**TOPIC 2 : BRUTE FORCE**

* 1. Write a program to perform the following
  + An empty list
  + A list with one element
  + A list with all identical elements
  + A list with negative numbers

**Test Cases:**

1. **Input:** []
   * **Expected Output:** []
2. **Input:** [1]
   * **Expected Output:** [1]
3. **Input:** [7, 7, 7, 7]
   * **Expected Output:** [7, 7, 7, 7]
4. **Input:** [-5, -1, -3, -2, -4]
   * **Expected Output:** [-5, -4, -3, -2, -1]

def fun(input\_list):

    return sorted(input\_list)

a = [

    [],

    [1],

    [7, 7, 7, 7],

    [-5, -1, -3, -2, -4]]

for i in a:

    result = fun(i)

    print(result)

* 1. Describe the Selection Sort algorithm's process of sorting an array. Selection Sort works by dividing the array into a sorted and an unsorted region. Initially, the sorted region is empty, and the unsorted region contains all elements. The algorithm repeatedly selects the smallest element from the unsorted region and swaps it with the leftmost unsorted element, then moves the boundary of the sorted region one element to the right. Explain why Selection Sort is simple to understand and implement but is inefficient for large datasets. Provide examples to illustrate step-by-step how Selection Sort rearranges the elements into ascending order, ensuring clarity in your explanation of the algorithm's mechanics and effectiveness.

**Sorting a Random Array**:

**Input**: [5, 2, 9, 1, 5, 6]

**Output**: [1, 2, 5, 5, 6, 9]

**Sorting a Reverse Sorted Array**:

**Input**: [10, 8, 6, 4, 2]

**Output**: [2, 4, 6, 8, 10]

**Sorting an Already Sorted Array**:

**Input**: [1, 2, 3, 4, 5]

**Output**: [1, 2, 3, 4, 5]

def selection\_sort(arr):

    n = len(arr)

    for i in range(n):

        min = i

        for j in range(i + 1, n):

            if arr[j] < arr[min]:

                min = j

        arr[i], arr[min] = arr[min], arr[i]

    return arr

arr = [5, 2, 9, 1, 5, 6]

sorted\_arr = selection\_sort(arr)

print("Sorted Array:", sorted\_arr)

* 1. Write code to modify bubble\_sort function to stop early if the list becomes sorted before all passes are completed.

def bubble\_sort(arr):

n = len(arr)

for i in range(n):

swapped = False

for j in range(0, n-i-1):

if arr[j] > arr[j+1]:

arr[j], arr[j+1] = arr[j+1], arr[j]

swapped = True

if not swapped:

break

return arr

a=[9,2,8,3,7,4,6,5]

sorted\_arr = bubble\_sort(a)

print(sorted\_arr)

* 1. **Test Cases:**
* Test your optimized function with the following lists:
  1. **Input:** [64, 25, 12, 22, 11]
     + **Expected Output:** [11, 12, 22, 25, 64]
  2. **Input:** [29, 10, 14, 37, 13]
     + **Expected Output:** [10, 13, 14, 29, 37]
  3. **Input:** [3, 5, 2, 1, 4]
     + **Expected Output:** [1, 2, 3, 4, 5]
  4. **Input:** [1, 2, 3, 4, 5] (Already sorted list)
     + **Expected Output:** [1, 2, 3, 4, 5]
  5. **Input:** [5, 4, 3, 2, 1] (Reverse sorted list)
     + **Expected Output:** [1, 2, 3, 4, 5]

def selection\_sort(arr):

n = len(arr)

for i in range(n):

min = i

for j in range(i + 1, n):

if arr[j] < arr[min]:

min = j

arr[i], arr[min] = arr[min], arr[i]

return arr

arr = [5, 2, 9, 1, 5, 6]

sorted\_arr = selection\_sort(arr)

print("Sorted Array:", sorted\_arr)

Write code for Insertion Sort that manages arrays with duplicate elements during the sorting process. Ensure the algorithm's behavior when encountering duplicate values, including whether it preserves the relative order of duplicates and how it affects the overall sorting outcome.

**Examples:**

**1. Array with Duplicates**:

* + **Input**: [3, 1, 4, 1, 5, 9, 2, 6, 5, 3]
  + **Output**: [1, 1, 2, 3, 3, 4, 5, 5, 6, 9]

1. **All Identical Elements**:
   * **Input**: [5, 5, 5, 5, 5]
   * **Output**: [5, 5, 5, 5, 5]
2. **Mixed Duplicates**:
   * **Input**: [2, 3, 1, 3, 2, 1, 1, 3]
   * **Output**: [1, 1, 1, 2, 2, 3, 3, 3]

def insertion\_sort(arr):

n = len(arr)

for i in range(1, n):

key = arr[i]

j = i - 1

while j >= 0 and arr[j] > key:

arr[j + 1] = arr[j]

j -= 1

arr[j + 1] = key

return arr

arr = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3]

sorted\_arr = insertion\_sort(arr)

print(sorted\_arr)

* 1. Given an array arr of positive integers sorted in a strictly increasing order, and an integer k. return the kth positive integer that is missing from this array.

Example 1:

Input: arr = [2,3,4,7,11], k = 5

Output: 9

Explanation: The missing positive integers are [1,5,6,8,9,10,12,13,...]. The 5th missing positive integer is 9.

Example 2:

Input: arr = [1,2,3,4], k = 2

Output: 6

Explanation: The missing positive integers are [5,6,7,...]. The 2nd missing positive integer is 6.

def find\_kth\_missing(arr, k):

    missing\_count = 0

    i = 0

    expected = 1

    while missing\_count < k:

        if i < len(arr) and arr[i] == expected:

            i += 1

        else:

            missing\_count += 1

            if missing\_count == k:

                return expected

        expected += 1

    return expected + k - 1

arr1, k1 = [2, 3, 4, 7, 11], 5

arr2, k2 = [1, 2, 3, 4], 2

print(find\_kth\_missing(arr1, k1))

print(find\_kth\_missing(arr2, k2))

* 1. A peak element is an element that is strictly greater than its neighbors. Given a 0-indexed integer array nums, find a peak element, and return its index. If the array contains multiple peaks, return the index to any of the peaks. You may imagine that nums[-1] = nums[n] = -∞. In other words, an element is always considered to be strictly greater than a neighbor that is outside the array. You must write an algorithm that runs in O(log n) time.

Example 1:

Input: nums = [1,2,3,1]

Output: 2

Explanation: 3 is a peak element and your function should return the index number 2.

Example 2:

Input: nums = [1,2,1,3,5,6,4]

Output: 5

Explanation: Your function can return either index number 1 where the peak element is 2, or index number 5 where the peak element is 6.

def find\_peak\_element(nums):

    left, right = 0, len(nums) - 1

    while left < right:

        mid = left + (right - left) // 2

        if nums[mid] > nums[mid + 1]:

            right = mid

        else:

            left = mid + 1

    return left

nums1 = [1, 2, 3, 1]

nums2 = [1, 2, 1, 3, 5, 6, 4]

print(find\_peak\_element(nums1))

print(find\_peak\_element(nums2))

* 1. Given two strings needle and haystack, return the index of the first occurrence of needle in haystack, or -1 if needle is not part of haystack.

Example 1:

Input: haystack = "sadbutsad", needle = "sad"

Output: 0

Explanation: "sad" occurs at index 0 and 6.

The first occurrence is at index 0, so we return 0.

Example 2:

Input: haystack = "leetcode", needle = "leeto"

Output: -1

Explanation: "leeto" did not occur in "leetcode", so we return -1.

def str\_str(haystack, needle):

    return haystack.find(needle)

haystack1, needle1 = "sadbutsad", "sad"

haystack2, needle2 = "leetcode", "leeto"

print(str\_str(haystack1, needle1))

print(str\_str(haystack2, needle2))

* 1. Given an array of string words, return all strings in words that is a substring of another word. You can return the answer in any order. A substring is a contiguous sequence of characters within a string

Example 1:

Input: words = ["mass","as","hero","superhero"]

Output: ["as","hero"]

Explanation: "as" is substring of "mass" and "hero" is substring of "superhero".

["hero","as"] is also a valid answer.

Example 2:

Input: words = ["leetcode","et","code"]

Output: ["et","code"]

Explanation: "et", "code" are substring of "leetcode".

Example 3:

Input: words = ["blue","green","bu"]

Output: []

Explanation: No string of words is substring of another string.

def string\_matching(words):

    result = []

    n = len(words)

    for i in range(n):

        for j in range(n):

            if i != j and words[i] in words[j]:

                result.append(words[i])

                break

    return result

a= ["mass", "as", "hero", "superhero"]

print(string\_matching(a))

* 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

def distance(point1, point2):

    return ((point2[0] - point1[0])\*\*2 + (point2[1] - point1[1])\*\*2)\*\*0.5

def closest\_pair\_brute\_force(points):

    n = len(points)

    if n < 2:

        return None, float('inf')

    min\_distance = float('inf')

    closest\_pair = None

    for i in range(n):

        for j in range(i + 1, n):

            dist = distance(points[i], points[j])

            if dist < min\_distance:

                min\_distance = dist

                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("Closest pair:", closest\_pair)

print("Minimum distance:", min\_distance)

* 1. 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

def euclidean\_distance(point1, point2):

    # Calculate Euclidean distance between two points

    return ((point2[0] - point1[0]) \*\* 2 + (point2[1] - point1[1]) \*\* 2) \*\* 0.5

def closest\_pair\_brute\_force(points):

    n = len(points)

    if n < 2:

        return None, float('inf')

    min\_distance = float('inf')

    closest\_pair = None

    for i in range(n):

        for j in range(i + 1, n):

            dist = euclidean\_distance(points[i], points[j])

            if dist < min\_distance:

                min\_distance = dist

                closest\_pair = (points[i], points[j])

    return closest\_pair, min\_distance

def orientation(p, q, r):

    # Function to determine orientation of three points

    val = (q[1] - p[1]) \* (r[0] - q[0]) - (q[0] - p[0]) \* (r[1] - q[1])

    if val == 0:

        return 0  # Collinear

    elif val > 0:

        return 1  # Clockwise

    else:

        return 2  # Counter-clockwise

def on\_segment(p, q, r):

    # Check if point q lies on segment pr

    if (q[0] <= max(p[0], r[0]) and q[0] >= min(p[0], r[0]) and

        q[1] <= max(p[1], r[1]) and q[1] >= min(p[1], r[1])):

        return True

    return False

def intersect(p1, q1, p2, q2):

    # Check if line segment p1q1 intersects with line segment p2q2

    o1 = orientation(p1, q1, p2)

    o2 = orientation(p1, q1, q2)

    o3 = orientation(p2, q2, p1)

    o4 = orientation(p2, q2, q1)

    # General case

    if o1 != o2 and o3 != o4:

        return True

    # Special cases

    # p1, q1 and p2 are collinear and p2 lies on segment p1q1

    if o1 == 0 and on\_segment(p1, p2, q1):

        return True

    # p1, q1 and q2 are collinear and q2 lies on segment p1q1

    if o2 == 0 and on\_segment(p1, q2, q1):

        return True

    # p2, q2 and p1 are collinear and p1 lies on segment p2q2

    if o3 == 0 and on\_segment(p2, p1, q2):

        return True

    # p2, q2 and q1 are collinear and q1 lies on segment p2q2

    if o4 == 0 and on\_segment(p2, q1, q2):

        return True

    return False

def convex\_hull\_brute\_force(points):

    n = len(points)

    if n < 3:

        return None

    hull = []

    for i in range(n):

        for j in range(i + 1, n):

            for k in range(j + 1, n):

                if orientation(points[i], points[j], points[k]) == 0:

                    # Check if points[i], points[j], points[k] are collinear

                    collinear = True

                    for m in range(n):

                        if m != i and m != j and m != k:

                            if orientation(points[i], points[j], points[m]) != 0:

                                collinear = False

                                break

                    if collinear:

                        # Check if this line segment intersects with any other line segment

                        valid = True

                        for p in range(n):

                            if p != i and p != j and p != k:

                                for q in range(p + 1, n):

                                    if q != i and q != j and q != k:

                                        if intersect(points[i], points[k], points[p], points[q]):

                                            valid = False

                                            break

                                if not valid:

                                    break

                        if valid:

                            if points[i] not in hull:

                                hull.append(points[i])

                            if points[k] not in hull:

                                hull.append(points[k])

    return hull

# Test the functions with the given points

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

# Find closest pair of points

closest\_pair, min\_distance = closest\_pair\_brute\_force(points)

print(f"Closest pair: {closest\_pair} Minimum distance: {min\_distance}")

# Find convex hull using brute force

convex\_hull = convex\_hull\_brute\_force(points)

print("Convex hull points:", convex\_hull)

* 1. 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)]

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

    elif val > 0:

        return 1

    else:

        return 2

def convex\_hull(points):

    n = len(points)

    if n < 3:

        return points

    hull = []

    l = 0

    for i in range(1, n):

        if points[i][0] < points[l][0]:

            l = i

    p = l

    while True:

        hull.append(points[p])

        q = (p + 1) % n

        for i in range(n):

            if orientation(points[p], points[i], points[q]) == 2:

                q = i

        p = q

        if p == l:

            break

    return hull

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

convex\_hull\_points = convex\_hull(points)

print("Convex Hull:", convex\_hull\_points)

* 1. 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:
   * Generate all possible permutations of the cities (excluding the starting city) using itertools.permutations.
   * 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).
3. 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)]

def distance(city1, city2):

    return ((city1[0] - city2[0])\*\*2 + (city1[1] - city2[1])\*\*2) \*\* 0.5

def permute(cities):

    if len(cities) <= 1:

        yield cities

    else:

        for perm in permute(cities[1:]):

            for i in range(len(cities)):

                yield perm[:i] + cities[0:1] + perm[i:]

def tsp(cities):

    n = len(cities)

    shortest\_path = None

    min\_distance = float('inf')

    for path in permute(cities[1:]):

        full\_path = [cities[0]] + list(path) + [cities[0]]

        total\_distance = 0

        for i in range(n):

            total\_distance += distance(full\_path[i], full\_path[i+1])

        if total\_distance < min\_distance:

            min\_distance = total\_distance

            shortest\_path = full\_path

    return min\_distance, shortest\_path

cities1 = [(1, 2), (4, 5), (7, 1), (3, 6)]

shortest\_distance1, shortest\_path1 = tsp(cities1)

print("Test Case 1:")

print("Shortest Distance:", shortest\_distance1)

print("Shortest Path:", shortest\_path1)

* 1. 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]]

1. **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

* 1. 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

1. **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: 10