CHAPTER 2:-BRUTE FORCE

1. Write a program to perform the following o An empty list o A list with one element o A list with all identical elements o A list with negative numbers **Test Cases:**
   1. **Input:** [] o **Expected Output:** []
   2. **Input:** [1] o **Expected Output:** [1]
   3. **Input:** [7, 7, 7, 7] o **Expected Output:** [7, 7, 7, 7]
   4. **Input:** [-5, -1, -3, -2, -4] o **Expected Output:** [-5, -4, -3, -2, -1]

**Program:-** # An empty list empty\_list = []

# A list with one element single\_element\_list = [42]

# A list with all identical elements identical\_elements\_list = [7, 7, 7, 7, 7]

# A list with negative numbers negative\_numbers\_list = [-3, -7, -11, -5]

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]

**Program:-** def selection\_sort(arr):

n = len(arr) for i in range(n): min\_idx = i for j in range(i+1, n): if arr[j] < arr[min\_idx]:

min\_idx = j

arr[i], arr[min\_idx] = arr[min\_idx], arr[i] return arr

# Sorting a Random Array random\_array = [5, 2, 9, 1, 5, 6]

sorted\_random\_array = selection\_sort(random\_array) print("Random Array Sorted:", sorted\_random\_array)

# Sorting a Reverse Sorted Array reverse\_sorted\_array = [10, 8, 6, 4, 2] sorted\_reverse\_array = selection\_sort(reverse\_sorted\_array) print("Reverse Sorted Array Sorted:", sorted\_reverse\_array)

# Sorting an Already Sorted Array already\_sorted\_array = [1, 2, 3, 4, 5]

sorted\_already\_sorted\_array = selection\_sort(already\_sorted\_array) print("Already Sorted Array Sorted:", sorted\_already\_sorted\_array)

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

**Test Cases:**

Test your optimized function with the following lists:

1. **Input:** [64, 25, 12, 22, 11]

**Program:-**

def bubble\_sort\_optimized(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

# Test the optimized function

input\_list = [64, 25, 12, 22, 11]

output\_list = bubble\_sort\_optimized(input\_list) print(output\_list)

4. 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] o **Output**: [1, 1, 2, 3, 3, 4, 5, 5, 6, 9]
2. **All Identical Elements**:
   * **Input**: [5, 5, 5, 5, 5] o **Output**: [5, 5, 5, 5, 5]
3. **Mixed Duplicates**:
   * **Input**: [2, 3, 1, 3, 2, 1, 1, 3] o **Output**: [1, 1, 1, 2, 2, 3, 3, 3]

**Program:-**

def insertion\_sort\_with\_duplicates(arr):

"""

Performs Insertion Sort on an array with duplicate elements.

The algorithm preserves the relative order of duplicate elements.

"""

for i in range(1, len(arr)):

key = arr[i] j = i - 1

# Shift elements greater than the key to the right while j >= 0 and arr[j] > key:

arr[j + 1] = arr[j]

j -= 1

# Find the correct position to insert the key

# If there are duplicate elements, insert the key after the last occurrence of the duplicate while j >= 0 and arr[j] == key: j -= 1

arr[j + 1] = key

return 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

**Program:-**

def findKthPositive(arr, k):

missing = []

i = 1

while len(missing) < k:

if i not in arr:

missing.append(i)

i += 1

return missing[-1]

# Example

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

output = findKthPositive(arr, k)

print(output)

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

**Program:-**

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]:

left = mid + 1 else: right = mid

return left

7. 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

**Program:-**

def strStr(haystack, needle): return haystack.find(needle)

# Example haystack = "sadbutsad" needle = "sad"

output = strStr(haystack, needle) print(output)

8. 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.

**Program:-**

def stringMatching(words): return [word for word in words if any(other\_word != word and other\_word.find(word) != -1 for other\_word in words)]

# Example

words = ["mass", "as", "hero", "superhero"] print(stringMatching(words))

9. 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)]

**Program:-**

import math

def closest\_pair\_of\_points(points):

"""

Finds the closest pair of points in a set of 2D points using the brute force approach.

Args:

points (list): A list of 2D points represented as tuples (x, y).

Returns:

tuple: The pair of points with the smallest Euclidean distance.

"""

min\_distance = float('inf') closest\_pair = None

for i in range(len(points)): for j in range(i+1, len(points)): x1, y1 = points[i] x2, y2 = points[j]

distance = math.sqrt((x1 - x2)\*\*2 + (y1 - y2)\*\*2)

if distance < min\_distance: min\_distance = distance

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

return closest\_pair

# Example usage

points = [(1, 2), (4, 5), (7, 8), (3, 1)] closest\_pair = closest\_pair\_of\_points(points) print(f"The closest pair of points is: {closest\_pair}")

10. 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?

**Program:-**

import math

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

def closest\_pair\_brute\_force(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

# Sample set of points sample\_points = [(10, 0), (11, 5), (5, 3), (9, 3.5), (15, 3), (12.5, 7), (6, 6.5), (7.5,

4.5)]

# Finding the closest pair of points

closest\_pair = closest\_pair\_brute\_force(sample\_points) print("Closest Pair of Points:", closest\_pair)

11. 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)]

**Program:-**

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

"""

Finds the convex hull of a set of 2D points using the brute force approach.

Args:

points (list): A list of 2D points represented as (x, y) tuples.

Returns:

list: A list of points that form the convex hull.

"""

if len(points) <= 3: return points

hull = [] for i in range(len(points)): for j in range(i+1, len(points)): for k in range(j+1, len(points)): p1, p2, p3 = points[i], points[j], points[k]

# Check if the three points form a counter-clockwise turn if (p2[0] - p1[0]) \* (p3[1] - p1[1]) - (p2[1] - p1[1]) \* (p3[0] - p1[0]) > 0:

hull.append(p1)

hull.append(p2) hull.append(p3)

# Remove duplicate points hull = list(set(hull))

return hull

# Example usage

points = [(1, 1), (4, 6), (8, 1), (0, 0), (3, 3)] convex\_hull = convex\_hull\_brute\_force(points) print(convex\_hull)

12. 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:

**Program:-**

import itertools

def calculate\_distance(city1, city2):

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

def total\_distance(path, cities):

distance = 0 for i in range(len(path) - 1):

distance += calculate\_distance(cities[path[i]], cities[path[i + 1]]) distance += calculate\_distance(cities[path[-1]], cities[path[0]]) return distance

def tsp\_exhaustive\_search(cities):

all\_paths = list(itertools.permutations(range(len(cities)))) min\_distance = float('inf') best\_path = None for path in all\_paths:

distance = total\_distance(path, cities) if distance < min\_distance: min\_distance = distance best\_path = path

return best\_path, min\_distance

# Example Usage

cities = [(0, 0), (1, 2), (3, 1), (5, 3)]

best\_path, min\_distance = tsp\_exhaustive\_search(cities) print("Best Path:", best\_path)

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

13. 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).

**Program:-**

import itertools

def total\_cost(assignment, cost\_matrix):

total = 0

for i, j in enumerate(assignment): total += cost\_matrix[i][j] return total

def assignment\_problem(cost\_matrix): workers = list(range(len(cost\_matrix))) min\_cost = float('inf')

best\_assignment = None

for perm in itertools.permutations(workers): cost = total\_cost(perm, cost\_matrix) if cost < min\_cost: min\_cost = cost best\_assignment = perm

return best\_assignment

# Test Cases cost\_matrix = [ [9, 2, 7],

[9, 7, 4],

[8, 3, 6]

]

best\_assignment = assignment\_problem(cost\_matrix)

print(best\_assignment)

14. 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

**Program:-**

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

# Test Case 1 items\_1 = [0, 1, 2] weights\_1 = [2, 3, 1] values\_1 = [4, 5, 3]

capacity\_1 = 4

# Test Case 2 items\_2 = [0, 1, 2, 3] weights\_2 = [1, 2, 3, 4] values\_2 = [2, 4, 6, 3]

capacity\_2 = 6