CHAPTER 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]

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]
```

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

5. 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)
```

6. 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

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

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
```