

26. 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

**Aim :** To find the two points in a 2D plane that are closest to each other, and calculate the minimum distance 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:**

```
1  import math
2  def euclidean_distance(point1, point2):
3      x1, y1 = point1
4      x2, y2 = point2
5      return math.sqrt((x1 - x2) ** 2 + (y1 - y2) ** 2)
6  def closest_pair_brute_force(points):
7      n = len(points)
8      min_distance = float('inf')
9      closest_pair = None
10
11     for i in range(n):
12         for j in range(i + 1, n):
13             distance = euclidean_distance(points[i], points[j])
14             if distance < min_distance:
15                 min_distance = distance
16                 closest_pair = (points[i], points[j])
17
18     return closest_pair, min_distance
19
20  points = [(1, 2), (4, 5), (7, 8), (3, 1)]
21  closest_pair, min_distance = closest_pair_brute_force(points)
22  print(f"Closest pair: {closest_pair[0]} - {closest_pair[1]}")
23  print(f"Minimum distance: {min_distance:.15f}")
```

Input:

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

Output:

### 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 distance between 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 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 same line?**

**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 convex hull 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:**

```

1 import math
2
3 def euclidean_distance(p1, p2):
4     return math.sqrt((p1[0] - p2[0])**2 + (p1[1] - p2[1])**2)
5
6 def closest_pair(points):
7     min_distance = float('inf')
8     closest_pair = None
9
10    for i in range(len(points)):
11        for j in range(i + 1, len(points)):
12            distance = euclidean_distance(points[i], points[j])
13            if distance < min_distance:
14                min_distance = distance
15                closest_pair = (points[i], points[j])
16
17    return closest_pair, min_distance
18
19 def convex_hull(points):
20     convex_hull_points = []
21
22    for i in range(len(points)):
23        for j in range(i + 1, len(points)):
24            for k in range(j + 1, len(points)):
25                if is_convex(points[i], points[j], points[k]):
26                    convex_hull_points.extend([points[i], points[j], points[k]])
27
28    return list(set(convex_hull_points))
29
30 def is_convex(p1, p2, p3):
31     return (p2[0] - p1[0]) * (p3[1] - p1[1]) - (p2[1] - p1[1]) * (p3[0] - p1[0]) > 0
32
33 # Test Cases
34 points = [(10, 0), (11, 5), (5, 3), (9, 3.5), (15, 3), (12.5, 7), (6, 6.5), (7.5, 4.5)]
35
36 closest_pair_points, min_distance = closest_pair(points)
37 print("Closest Pair:", closest_pair_points)
38 print("Minimum Distance:", min_distance)
39
40 convex_hull_points = convex_hull(points)

```

Input:

```

# Test Cases
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)]

```

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

**Result:** The program will output the closest pair of points and the minimum distance, and the points that form 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:**

```
1 def orientation(p, q, r):
2     return (q[1] - p[1]) * (r[0] - q[0]) - (q[0] - p[0]) * (r[1] -
3         q[1])
4 points = [(1, 1), (4, 6), (8, 1), (0, 0), (3, 3)]
5 hull = []
6 for i in range(len(points)):
7     for j in range(i + 1, len(points)):
8         p, q = points[i], points[j]
9         valid = True
10        for k in range(len(points)):
11            if k != i and k != j:
12                r = points[k]
13                if orientation(p, q, r) > 0:
14                    valid = False
15                    break
16        if valid:
17            hull.append(p)
18            hull.append(q)
19 hull = list(set(hull))
20 hull.sort(key=lambda x: (x[0], x[1]))
21 print("Convex Hull:", hull)
```

**Input:**

```
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 executed successfully.

**29.** 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).

**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:**

```

1 import itertools
2 import math
3
4 def distance(city1, city2):
5     """
6     Calculate the Euclidean distance between two cities.
7     """
8     return math.sqrt((city1[0] - city2[0])**2 + (city1[1] - city2[1])**2)
9
10 def tsp(cities):
11     """
12     Solve the TSP using exhaustive search.
13     """
14     start_city = cities[0]
15
16     permutations = list(itertools.permutations(cities[1:]))
17
18     min_distance = float('inf')
19     shortest_path = None
20     for path in permutations:
21         total_distance = 0
22         current_city = start_city
23         for city in path:
24             total_distance += distance(current_city, city)
25             current_city = city
26         total_distance += distance(current_city, start_city)
27
28         if total_distance < min_distance:
29             min_distance = total_distance
30             shortest_path = [start_city] + list(path) + [start_city]
31
32     return min_distance, shortest_path
33 cities = [(0, 0), (10, 0), (5, 3), (0, 5)]
34 min_distance, shortest_path = tsp(cities)
35 print("Minimum Distance:", min_distance)
36 print("Shortest Path:", shortest_path)

```

Input:

```
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 successfully executed.

30. You are given a cost matrix where each element  $\text{cost}[i][j]$  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 :

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:**



```

1 import itertools
2
3 def total_cost(assignment, cost_matrix):
4     return sum(cost_matrix[i][j] for i, j in assignment)
5
6 def assignment_problem(cost_matrix):
7     num_workers = len(cost_matrix)
8     permutations = list(itertools.permutations(range(num_workers)))
9     min_cost = float('inf')
10    optimal_assignment = None
11
12    for permutation in permutations:
13        assignment = list(zip(range(num_workers), permutation))
14        cost = total_cost(assignment, cost_matrix)
15        if cost < min_cost:
16            min_cost = cost
17            optimal_assignment = assignment
18
19    return optimal_assignment, min_cost
20
21 cost_matrices = [
22     [[3, 10, 7], [8, 5, 12], [4, 6, 9]],
23     [[15, 9, 4], [8, 7, 18], [6, 12, 11]]
24 ]
25 for cost_matrix in cost_matrices:
26     assignment, cost = assignment_problem(cost_matrix)
27     print("Optimal Assignment:", assignment)
28     print("Total Cost:", cost)
29     print()

```

Input:

```

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

```

Time complexity:  $T(n) = O(n!)$

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:**

- 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

**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:**

```

1 import itertools
2
3 def total_value(items, values):
4     return sum(values[i] for i in items)
5
6 def is_feasible(items, weights, capacity):
7     return sum(weights[i] for i in items) <= capacity
8
9 def knapsack_problem(weights, values, capacity):
10    num_items = len(weights)
11    max_value = 0
12    optimal_selection = None
13
14    for r in range(num_items + 1):
15        for combination in itertools.combinations(range(num_items), r):
16            if is_feasible(combination, weights, capacity):
17                value = total_value(combination, values)
18                if value > max_value:
19                    max_value = value
20                    optimal_selection = combination
21
22    return list(optimal_selection), max_value
23
24 # Test Cases
25 test_cases = [
26     ([2, 3, 1], [4, 5, 3], 4),
27     ([1, 2, 3, 4], [2, 4, 6, 3], 6)
28 ]
29
30 for weights, values, capacity in test_cases:
31     selection, value = knapsack_problem(weights, values, capacity)
32     print("Optimal Selection:", selection)
33     print("Total Value:", value)
34     print()

```

Input:

```

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

```

Output:

```

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.