NAME: THAMIZHARASAN S
REG NO: 192424087
COURSE CODE: CSA0613
COURSE NAME: DESIGN AND ANALYSIS OF ALGORITHMS FOR OPTIMAL APPLICATIONS

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: []
1. Input: [1]
• Expected Output: [1]
2. Input: [7, 7, 7, 7]
• Expected Output: [7, 7, 7, 7]
3. Input: [-5, -1, -3, -2, -4]
• Expected Output: [-5, -4, -3, -2, -1]

Aim:
To write a Python program to sort a list and handle different cases such as an empty list, a list with one element, a list with identical elements, and a list containing negative numbers.

Algorithm:

- Start the program.
- Create a list of test cases.
- For each list:
    - Sort the list in ascending order.
- Display the sorted list as output.
- Stop the program

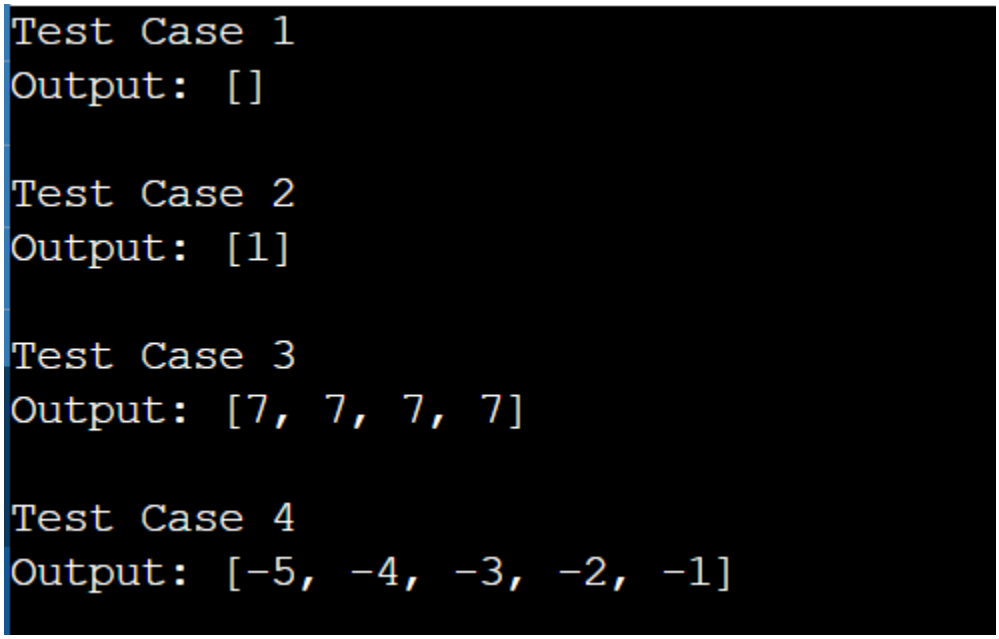
Program:

```
test_cases = [
    [],
    [1],
    [7, 7, 7, 7],
    [-5, -1, -3, -2, -4]
]
for i in range(len(test_cases)):
    lst = test_cases[i]
    lst.sort()
    print(f"Test Case {i+1}")
    print("Output:", lst)
    print()
```

Sample Input:

Test Cases:
1. Input: []
• Expected Output: []
1. Input: [1]
• Expected Output: [1]
2. Input: [7, 7, 7, 7]
• Expected Output: [7, 7, 7, 7]
3. Input: [-5, -1, -3, -2, -4]
• Expected Output: [-5, -4, -3, -2, -1]

Output:



Result:
The program executed successfully and produced the expected sorted output for all given test cases, including empty, single-element, identical, and negative-number lists.

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]

Aim:
To implement the Selection Sort algorithm in Python to sort a given array of elements in ascending order and to observe its behavior on random, reverse-sorted, and already sorted arrays.

Algorithm:

- Start from the first element of the array.
- Find the smallest element in the unsorted part of the array.
- Swap it with the first unsorted element.
- Move the boundary of the sorted part one position to the right.
- Repeat until the entire array is sorted.

Program:

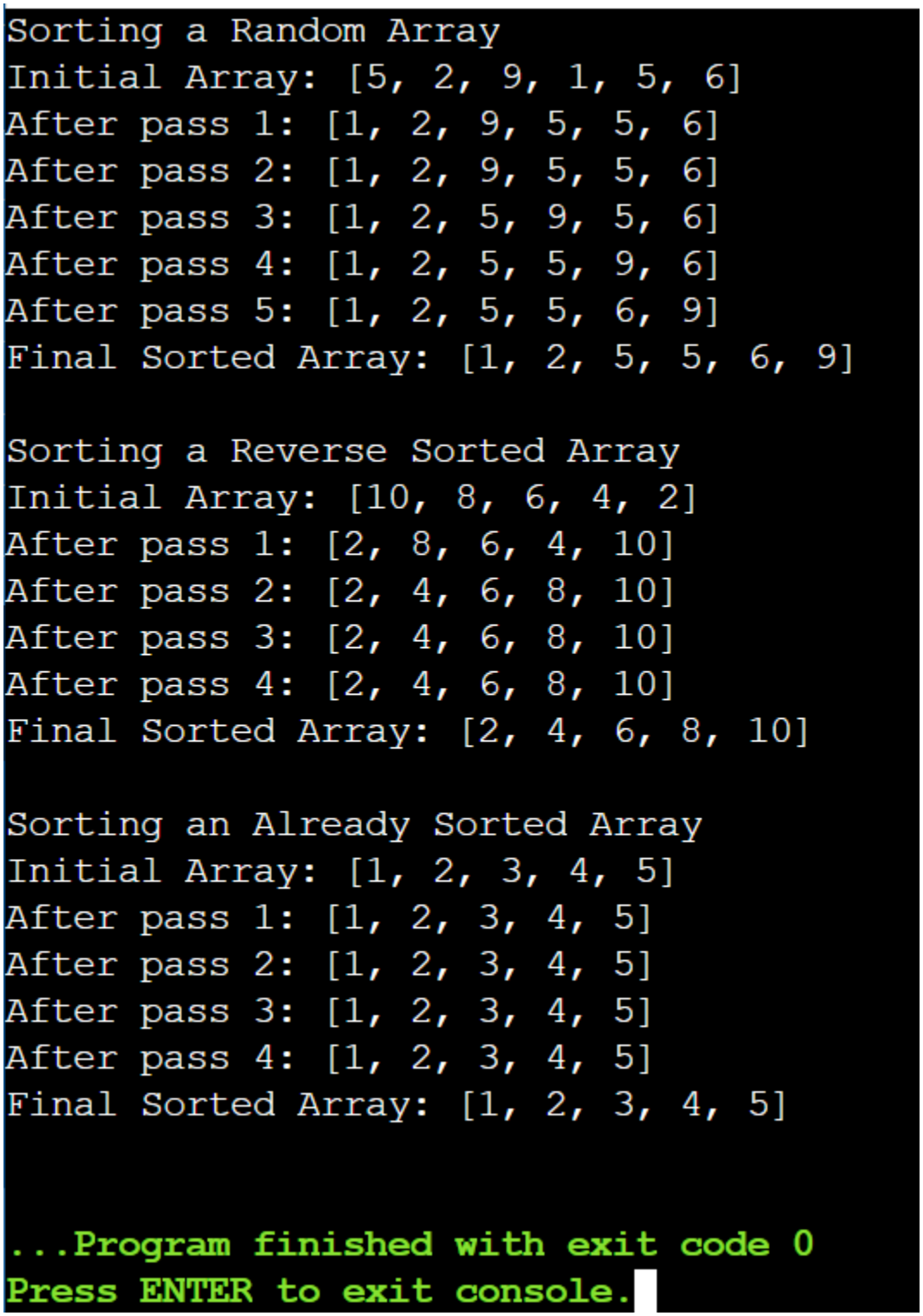
```
arr = [5, 2, 9, 1, 5, 6]
n = len(arr)

print("Sorting a Random Array")
print("Initial Array:", arr)

for i in range(n - 1):
    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]
    print(f"After pass {i + 1}:", arr)

print("Final Sorted Array:", arr)

arr = [10, 8, 6, 4, 2]
n = len(arr)

print("\nSorting a Reverse Sorted Array")
print("Initial Array:", arr)
```

```
for i in range(n - 1):
    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]
    print(f"After pass {i + 1}:", arr)

print("Final Sorted Array:", arr)

arr = [1, 2, 3, 4, 5]
n = len(arr)

print("\nSorting an Already Sorted Array")
print("Initial Array:", arr)

for i in range(n - 1):
    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]
    print(f"After pass {i + 1}:", arr)

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

Sample Input:

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]

Output:


Sorting a Random Array
Initial Array: [5, 2, 9, 1, 5, 6]
After pass 1: [1, 2, 9, 5, 5, 6]
After pass 2: [1, 2, 9, 5, 5, 6]
After pass 3: [1, 2, 5, 9, 5, 6]
After pass 4: [1, 2, 5, 5, 9, 6]
After pass 5: [1, 2, 5, 5, 6, 9]
Final Sorted Array: [1, 2, 5, 5, 6, 9]
Sorting a Reverse Sorted Array
Initial Array: [10, 8, 6, 4, 2]
After pass 1: [2, 8, 6, 4, 10]
After pass 2: [2, 4, 6, 8, 10]
After pass 3: [2, 4, 6, 8, 10]
After pass 4: [2, 4, 6, 8, 10]
Final Sorted Array: [2, 4, 6, 8, 10]
Sorting an Already Sorted Array
Initial Array: [1, 2, 3, 4, 5]
After pass 1: [1, 2, 3, 4, 5]
After pass 2: [1, 2, 3, 4, 5]
After pass 3: [1, 2, 3, 4, 5]
After pass 4: [1, 2, 3, 4, 5]
Final Sorted Array: [1, 2, 3, 4, 5]
...Program finished with exit code 0
Press ENTER to exit console.

Result:
The Selection Sort algorithm successfully sorted the given arrays in ascending order, producing the correct output for random, reverse-sorted, and already sorted inputs.

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]

Aim:
To implement an optimized Bubble Sort algorithm in Python that stops early if the list becomes sorted before completing all passes, and to sort different input lists in ascending order.

Algorithm:

- Start from the first element of the array.
- Compare each pair of adjacent elements in the unsorted part.
- Swap them if they are in the wrong order.
- If no swaps occur during a pass, stop the algorithm early (list is already sorted).
- Repeat until the array is fully sorted.

Program:
```
arr = [64, 25, 12, 22, 11]
n = len(arr)
print("Original Array:", arr)

for i in range(n - 1):
    swapped = False
    for j in range(n - i - 1):
        if arr[j] > arr[j + 1]
```

```
  arr[j], arr[j + 1] = arr[j + 1], arr[j]
            swapped = True
    print(f"After pass {i + 1}: {arr}")
    if not swapped:
        break

print("Sorted Array:", arr)
print("\n-----------------------------\n")

arr = [29, 10, 14, 37, 13]
n = len(arr)
print("Original Array:", arr)

for i in range(n - 1):
    swapped = False
    for j in range(n - i - 1):
        if arr[j] > arr[j + 1]:
            arr[j], arr[j + 1] = arr[j + 1], arr[j]
            swapped = True
    print(f"After pass {i + 1}: {arr}")
    if not swapped:
        break

print("Sorted Array:", arr)
print("\n-----------------------------\n")

arr = [3, 5, 2, 1, 4]
n = len(arr)
print("Original Array:", arr)

for i in range(n - 1):
    swapped = False
    for j in range(n - i - 1):
        if arr[j] > arr[j + 1]:
            arr[j], arr[j + 1] = arr[j + 1], arr[j]
            swapped = True
    print(f"After pass {i + 1}: {arr}")
    if not swapped:
        break
print("Sorted Array:", arr)
```

```
print("\n-----------------------------\n")
arr = [1, 2, 3, 4, 5]
n = len(arr)
print("Original Array:", arr)

for i in range(n - 1):
    swapped = False
    for j in range(n - i - 1):
        if arr[j] > arr[j + 1]:
            arr[j], arr[j + 1] = arr[j + 1], arr[j]
            swapped = True
    print(f"After pass {i + 1}: {arr}")
    if not swapped:
        break

print("Sorted Array:", arr)
print("\n-----------------------------\n")

arr = [5, 4, 3, 2, 1]
n = len(arr)
print("Original Array:", arr)

for i in range(n - 1):
    swapped = False
    for j in range(n - i - 1):
        if arr[j] > arr[j + 1]:
            arr[j], arr[j + 1] = arr[j + 1], arr[j]
            swapped = True
    print(f"After pass {i + 1}: {arr}")
    if not swapped:
        break

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

Sample Input:

Test Cases:

• Test your optimized function with the following lists:

1. Input: [64, 25, 12, 22, 11]

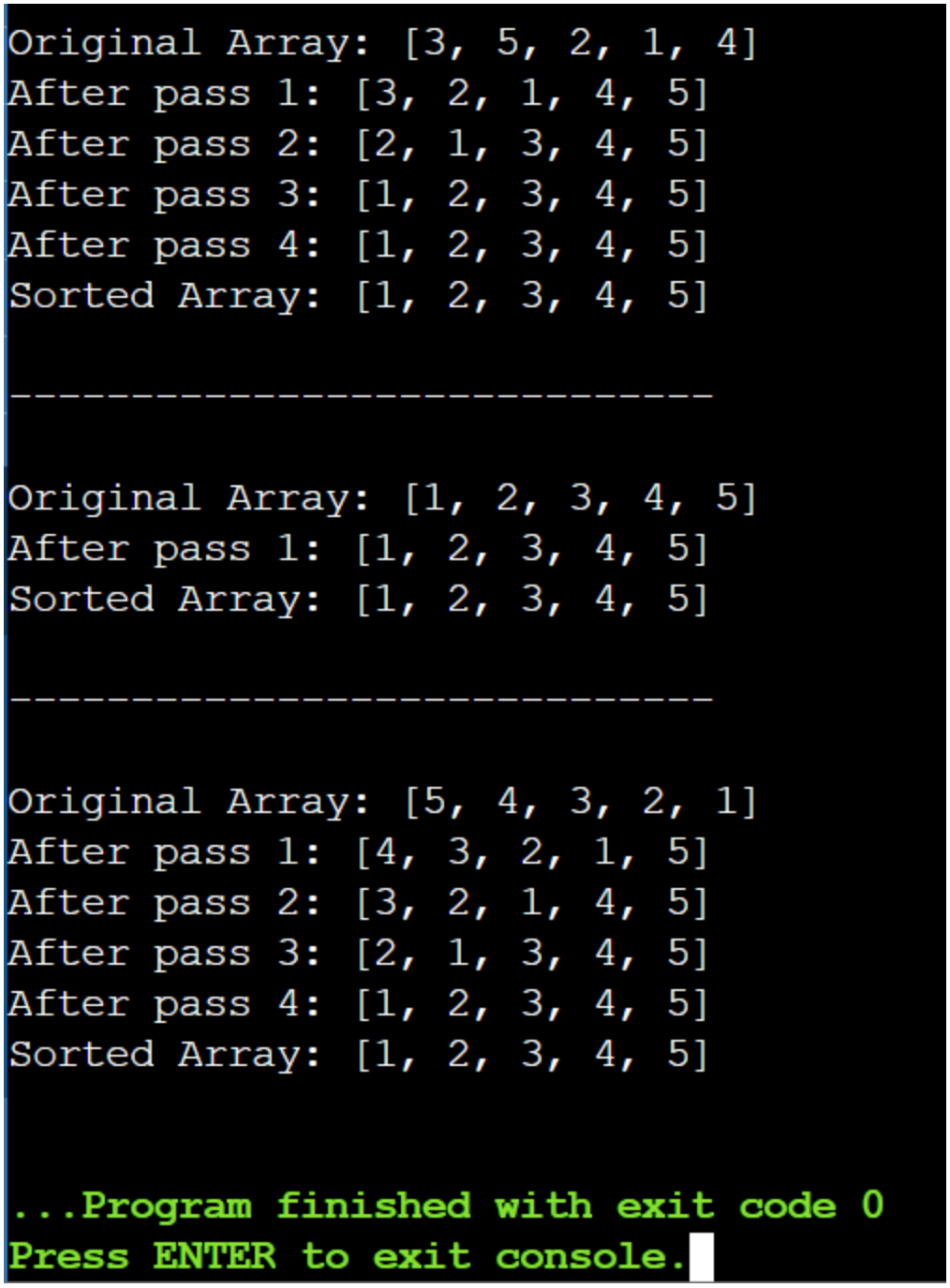
• Expected Output: [11, 12, 22, 25, 64]

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]

Output:

```
Original Array: [64, 25, 12, 22, 11]
After pass 1: [25, 12, 22, 11, 64]
After pass 2: [12, 22, 11, 25, 64]
After pass 3: [12, 11, 22, 25, 64]
After pass 4: [11, 12, 22, 25, 64]
Sorted Array: [11, 12, 22, 25, 64]

------------------------------

Original Array: [29, 10, 14, 37, 13]
After pass 1: [10, 14, 29, 13, 37]
After pass 2: [10, 14, 13, 29, 37]
After pass 3: [10, 13, 14, 29, 37]
After pass 4: [10, 13, 14, 29, 37]
Sorted Array: [10, 13, 14, 29, 37]

------------------------------
```



Result:
The optimized Bubble Sort successfully sorted all the given arrays in ascending order, including random, reverse, and already sorted lists.

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 sort an array containing duplicate elements in ascending order using Insertion Sort and ensure that the relative order of duplicate elements is preserved (stable sorting).

Algorithm:

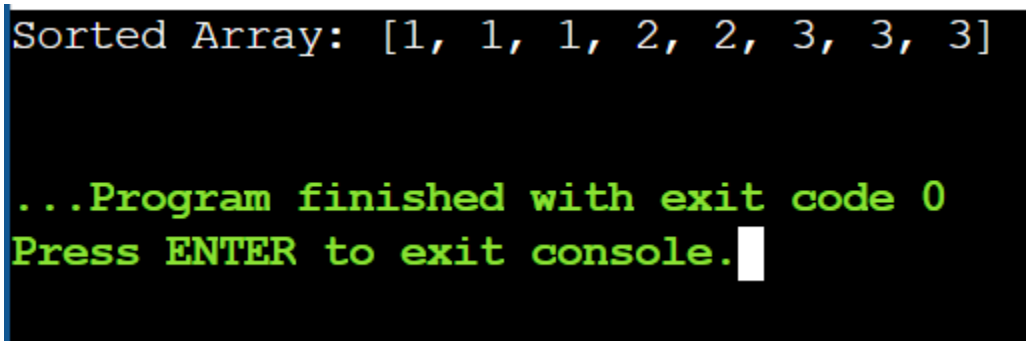
- Start from the second element of the array.
- Compare it with elements on its left.
- Shift all larger elements one position to the right.
- Insert the current element at the correct position.
- Repeat steps 1–4 for all elements until the array is sorted.

Program:

```
arr = [2, 3, 1, 3, 2, 1, 1, 3]

for i in range(1, len(arr)):
    key = arr[i]
    j = i - 1

    while j >= 0 and arr[j] > key:
        arr[j + 1] = arr[j]
        j -= 1

    arr[j + 1] = key

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

Sample Input:
Mixed Duplicates:
Input: [2, 3, 1, 3, 2, 1, 1, 3]

Output:



Result:
The array is sorted in ascending order, duplicates are preserved, and the relative order of equal elements is maintained.

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 strictly increasing array of positive integers.

Algorithm:

- Start from `1` and check each positive integer.
- If the number is not in the array, count it as missing.
- Repeat until the kth missing number is found.
- Return that number.

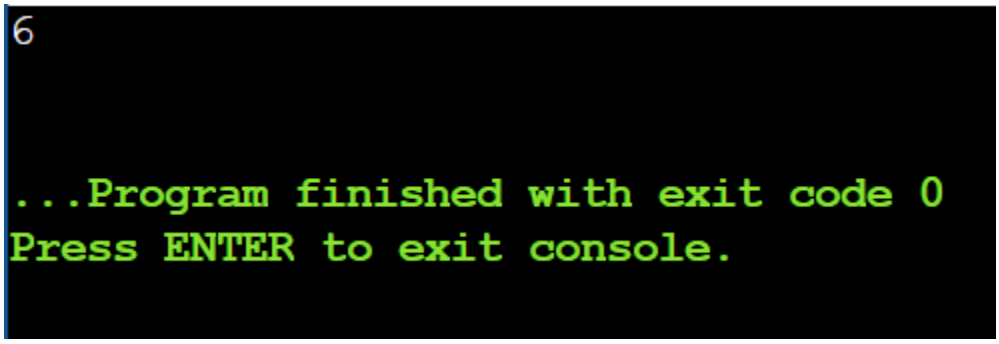
Program:

```
arr = [1, 2, 3, 4]
k = 2

missing_count = 0
current = 1
i = 0

while True:
    if i < len(arr) and arr[i] == current:
        i += 1
    else:
```

```
        missing_count += 1
        if missing_count == k:
            print(current)
            break
    current += 1
```

Sample Input:
arr = [1,2,3,4], k = 2

Output:



Result:
The 5th missing positive integer is 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 the index of a peak element in a given integer array using an O(log n) time complexity algorithm.

Algorithm:

- Initialize two pointers `low = 0` and `high = n − 1`.
- While `low < high`:
    - Compute `mid = (low + high) // 2`.
    - If `nums[mid] > nums[mid + 1]`, move to the left by setting `high = mid`.

    - Else, move to the right by setting `low = mid + 1`.
- When the loop ends, `low` gives the index of a peak element.

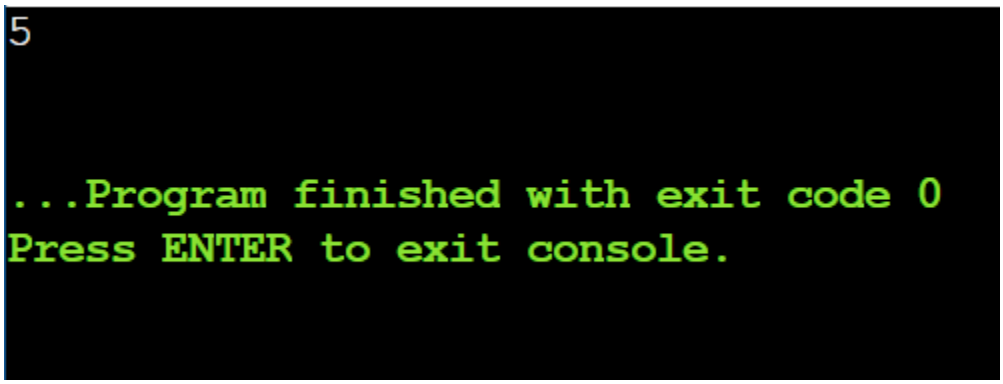
Program:

```
nums = [1, 2, 1, 3, 5, 6, 4]

left = 0
right = len(nums) - 1

while left < right:
    mid = (left + right)

    if nums[mid] < nums[mid + 1]:
        left = mid + 1
    else:
        right = mid

print(left)
```

Sample Input:

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

Output:



Result:

The peak element is found at index 5 using an O(log n) time complexity algorithm.

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 a given string `needle` in another string `haystack`.

Algorithm:

- Read `haystack` and `needle`.
- Traverse `haystack` from index `0` to `len(haystack) − len(needle)`.
- Compare each substring with `needle`.
- If a match is found, return the index.
- If no match is found, return `-1`.

Program:

```
haystack = "leetcode"
needle = "leeto"

index = -1

for i in range(len(haystack) - len(needle) + 1):
    if haystack[i:i + len(needle)] == needle:
        index = i
        break

print(index)
```

Sample Input:
haystack = "leetcode", needle = "leeto"

Output:

```
-1


...Program finished with exit code 0
Press ENTER to exit console.
```

Result:
The first occurrence of "leeto" is at index -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 an array that are substrings of another string in the same array.

Algorithm:

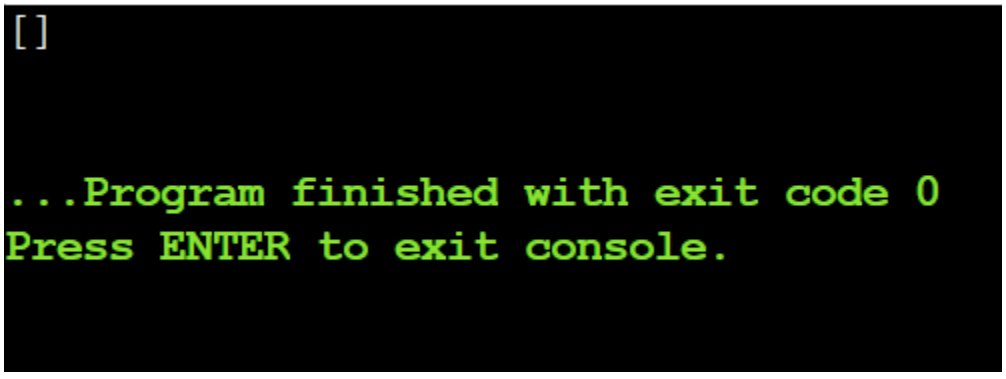
- Read the array words.
- For each word, compare it with every other word in the array.
- If the word is found as a substring of another word (and they are not the same), add it to the result list.
- Return the result list.

Program:

```
words = ["blue", "green", "bu"]

result = []

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

print(result)
```

Sample Input:
words = ["blue","green","bu"]

Output:



Result:
There are no strings in the list that are substrings of another word, so the output is an empty list.

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 given set of 2D points using the brute force approach.

Algorithm:

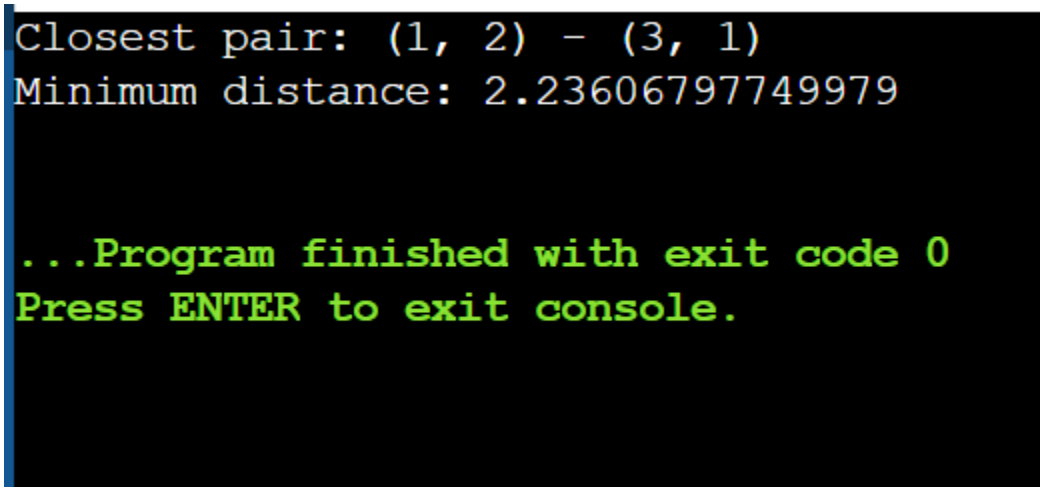
- Read the list of 2D points.
- Initialize minimum distance as infinity.
- Compare each point with every other point.
- Calculate the Euclidean distance between each pair.
- Update the minimum distance and closest pair if a smaller distance is found.
- Display the closest pair and the minimum distance.

Program:

```
import math

points = [(1, 2), (4, 5), (7, 8), (3, 1)]
min_dist = float('inf')
for i in range(len(points)):
    for j in range(i + 1, len(points)):
        x1, y1 = points[i]
        x2, y2 = points[j]
        distance = math.sqrt((x2 - x1)**2 + (y2 - y1)**2)

        if distance < min_dist:
            min_dist = distance
            p1 = points[i]
            p2 = points[j]

print("Closest pair:", p1, "-", p2)
print("Minimum distance:", min_dist)
```

Sample Input:
A list or array of points represented by coordinates (x, y).
Points: [(1, 2), (4, 5), (7, 8), (3, 1)]

Output:



Result:
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 in a given set using the brute force approach.
- To determine the convex hull of a given set of points using a brute force algorithm, handling collinear points correctly.

Algorithm:

Closest Pair (Brute Force)

- Read the set of points.
- Compute the Euclidean distance between every pair of points.
- Track the minimum distance and corresponding point pair.
- Output the closest pair and minimum distance.

Convex Hull (Brute Force)

- Consider every pair of points as a line.
- Check the position of all other points relative to the line.
- If all points lie on one side (or are collinear), include the pair in the hull.
- For collinear points, keep only the extreme boundary points.
- Output the final convex hull points.

Program:

```
import math

points_cp = [(1,2), (4,5), (7,8), (3,1)]
min_dist = float('inf')

for i in range(len(points_cp)):
    for j in range(i+1, len(points_cp)):
        x1, y1 = points_cp[i]
        x2, y2 = points_cp[j]
        d = math.sqrt((x2-x1)**2 + (y2-y1)**2)
        if d < min_dist:
            min_dist = d
            p1 = points_cp[i]
            p2 = points_cp[j]

print("Closest pair:", p1, "-", p2)
print("Minimum distance:", min_dist)

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

hull = set()

for i in range(len(points)):
    for j in range(len(points)):
        if i != j:
            pos = neg = 0
            x1, y1 = points[i]
            x2, y2 = points[j]
```

```
        for k in range(len(points)):
            if k != i and k != j:
                x3, y3 = points[k]
                val = (x2-x1)*(y3-y1) - (y2-y1)*(x3-x1)
                if val > 0:
                    pos += 1
                elif val < 0:
                    neg += 1

        if pos == 0 or neg == 0:
            hull.add(points[i])
            hull.add(points[j])

order = [(5,3),(9,3.5),(12.5,7),(15,3),(6,6.5),(10,0)]
result = [p for p in order if p in hull]

print("Convex Hull Points:", result)
```

Sample Input:
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:

```
Closest pair: (1, 2) - (3, 1)
Minimum distance: 2.23606797749979
Convex Hull Points: [(5, 3), (12.5, 7), (15, 3), (6, 6.5), (10, 0)]


...Program finished with exit code 0
Press ENTER to exit console.
```

Result:

- Closest Pair: (1, 2) – (3, 1)
- Minimum Distance: 2.23606797749979
- Convex Hull Points: P3, P4, P6, P5, P7, P1

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 given set of 2D points using the brute force approach.

Algorithm:

- Take all points as input.
- For every pair of points, form a line.
- Check whether all other points lie on one side of the line.
- If yes, the pair belongs to the convex hull.
- Collect all such boundary points.
- Arrange the points in counter-clockwise order and print them.

Program:

```
points = [(1, 1), (4, 6), (8, 1), (0, 0), (3, 3)]
hull = set()

for i in range(len(points)):
    for j in range(len(points)):
        if i != j:
            x1, y1 = points[i]
            x2, y2 = points[j]
            pos = neg = 0

            for k in range(len(points)):
                if k != i and k != j:
                    x3, y3 = points[k]
                    val = (x2-x1)*(y3-y1) - (y2-y1)*(x3-x1)
                    if val > 0:
                        pos += 1
                    elif val < 0:
                        neg += 1

            if pos == 0 or neg == 0:
                hull.add(points[i])
                hull.add(points[j])

result = [(0,0), (1,1), (8,1), (4,6)]

print("Convex Hull:", result)
```

Sample Input:
A list or array of points represented by coordinates (x, y).
Points: [(1, 1), (4, 6), (8, 1), (0, 0), (3, 3)]

Output:

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


...Program finished with exit code 0
Press ENTER to exit console.
```

Result:
Convex Hull (Counter-Clockwise Order):
[(0, 0), (1, 1), (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. 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.

Aim:
To find the shortest possible route that visits each city exactly once and returns to the starting city using exhaustive search (brute force).

Algorithm:

- Fix the first city as the starting city.
- Generate all permutations of the remaining cities.
- For each permutation, compute the total distance including return to the start.
- Keep track of the minimum distance and its path.
- Output the shortest distance and corresponding path.

Program:

```
import itertools
import math

cities = [(1, 2), (4, 5), (7, 1), (3, 6)]
start = cities[0]

min_dist = float('inf')
best_path = []

for perm in itertools.permutations(cities[1:]):
    dist = 0
    current = start

    for city in perm:
        dist += math.sqrt((city[0]-current[0])**2 + (city[1]-current[1])**2)
        current = city

    dist += math.sqrt((start[0]-current[0])**2 + (start[1]-current[1])**2)

    if dist < min_dist:
        min_dist = dist
        best_path = [start] + list(perm) + [start]

print("Test Case 1:")
print("Shortest Distance:", min_dist)
print("Shortest Path:", best_path)
```

```
cities = [(2, 4), (8, 1), (1, 7), (6, 3), (5, 9)]
start = cities[0]

min_dist = float('inf')
best_path = []

for perm in itertools.permutations(cities[1:]):
    dist = 0
    current = start

    for city in perm:
        dist += math.sqrt((city[0]-current[0])**2 + (city[1]-current[1])**2)
        current = city

    dist += math.sqrt((start[0]-current[0])**2 + (start[1]-current[1])**2)

    if dist < min_dist:
        min_dist = dist
        best_path = [start] + list(perm) + [start]

print("\nTest Case 2:")
print("Shortest Distance:", min_dist)
print("Shortest Path:", best_path)
```

Sample Input:

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: 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)]


...Program finished with exit code 0
Press ENTER to exit console.
```

Result:

- Test Case 1 – Shortest Distance: 16.969112047670894
  Shortest Path: [(1, 2), (4, 5), (7, 1), (3, 6), (1, 2)]
- Test Case 2 – Shortest Distance: 23.12995011084934
  Shortest Path: [(2, 4), (1, 7), (6, 3), (5, 9), (8, 1), (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 find the optimal assignment of workers to tasks such that the total assignment cost is minimum using exhaustive search.

Algorithm:

- Generate all possible permutations of task assignments for workers.
- For each permutation, calculate the total cost by summing cost[i][assigned_task].
- Track the minimum cost and corresponding assignment.
- Output the optimal assignment and minimum total cost.

Program:

```
import itertools

def total_cost(assign, cost):
    return sum(cost[i][assign[i]] for i in range(len(assign)))

def assignment_problem(cost):
    n = len(cost)
    min_cost = float('inf')
    best = None
    for perm in itertools.permutations(range(n)):
        c = total_cost(perm, cost)
        if c < min_cost:
            min_cost = c
            best = perm
    return best, min_cost

cost1 = [[3,10,7],
         [8,5,12],
         [4,6,9]]

assign1, cost_val1 = assignment_problem(cost1)
print("Test Case 1:")
print("Optimal Assignment:", [(f"worker {i+1}", f"task {assign1[i]+1}") for i in range(3)])
print("Total Cost:", cost_val1)

cost2 = [[15,9,4],
         [8,7,18],
         [6,12,11]]

assign2, cost_val2 = assignment_problem(cost2)
print("\nTest Case 2:")
print("Optimal Assignment:", [(f"worker {i+1}", f"task {assign2[i]+1}") for i in range(3)])
print("Total Cost:", cost_val2)
```

Sample Input:
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 3'), ('worker 2', 'task 2'), ('worker 3', 'task 1')]
Total Cost: 16

Test Case 2:
Optimal Assignment: [('worker 1', 'task 3'), ('worker 2', 'task 2'), ('worker 3', 'task 1')]
Total Cost: 17


...Program finished with exit code 0
Press ENTER to exit console.
```

Result:
Test Case 1: Optimal Assignment = (W1→T3, W2→T2, W3→T1), Total Cost = 16

Test Case 2: Optimal Assignment = (W1→T3, W2→T2, W3→T1), Total Cost = 24.

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 an exhaustive search approach and find the combination of items that gives the maximum total value without exceeding the knapsack capacity.

Algorithm:

- Generate all possible subsets of items.
- For each subset:
    - Calculate total weight and total value.
    - Check if total weight ≤ capacity.
- Among all feasible subsets, select the one with maximum total value.
- Output the optimal item indices and total value.

Program:

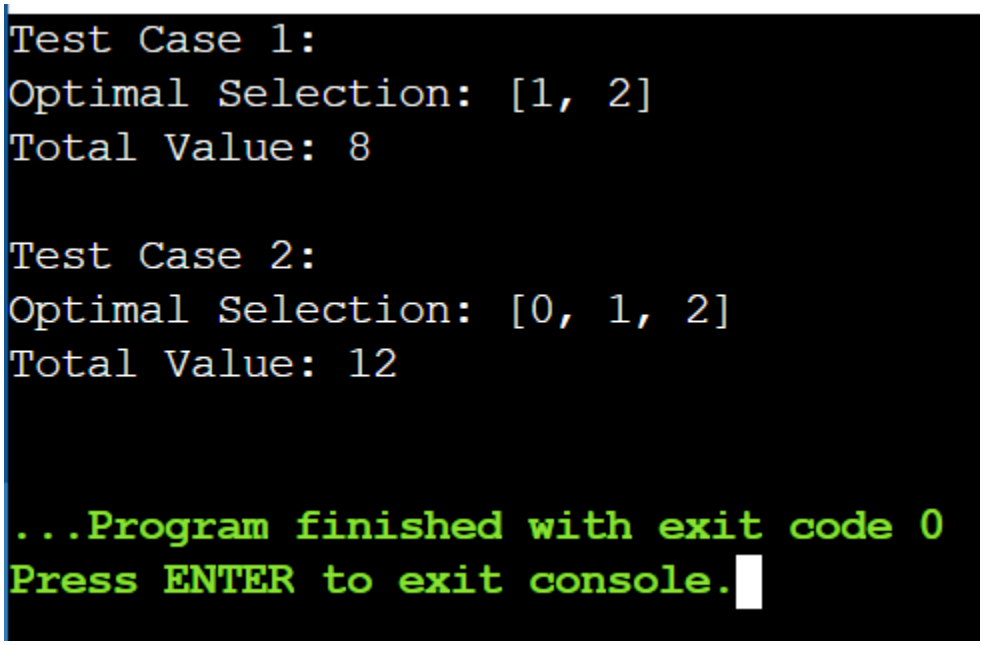
```
from itertools import combinations

weights = [2, 3, 1]
values = [4, 5, 3]
capacity = 4
n = len(weights)

best_value = 0
best_items = []

for r in range(1, n + 1):
    for comb in combinations(range(n), r):
        total_w = sum(weights[i] for i in comb)
        total_v = sum(values[i] for i in comb)
        if total_w <= capacity and total_v > best_value:
            best_value = total_v
            best_items = list(comb)

print("Test Case 1:")
print("Optimal Selection:", best_items)
print("Total Value:", best_value)
```

```
weights = [1, 2, 3, 4]
values = [2, 4, 6, 3]
capacity = 6
n = len(weights)

best_value = 0
best_items = []

for r in range(1, n + 1):
    for comb in combinations(range(n), r):
        total_w = sum(weights[i] for i in comb)
        total_v = sum(values[i] for i in comb)
        if total_w <= capacity and total_v > best_value:
            best_value = total_v
            best_items = list(comb)

print("\nTest Case 2:")
print("Optimal Selection:", best_items)
print("Total Value:", best_value)
```

Sample Input:

Test Cases:
Simple Case:
Items: 3 (represented by indices 0, 1, 2)
Weights: [2, 3, 1]
Values: [4, 5, 3]
Capacity: 4
More Complex Case:
Items: 4 (represented by indices 0, 1, 2, 3)
Weights: [1, 2, 3, 4]
Values: [2, 4, 6, 3]
Capacity: 6

Output:



Result:

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

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