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]

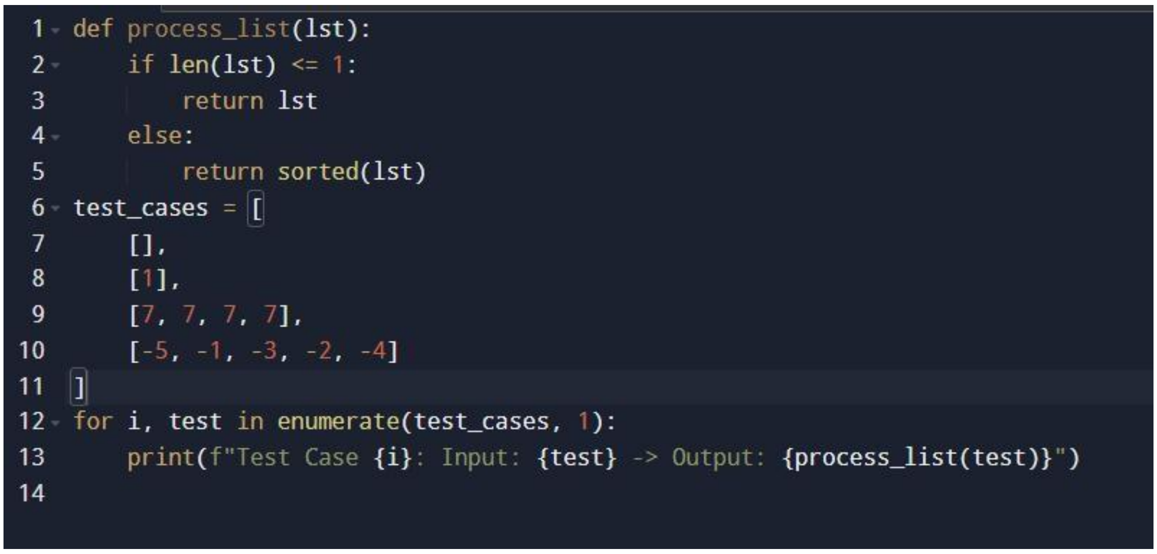
- **Expected Output:** [1]

2. **Input:** [7, 7, 7, 7]

- **Expected Output:** [7, 7, 7, 7]

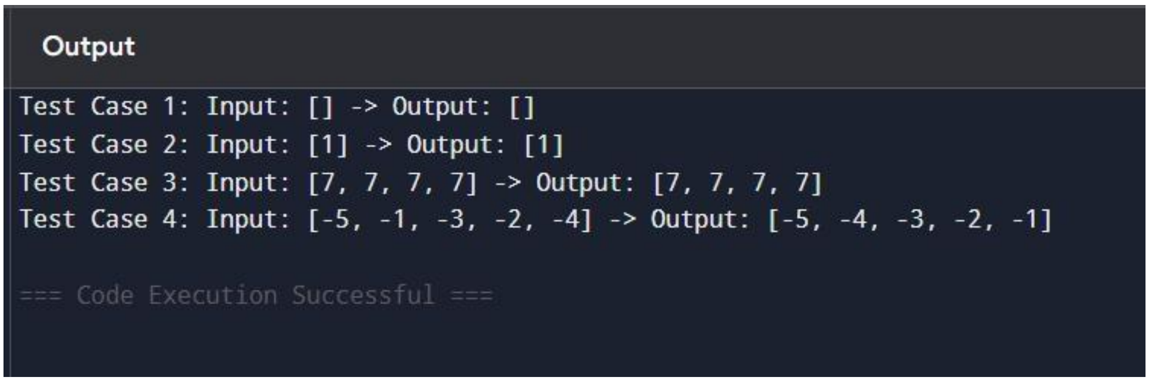
3. **Input:** [-5, -1, -3, -2, -4]

- **Expected Output:** [-5, -4, -3, -2, -1]



```python
def process_list(lst):
    if len(lst) <= 1:
        return lst
    else:
        return sorted(lst)
test_cases = [
    [],
    [1],
    [7, 7, 7, 7],
    [-5, -1, -3, -2, -4]
]
for i, test in enumerate(test_cases, 1):
    print(f"Test Case {i}: Input: {test} -> Output: {process_list(test)}")

```



Output

```
Test Case 1: Input: [] -> Output: []
Test Case 2: Input: [1] -> Output: [1]
Test Case 3: Input: [7, 7, 7, 7] -> Output: [7, 7, 7, 7]
Test Case 4: Input: [-5, -1, -3, -2, -4] -> Output: [-5, -4, -3, -2, -1]

=== Code Execution Successful ===
```

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:

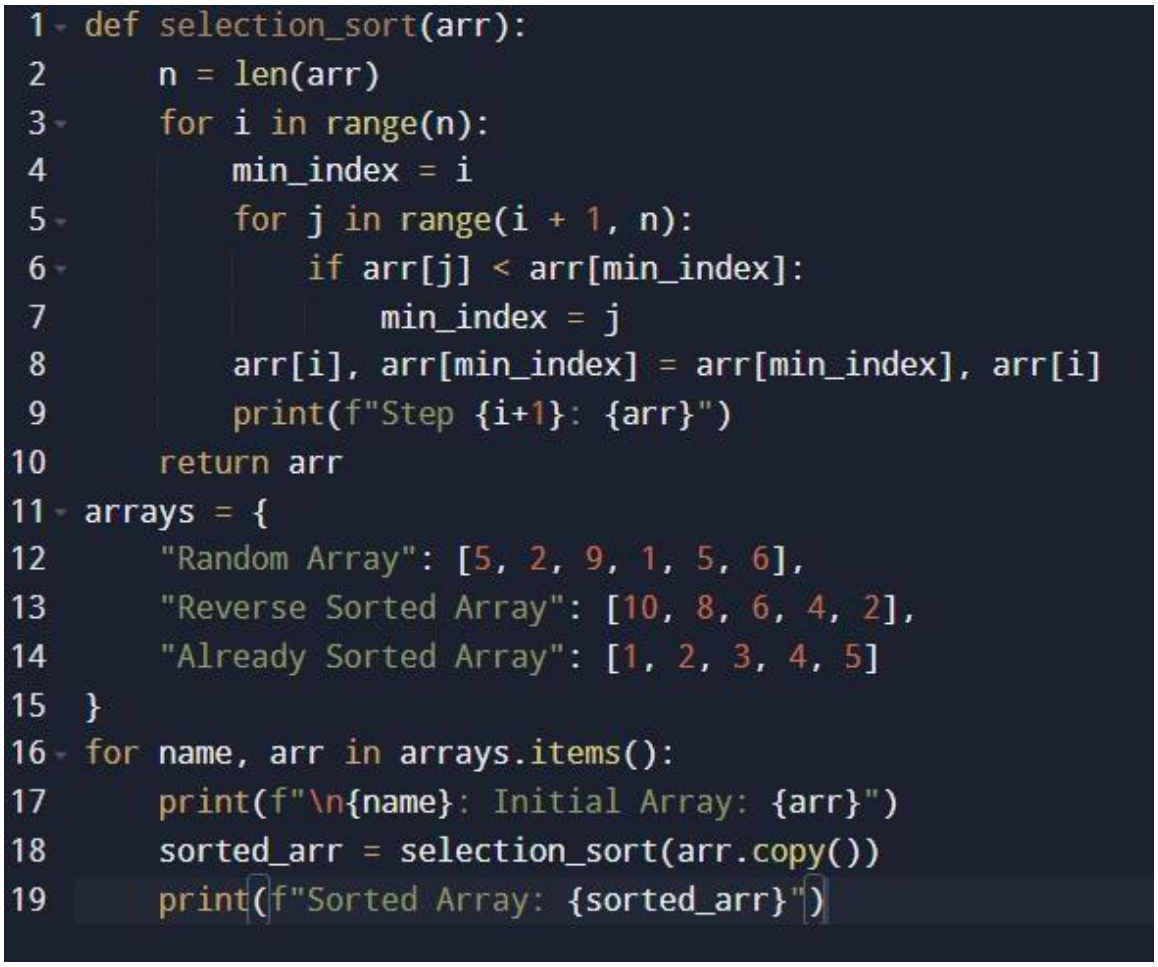
Input: [10, 8, 6, 4, 2]

Output: [2, 4, 6, 8, 10]

Sorting an Already Sorted Array:

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

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



```python
def selection_sort(arr):
    n = len(arr)
    for i in range(n):
        min_index = i
        for j in range(i + 1, n):
            if arr[j] < arr[min_index]:
                min_index = j
        arr[i], arr[min_index] = arr[min_index], arr[i]
        print(f"Step {i+1}: {arr}")
    return arr
arrays = {
    "Random Array": [5, 2, 9, 1, 5, 6],
    "Reverse Sorted Array": [10, 8, 6, 4, 2],
    "Already Sorted Array": [1, 2, 3, 4, 5]
}
for name, arr in arrays.items():
    print(f"\n{name}: Initial Array: {arr}")
    sorted_arr = selection_sort(arr.copy())
    print(f"Sorted Array: {sorted_arr}")
```

2. 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]

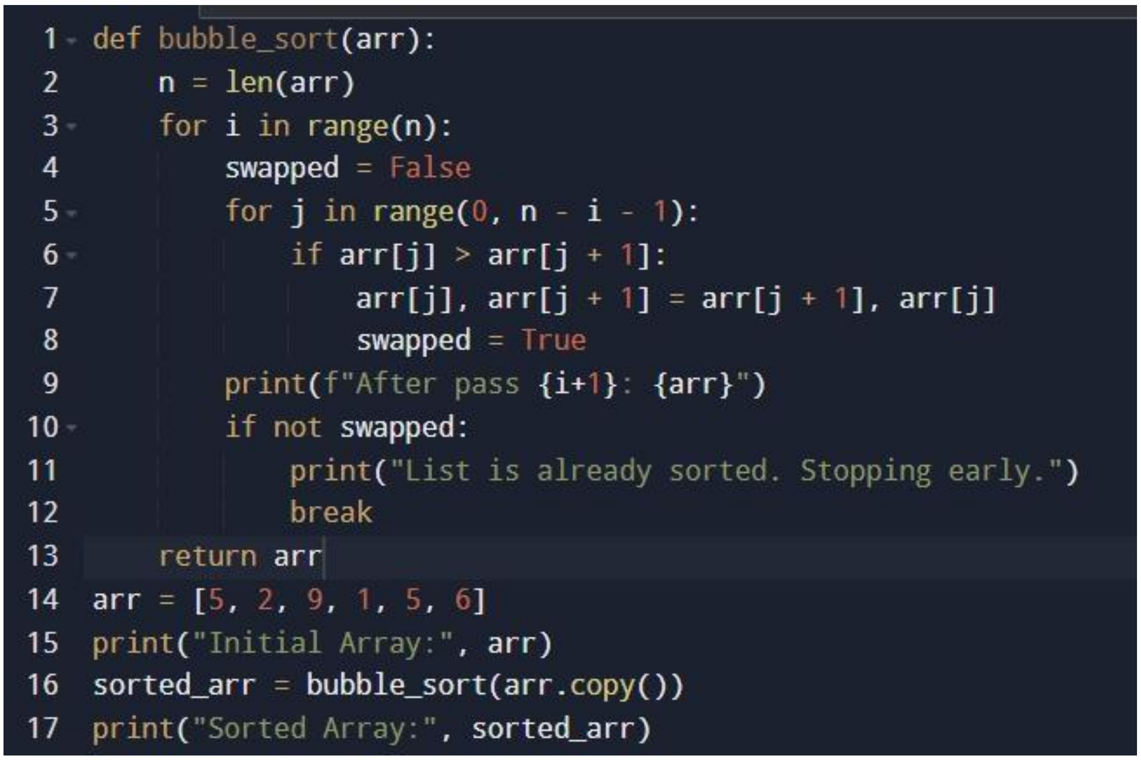
- **Expected Output:** [1, 2, 3, 4, 5]

4. **Input:** [1, 2, 3, 4, 5] (Already sorted list)

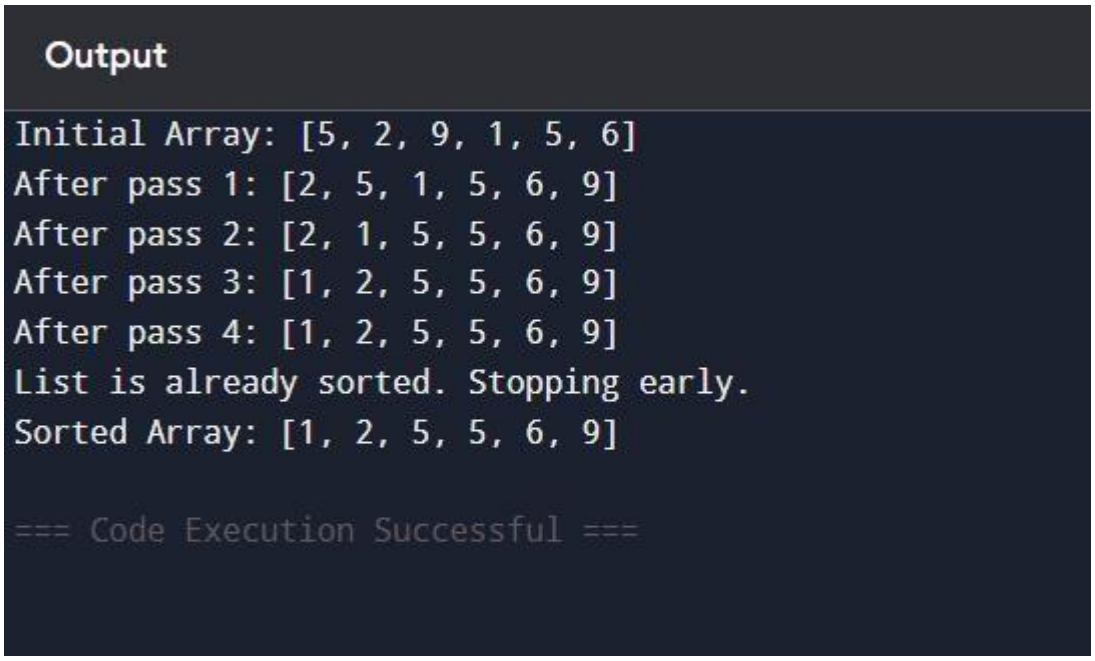
- **Expected Output:** [1, 2, 3, 4, 5]

5. **Input:** [5, 4, 3, 2, 1] (Reverse sorted list)

- **Expected Output:** [1, 2, 3, 4, 5]



```python
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
        print(f"After pass {i+1}: {arr}")
        if not swapped:
            print("List is already sorted. Stopping early.")
            break
    return arr
arr = [5, 2, 9, 1, 5, 6]
print("Initial Array:", arr)
sorted_arr = bubble_sort(arr.copy())
print("Sorted Array:", sorted_arr)
```


Output

```
Initial Array: [5, 2, 9, 1, 5, 6]
After pass 1: [2, 5, 1, 5, 6, 9]
After pass 2: [2, 1, 5, 5, 6, 9]
After pass 3: [1, 2, 5, 5, 6, 9]
After pass 4: [1, 2, 5, 5, 6, 9]
List is already sorted. Stopping early.
Sorted Array: [1, 2, 5, 5, 6, 9]

=== Code Execution Successful ===
```

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]

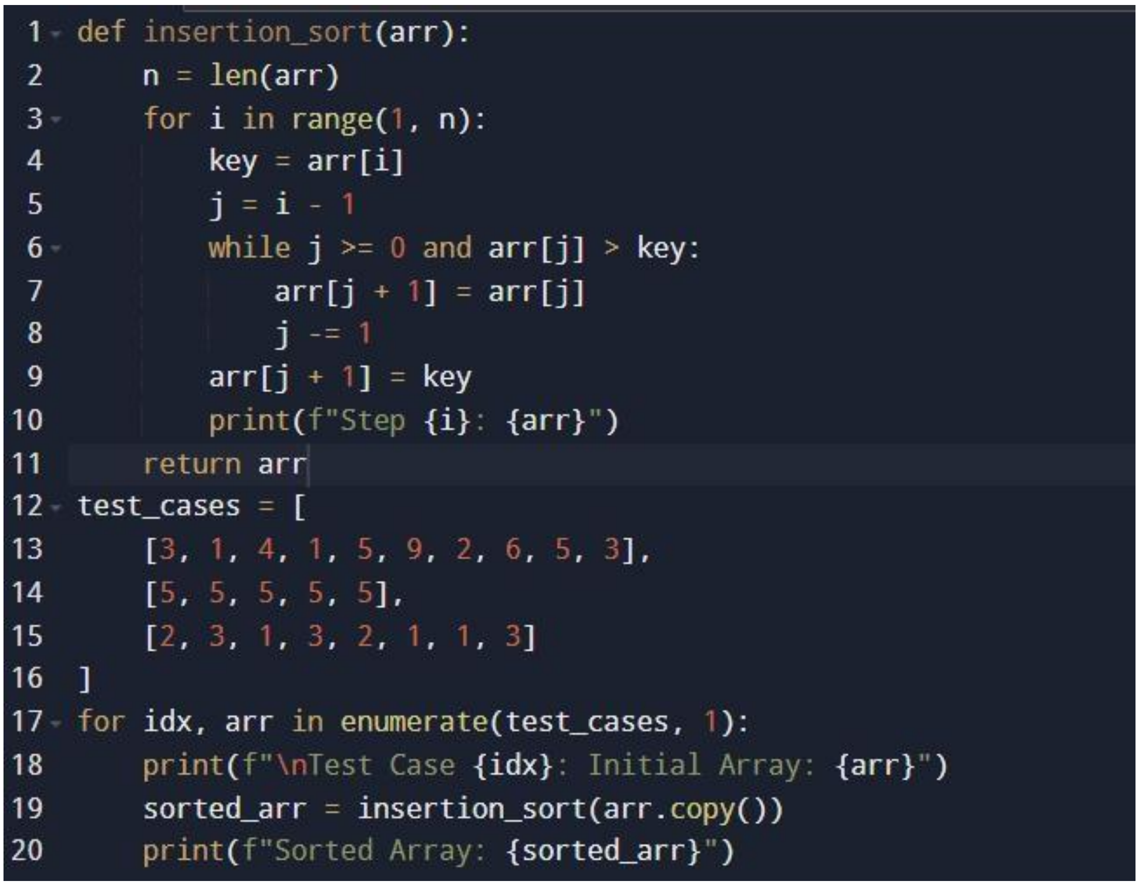
1. **All Identical Elements**:

- ***Input***: [5, 5, 5, 5, 5]
- ***Output***: [5, 5, 5, 5, 5]

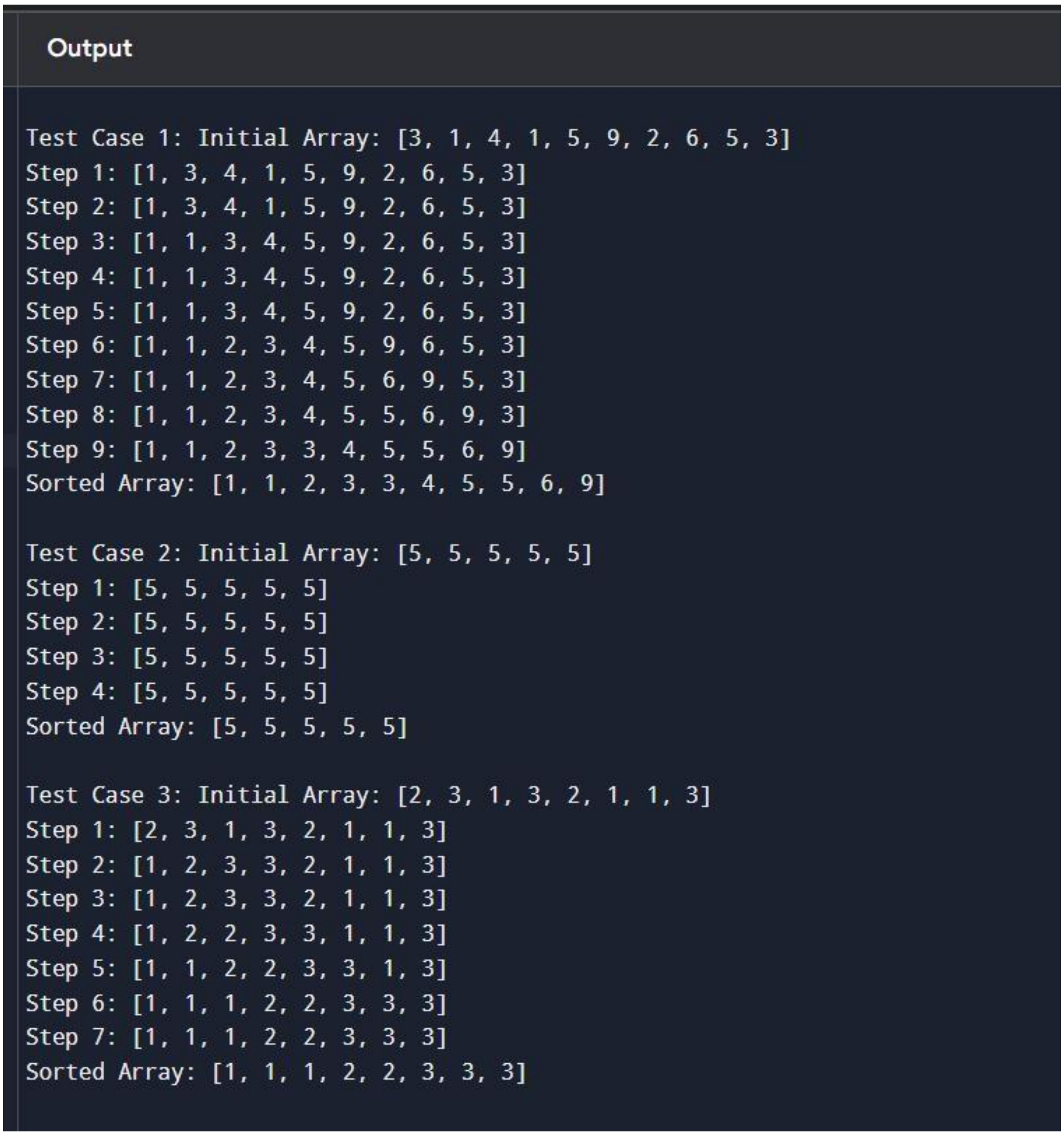
2. **Mixed Duplicates**:

- ***Input***: [2, 3, 1, 3, 2, 1, 1, 3]

- ***Output***: [1, 1, 1, 2, 2, 3, 3, 3]



```python
def insertion_sort(arr):
    n = len(arr)
    for i in range(1, n):
        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(f"Step {i}: {arr}")
    return arr
test_cases = [
    [3, 1, 4, 1, 5, 9, 2, 6, 5, 3],
    [5, 5, 5, 5, 5],
    [2, 3, 1, 3, 2, 1, 1, 3]
]
for idx, arr in enumerate(test_cases, 1):
    print(f"\nTest Case {idx}: Initial Array: {arr}")
    sorted_arr = insertion_sort(arr.copy())
    print(f"Sorted Array: {sorted_arr}")
```


Output

```
Test Case 1: Initial Array: [3, 1, 4, 1, 5, 9, 2, 6, 5, 3]
Step 1: [1, 3, 4, 1, 5, 9, 2, 6, 5, 3]
Step 2: [1, 3, 4, 1, 5, 9, 2, 6, 5, 3]
Step 3: [1, 1, 3, 4, 5, 9, 2, 6, 5, 3]
Step 4: [1, 1, 3, 4, 5, 9, 2, 6, 5, 3]
Step 5: [1, 1, 3, 4, 5, 9, 2, 6, 5, 3]
Step 6: [1, 1, 2, 3, 4, 5, 9, 6, 5, 3]
Step 7: [1, 1, 2, 3, 4, 5, 6, 9, 5, 3]
Step 8: [1, 1, 2, 3, 4, 5, 5, 6, 9, 3]
Step 9: [1, 1, 2, 3, 3, 4, 5, 5, 6, 9]
Sorted Array: [1, 1, 2, 3, 3, 4, 5, 5, 6, 9]

Test Case 2: Initial Array: [5, 5, 5, 5, 5]
Step 1: [5, 5, 5, 5, 5]
Step 2: [5, 5, 5, 5, 5]
Step 3: [5, 5, 5, 5, 5]
Step 4: [5, 5, 5, 5, 5]
Sorted Array: [5, 5, 5, 5, 5]

Test Case 3: Initial Array: [2, 3, 1, 3, 2, 1, 1, 3]
Step 1: [2, 3, 1, 3, 2, 1, 1, 3]
Step 2: [1, 2, 3, 3, 2, 1, 1, 3]
Step 3: [1, 2, 3, 3, 2, 1, 1, 3]
Step 4: [1, 2, 2, 3, 3, 1, 1, 3]
Step 5: [1, 1, 2, 2, 3, 3, 1, 3]
Step 6: [1, 1, 1, 2, 2, 3, 3, 3]
Step 7: [1, 1, 1, 2, 2, 3, 3, 3]
Sorted Array: [1, 1, 1, 2, 2, 3, 3, 3]
```

5. Given an array arr of positive integers sorted in a strictly increasing order, and an integer k. return the kth positive integer that is missing from this array.

Example 1:

Input: arr = [2,3,4,7,11], k = 5

Output: 9

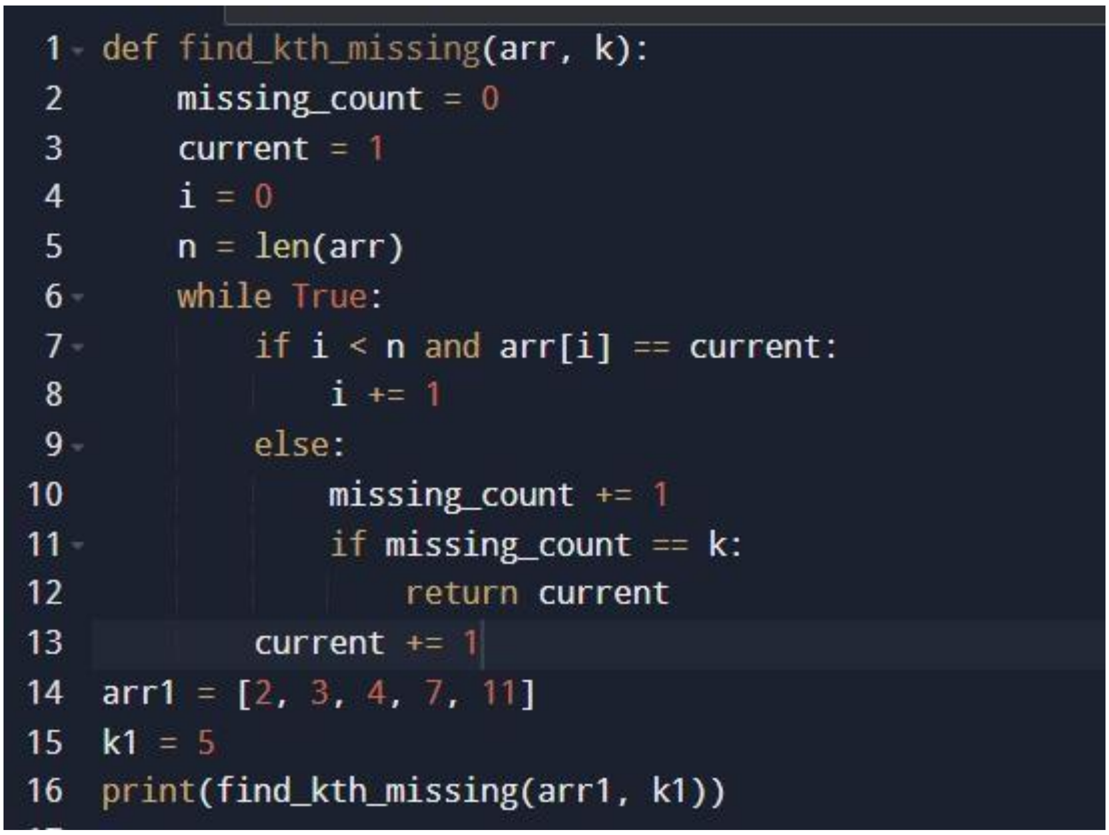
Explanation: The missing positive integers are [1,5,6,8,9,10,12,13,...]. The 5th missing positive integer is 9.

Example 2:

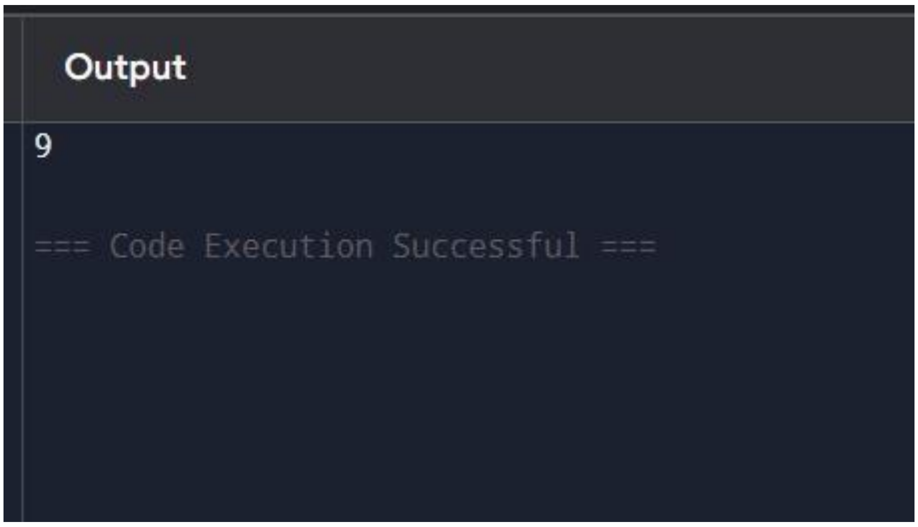
Input: arr = [1,2,3,4], k = 2

Output: 6

Explanation: The missing positive integers are [5,6,7,...]. The 2nd missing positive integer is 6.



```python
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
arr1 = [2, 3, 4, 7, 11]
k1 = 5
print(find_kth_missing(arr1, k1))
```


Output

```
9

=== Code Execution Successful ===
```

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

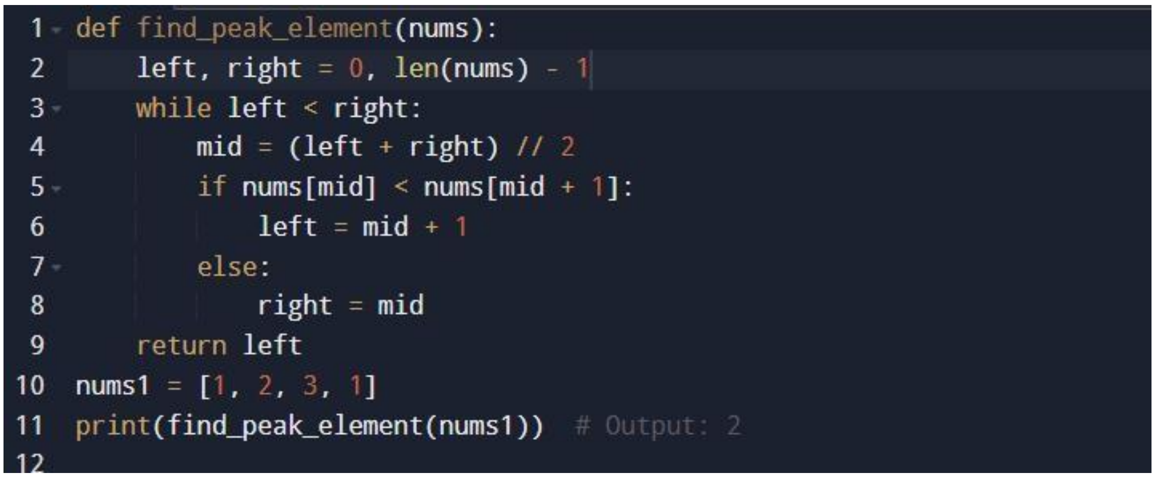
Explanation: 3 is a peak element and your function should return the index number 2.

Example 2:

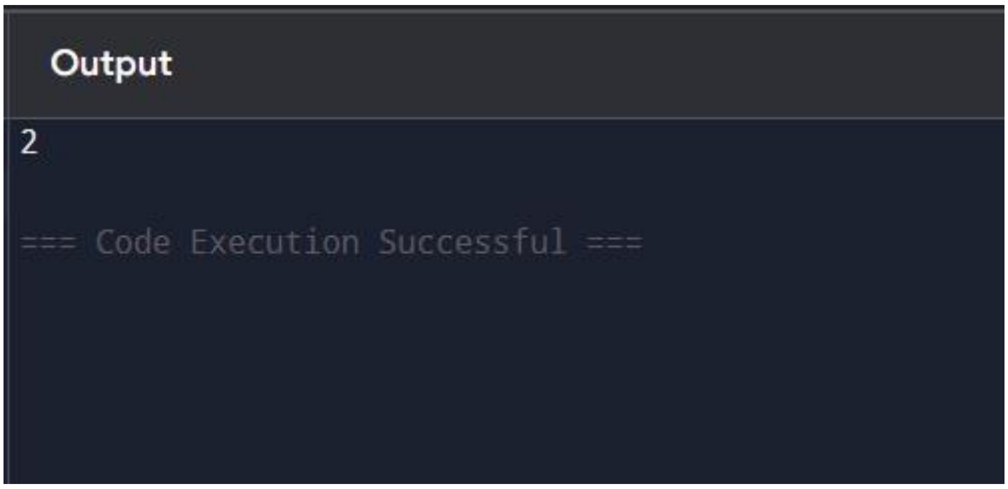
Input: nums = [1,2,1,3,5,6,4]

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.



```python
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
nums1 = [1, 2, 3, 1]
print(find_peak_element(nums1))  # Output: 2
```



Output

```
2

=== Code Execution Successful ===
```

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.

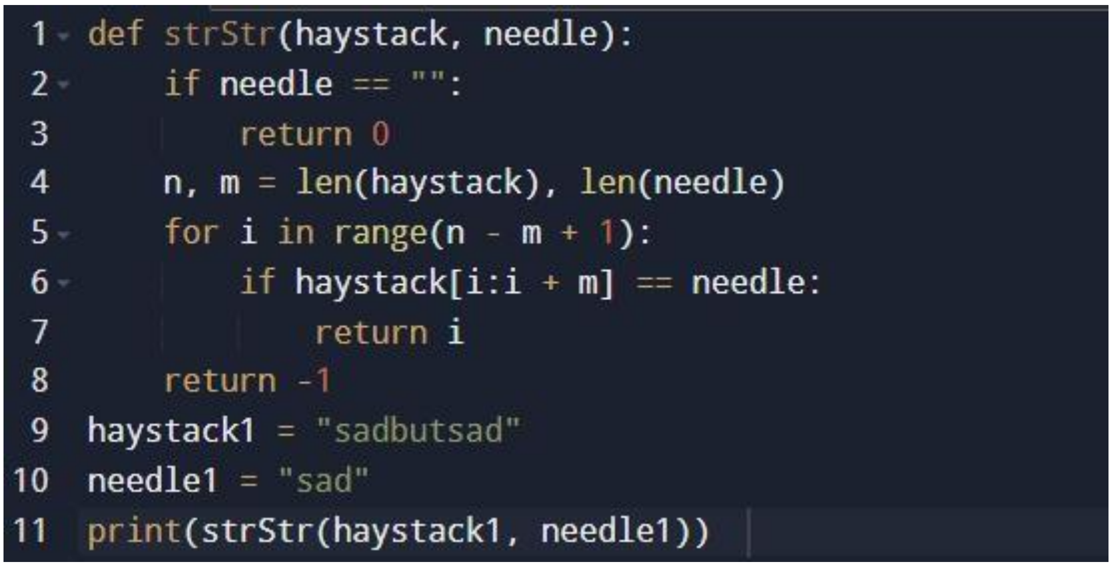
The first occurrence is at index 0, so we return 0.

Example 2:

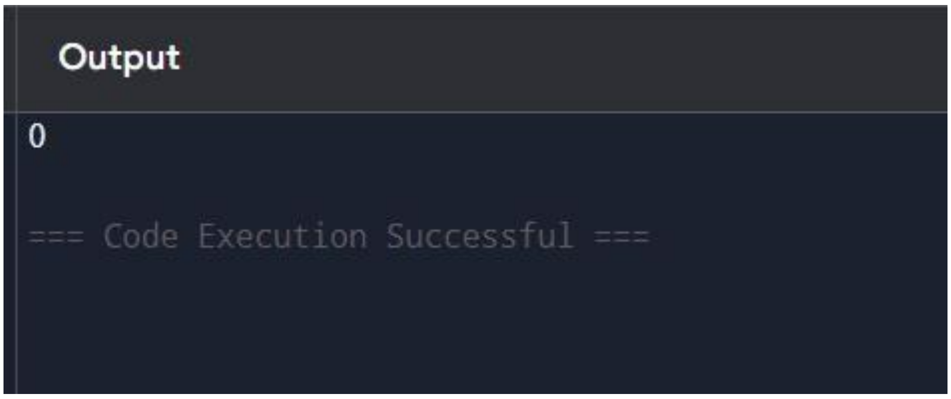
Input: haystack = "leetcode", needle = "leeto"

Output: -1

Explanation: "leeto" did not occur in "leetcode", so we return -1.



```python
def strStr(haystack, needle):
    if 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
haystack1 = "sadbutsad"
needle1 = "sad"
print(strStr(haystack1, needle1))
```



Output

```
0

=== Code Execution Successful ===
```

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"]

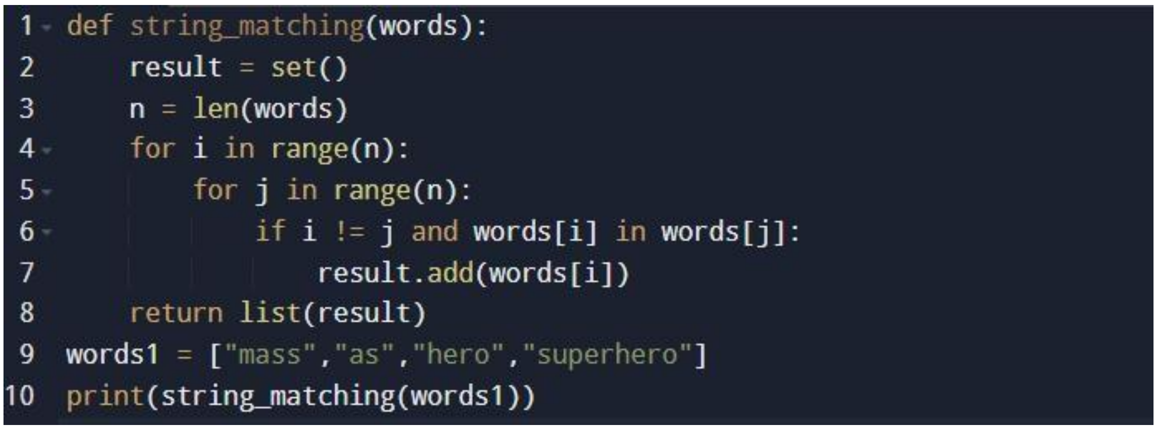
Explanation: "et", "code" are substring of "leetcode".

Example 3:

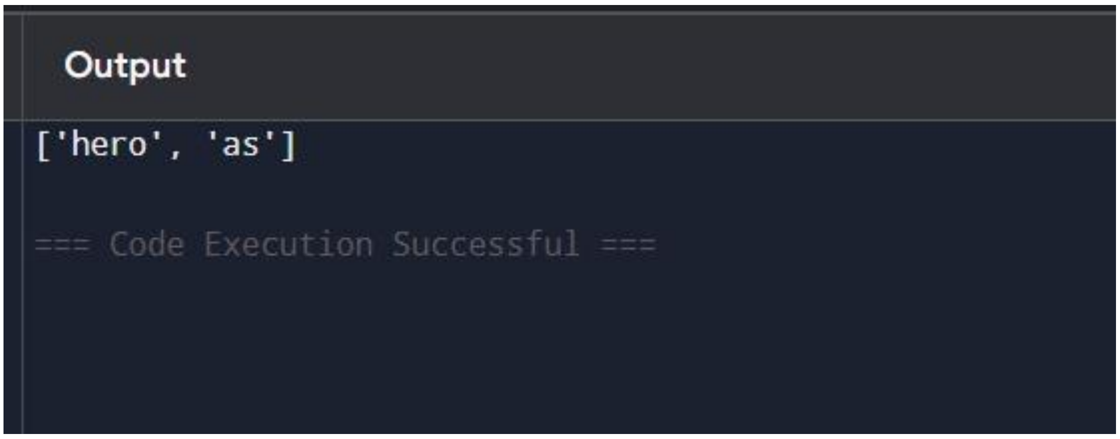
Input: words = ["blue","green","bu"]

Output: []

Explanation: No string of words is substring of another string.



```python
def string_matching(words):
    result = set()
    n = len(words)
    for i in range(n):
        for j in range(n):
            if i != j and words[i] in words[j]:
                result.add(words[i])
    return list(result)
words1 = ["mass","as","hero","superhero"]
print(string_matching(words1))
```



Output

```
['hero', 'as']

=== Code Execution Successful ===
```

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).

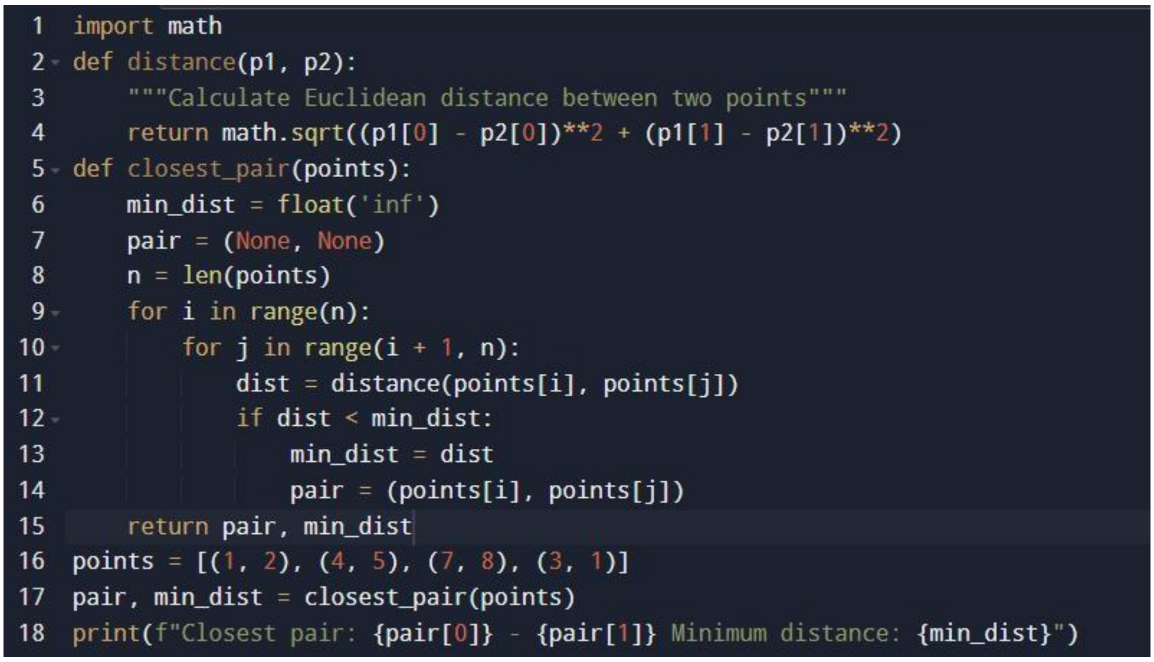
Points: [(1, 2), (4, 5), (7, 8), (3, 1)]

Output:

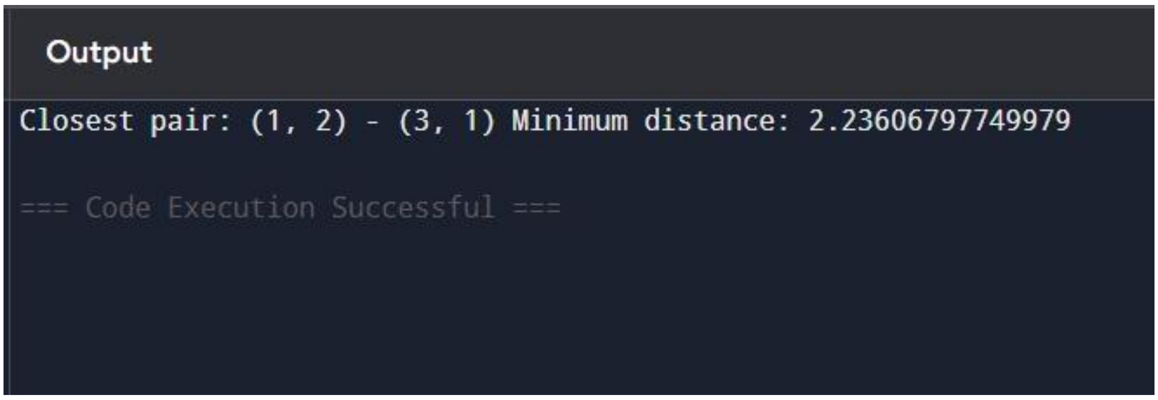
The two points with the minimum distance between them.

The minimum distance itself.

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



```python
import math
def distance(p1, p2):
    """Calculate Euclidean distance between two points"""
    return math.sqrt((p1[0] - p2[0])**2 + (p1[1] - p2[1])**2)
def closest_pair(points):
    min_dist = float('inf')
    pair = (None, None)
    n = len(points)
    for i in range(n):
        for j in range(i + 1, n):
            dist = distance(points[i], points[j])
            if dist < min_dist:
                min_dist = dist
                pair = (points[i], points[j])
    return pair, min_dist
points = [(1, 2), (4, 5), (7, 8), (3, 1)]
pair, min_dist = closest_pair(points)
print(f"Closest pair: {pair[0]} - {pair[1]} Minimum distance: {min_dist}")
```



Output

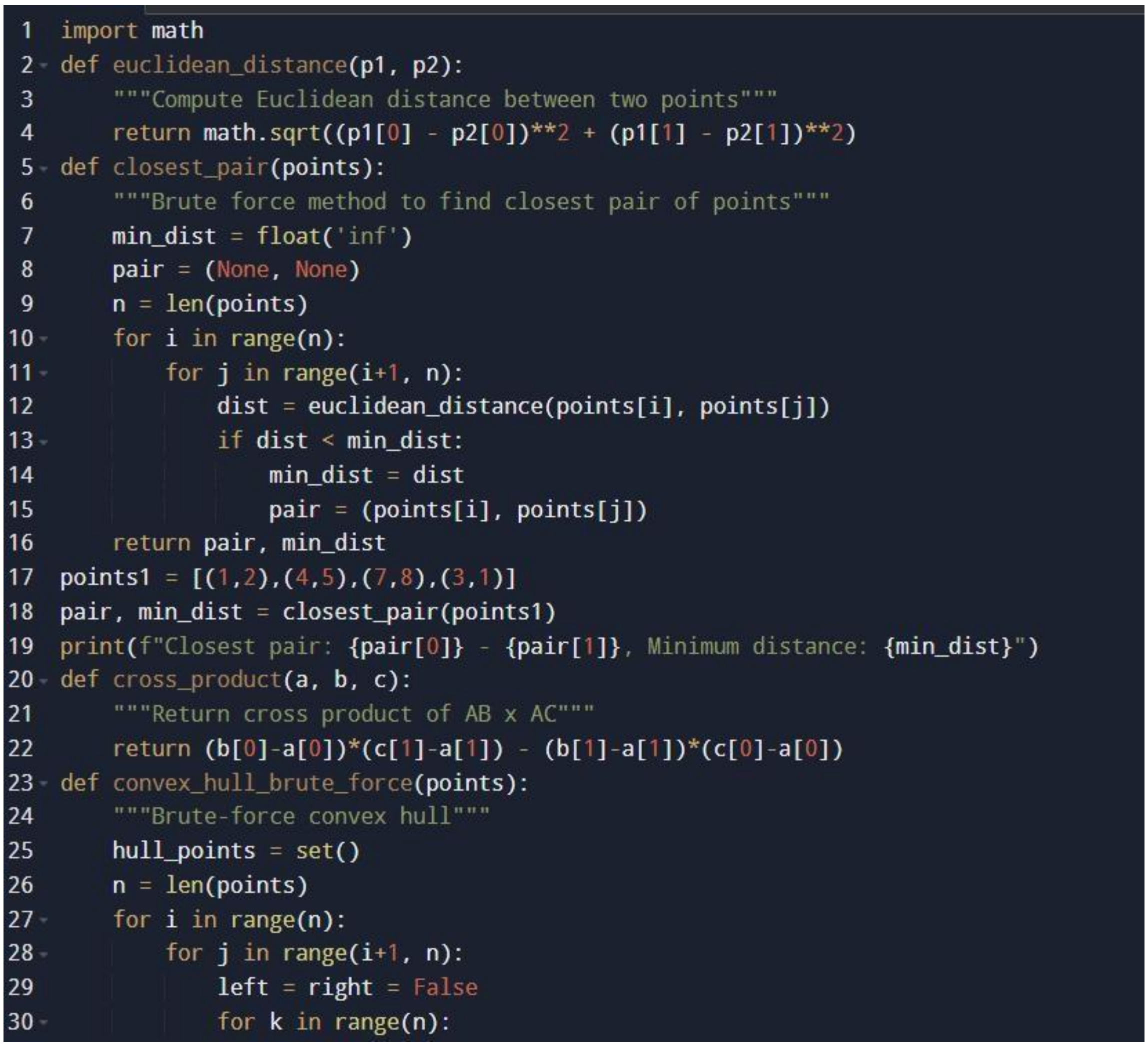
```
Closest pair: (1, 2) - (3, 1) Minimum distance: 2.23606797749979

=== Code Execution Successful ===
```

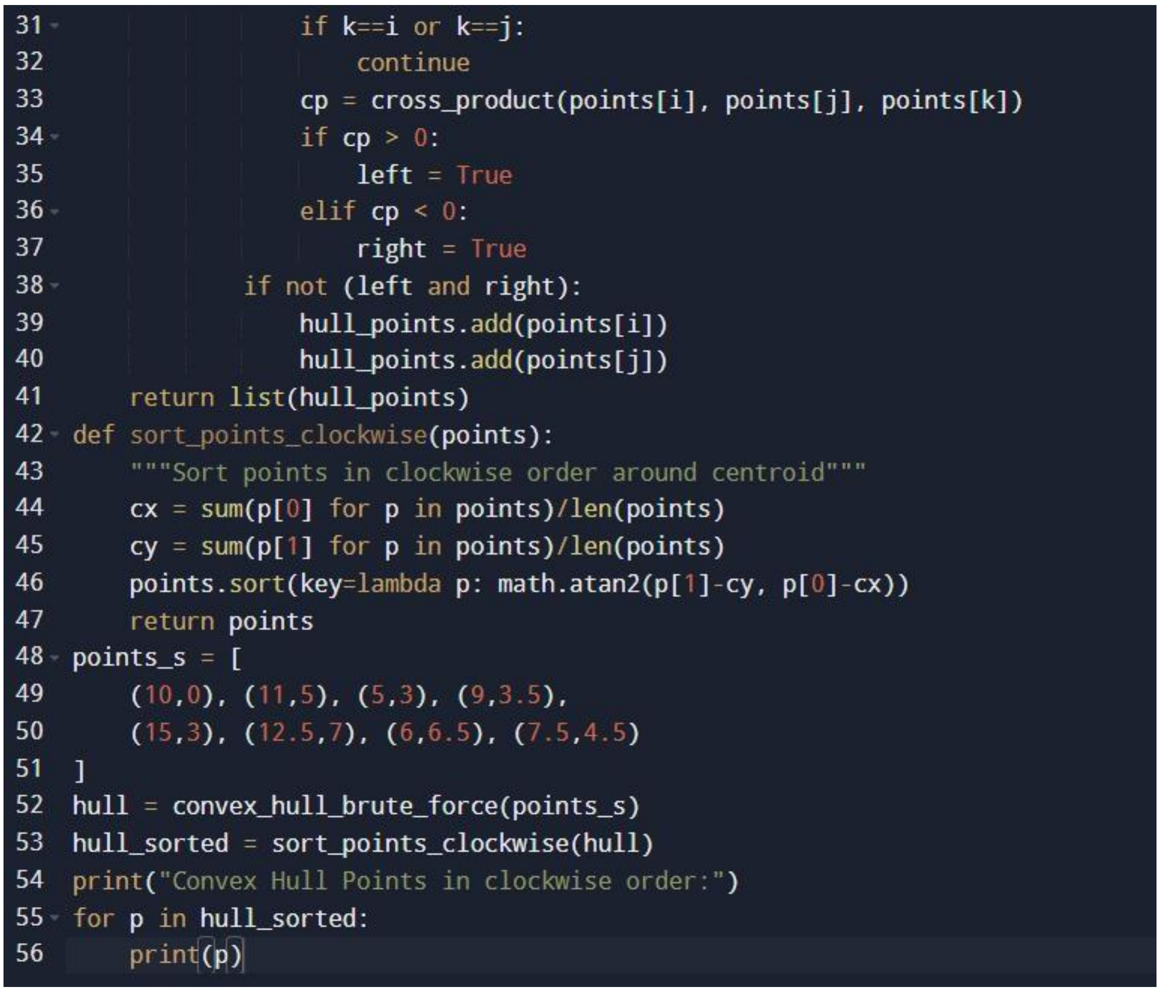
10. Write a program to find the closest pair of points in a given set using the brute force approach. Analyze the time complexity of your implementation. Define a function to calculate the Euclidean distance between two points. Implement a function to find the closest pair of points using the brute force 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).

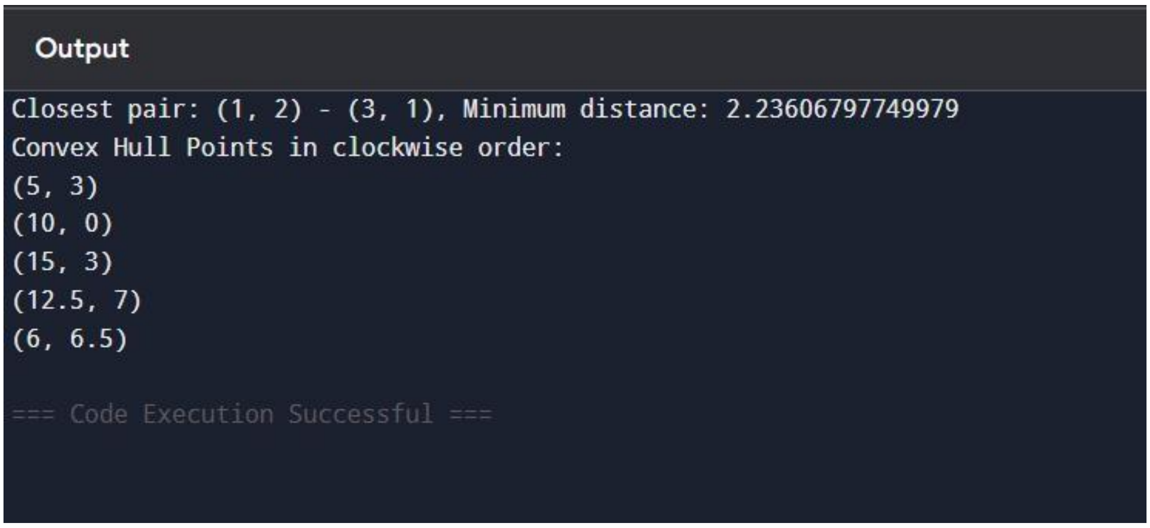
output: P3, P4, P6, P5, P7, P1



```python
import math
def euclidean_distance(p1, p2):
    """Compute Euclidean distance between two points"""
    return math.sqrt((p1[0] - p2[0])**2 + (p1[1] - p2[1])**2)
def closest_pair(points):
    """Brute force method to find closest pair of points"""
    min_dist = float('inf')
    pair = (None, 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_dist:
                min_dist = dist
                pair = (points[i], points[j])
    return pair, min_dist
points1 = [(1,2),(4,5),(7,8),(3,1)]
pair, min_dist = closest_pair(points1)
print(f"Closest pair: {pair[0]} - {pair[1]}, Minimum distance: {min_dist}")
def cross_product(a, b, c):
    """Return cross product of AB x AC"""
    return (b[0]-a[0])*(c[1]-a[1]) - (b[1]-a[1])*(c[0]-a[0])
def convex_hull_brute_force(points):
    """Brute-force convex hull"""
    hull_points = set()
    n = len(points)
    for i in range(n):
        for j in range(i+1, n):
            left = right = False
            for k in range(n):
```



```python
                if k==i or k==j:
                    continue
                cp = cross_product(points[i], points[j], points[k])
                if cp > 0:
                    left = True
                elif cp < 0:
                    right = True
            if not (left and right):
                hull_points.add(points[i])
                hull_points.add(points[j])
    return list(hull_points)
def sort_points_clockwise(points):
    """Sort points in clockwise order around centroid"""
    cx = sum(p[0] for p in points)/len(points)
    cy = sum(p[1] for p in points)/len(points)
    points.sort(key=lambda p: math.atan2(p[1]-cy, p[0]-cx))
    return points
points_s = [
    (10,0), (11,5), (5,3), (9,3.5),
    (15,3), (12.5,7), (6,6.5), (7.5,4.5)
]
hull = convex_hull_brute_force(points_s)
hull_sorted = sort_points_clockwise(hull)
print("Convex Hull Points in clockwise order:")
for p in hull_sorted:
    print(p)
```



Output

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

=== Code Execution Successful ===
```

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

Input:

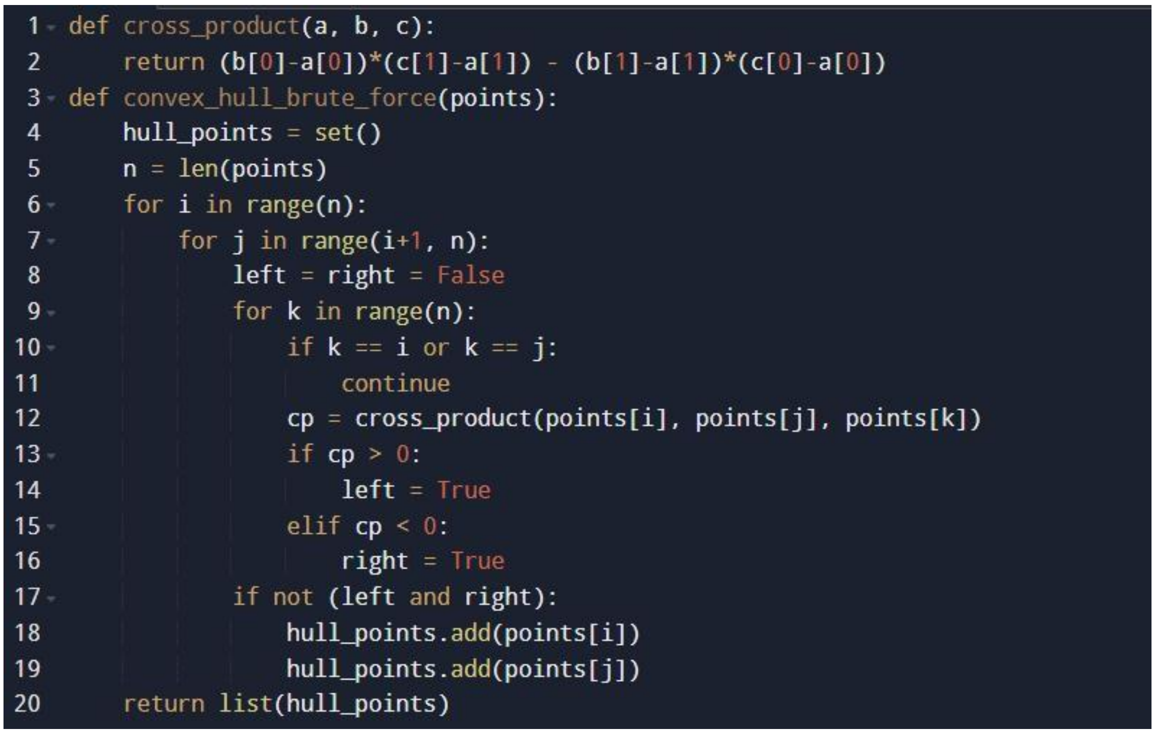
A list or array of points represented by coordinates (x, y).

Points: [(1, 1), (4, 6), (8, 1), (0, 0), (3, 3)]

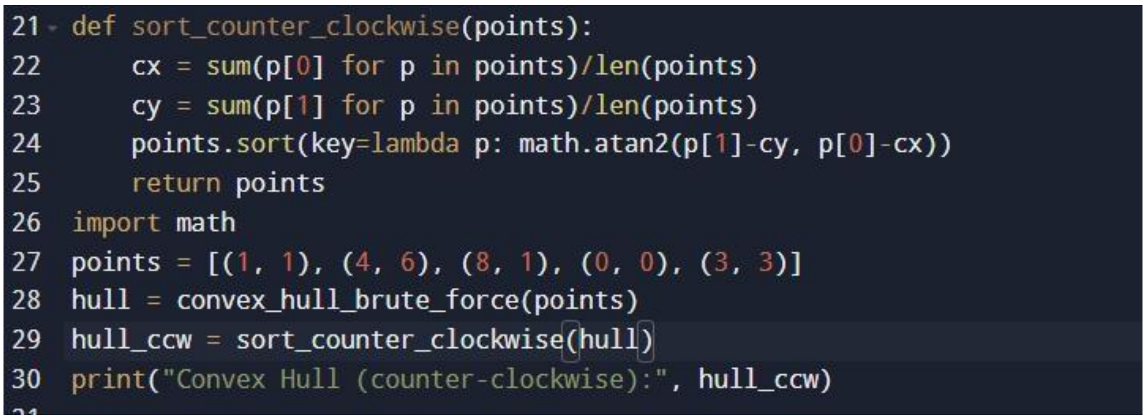
Output:

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

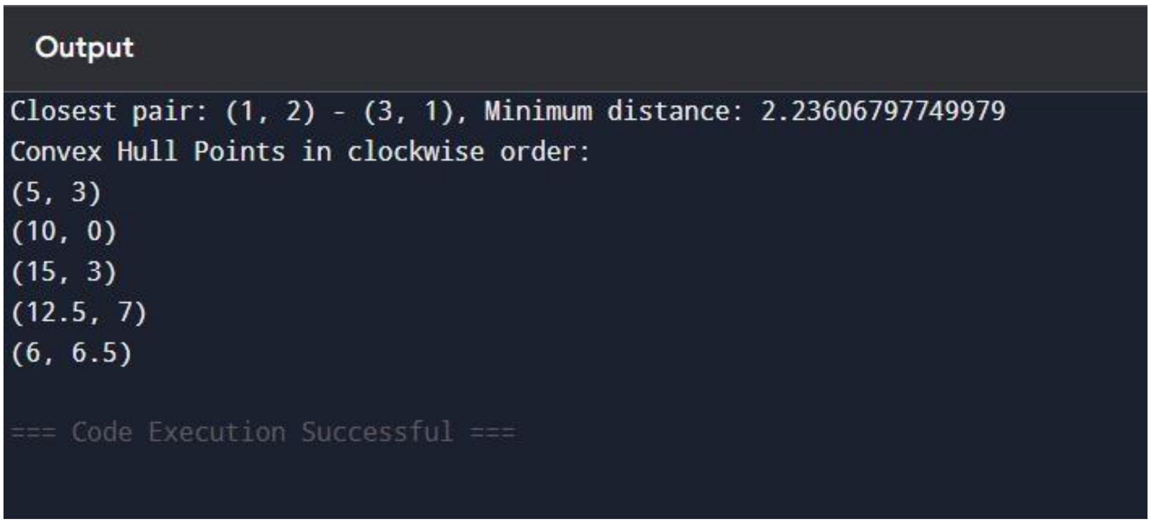
Convex Hull: [(0, 0), (1, 1), (8, 1), (4, 6)]



```python
def cross_product(a, b, c):
    return (b[0]-a[0])*(c[1]-a[1]) - (b[1]-a[1])*(c[0]-a[0])
def convex_hull_brute_force(points):
    hull_points = set()
    n = len(points)
    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
                cp = cross_product(points[i], points[j], points[k])
                if cp > 0:
                    left = True
                elif cp < 0:
                    right = True
            if not (left and right):
                hull_points.add(points[i])
                hull_points.add(points[j])
    return list(hull_points)
```



```python
def sort_counter_clockwise(points):
    cx = sum(p[0] for p in points)/len(points)
    cy = sum(p[1] for p in points)/len(points)
    points.sort(key=lambda p: math.atan2(p[1]-cy, p[0]-cx))
    return points
import math
points = [(1, 1), (4, 6), (8, 1), (0, 0), (3, 3)]
hull = convex_hull_brute_force(points)
hull_ccw = sort_counter_clockwise(hull)
print("Convex Hull (counter-clockwise):", hull_ccw)
```


Output

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

=== Code Execution Successful ===
```

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:

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:

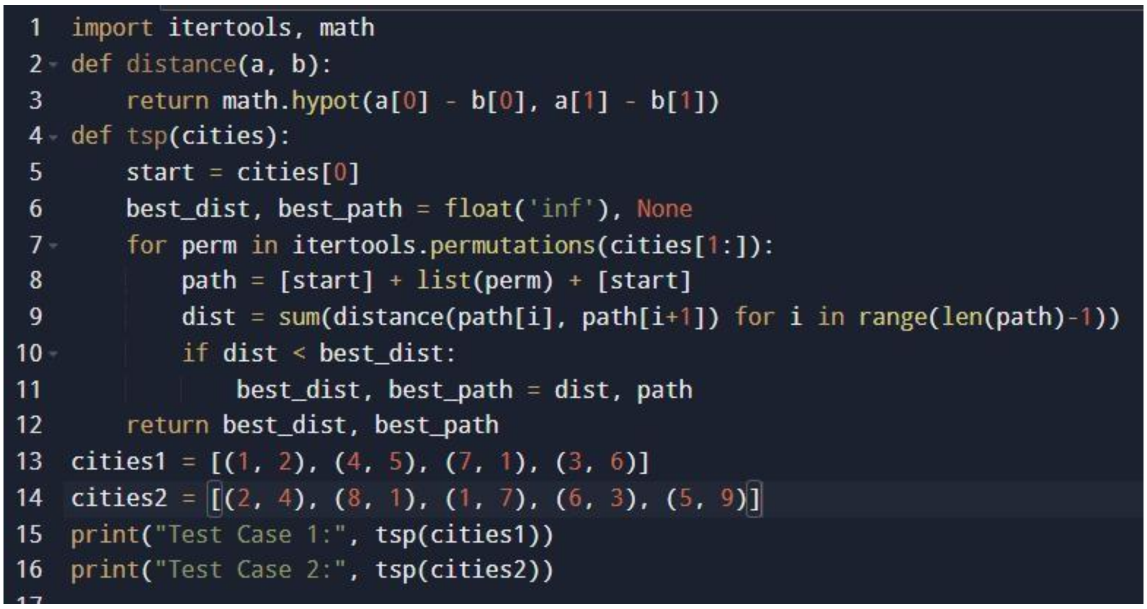
Shortest Distance: 7.0710678118654755

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

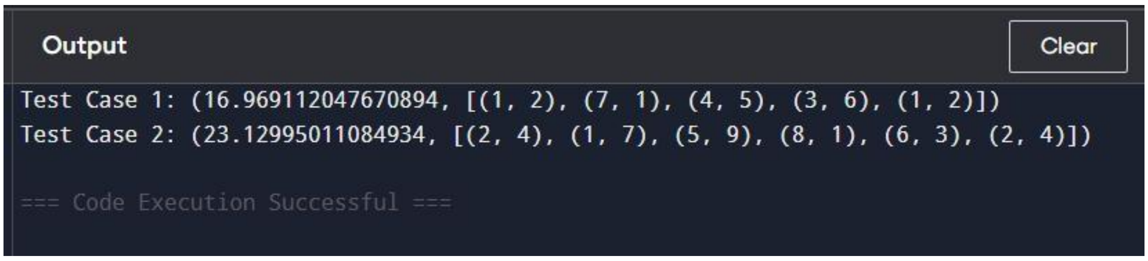
Test Case 2:

Shortest Distance: 14.142135623730951

Shortest Path: [(2, 4), (1, 7), (6, 3), (5, 9), (8, 1), (2, 4)]



```python
import itertools, math
def distance(a, b):
    return math.hypot(a[0] - b[0], a[1] - b[1])
def tsp(cities):
    start = cities[0]
    best_dist, best_path = float('inf'), None
    for perm in itertools.permutations(cities[1:]):
        path = [start] + list(perm) + [start]
        dist = sum(distance(path[i], path[i+1]) for i in range(len(path)-1))
        if dist < best_dist:
            best_dist, best_path = dist, path
    return best_dist, best_path
cities1 = [(1, 2), (4, 5), (7, 1), (3, 6)]
cities2 = [(2, 4), (8, 1), (1, 7), (6, 3), (5, 9)]
print("Test Case 1:", tsp(cities1))
print("Test Case 2:", tsp(cities2))
```



```
Test Case 1: (16.969112047670894, [(1, 2), (7, 1), (4, 5), (3, 6), (1, 2)])
Test Case 2: (23.12995011084934, [(2, 4), (1, 7), (5, 9), (8, 1), (6, 3), (2, 4)])

=== Code Execution Successful ===
```

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:

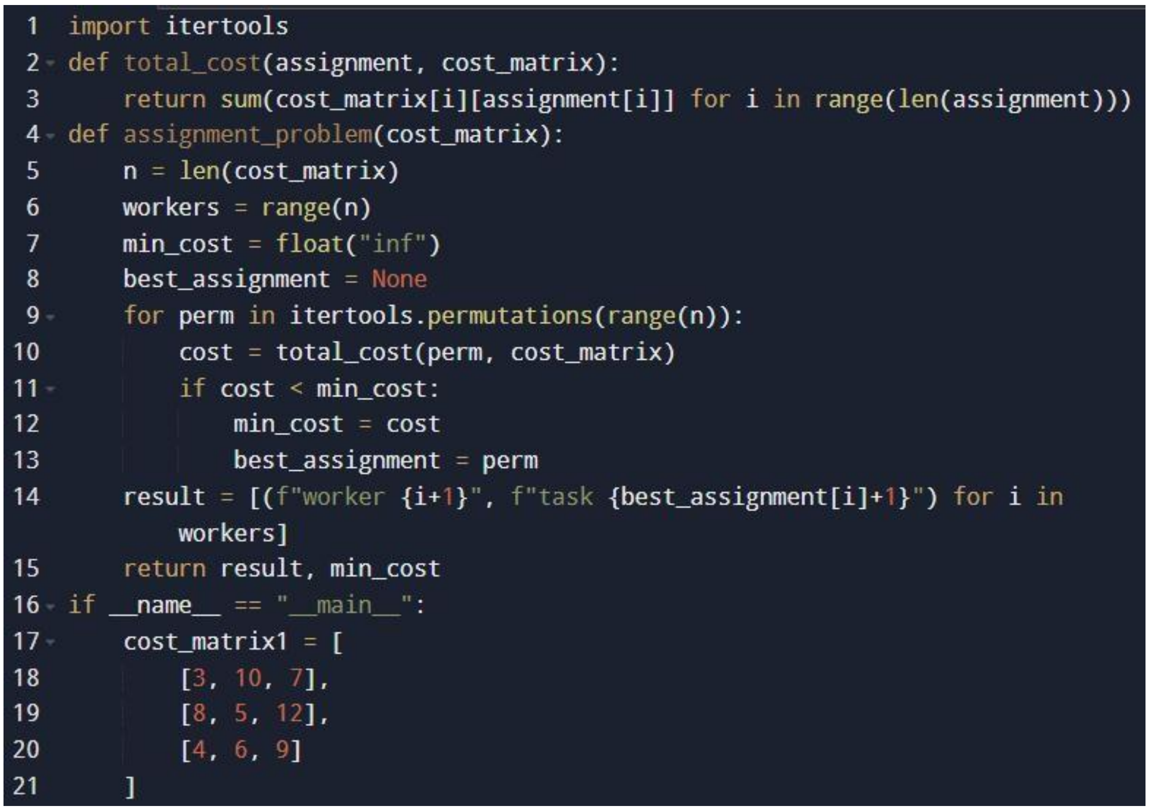
Optimal Assignment: [(worker 1, task 2), (worker 2, task 1), (worker 3, task 3)]

Total Cost: 19

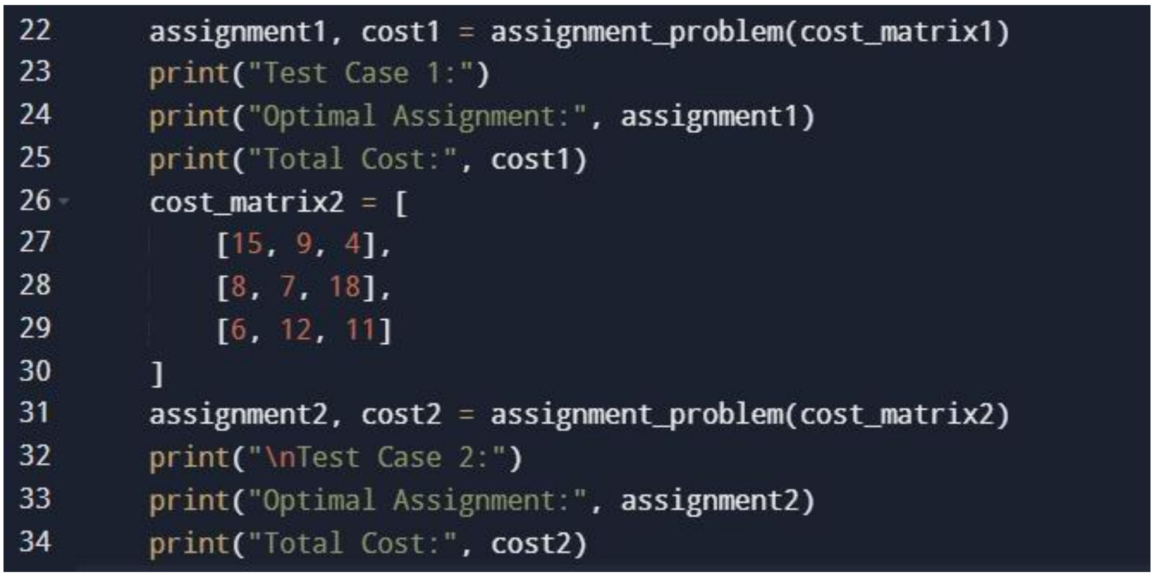
Test Case 2:

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

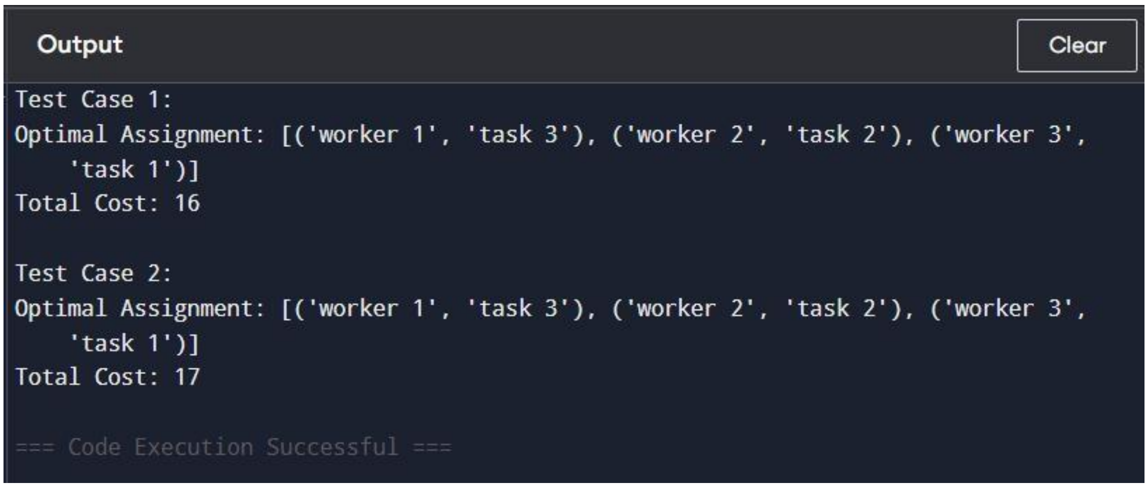
Total Cost: 24



```python
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)
    workers = range(n)
    min_cost = float("inf")
    best_assignment = None
    for perm in itertools.permutations(range(n)):
        cost = total_cost(perm, cost_matrix)
        if cost < min_cost:
            min_cost = cost
            best_assignment = perm
    result = [(f"worker {i+1}", f"task {best_assignment[i]+1}") for i in workers]
    return result, min_cost
if __name__ == "__main__":
    cost_matrix1 = [
        [3, 10, 7],
        [8, 5, 12],
        [4, 6, 9]
    ]
```



```python
    assignment1, cost1 = assignment_problem(cost_matrix1)
    print("Test Case 1:")
    print("Optimal Assignment:", assignment1)
    print("Total Cost:", cost1)
    cost_matrix2 = [
        [15, 9, 4],
        [8, 7, 18],
        [6, 12, 11]
    ]
    assignment2, cost2 = assignment_problem(cost_matrix2)
    print("\nTest Case 2:")
    print("Optimal Assignment:", assignment2)
    print("Total Cost:", cost2)
```



```
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

=== Code Execution Successful ===
```

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.

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:

Test Case 1:

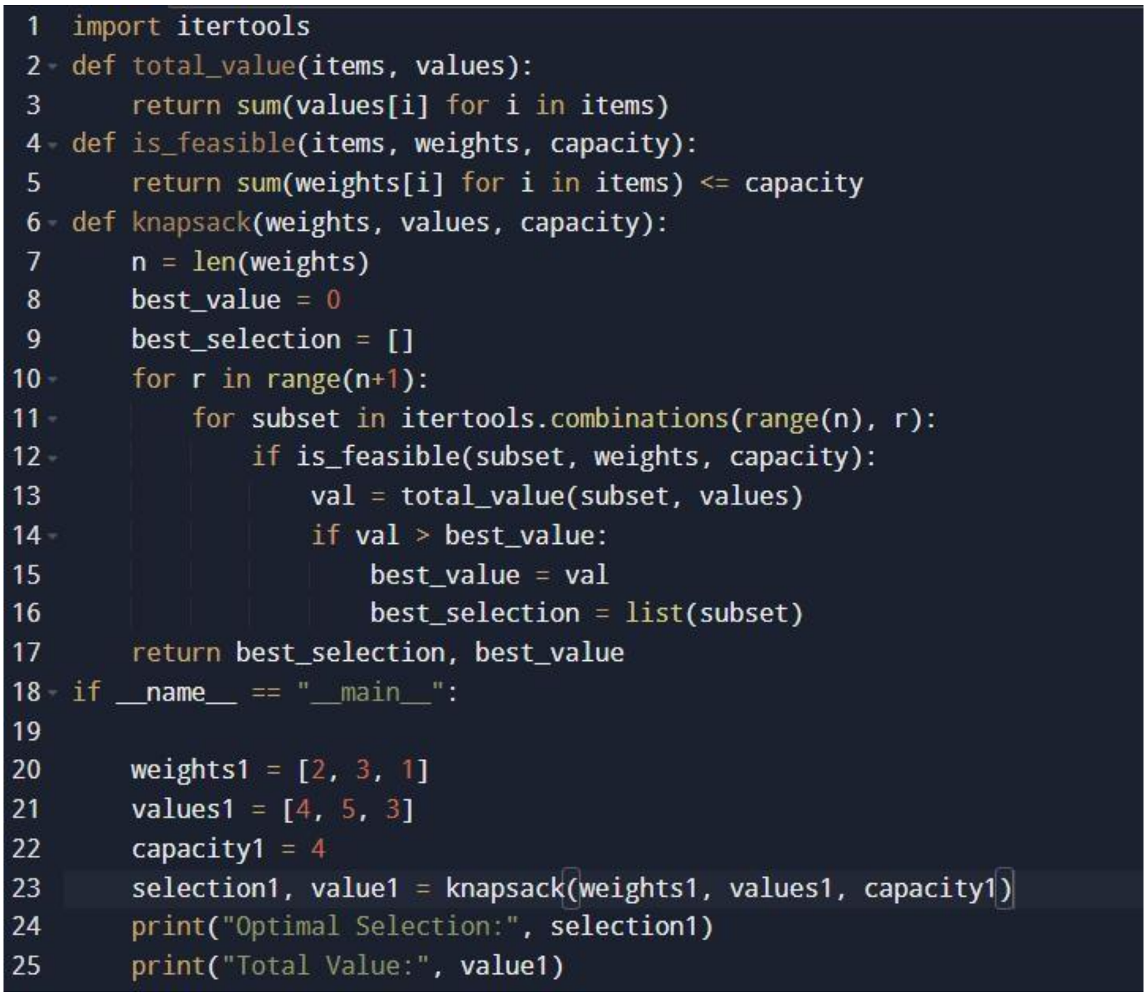
Optimal Selection: [0, 2] (Items with indices 0 and 2)

Total Value: 7

Test Case 2:

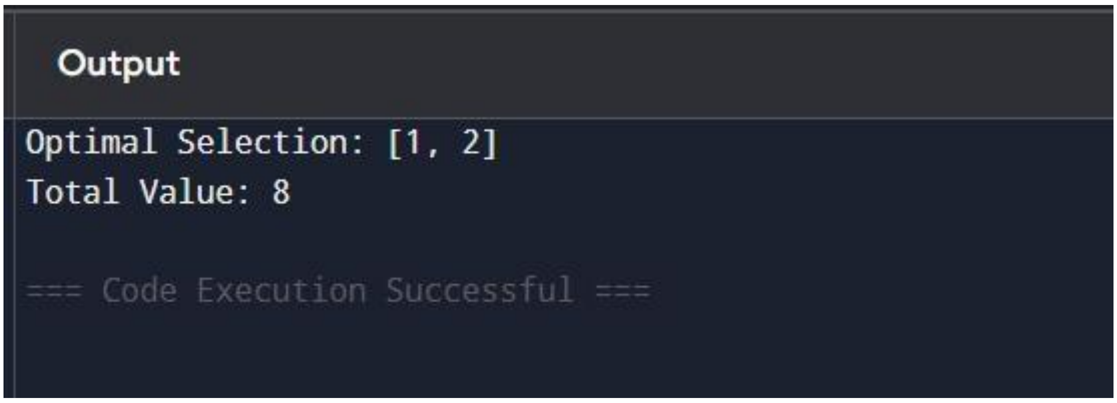
Optimal Selection: [0, 1, 2] (Items with indices 0, 1, and 2)

Total Value: 10



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

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



Output

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


=== Code Execution Successful ===
```