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]

Program:

```
def process_list(input_list):

   if not input_list:

      # If the list is empty

      return input_list

   elif len(input_list) == 1:

      # If the list has only one element

      return input_list

   elif all(x == input_list[0] for x in input_list):

      # If all elements in the list are identical

      return input_list

   else:

      # If the list contains negative numbers

      return sorted(input_list)



# Test cases

test_cases = [
   ([], []),

   ([1], [1]),

   ([7, 7, 7, 7], [7, 7, 7, 7]),

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

]


for i, (input_list, expected_output) in enumerate(test_cases, 1):

   output = process_list(input_list)

   assert output == expected_output, f"Test case {i} failed: expected {expected_output}, got
{output}"

   print(f"Test case {i} passed: {output}")


custom_tests = [

   ([10, 10, 10], [10, 10, 10]),

   ([-10, 0, 5, -20, 15], [-20, -10, 0, 5, 15]),

   ([], []),

   ([3], [3])

]


for i, input_list in enumerate(custom_tests, 1):

   output = process_list(input_list)

   print(f"Custom test case {i}: {output}")
```

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


# Test cases


# Sorting a Random Array

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

print("Random Array Before Sorting:", random_array)

sorted_random_array = selection_sort(random_array)

print("Random Array After Sorting:", sorted_random_array)


# Sorting a Reverse Sorted Array

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

print("\nReverse Sorted Array Before Sorting:", reverse_sorted_array)

sorted_reverse_sorted_array = selection_sort(reverse_sorted_array)
```

```
print("Reverse Sorted Array After Sorting:", sorted_reverse_sorted_array)


# Sorting an Already Sorted Array

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

print("\nAlready Sorted Array Before Sorting:", already_sorted_array)

sorted_already_sorted_array = selection_sort(already_sorted_array)

print("Already Sorted Array After Sorting:", 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.

Program:

```
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

# Test cases

# Sorting a Random Array
random_array = [5, 2, 9, 1, 5, 6]
print("Random Array Before Sorting:", random_array)
sorted_random_array = bubble_sort(random_array)
print("Random Array After Sorting:", sorted_random_array)

# Sorting a Reverse Sorted Array
reverse_sorted_array = [10, 8, 6, 4, 2]
print("\nReverse Sorted Array Before Sorting:", reverse_sorted_array)
sorted_reverse_sorted_array = bubble_sort(reverse_sorted_array)
print("Reverse Sorted Array After Sorting:", sorted_reverse_sorted_array)

# Sorting an Already Sorted Array
already_sorted_array = [1, 2, 3, 4, 5]
print("\nAlready Sorted Array Before Sorting:", already_sorted_array)
sorted_already_sorted_array = bubble_sort(already_sorted_array)
```

```
print("Already Sorted Array After Sorting:", sorted_already_sorted_array)
```

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.

<u>Examples:</u>

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]

Program:

```
def insertion_sort(arr):

    n = len(arr)

    for i in range(1, n):

        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



# Test cases



# Array with Duplicates

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

print("Array with Duplicates Before Sorting:", array_with_duplicates)
```

```
sorted_array_with_duplicates = insertion_sort(array_with_duplicates)

print("Array with Duplicates After Sorting:", sorted_array_with_duplicates)



# All Identical Elements

all_identical_elements = [5, 5, 5, 5, 5]

print("\nAll Identical Elements Before Sorting:", all_identical_elements)

sorted_all_identical_elements = insertion_sort(all_identical_elements)

print("All Identical Elements After Sorting:", sorted_all_identical_elements)



# Mixed Duplicates

mixed_duplicates = [2, 3, 1, 3, 2, 1, 1, 3]

print("\nMixed Duplicates Before Sorting:", mixed_duplicates)

sorted_mixed_duplicates = insertion_sort(mixed_duplicates)

print("Mixed Duplicates After Sorting:", sorted_mixed_duplicates)
```

6.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.

Program:

```
def find_kth_missing_positive(arr, k):

   missing_count = 0

   current = 1

   i = 0



   while missing_count < k:
        if i < len(arr) and arr[i] == current:

            i += 1

        else:

            missing_count += 1

        if missing_count < k:

            current += 1


    return current



# Test cases



# Example 1

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

k1 = 5

print(f"Test case 1 - Expected Output: 9, Actual Output: {find_kth_missing_positive(arr1, k1)}")



# Example 2

arr2 = [1, 2, 3, 4]

k2 = 2

print(f"Test case 2 - Expected Output: 6, Actual Output: {find_kth_missing_positive(arr2, k2)}")
```

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.

Program:

```
def find_peak_element(nums):

    n = len(nums)

    left, right = 0, n - 1



    while left <= right:

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



        # Check if mid is a peak

        if (mid == 0 or nums[mid-1] <= nums[mid]) and (mid == n-1 or nums[mid] >= nums[mid+1]):

            return mid

        elif mid > 0 and nums[mid-1] > nums[mid]:

            right = mid - 1

        else:

            left = mid + 1



    return -1  # Normally unreachable, as per problem statement



# Example usage:

nums1 = [1, 2, 3, 1]

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



print(find_peak_element(nums1))  # Output: 2

print(find_peak_element(nums2))  # Output: 5
```

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.

Program:

```
def strStr(haystack, needle):

    n = len(haystack)

    m = len(needle)



    for i in range(n - m + 1):

        if haystack[i:i+m] == needle:

            return i



    return -1



# Example usage:

haystack1 = "sadbutsad"

needle1 = "sad"

print(strStr(haystack1, needle1))  # Output: 0



haystack2 = "leetcode"

needle2 = "leeto"

print(strStr(haystack2, needle2))  # Output: -1
```

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.

Program:

```
def substring_words(words):
    result = []
    n = len(words)

    for i in range(n):
        for j in range(n):
            if i != j and (words[i] in words[j] or words[j] in words[i]):
                if words[i] not in result:
                    result.append(words[i])
                if words[j] not in result:
                    result.append(words[j])

    return result

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

words2 = ["leetcode", "et", "code"]
print(substring_words(words2))  # Output: ["et", "code"]

words3 = ["blue", "green", "bu"]
print(substring_words(words3))  # 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

Program:

```
import math
```

```
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):

            x1, y1 = points[i]

            x2, y2 = points[j]

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

            if distance < min_distance:

                min_distance = distance

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


    return closest_pair, min_distance


# Example usage:

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

closest_pair, min_distance = closest_pair_brute_force(points)


if closest_pair:

    point1, point2 = closest_pair

    print(f"Closest pair: {point1} - {point2}")

    print(f"Minimum distance: {min_distance}")
```

```
else:

   print("No points or less than 2 points provided.")
```

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

program:

```
import math

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):
            x1, y1 = points[i]
            x2, y2 = points[j]
            distance = math.sqrt((x2 - x1)**2 + (y2 - y1)**2)
            if distance < min_distance:
                min_distance = distance
                closest_pair = (points[i], points[j])

    return closest_pair, min_distance

# Example usage:
points = [(1, 2), (4, 5), (7, 8), (3, 1)]
closest_pair, min_distance = closest_pair_brute_force(points)

if closest_pair:
    point1, point2 = closest_pair
    print(f"Closest pair: {point1} - {point2}")
    print(f"Minimum distance: {min_distance}")
else:
    print("No points or less than 2 points provided.")
```

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)]

Program:

```
def cross_product(o, a, b):
    """ Returns the cross product of vector OA and OB.
        A positive cross product indicates a counter-clockwise turn, 0 indicates a collinear point,
        and negative indicates a clockwise turn. """
    return (a[0] - o[0]) * (b[1] - o[1]) - (a[1] - o[1]) * (b[0] - o[0])

def convex_hull_brute_force(points):
    """ Returns the points on the convex hull of points in CCW order using the brute force
approach (Jarvis March). """
    n = len(points)
    if n < 3:
        return None

    # Sort points lexicographically (by x, then by y)
    points = sorted(points)

    hull = []

    p = 0
    while True:
        hull.append(points[p])

        q = (p + 1) % n
        for r in range(n):
            if cross_product(points[p], points[q], points[r]) < 0:
                q = r



        p = q
        if p == 0:
            break

    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:", 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:

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)]

Program:

```
import itertools

import math



def distance(city1, city2):

   """ Calculate Euclidean distance between two cities (points). """

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



def tsp(cities):

   """ Solve the Traveling Salesperson Problem using exhaustive search (brute force). """
```

```
n = len(cities)

if n < 2:

    return None, None


# Generate all permutations of cities (excluding starting city)

all_permutations = itertools.permutations(range(1, n))


min_distance = float('inf')

shortest_path = None


# Iterate over all permutations

for perm in all_permutations:

    # Construct the full path including the starting city at both start and end

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


    # Calculate total distance for this path

    total_distance = 0

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

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


    # Update minimum distance and shortest path if found a shorter path

    if total_distance < min_distance:

        min_distance = total_distance

        shortest_path = path


# Convert path indices to actual cities

shortest_path_cities = [cities[idx] for idx in shortest_path]
```

```
    return min_distance, shortest_path_cities


# Test Cases

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

cities2 = [(2, 4), (8, 1), (1, 7), (6, 3), (5, 9)]


# Execute TSP and print results for each test case

print("Test Case 1:")

shortest_distance, shortest_path = tsp(cities1)

print(f"Shortest Distance: {shortest_distance}")

print(f"Shortest Path: {shortest_path}")


print("\nTest Case 2:")

shortest_distance, shortest_path = tsp(cities2)

print(f"Shortest Distance: {shortest_distance}")

print(f"Shortest Path: {shortest_path}")
```

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
```

Program:

```
import itertools



def total_cost(assignment, cost_matrix):

    """ Calculate the total cost of the assignment using the given cost matrix. """

    total = 0

    for worker, task in assignment:

        total += cost_matrix[worker][task]

    return total



def assignment_problem(cost_matrix):

    """ Solve the assignment problem using exhaustive search (brute force). """

    num_workers = len(cost_matrix)

    num_tasks = len(cost_matrix[0])



    # Generate all permutations of worker indices

    all_permutations = itertools.permutations(range(num_tasks))



    min_cost = float('inf')

    optimal_assignment = None



    # Iterate over all permutations
    for perm in all_permutations:
        current_assignment = [(worker, perm[worker]) for worker in range(num_workers)]
        current_cost = total_cost(current_assignment, cost_matrix)

        # Update minimum cost and optimal assignment
        if current_cost < min_cost:
            min_cost = current_cost
            optimal_assignment = current_assignment

    # Prepare the output format for optimal assignment
    formatted_optimal_assignment = [(f"(worker {worker + 1}, task {task + 1})") for worker, task in
optimal_assignment]

    return formatted_optimal_assignment, 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]
]
```

```
# Execute assignment_problem and print results for each test case

print("Test Case 1:")

optimal_assignment, total_cost = assignment_problem(cost_matrix1)

print(f"Optimal Assignment: {optimal_assignment}")

print(f"Total Cost: {total_cost}")



print("\nTest Case 2:")

optimal_assignment, total_cost = assignment_problem(cost_matrix2)

print(f"Optimal Assignment: {optimal_assignment}")

print(f"Total Cost: {total_cost}")
```

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:

```
import itertools



def total_value(items, values):

    """ Calculate the total value of selected items using the given values list. """

    total = 0
    for item in items:
        total += values[item]
    return total

def is_feasible(items, weights, capacity):
    """ Check if selecting the items (given by their indices) does not exceed the knapsack capacity. """
    total_weight = sum(weights[item] for item in items)
    return total_weight <= capacity

def knapsack_01_exhaustive(weights, values, capacity):
    """ Solve the 0-1 Knapsack Problem using exhaustive search (brute force). """
    num_items = len(weights)

    # Generate all possible subsets of items
    all_subsets = []
    for r in range(num_items + 1):
        all_subsets.extend(itertools.combinations(range(num_items), r))

    max_value = 0
    optimal_subset = None

    # Iterate over all subsets
    for subset in all_subsets:
        if is_feasible(subset, weights, capacity):
            subset_value = total_value(subset, values)
            if subset_value > max_value:
                max_value = subset_value
            optimal_subset = subset

    # Prepare output format for optimal subset (indices or some representation)
    optimal_subset_indices = list(optimal_subset) if optimal_subset else []

    return max_value, optimal_subset_indices

# Test Cases
weights1 = [2, 3, 1]
values1 = [4, 5, 3]
capacity1 = 4

weights2 = [1, 2, 3, 4]
values2 = [2, 4, 6, 3]
capacity2 = 6

# Execute knapsack_01_exhaustive and print results for each test case
print("Test Case 1:")
max_value, optimal_subset = knapsack_01_exhaustive(weights1, values1, capacity1)
print(f"Maximum Value: {max_value}")
print(f"Optimal Subset: {optimal_subset}")

print("\nTest Case 2:")
max_value, optimal_subset = knapsack_01_exhaustive(weights2, values2, capacity2)
print(f"Maximum Value: {max_value}")
print(f"Optimal Subset: {optimal_subset}")
```