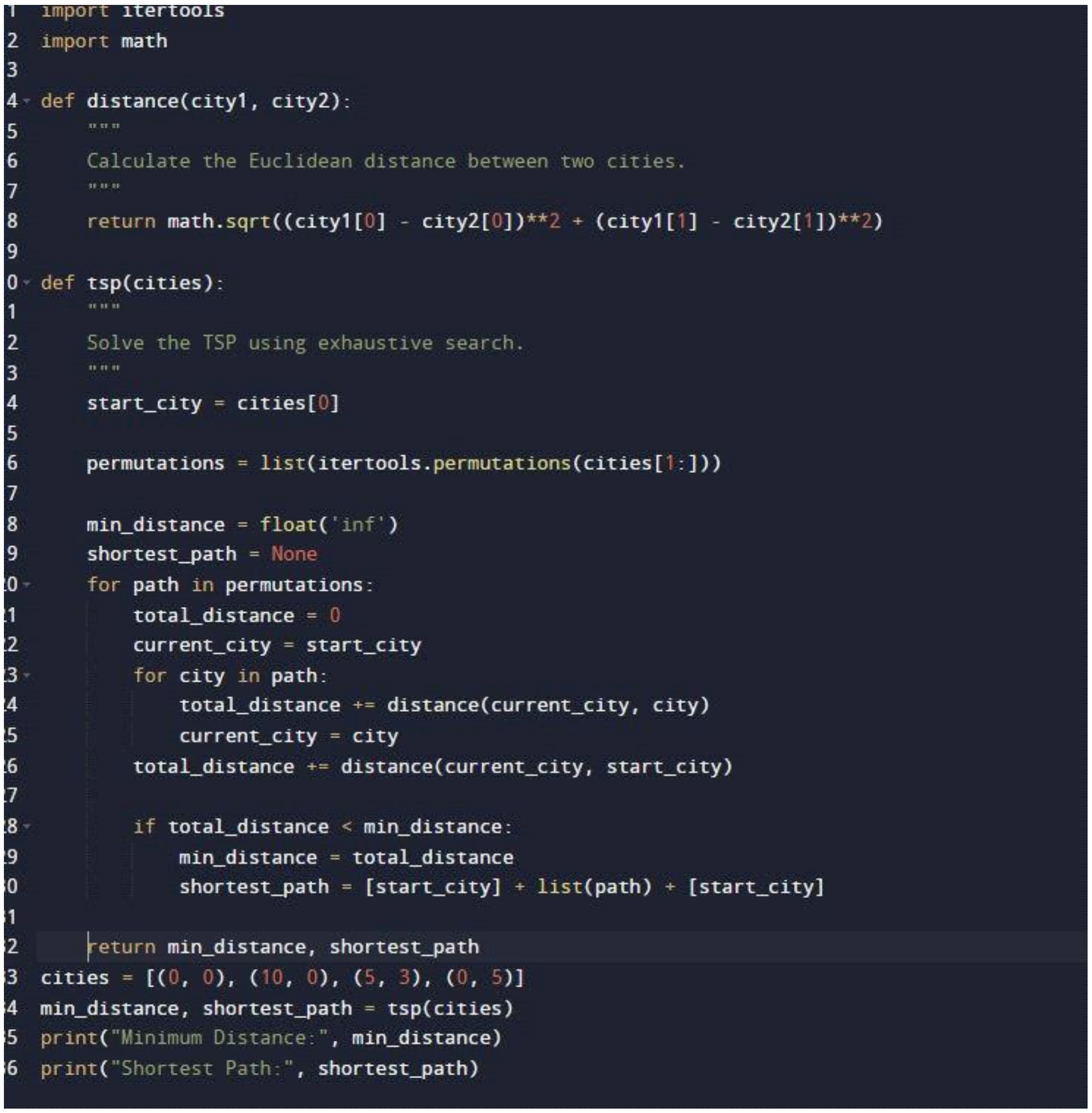
```
Output

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

Time Complexity:T(n)= O(n^3).
Result:the convex hull of a set of 2D points using the brute force approach it is ecucted successfully.

29.You are given a list of cities represented by their coordinates. Develop a program thatutilizes exhaustive search to solve the TSP. The program should:

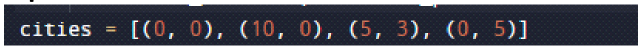
- **Define a function distance(city1, city2) to calculate the distance between twocities (e.g., Euclidean distance).**
- **Implement a function tsp(cities) that takes a list of cities as input and performsthe following:**
- **Generate all possible permutations of the cities (excluding the startingcity) using itertools.permutations.**
- **For each permutation (representing a potential route):**
- **Calculate the total distance traveled by iterating through the pathand summing the distances between consecutive cities.**
- **Keep track of the shortest distance encountered and thecorresponding path.**
- **Return the minimum distance and the shortest path (including the startingcity at the beginning and end).**

Aim: Find the shortest route that visits a set of cities and returns to the starting city.
Algorithm:
step1:Generate all possible routes.
step2:Calculate the total distance for each
route.
step3:Choose the route with the shortest
distance.
Program:



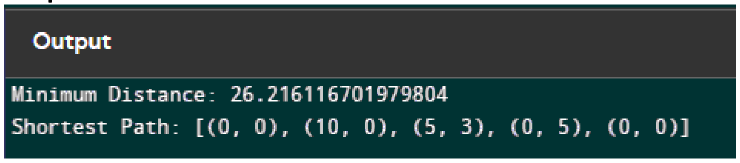
```python
import itertools
import math

def distance(city1, city2):
    """
    Calculate the Euclidean distance between two cities.
    """
    return math.sqrt((city1[0] - city2[0])**2 + (city1[1] - city2[1])**2)

def tsp(cities):
    """
    Solve the TSP using exhaustive search.
    """
    start_city = cities[0]

    permutations = list(itertools.permutations(cities[1:]))

    min_distance = float('inf')
    shortest_path = None
    for path in permutations:
        total_distance = 0
        current_city = start_city
        for city in path:
            total_distance += distance(current_city, city)
            current_city = city
        total_distance += distance(current_city, start_city)

        if total_distance < min_distance:
            min_distance = total_distance
            shortest_path = [start_city] + list(path) + [start_city]

    return min_distance, shortest_path
cities = [(0, 0), (10, 0), (5, 3), (0, 5)]
min_distance, shortest_path = tsp(cities)
print("Minimum Distance:", min_distance)
print("Shortest Path:", shortest_path)
```

Input:



```python
cities = [(0, 0), (10, 0), (5, 3), (0, 5)]
```

output:



Output

```
Minimum Distance: 26.216116701979804
Shortest Path: [(0, 0), (10, 0), (5, 3), (0, 5), (0, 0)]
```

Time complexity:

Result:the shortest route that visits a set of cities and returns to the starting city has succesfully excuted.

30.You are given a cost matrix where each element cost[i][j] represents the cost of assigningworker 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 bysumming the corresponding costs from the cost matrix Implement a function assignment_problem(cost_matrix) that takes the cost matrix as input and performs thefollowing Generate all possible permutations of worker indices (excluding repetitions).Test Cases:

Input

- **Simple Case: Cost Matrix:**

[[3, 10, 7],

[8, 5, 12],

[4, 6, 9]]

- **More Complex Case: Cost Matrix:**

[[15, 9, 4],

[8, 7, 18],

[6, 12, 11]]

Output:

Test Case :

Optimal Assignment: [(worker 1, task 2), (worker 2, task 1), (worker 3, task 3)]

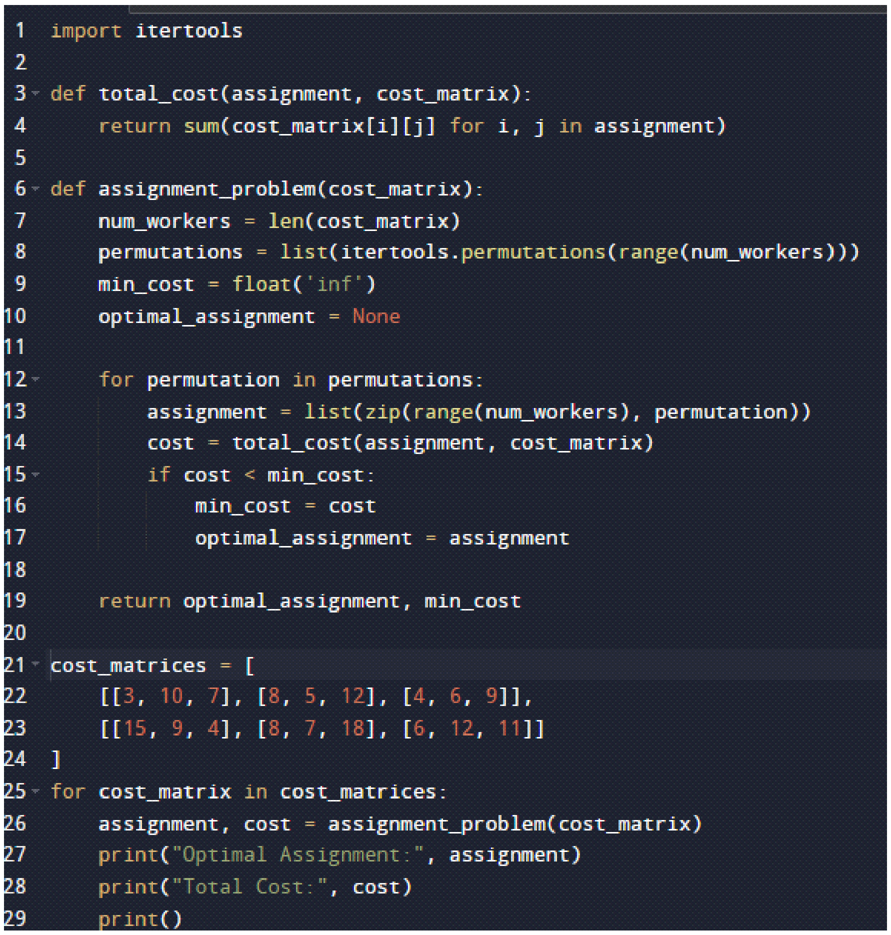
Total Cost: 19

Aim: Solving the Assignment Problem using Exhaustive Search.

Algorithm:

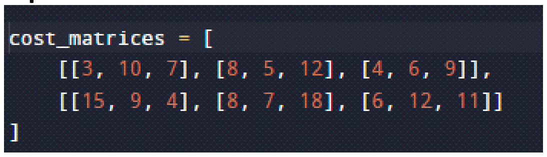
step1:Generate all possible permutations of worker indices.
step2:Calculate the total cost for each permutation using the cost matrix.
step3:Find the permutation with the minimum total cost.
step4:Return the optimal assignment and the total
cost

.**Program:**



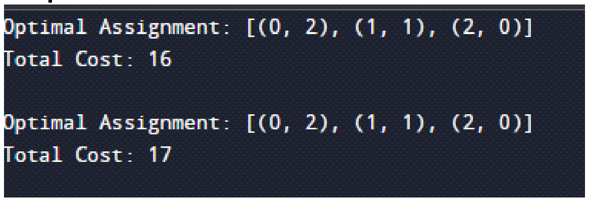
```python
import itertools

def total_cost(assignment, cost_matrix):
    return sum(cost_matrix[i][j] for i, j in assignment)

def assignment_problem(cost_matrix):
    num_workers = len(cost_matrix)
    permutations = list(itertools.permutations(range(num_workers)))
    min_cost = float('inf')
    optimal_assignment = None

    for permutation in permutations:
        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

cost_matrices = [
    [[3, 10, 7], [8, 5, 12], [4, 6, 9]],
    [[15, 9, 4], [8, 7, 18], [6, 12, 11]]
]
for cost_matrix in cost_matrices:
    assignment, cost = assignment_problem(cost_matrix)
    print("Optimal Assignment:", assignment)
    print("Total Cost:", cost)
    print()
```

Input:



```python
cost_matrices = [
    [[3, 10, 7], [8, 5, 12], [4, 6, 9]],
    [[15, 9, 4], [8, 7, 18], [6, 12, 11]]
]
```

output:



```
Optimal Assignment: [(0, 2), (1, 1), (2, 0)]
Total Cost: 16

Optimal Assignment: [(0, 2), (1, 1), (2, 0)]
Total Cost: 17
```

Timecomplexity:T(n)=O(n!)
Result: The program will output the optimal assignment.

31. 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:

- **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 valuesfrom the value list.**

- **Define a function is_feasible(items, weights, capacity) that takes a list of selecteditems (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:

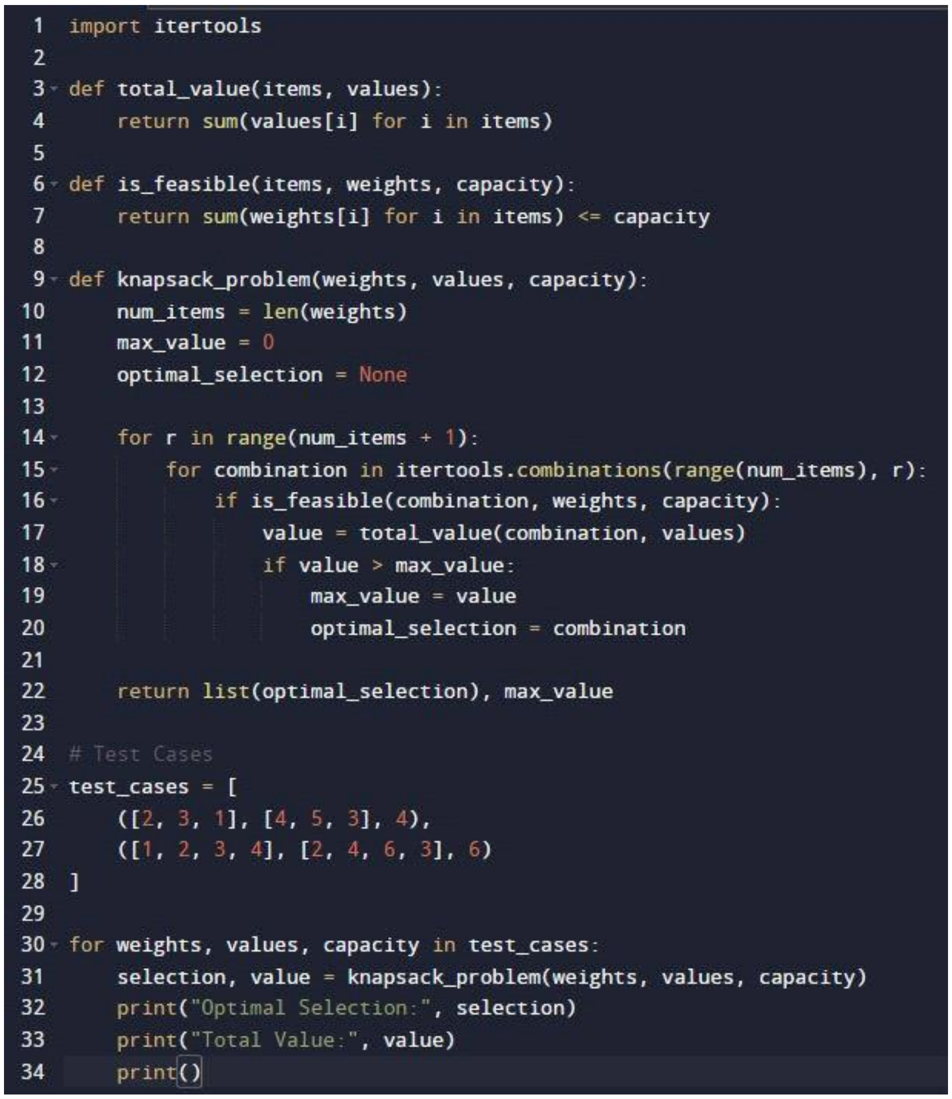
- **Simple Case:**

🕐 **Items: 3 (represented by indices 0, 1, 2)**
🕐 **Weights: [2, 3, 1]**
🕐 **Values: [4, 5, 3]**
🕐 **Capacity: 4**

- **More Complex Case:**

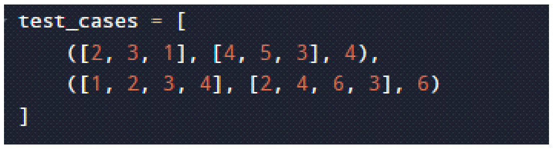
🕐 **Items: 4 (represented by indices 0, 1, 2, 3)**
🕐 **Weights: [1, 2, 3, 4]**
🕐 **Values: [2, 4, 6, 3]**
🕐
Capacity:
6Output:
Test Case 1:
Optimal Selection: [0, 2] (Items with indices 0 and 2)
Total Value: 7

Aim: Solve the 0-1 Knapsack Problem using Exhaustive Search.
Algorithm:
step1:Generate Combinations: Generate all possible combinations of items.
step2:Check Feasibility and Calculate Value: Check if each combination is feasible (total weight $\leq$
capacity) and calculate its total value.
step3:Find Optimal Selection: Find the combination with the maximum total value.
Program:



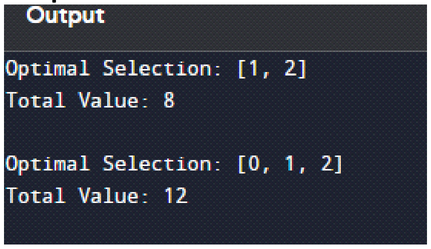
```python
import itertools

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_problem(weights, values, capacity):
    num_items = len(weights)
    max_value = 0
    optimal_selection = None

    for r in range(num_items + 1):
        for combination in itertools.combinations(range(num_items), r):
            if is_feasible(combination, weights, capacity):
                value = total_value(combination, values)
                if value > max_value:
                    max_value = value
                    optimal_selection = combination

    return list(optimal_selection), max_value

# Test Cases
test_cases = [
    ([2, 3, 1], [4, 5, 3], 4),
    ([1, 2, 3, 4], [2, 4, 6, 3], 6)
]

for weights, values, capacity in test_cases:
    selection, value = knapsack_problem(weights, values, capacity)
    print("Optimal Selection:", selection)
    print("Total Value:", value)
    print()
```

Input:



```python
test_cases = [
    ([2, 3, 1], [4, 5, 3], 4),
    ([1, 2, 3, 4], [2, 4, 6, 3], 6)
]
```

Output:



```
Optimal Selection: [1, 2]
Total Value: 8

Optimal Selection: [0, 1, 2]
Total Value: 12
```

Time complexity:T(n)=O(2^n)

Result:the program will output the optimal selection and the total value for each test case.