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]

Sol :

```
def process_list(input_list):

   if not input_list or len(input_list) == 1 or all(x == input_list[0] for x in input_list):

      return input_list

   else:

      return sorted(input_list)


# Input from user

user_input = input("Enter a list of numbers separated by commas: ")

input_list = list(map(int, user_input.split(',')))


# Process and print the result

result = process_list(input_list)

print(f"Processed list: {result}")
```

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]

SOL:

```
def selection_sort(arr):

        n = len(arr)

    for i in range(n):

        min_index = i

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

            if arr[j] < arr[min_index]:

                    min_index = j

        arr[i], arr[min_index] = arr[min_index], arr[i]

    return arr


# Sorting a Random Array
```

```
print(selection_sort([5, 2, 9, 1, 5, 6]))  # Output: [1, 2, 5, 5, 6, 9]


# Sorting a Reverse Sorted Array

print(selection_sort([10, 8, 6, 4, 2]))   # Output: [2, 4, 6, 8, 10]


# Sorting an Already Sorted Array

print(selection_sort([1, 2, 3, 4, 5]))     # Output: [1, 2, 3, 4, 5]
```

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]
        - § **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]

Sol:

```
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


# Example usage
print(bubble_sort([64, 34, 25, 12, 22, 11, 90]))  # Output: [11, 12, 22, 25, 34, 64, 90]
print(bubble_sort([1, 2, 3, 4, 5]))               # Output: [1, 2, 3, 4, 5]
print(bubble_sort([5, 4, 3, 2, 1]))               # Output: [1, 2, 3, 4, 5]
```

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]

Sol:

```
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1

        arr[j + 1] = key

    return arr


# Example usage

print(insertion_sort([3, 1, 4, 1, 5, 9, 2, 6, 5, 3]))  # Output: [1, 1, 2, 3, 3, 4, 5, 5, 6, 9]

print(insertion_sort([5, 5, 5, 5, 5]))                  # Output: [5, 5, 5, 5, 5]

print(insertion_sort([2, 3, 1, 3, 2, 1, 1, 3]))        # Output: [1, 1, 1, 2, 2, 3, 3, 3]
```

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
> 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.

Sol:

```
def find_kth_missing(arr, k):

    missing_count = 0

    current = 1

    index = 0


    while missing_count < k:

        if index < len(arr) and arr[index] == current:

            index += 1

        else:
            missing_count += 1

            if missing_count == k:

                return current

        current += 1


# Example usage

print(find_kth_missing([2, 3, 4, 7, 11], 5))  # Output: 9

print(find_kth_missing([1, 2, 3, 4], 2))      # Output: 6
```

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
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.

Sol:

```
def find_peak_element(nums):

    left, right = 0, len(nums) - 1


    while left < right:

        mid = (left + right) // 2

        if nums[mid] > nums[mid + 1]:

            right = mid
        else:
            left = mid + 1

    return left


# Example usage
print(find_peak_element([1, 2, 3, 1]))        # Output: 2
print(find_peak_element([1, 2, 1, 3, 5, 6, 4]))  # Output: 5 (or 1)
```

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
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.
```

Sol:

```
def str_str(haystack, needle):
    return haystack.find(needle)


# Example usage
print(str_str("sadbutsad", "sad"))    # Output: 0
print(str_str("leetcode", "leeto"))   # Output: -1
```

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.

Sol:
```
def find_substrings(words):
    substrings = []
    for i, word in enumerate(words):
        for j, other_word in enumerate(words):
            if i != j and word in other_word:
                substrings.append(word)
                break
    return substrings


# Example 1
words1 = ["mass", "as", "hero", "superhero"]
print("Example 1 Output:", find_substrings(words1))


# Example 2
```

```
words2 = ["leetcode", "et", "code"]
print("Example 2 Output:", find_substrings(words2))


# Example 3
words3 = ["blue", "green", "bu"]
print("Example 3 Output:", find_substrings(words3))
```

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)]

Output:

· The two points with the minimum distance between them.

· The minimum distance itself.

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

Sol : import math

```
def euclidean_distance(point1, point2):
    return math.sqrt((point1[0] - point2[0]) ** 2 + (point1[1] - point2[1]) ** 2)


def closest_pair(points):
    min_distance = float('inf')
    closest_points = None
    n = len(points)
    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_points = (points[i], points[j])


    return closest_points, min_distance


# Test case

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

closest_points, min_distance = closest_pair(points)

print("Closest pair:", closest_points)

print("Minimum distance:", min_distance)
```

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?

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

Sol:

```
import math


# Function to calculate Euclidean distance
def euclidean_distance(p1, p2):
    return math.sqrt((p1[0] - p2[0]) ** 2 + (p1[1] - p2[1]) ** 2)


# Function to find the closest pair of points using brute force
def closest_pair(points):
    min_distance = float('inf')
    closest_points = None
    n = len(points)

    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_points = (points[i], points[j])

    return closest_points, min_distance


# Sample set of points
points = [(10,0), (11,5), (5, 3), (9, 3.5), (15, 3), (12.5, 7), (6, 6.5), (7.5, 4.5)]
closest_points, min_distance = closest_pair(points)
```

```
print("Closest pair of points:", closest_points)

print("Minimum distance:", min_distance)
```

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)]

Output:

· The list of points that form the convex hull in counter-clockwise order.

Convex Hull: [(0, 0), (1, 1), (8, 1), (4, 6)]

Sol :
```
def cross_product(o, a, b):

    return (a[0] - o[0]) * (b[1] - o[1]) - (a[1] - o[1]) * (b[0] - o[0])


def convex_hull(points):

    hull = []

    n = len(points)


    if n < 3:

        return points


    # Find the leftmost point

    leftmost = min(points, key=lambda p: p[0])

    point_on_hull = leftmost
    while True:
        hull.append(point_on_hull)
        endpoint = points[0]
        for j in range(1, n):
            if endpoint == point_on_hull or cross_product(point_on_hull, endpoint, points[j]) < 0:
                endpoint = points[j]
        point_on_hull = endpoint
        if point_on_hull == hull[0]:
            break

    return hull

# Test case
points = [(1, 1), (4, 6), (8, 1), (0, 0), (3, 3)]
hull = convex_hull(points)
print("Convex Hull:", 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:

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)]

Sol: from itertools import permutations

```
import math


def distance(city1, city2):
    return math.sqrt((city1[0] - city2[0]) ** 2 + (city1[1] - city2[1]) ** 2)


def tsp(cities):
    n = len(cities)
    min_distance = float('inf')
    best_path = []

    for perm in permutations(cities[1:]):
        current_path = [cities[0]] + list(perm) + [cities[0]]
        current_distance = sum(distance(current_path[i], current_path[i + 1]) for i in
range(n))

        if current_distance < min_distance:
            min_distance = current_distance
            best_path = current_path

    return min_distance, best_path


# Test cases
cities1 = [(1, 2), (4, 5), (7, 1), (3, 6)]
cities2 = [(2, 4), (8, 1), (1, 7), (6, 3), (5, 9)]
```

print("Test Case 1:\nShortest Distance:", tsp(cities1)[0], "\nShortest Path:", tsp(cities1)[1])

print("Test Case 2:\nShortest Distance:", tsp(cities2)[0], "\nShortest Path:", tsp(cities2)[1])

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).

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 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

Sol:
```
from itertools import permutations


def total_cost(assignment, cost_matrix):

    return sum(cost_matrix[i][assignment[i]] for i in range(len(assignment)))


def assignment_problem(cost_matrix):

    n = len(cost_matrix)

    min_cost = float('inf')

    best_assignment = []


    for perm in permutations(range(n)):

        current_cost = total_cost(perm, cost_matrix)

        if current_cost < min_cost:

            min_cost = current_cost

            best_assignment = perm


    return [(i + 1, best_assignment[i] + 1) for i in range(n)], min_cost


# Test cases

cost_matrix1 = [[3, 10, 7], [8, 5, 12], [4, 6, 9]]
```

```
cost_matrix2 = [[15, 9, 4], [8, 7, 18], [6, 12, 11]]


print("Test Case 1:", assignment_problem(cost_matrix1))

print("Test Case 2:", assignment_problem(cost_matrix2))
```

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

Sol:

```
from itertools import combinations

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(weights, values, capacity):
    n = len(weights)
    best_value = 0
    best_combination = []

    for r in range(n + 1):
        for combination in combinations(range(n), r):
            if is_feasible(combination, weights, capacity):
                current_value = total_value(combination, values)
                if current_value > best_value:
                    best_value = current_value
                    best_combination = combination

    return best_combination, best_value

# Test cases
print("Test Case 1:", knapsack([2, 3, 1], [4, 5, 3], 4))
print("Test Case 2:", knapsack([1, 2, 3, 4], [2, 4, 6, 3], 6))
```