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]**

```
 # An empty list

empty_list = []

print(empty_list)

# A list with one element

single_element_list = [1]

print(single_element_list)

# A list with all identical elements

identical_elements_list = [7, 7, 7, 7]

print(identical_elements_list)

# A list with negative numbers

negative_numbers_list = [-5, -1, -3, -2, -4]

negative_numbers_list.sort()

print(negative_numbers_list)
```

output:

[]

[1]

[7, 7, 7, 7]

[-5, -4, -3, -2, -1]

Algorithm:

- Create an empty list.
- Create a list with one element and print it.
- Create a list with identical elements and print it.
- Create a list with negative numbers.
- Sort the list with negative numbers and print it.

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]

```
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]

# Sorting a Random Array

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

selection_sort(arr)

print("Random Array Sorted:", arr)

# Sorting a Reverse Sorted Array

arr = [10, 8, 6, 4, 2]

selection_sort(arr)

print("Reverse Sorted Array Sorted:", arr)

# Sorting an Already Sorted Array

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

selection_sort(arr)

print("Already Sorted Array Sorted:", arr)
```

output:

```
Random Array Sorted: [1, 2, 5, 5, 6, 9]
```

Reverse Sorted Array Sorted: [2, 4, 6, 8, 10]

Already Sorted Array Sorted: [1, 2, 3, 4, 5]

Algorithm:

- Define a selection sort function that iterates over the array, finding the minimum element in the unsorted part and swapping it with the first unsorted element.
- Initialize a random array and sort it using the selection sort function.
- Print the sorted random array.
- Initialize a reverse sorted array and sort it using the selection sort function.
- Print the sorted reverse array, initialize an already sorted array, sort it using the selection sort function, and print the result.

3.Write code to modify bubble_sort function to stop early if the list becomes sorted before all passes are completed.
Test Cases:
Input: [64, 25, 12, 22, 11]
Expected Output: [11, 12, 22, 25, 64]
Input: [29, 10, 14, 37, 13]
Expected Output: [10, 13, 14, 29, 37]
Input: [3, 5, 2, 1, 4]
Expected Output: [1, 2, 3, 4, 5]
Input: [1, 2, 3, 4, 5] (Already sorted list)
Expected Output: [1, 2, 3, 4, 5]
Input: [5, 4, 3, 2, 1] (Reverse sorted list)
Expected Output: [1, 2, 3, 4, 5]

```
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

print(bubble_sort_optimized([64, 25, 12, 22, 11]))
print(bubble_sort_optimized([1, 2, 3, 4, 5]))
print(bubble_sort_optimized([5, 4, 3, 2, 1]))
```

output:

[11, 12, 22, 25, 64]

[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5]

Algorithm:

- Define an optimized bubble sort function that iterates over the array, swapping adjacent elements if they are in the wrong order.
- Use a flag to detect if any swaps were made during the iteration; if no swaps were made, terminate early.
- Sort and print an unsorted array using the bubble sort function.

- Sort and print an already sorted array using the bubble sort function.
- Sort and print a reverse sorted array using the bubble sort function.

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

```
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
# Array with Duplicates
input_arr1 = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3]
output_arr1 = insertion_sort(input_arr1)
print(output_arr1)
# All Identical Elements
input_arr2 = [5, 5, 5, 5, 5]
output_arr2 = insertion_sort(input_arr2)
print(output_arr2)
# Mixed Duplicates
input_arr3 = [2, 3, 1, 3, 2, 1, 1, 3]
```

```
output_arr3 = insertion_sort(input_arr3)

print(output_arr3)
```

output:

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

[5, 5, 5, 5, 5]

[1, 1, 1, 2, 2, 3, 3, 3]

Algorithm:

- Define an insertion sort function that iterates over the array starting from the second element.
- For each element, compare it with elements before it and insert it into the correct position in the sorted portion of the array.
- Initialize an array with duplicates and sort it using the insertion sort function.
- Initialize an array with all identical elements and sort it using the insertion sort function.
- Initialize an array with mixed duplicates and sort it using the insertion sort function.

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.

```
def findKthPositive(arr, k):
    missing = []
    num = 1
    while len(missing) < k:
        if num not in arr:
            missing.append(num)
        num += 1
    return missing[-1]
# Example 1
arr1 = [2, 3, 4, 7, 11]
k1 = 5
output1 = findKthPositive(arr1, k1)
print(output1)
```

Example 2

arr2 = [1, 2, 3, 4]

k2 = 2

output2 = findKthPositive(arr2, k2)

print(output2)

output:

9

6

Algorithm:

- Initialize an empty list missing to keep track of missing numbers and set num to 1.
- Loop until the length of missing is less than k:
  - If num is not in arr, append num to missing.
  - Increment num by 1.
- Return the last element in missing.

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.

```
def find_peak(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

nums=[1,2,3,1]

print(find_peak(nums))
```

output:2

Algorithm:

- Initialize left to 0 and right to the last index of nums.
- Loop while left is less than right:
  - Calculate mid as the average of left and right.
  - If the middle element nums[mid] is less than the next element nums[mid + 1], set left to mid + 1.
  - Otherwise, set right to mid.
- Return left as the index of the peak element.

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.

```
haystack = "leetcode"

needle = "leeto"

index = haystack.find(needle)

print("index:",index)
```

output:

index: -1

Algorithm:

- Assign the string "leetcode" to the variable haystack.
- Assign the string "leeto" to the variable needle.
- Use the find method of haystack to search for needle and store the result in index.
- Print the label "index:" followed by the value of index.
- End.

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.
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.

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

output = [word for word in words if any(word in other_word for other_word in words if word != other_word)]

print(output)

Algorithm:

- nitialize the list words with the given words.
- Create a list comprehension to iterate over each word in words.
- Inside the list comprehension, use any to check if the word is a substring of any other_word in words.
- Ensure word is not compared with itself by adding the condition if word != other_word.
- Print the resulting list output.

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

```
import math

def brute_force_closest_pair(points):

    min_distance = float('inf')

    pair = None

    for i in range(len(points)):

        for j in range(i + 1, len(points)):

            distance = math.sqrt((points[i][0] - points[j][0])**2 + (points[i][1] - points[j][1])**2)

            if distance < min_distance:

                min_distance = distance

                pair = (points[i], points[j])
```

```
    return pair, min_distance

# Input points

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

# Find the closest pair of points using brute force

closest_pair, min_distance = brute_force_closest_pair(points)

# Output the result

print(f"Closest pair: {closest_pair[0]} - {closest_pair[1]} Minimum distance: {min_distance}")
```

output:

Closest pair: (1, 2) - (3, 1) Minimum distance: 2.23606797

Algorithm:

- Initialize min_distance to infinity and pair to None.
- Loop over each point i in points.
- For each point i, loop over each subsequent point j in points.
- Calculate the Euclidean distance between points i and j, update min_distance and pair if this distance is smaller.
- Return the closest pair and the minimum 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

```
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
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(points)
print(closest_pair)
```

output:

((9, 3.5), (7.5, 4.5))

Algorithm:

- Initialize min_distance to infinity and closest_pair to None.
- Loop over each point i in points.
- For each point i, loop over each subsequent point j in points.
- Calculate the Euclidean distance between points i and j using the euclidean_distance function, updating min_distance and closest_pair if this distance is smaller.
- Return 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)]
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
    return 1 if val > 0 else 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
# Input Points
points = [(1, 1), (4, 6), (8, 1), (0, 0), (3, 3)]
# Finding Convex Hull
convex_hull_points = convex_hull(points)
# Output Convex Hull Points
print("Convex Hull:", convex_hull_points)
```

output:

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

Algorithm:

- Calculate the number of points n in the input points.
- If n is less than 3, return points as the hull since a convex hull requires at least 3 points.
- Initialize an empty list hull to store the points of the convex hull.
- Find the leftmost point l in points based on the x-coordinate.
- Use a loop to construct the convex hull:

    - Start from p = l and add points[p] to hull.
    - Find the next point q such that it maximizes the angle formed by points[p], points[q], and points[i] (where i iterates through all points).
    - Update p to q and continue until returning to the starting point l.

- Return hull as the convex hull of points.

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:
Define a function distance(city1, city2) to calculate the distance between two cities (e.g., Euclidean distance).
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).
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:
Simple Case: Four cities with basic coordinates (e.g., [(1, 2), (4, 5), (7, 1), (3, 6)])
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)]

```
import itertools

import math

def distance(city1, city2):

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

def tsp(cities):

    min_distance = float('inf')

    shortest_path = None

    for perm in itertools.permutations(cities[1:]):

        total_distance = 0

        path = [cities[0]] + list(perm) + [cities[0]]

        for i in range(len(path) - 1):

            total_distance += distance(path[i], path[i+1])

        if total_distance < min_distance:

            min_distance = total_distance

            shortest_path = path

    return min_distance, shortest_path
```

Test Cases

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)

output:

Test Case 1:

Shortest Distance: 16.969112047670894

Shortest Path: [(1, 2), (7, 1), (4, 5), (3, 6), (1, 2)]

Algorithm:

- Initialize min_distance to infinity and shortest_path to None.
- Iterate over all permutations of cities[1:] using itertools.permutations.
- For each permutation, compute the total distance of the path that starts and ends at cities[0].
- Update min_distance and shortest_path if the current path distance is shorter than min_distance.
- Return min_distance and shortest_path as the shortest distance and path found for the TSP.

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
Simple Case: Cost Matrix:
[[3, 10, 7],
[8, 5, 12],
[4, 6, 9]]
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

import itertools

def total_cost(assignment, cost_matrix):

    return sum(cost_matrix[worker][task] for worker, task in assignment)

```
def assignment_problem(cost_matrix):
    workers = range(len(cost_matrix))
    min_cost = float('inf')
    optimal_assignment = None
    for permutation in itertools.permutations(workers):
        assignment = list(zip(permutation, range(len(cost_matrix))))
        cost = total_cost(assignment, cost_matrix)
        if cost < min_cost:
            min_cost = cost
            optimal_assignment = assignment
    return optimal_assignment, min_cost
# Test Cases
cost_matrix_1 = [[3, 10, 7], [8, 5, 12], [4, 6, 9]]
optimal_assignment_1, total_cost_1 = assignment_problem(cost_matrix_1)
print("Test Case 1:")
print("Optimal Assignment:", [(f"worker {worker + 1}", f"task {task + 1}") for worker, task in
optimal_assignment_1])
print("Total Cost:", total_cost_1)
```

output:

Test Case 1:

Optimal Assignment: [('worker 3', 'task 1'), ('worker 2', 'task 2'), ('worker 1', 'task 3')]

Total Cost: 16

Algorithm:

- Initialize workers as a range from 0 to the length of cost_matrix.
- Initialize min_cost to infinity and optimal_assignment to None.
- Iterate through all permutations of workers using itertools.permutations.
- For each permutation, create an assignment list where each worker is assigned to a task based on the permutation.
- Calculate the total cost using the total_cost function and update min_cost and optimal_assignment if the current assignment has a lower cost.
- Return optimal_assignment and min_cost as the optimal assignment and total cost found.

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**

```
items = 3

weights = [2, 3, 1]

values = [4, 5, 3]

capacity = 4

max_value = 0

best_set = []

for i in range(2**items):

    selected_items = [j for j in range(items) if (i & (1 << j))]

    total_weight = sum(weights[j] for j in selected_items)

    total_val = sum(values[j] for j in selected_items)

    if total_weight <= capacity and total_val > max_value:

        max_value = total_val

        best_set = selected_items

print("Best set of items:", best_set)

print("Total value:", max_value)
```

output:

Best set of items: [1, 2]

Total value: 8

Algorithm:

- Initialize items to 3, weights to [2, 3, 1], values to [4, 5, 3], and capacity to 4.
- Initialize max_value to 0 and best_set to an empty list.
- Iterate over all possible subsets of items using range(2**items).
- For each subset represented by i, compute selected_items which are items included in the subset.

- Calculate total_weight and total_val for the selected items. Update max_value and best_set if the current subset's total weight is within capacity and its total value is greater than max_value.
- Print the best_set and max_value.