NAME : SUMALATHA D K

REG NO. : 192424023

COURSE CODE : CSA0613

COURSE NAME : DESIGN AND ANALYSIS OF ALGORITHMS FOR OPTIMAL APPLICATIONS

SLOT : A

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**

AIM:

To write a program that handles different types of lists (empty, single element, all identical, and with negative numbers) and sorts them when necessary.

ALGORITHM:

1. Start
2. Read the input list nums
3. Check the type of list:
   - If empty, return []
   - If list has one element, return the same list
   - Otherwise, sort the list in ascending order
4. Return the processed list
5. Stop

CODE:

```
def process_list(nums):
    if len(nums) == 0:
        return []
    elif len(nums) == 1:
        return nums
    else:
        return sorted(nums)

# Test cases
print(process_list([]))              # Empty list
print(process_list([1]))             # Single element
print(process_list([7, 7, 7, 7]))    # All identical
print(process_list([-5, -1, -3, -2, -4])) # Negative numbers
```

INPUT:

```
print(process_list([]))
print(process_list([1]))
print(process_list([7, 7, 7, 7]))
print(process_list([-5, -1, -3, -2, -4]))
```

OUTPUT:

```
[]
[1]
[7, 7, 7, 7]
[-5, -4, -3, -2, -1]
```

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.

AIM:

To explain and demonstrate the Selection Sort algorithm, showing how it sorts arrays into ascending order and analyzing its simplicity and inefficiency for large datasets.

ALGORITHM:

1. Start
2. Divide the array into a sorted and an unsorted region. Initially, the sorted region is empty, and the unsorted region contains all elements.
3. For each position i from 0 to n-1:
   - Find the smallest element in the unsorted region (from index i to n-1)
   - Swap this smallest element with the element at index i
   - Move the boundary of the sorted region one element to the right
4. Repeat until the entire array is sorted
5. Stop

CODE:

```
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
print(selection_sort([5, 2, 9, 1, 5, 6]))
print(selection_sort([10, 8, 6, 4, 2]))
print(selection_sort([1, 2, 3, 4, 5]))
```

INPUT:

```
print(selection_sort([5, 2, 9, 1, 5, 6]))
print(selection_sort([10, 8, 6, 4, 2]))
print(selection_sort([1, 2, 3, 4, 5]))
```

OUTPUT:

```
[1, 2, 5, 5, 6, 9]
[2, 4, 6, 8, 10]
[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.**

AIM:

To optimize the Bubble Sort algorithm by stopping early if the list becomes sorted before completing all passes, reducing unnecessary comparisons.

ALGORITHM:

1. Read the input array nums
2. Let n be the length of the array
3. For each pass i from 0 to n-1:
    - Initialize a flag swapped = False
    - For each index j from 0 to n-i-2:
        - If nums[j] > nums[j+1], swap them and set swapped = True

- If swapped remains False after the inner loop, break the outer loop (array is sorted)

4. Return the sorted array
5. Stop

CODE:

```
def bubble_sort_optimized(nums):
    n = len(nums)
    for i in range(n):
        swapped = False
        for j in range(0, n - i - 1):
            if nums[j] > nums[j + 1]:
                nums[j], nums[j + 1] = nums[j + 1], nums[j]
                swapped = True
        if not swapped:
            break
    return nums

# Test cases
print(bubble_sort_optimized([64, 25, 12, 22, 11]))
print(bubble_sort_optimized([29, 10, 14, 37, 13]))
print(bubble_sort_optimized([3, 5, 2, 1, 4]))
print(bubble_sort_optimized([1, 2, 3, 4, 5]))
print(bubble_sort_optimized([5, 4, 3, 2, 1]))
```

INPUT:

```
print(bubble_sort_optimized([64, 25, 12, 22, 11]))
print(bubble_sort_optimized([29, 10, 14, 37, 13]))
print(bubble_sort_optimized([3, 5, 2, 1, 4]))
print(bubble_sort_optimized([1, 2, 3, 4, 5]))
print(bubble_sort_optimized([5, 4, 3, 2, 1]))
```

OUTPUT:

```
[11, 12, 22, 25, 64]
[10, 13, 14, 29, 37]
[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5]
[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.**

AIM:

To implement Insertion Sort that correctly handles arrays with duplicate elements while preserving their relative order (stable sort), ensuring duplicates are sorted in ascending order.

ALGORITHM:

1. Start
2. Read the input array nums
3. For index i from 1 to n-1:
    - Store key = nums[i]
    - Initialize j = i - 1
    - While j >= 0 and nums[j] > key:
        - Shift nums[j] one position to the right
        - Decrement j
    - Place key at position j + 1
4. Continue until all elements are processed
5. Return the sorted array
6. Stop

CODE:

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

# Test cases
print(insertion_sort([3, 1, 4, 1, 5, 9, 2, 6, 5, 3]))  # Array with duplicates
print(insertion_sort([5, 5, 5, 5, 5]))              # All identical elements
print(insertion_sort([2, 3, 1, 3, 2, 1, 1, 3]))     # Mixed duplicates
```

INPUT:

```
print(insertion_sort([3, 1, 4, 1, 5, 9, 2, 6, 5, 3]))
print(insertion_sort([5, 5, 5, 5, 5]))
print(insertion_sort([2, 3, 1, 3, 2, 1, 1, 3]))
```

OUTPUT:

```
[1, 1, 2, 3, 3, 4, 5, 5, 6, 9]
[5, 5, 5, 5, 5]
[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.**

AIM:

To find the kth missing positive integer from a sorted array of strictly increasing positive integers.

ALGORITHM:

1. Start
2. Read the input array arr and integer k
3. Initialize missing_count = 0 and current = 1
4. Initialize an index i = 0 to traverse the array
5. Loop until missing_count < k:
    - If i < len(arr) and arr[i] == current:
        - Move to the next element in the array (i += 1)
    - Else:
        - Increment missing_count
        - If missing_count == k, return current
    - Increment current
6. Stop

CODE:

```
def find_kth_missing(arr, k):
    missing_count = 0
    current = 1
    i = 0
    n = len(arr)

    while True:
        if i < n and arr[i] == current:
            i += 1
        else:
            missing_count += 1
            if missing_count == k:
                return current
        current += 1

# Test cases
print(find_kth_missing([2, 3, 4, 7, 11], 5))
print(find_kth_missing([1, 2, 3, 4], 2))
```

INPUT: **OUTPUT:**

```
print(find_kth_missing([2, 3, 4, 7, 11], 5))
print(find_kth_missing([1, 2, 3, 4], 2))
```

```
9
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.**

AIM:

To find a peak element in an array where a peak is defined as an element strictly greater than its neighbors, using an algorithm with O(log n) time complexity.

ALGORITHM:

1. Start
2. Read the input array nums

3. Initialize left = 0 and right = len(nums) - 1
4. While left < right:
    - Calculate mid = (left + right) // 2
    - If nums[mid] < nums[mid + 1]:
        - Move left = mid + 1 (peak must be on the right)
    - Else:
        - Move right = mid (peak is at mid or on the left)
5. When left == right, return left as the index of a peak element
6. Stop

CODE:

```
def find_peak_element(nums):
    left, right = 0, len(nums) - 1

    while left < right:
        mid = (left + right) // 2
        if nums[mid] < nums[mid + 1]:
            left = mid + 1
        else:
            right = mid
    return left

# Test cases
print(find_peak_element([1, 2, 3, 1]))
print(find_peak_element([1, 2, 1, 3, 5, 6, 4]))
```

INPUT: **OUTPUT:**

```
print(find_peak_element([1, 2, 3, 1]))
print(find_peak_element([1, 2, 1, 3, 5, 6, 4]))
```

```
2
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.

AIM:

To find the index of the first occurrence of the string needle in the string haystack and return -1 if needle is not found.

ALGORITHM:

1. Start
2. Read input strings haystack and needle
3. If needle is empty, return 0
4. Loop over i from 0 to len(haystack) - len(needle):
    - If the substring haystack[i:i+len(needle)] equals needle, return i
5. If the loop completes without finding a match, return -1
6. Stop

CODE:

```
def str_str(haystack, needle):
    if not needle:
        return 0

    n, m = len(haystack), len(needle)

    for i in range(n - m + 1):
        if haystack[i:i + m] == needle:
            return i
    return -1

# Test cases
print(str_str("sadbutsad", "sad"))
print(str_str("leetcode", "leeto"))
```

INPUT: **OUTPUT:**

```
print(str_str("sadbutsad", "sad"))
print(str_str("leetcode", "leeto"))
```

```
0
-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**

AIM:

To find all strings in a given list that are substrings of at least one other string in the list.

ALGORITHM:

1. Start
2. Read the input list words
3. Initialize an empty list result to store substrings
4. For each word i in words:
    - For each word j in words where i != j:
        - If i is a substring of j, add i to result and break
5. Return result
6. Stop

CODE:

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

    for i in range(n):
        for j in range(n):
            if i != j and words[i] in words[j]:
                result.append(words[i])
                break
    return result

# Test cases
print(string_matching(["mass","as","hero","superhero"]))
print(string_matching(["leetcode","et","code"]))
print(string_matching(["blue","green","bu"]))
```

INPUT:	OUTPUT:
print(string_matching(["mass","as","hero","superhero"]))	['as', 'hero']
print(string_matching(["leetcode","et","code"]))	['et', 'code']
print(string_matching(["blue","green","bu"]))	[]

9. Write a program that finds the closest pair of points in a set of 2D points using the brute force approach.

AIM:

To find the closest pair of points in a set of 2D points using the brute-force approach, calculating the Euclidean distance between all possible pairs.

ALGORITHM:

1. Start
2. Read the list of points
3. Initialize min_distance = infinity and closest_pair = None
4. For each point i in points:
    - For each point j after i in points:
        - Compute Euclidean distance between i and j
        - If distance < min_distance:
            - Update min_distance and closest_pair
5. Return closest_pair and min_distance
6. Stop

CODE:

```python
import math
def closest_pair_brute(points):
    min_distance = float('inf')
    closest_pair = None
    n = len(points)

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

# Test case
points = [(1, 2), (4, 5), (7, 8), (3, 1)]
pair, dist = closest_pair_brute(points)
print(f"Closest pair: {pair[0]} - {pair[1]}, Minimum distance: {dist}")
```

INPUT:

```python
points = [(1, 2), (4, 5), (7, 8), (3, 1)]
pair, dist = closest_pair_brute(points)
print(f"Closest pair: {pair[0]} - {pair[1]}, Minimum distance: {dist}")
```

OUTPUT:

```
Closest pair: (1, 2) - (3, 1), Minimum distance: 2.23606797749979
```

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?

AIM:

To find the closest pair of points and the convex hull of a set of 2D points using brute-force algorithms and analyze the time complexity of the implementations.

ALGORITHM:

1. Start
2. Read the set of points
3. Initialize min_distance = infinity and closest_pair = None
4. For each point i in points:
    - For each point j after i:
        - Compute Euclidean distance between i and j
        - If distance < min_distance, update min_distance and closest_pair
5. Return closest_pair and min_distance
6. Read the set of points S
7. For each pair of points (p, q) in S:
    - Check if all other points lie on the same side of the line formed by (p, q)
    - If yes, (p, q) is an edge of the convex hull
8. Include all unique points from these edges in order to get the convex hull
9. Stop

CODE:

```python
import math
def euclidean_distance(p1, p2):
    return math.sqrt((p1[0] - p2[0])**2 + (p1[1] - p2[1])**2)
def closest_pair(points):
    min_distance = float('inf')
    closest = None
    n = len(points)
    for i in range(n):
        for j in range(i+1, n):
            dist = euclidean_distance(points[i], points[j])
            if dist < min_distance:
                min_distance = dist
                closest = (points[i], points[j])
    return closest, min_distance
def convex_hull(points):
    def cross(o, a, b):
        return (a[0]-o[0])*(b[1]-o[1]) - (a[1]-o[1])*(b[0]-o[0])
    points = sorted(points)
    lower = []
    for p in points:
        while len(lower) >= 2 and cross(lower[-2], lower[-1], p) <= 0:
            lower.pop()
        lower.append(p)
    upper = []
    for p in reversed(points):
        while len(upper) >= 2 and cross(upper[-2], upper[-1], p) <= 0:
            upper.pop()
        upper.append(p)
    return lower[:-1] + upper[:-1]
# Test cases
points_set = [(10,0),(11,5),(5,3),(9,3.5),(15,3),(12.5,7),(6,6.5),(7.5,4.5)]

pair, dist = closest_pair(points_set)
print(f"Closest pair: {pair}, Minimum distance: {dist}")
hull_points = convex_hull(points_set)
print("Convex hull points in order:", hull_points)
```

INPUT:

```
points_set = [(10,0),(11,5),(5,3),(9,3.5),(15,3),(12.5,7),(6,6.5),(7.5,4.5)]
```

OUTPUT:

```
Closest pair: ((9, 3.5), (7.5, 4.5)), Minimum distance: 1.8027756377319946
Convex hull points in order: [(5, 3), (10, 0), (15, 3), (12.5, 7), (6, 6.5)]
```

11. Write a program that finds the convex hull of a set of 2D points using the brute force approach.

AIM:

To find the convex hull of a set of 2D points using the brute-force approach and return the points forming the hull in counter-clockwise order.

ALGORITHM:

1. Start
2. Read the list of points points
3. Initialize an empty list hull
4. For each pair of points (p, q) in points:
    - Assume the line through (p, q) forms an edge of the convex hull
    - For all other points r in points:
        - Compute the orientation (cross product) of (p, q, r)
        - If all points lie on the same side of the line (p, q), keep (p, q) as an edge
5. Add all unique points from these edges to hull
6. Sort the hull points in counter-clockwise order (optional using polar angle from centroid)
7. Return hull
8. Stop
9.

CODE:

```
import math
def cross(o, a, b):
    return (a[0]-o[0])*(b[1]-o[1]) - (a[1]-o[1])*(b[0]-o[0])
```

```
def convex_hull_bruteforce(points):
    n = len(points)
    hull_points = set()
    for i in range(n):
        for j in range(i+1, n):
            left = right = False
            for k in range(n):
                if k == i or k == j:
                    continue
                val = cross(points[i], points[j], points[k])
                if val > 0:
                    left = True
                elif val < 0:
                    right = True
            if not (left and right):
                hull_points.add(points[i])
                hull_points.add(points[j])

    hull_list = list(hull_points)
    cx = sum(x for x, y in hull_list)/len(hull_list)
    cy = sum(y for x, y in hull_list)/len(hull_list)
    hull_list.sort(key=lambda p: math.atan2(p[1]-cy, p[0]-cx))
    return hull_list
```

INPUT:

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

OUTPUT:

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

12. You are given a list of cities represented by their coordinates. Develop a program that utilizes exhaustive search to solve the TSP.

AIM:

To solve the Traveling Salesman Problem (TSP) using exhaustive search by generating all permutations of cities and finding the shortest possible route that visits each city once and returns to the starting city.

ALGORITHM:

1. Start
2. Read the list of cities as coordinates
3. Define distance(city1, city2) to calculate Euclidean distance
4. Fix the starting city as the first city in the list
5. Generate all permutations of the remaining cities using itertools.permutations
6. For each permutation:
    - Calculate total distance including returning to the starting city
    - If total distance < current minimum, update minimum and store the path
7. Return the minimum distance and corresponding path including the starting city at beginning and end
8. Stop

CODE:

```
import itertools
import math

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

# Exhaustive TSP solver
def tsp(cities):
    start = cities[0]
    min_dist = float('inf')
    shortest_path = []
```

```
    for perm in itertools.permutations(cities[1:]):
        path = [start] + list(perm) + [start]
        total_dist = sum(distance(path[i], path[i+1]) for i in range(len(path)-1))
        if total_dist < min_dist:
            min_dist = total_dist
            shortest_path = path
    return min_dist, shortest_path
```

INPUT:

```
# Test Case 1
cities1 = [(1, 2), (4, 5), (7, 1), (3, 6)]
dist1, path1 = tsp(cities1)
print("Test Case 1:")
print("Shortest Distance:", dist1)
print("Shortest Path:", path1)

# Test Case 2
cities2 = [(2, 4), (8, 1), (1, 7), (6, 3), (5, 9)]
dist2, path2 = tsp(cities2)
print("\nTest Case 2:")
print("Shortest Distance:", dist2)
print("Shortest Path:", path2)
```

OUTPUT:

```
Test Case 1:
Shortest Distance: 16.969112047670894
Shortest Path: [(1, 2), (7, 1), (4, 5), (3, 6), (1, 2)]

Test Case 2:
Shortest Distance: 23.12995011084934
Shortest Path: [(2, 4), (6, 3), (8, 1), (5, 9), (1, 7), (2, 4)]
```

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).

AIM:

To solve the assignment problem using exhaustive search by generating all possible worker-task assignments and selecting the one with the minimum total cost.

ALGORITHM:

1. Start
2. Read the cost matrix
3. Define total_cost(assignment, cost_matrix) to sum the costs of a given worker-task assignment
4. Generate all permutations of task indices using itertools.permutations
5. For each permutation:
    - Calculate total cost of assigning worker i to task perm[i]
    - If cost < current minimum, update minimum and store the assignment
6. Return the optimal assignment and the minimum total cost
7. Stop

CODE:

```
import itertools
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')
    optimal_assignment = None
    for perm in itertools.permutations(range(n)):
        cost = total_cost(perm, cost_matrix)
        if cost < min_cost:
            min_cost = cost
            optimal_assignment = perm
    assignment_pairs = [(i+1, task+1) for i, task in enumerate(optimal_assignment)]
    return assignment_pairs, min_cost

assignment2, cost2 = assignment_problem(cost_matrix2)
print("\nTest Case 2:")
print("Optimal Assignment:", assignment2)
print("Total Cost:", cost2)
```

INPUT:

```
# Test Case 1
cost_matrix1 = [
    [3, 10, 7],
    [8, 5, 12],
    [4, 6, 9]
]
assignment1, cost1 = assignment_problem(cost_matrix1)
print("Test Case 1:")
print("Optimal Assignment:", assignment1)
print("Total Cost:", cost1)

# Test Case 2
cost_matrix2 = [
    [15, 9, 4],
    [8, 7, 18],
    [6, 12, 11]
]
```

OUTPUT:

Test Case 1:

Optimal Assignment: [(1, 3), (2, 2), (3, 1)]

Total Cost: 16

Test Case 2:

Optimal Assignment: [(1, 3), (2, 2), (3, 1)]

Total Cost: 17

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:

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.

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.

AIM:

To solve the 0-1 Knapsack Problem using exhaustive search by generating all subsets of items and selecting the combination with maximum total value that does not exceed the knapsack capacity.

ALGORITHM:

1. Start

2. Read the list of item weights, values, and knapsack capacity
3. Define total_value(items, values) to sum the values of selected items
4. Define is_feasible(items, weights, capacity) to check if total weight ≤ capacity
5. Generate all possible subsets of item indices using itertools.combinations
6. For each subset:
    - Check feasibility using is_feasible
    - If feasible, calculate total value
    - If total value > current maximum, update maximum and store the subset
7. Return the optimal subset of items and corresponding maximum value
8. Stop

CODE:

```
import itertools
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)
    max_value = 0
    best_selection = []
    for r in range(1, n+1):
        for subset in itertools.combinations(range(n), r):
            if is_feasible(subset, weights, capacity):
                val = total_value(subset, values)
                if val > max_value:
                    max_value = val
                    best_selection = list(subset)
    return best_selection, max_value
```

INPUT:

```
# Test Case 1
weights1 = [2, 3, 1]
values1 = [4, 5, 3]
capacity1 = 4
selection1, value1 = knapsack(weights1, values1, capacity1)
print("Test Case 1:")
print("Optimal Selection:", selection1)
print("Total Value:", value1)

# Test Case 2
weights2 = [1, 2, 3, 4]
values2 = [2, 4, 6, 3]
capacity2 = 6
selection2, value2 = knapsack(weights2, values2, capacity2)
print("\nTest Case 2:")
print("Optimal Selection:", selection2)
print("Total Value:", value2)
```

OUTPUT:

```
Test Case 1:
Optimal Selection: [1, 2]
Total Value: 8

Test Case 2:
Optimal Selection: [0, 1, 2]
Total Value: 12
```