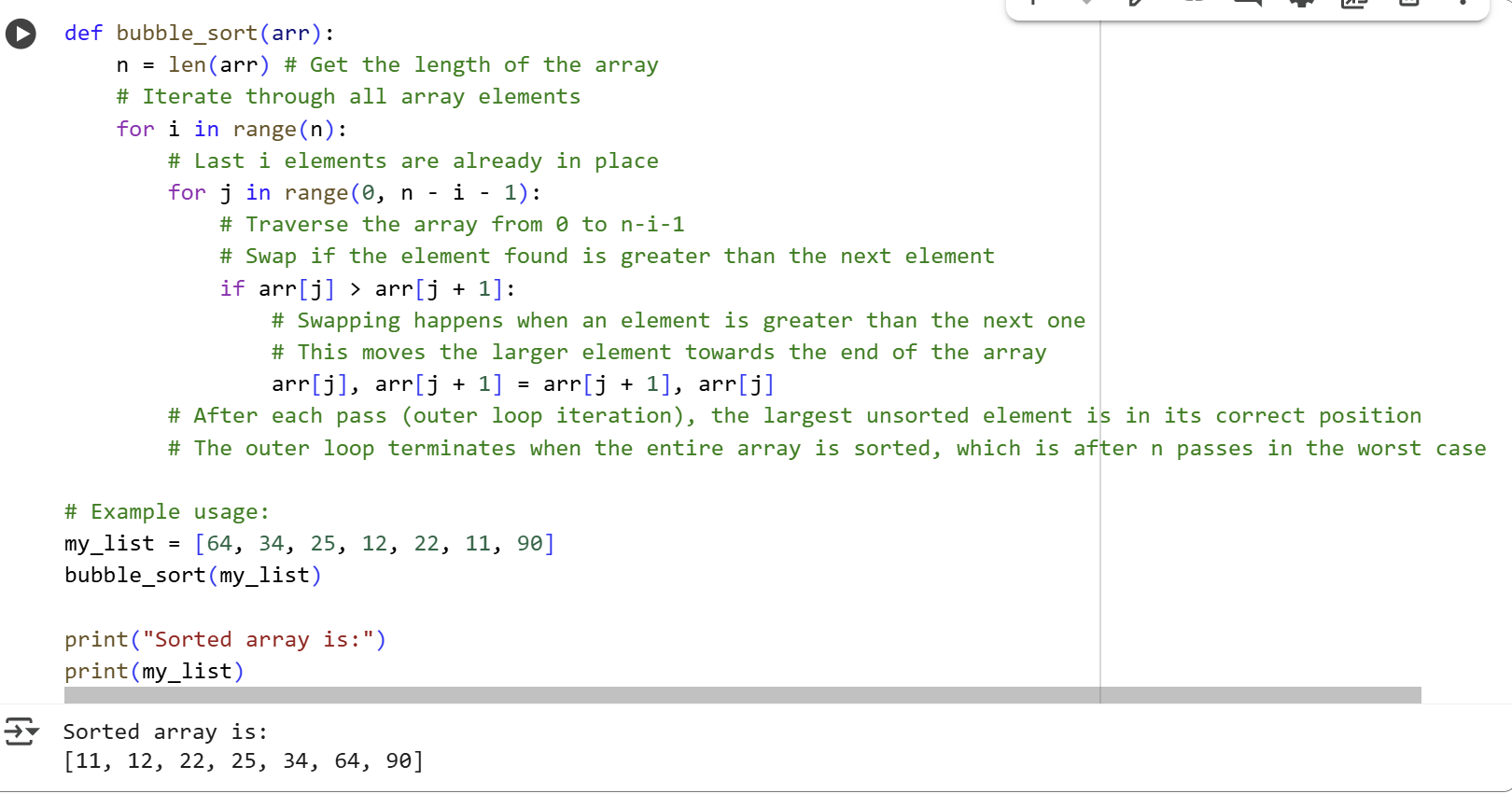
**AI Assignment-12**

**HTNO:2403A52134**

**Task-1:**



**Explanation:**

def bubble\_sort(arr):  
    n = len(arr) # Get the length of the array

This line defines a function called bubble\_sort that takes one argument, arr, which is the list or array to be sorted. Inside the function, n is assigned the length of the input array arr.

    # Iterate through all array elements  
    for i in range(n):

This is the start of the outer loop. It iterates n times, where n is the length of the array. The variable i keeps track of the number of passes completed. After each pass i, the last i elements of the array are sorted and in their correct position.

        # Last i elements are already in place  
        for j in range(0, n - i - 1):

This is the inner loop. It iterates through the unsorted portion of the array. n - i - 1 is the upper limit of the range. As the outer loop progresses (i.e., i increases), the upper limit of the inner loop decreases because the last i elements are already sorted.

            # Traverse the array from 0 to n-i-1  
            # Swap if the element found is greater than the next element  
            if arr[j] > arr[j + 1]:

This is the comparison step. Inside the inner loop, this if statement compares the current element arr[j] with the next element arr[j + 1].

                # Swapping happens when an element is greater than the next one  
                # This moves the larger element towards the end of the array  
                arr[j], arr[j + 1] = arr[j + 1], arr[j]

If the condition in the if statement is true (i.e., arr[j] is greater than arr[j + 1]), these two lines perform a swap. The values of arr[j] and arr[j + 1] are exchanged, effectively moving the larger element to the right (towards the end of the array).

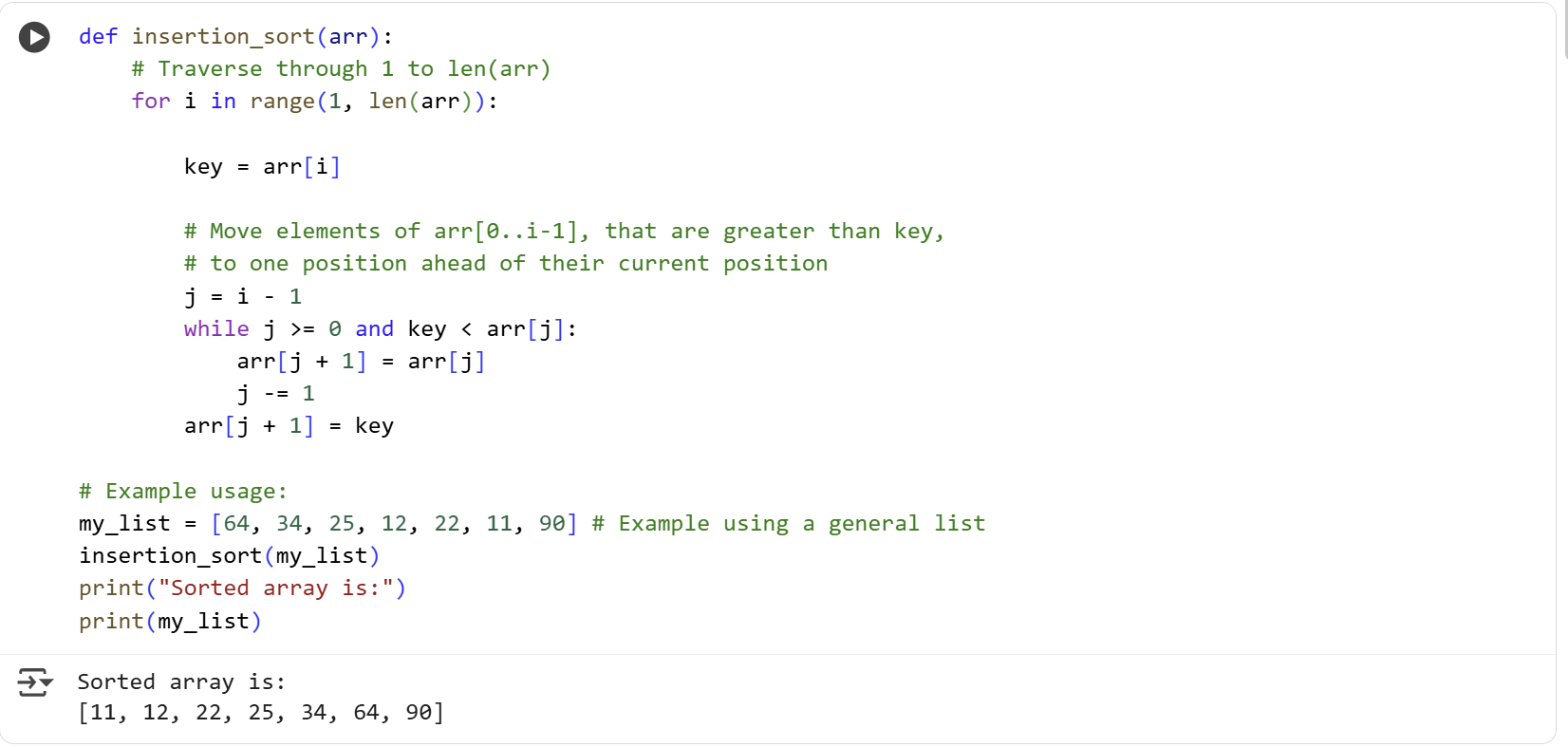
        # After each pass (outer loop iteration), the largest unsorted element is in its correct position  
        # The outer loop terminates when the entire array is sorted, which is after n passes in the worst case  
  
# Example usage:  
my\_list = [64, 34, 25, 12, 22, 11, 90]  
bubble\_sort(my\_list)

These lines are outside the function definition. my\_list is an example list that will be sorted. The bubble\_sort(my\_list) line calls the function to sort this list.

print("Sorted array is:")  
print(my\_list)

These lines print the sorted array after the bubble\_sort function has been executed.

**Task-2:**



**Explanation**:

def find\_duplicates\_brute\_force(arr):  
    """  
    Finds duplicate elements in a list using a brute-force approach (O(n^2)).  
  
    Args:  
        arr: The list to search for duplicates.  
  
    Returns:  
        A list of duplicate elements found in the input list.  
    """

This defines the function find\_duplicates\_brute\_force which takes one argument, arr, the list to search. The docstring explains its purpose, arguments, return value, and notes the O(n^2) time complexity.

    duplicates = []  
    n = len(arr)

An empty list duplicates is created to store unique duplicate elements. n stores the number of elements in the input list arr.

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

These are nested loops. The outer loop with index i iterates from the first element to the last. The inner loop with index j iterates from the element *after* i to the last element. This ensures every pair of elements is compared exactly once.

            if arr[i] == arr[j] and arr[i] not in duplicates:

This if statement checks if the element at index i is equal to the element at index j (meaning a duplicate is found) AND if this duplicate element is not already in the duplicates list.

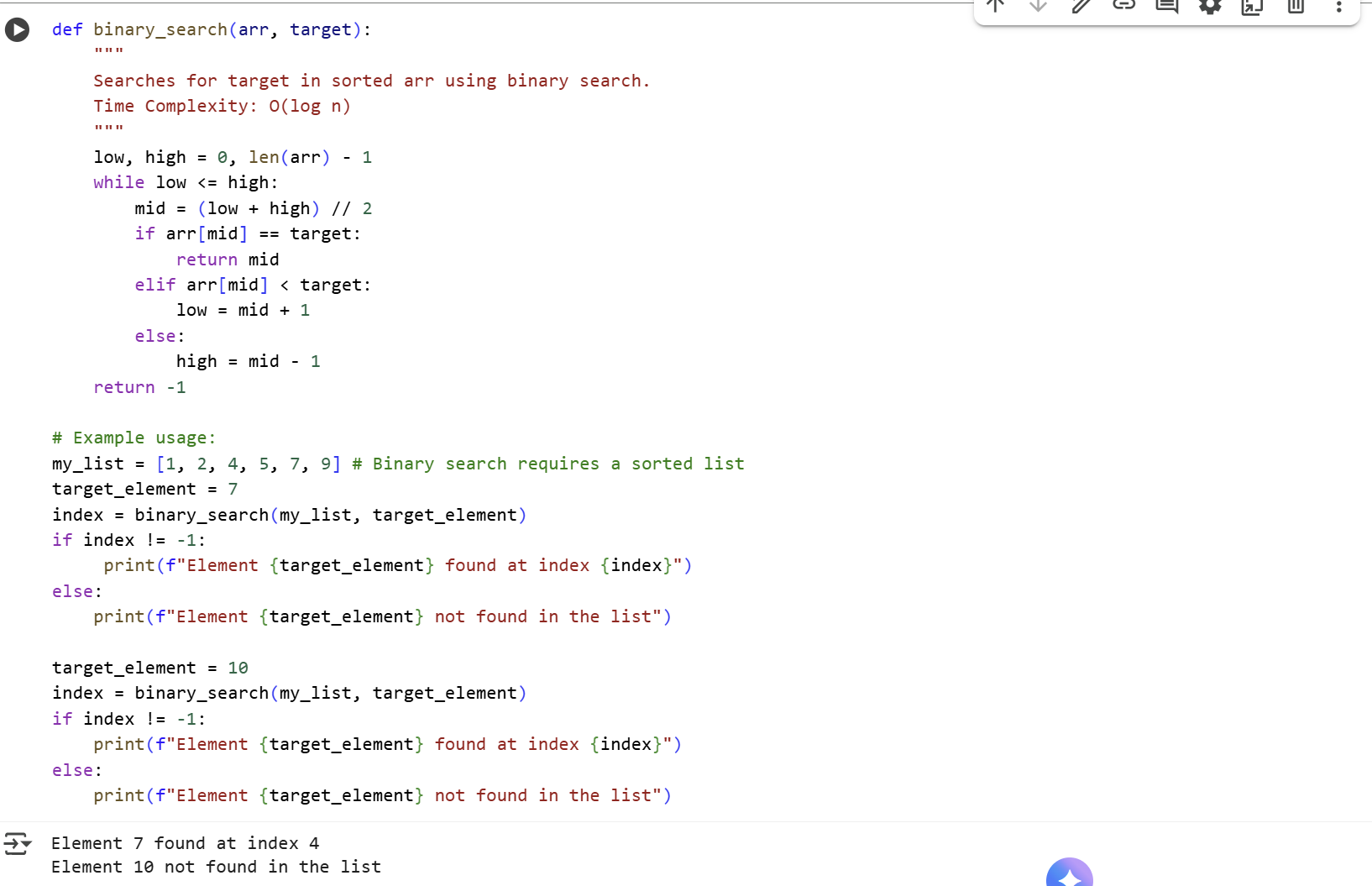
                duplicates.append(arr[i])

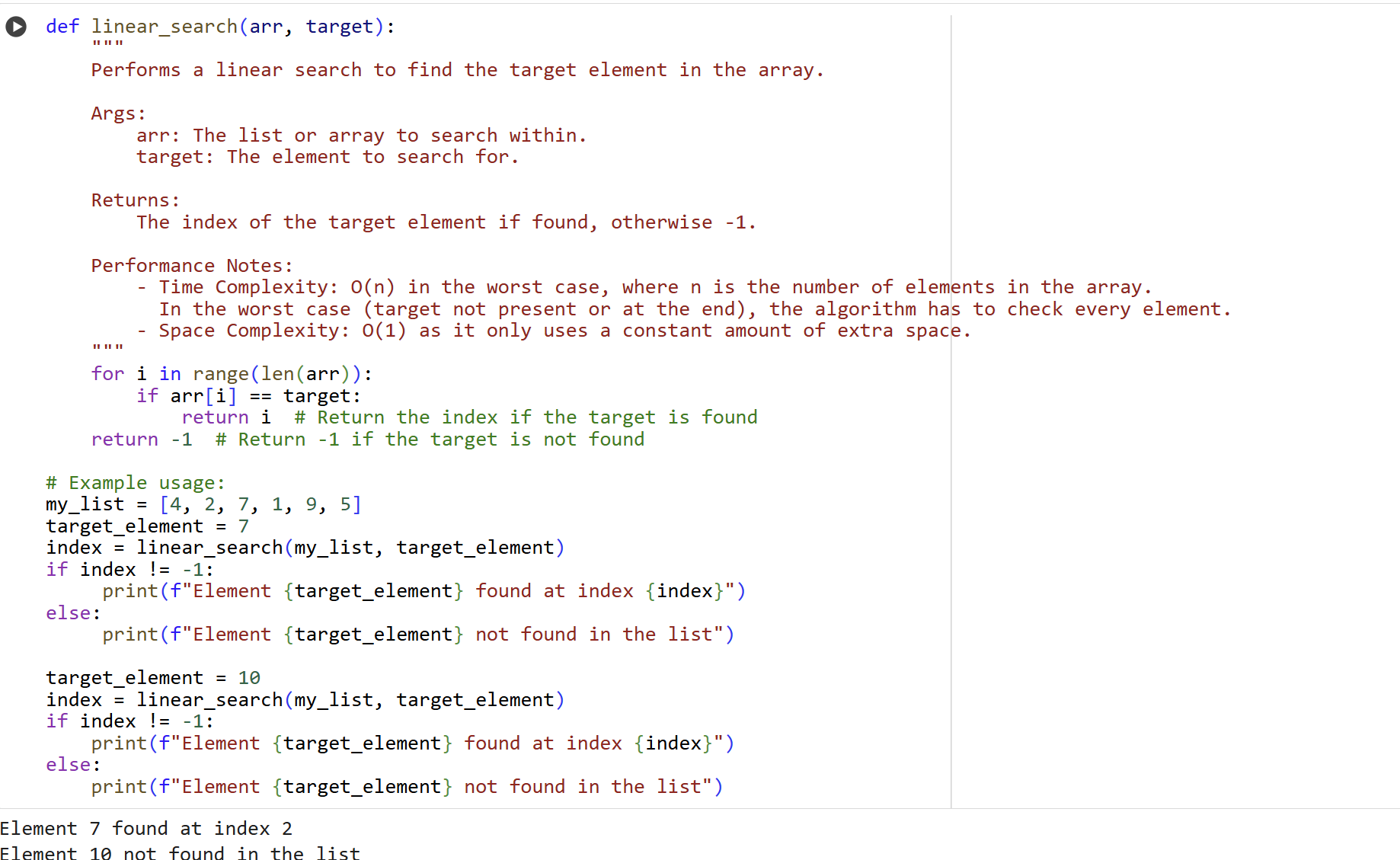
If both conditions in the if statement are true, the duplicate element arr[i] is added to the duplicates list.

    return duplicates

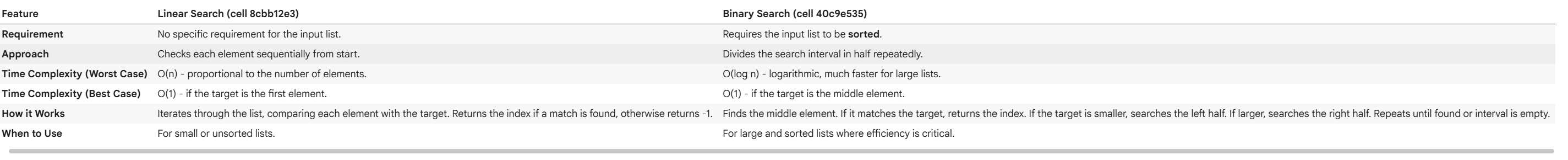
Finally, the function returns the list duplicates containing all the unique duplicate elements found.

**Task-3:**

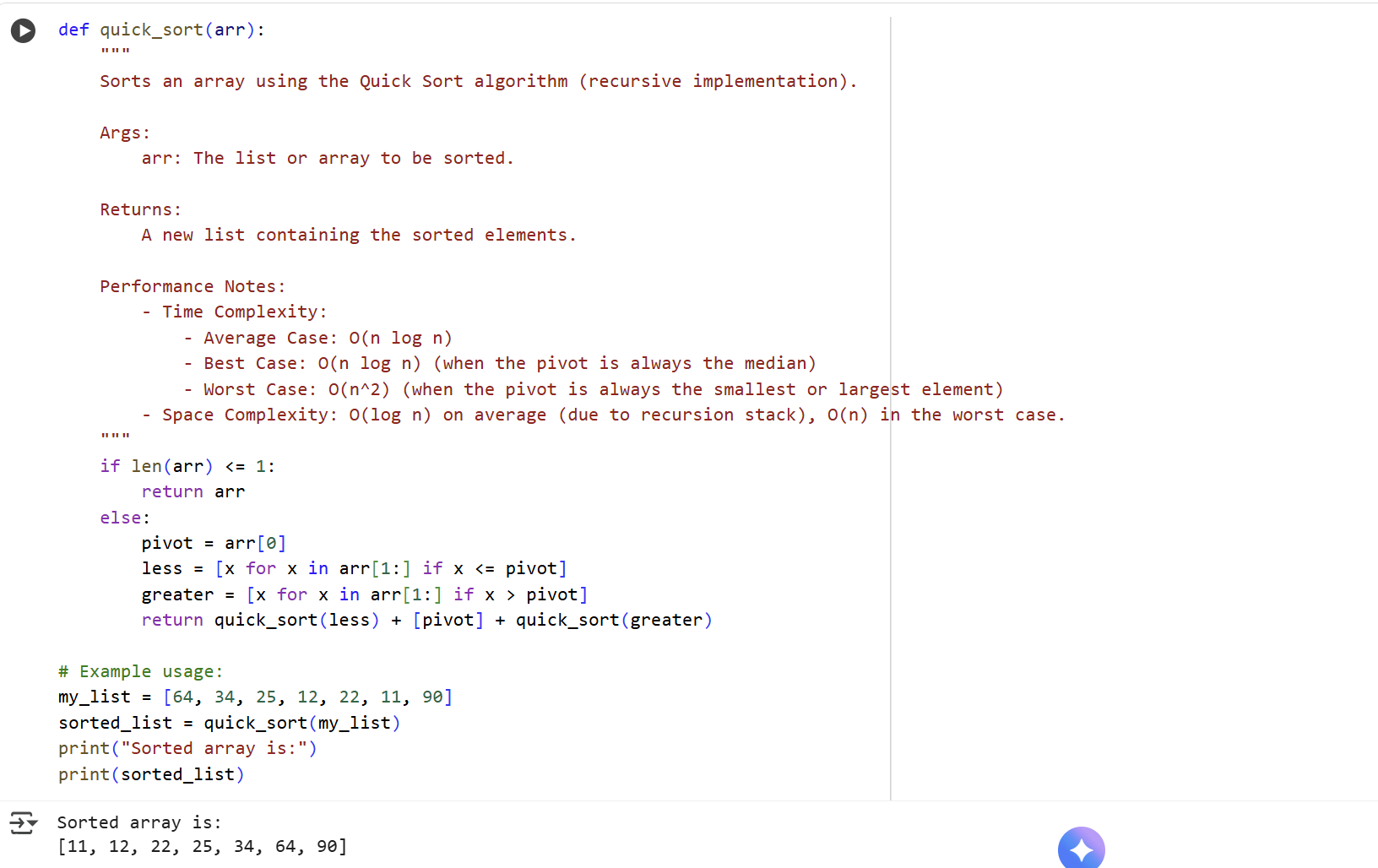


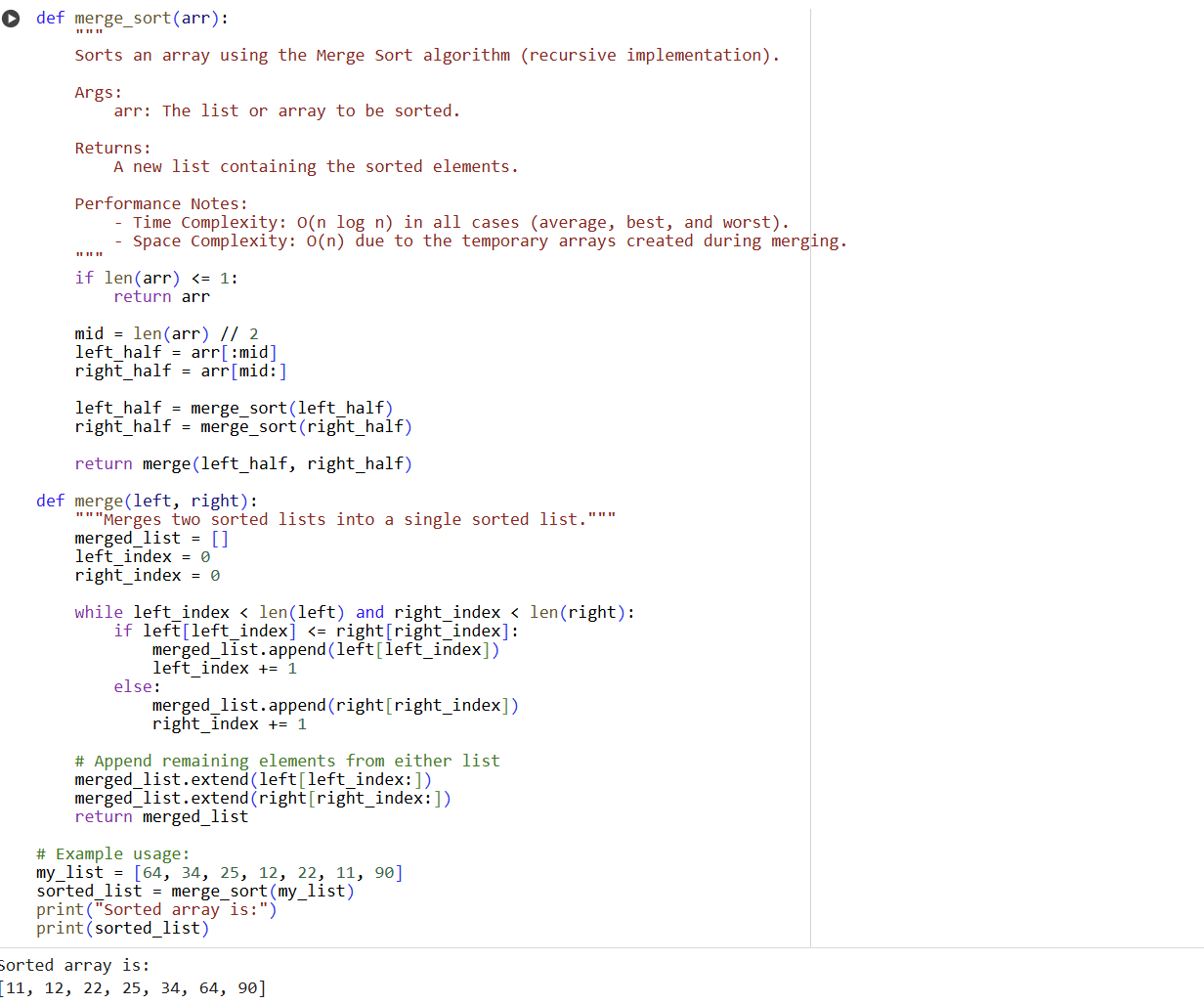


**Explanation:**



**Task-4:**



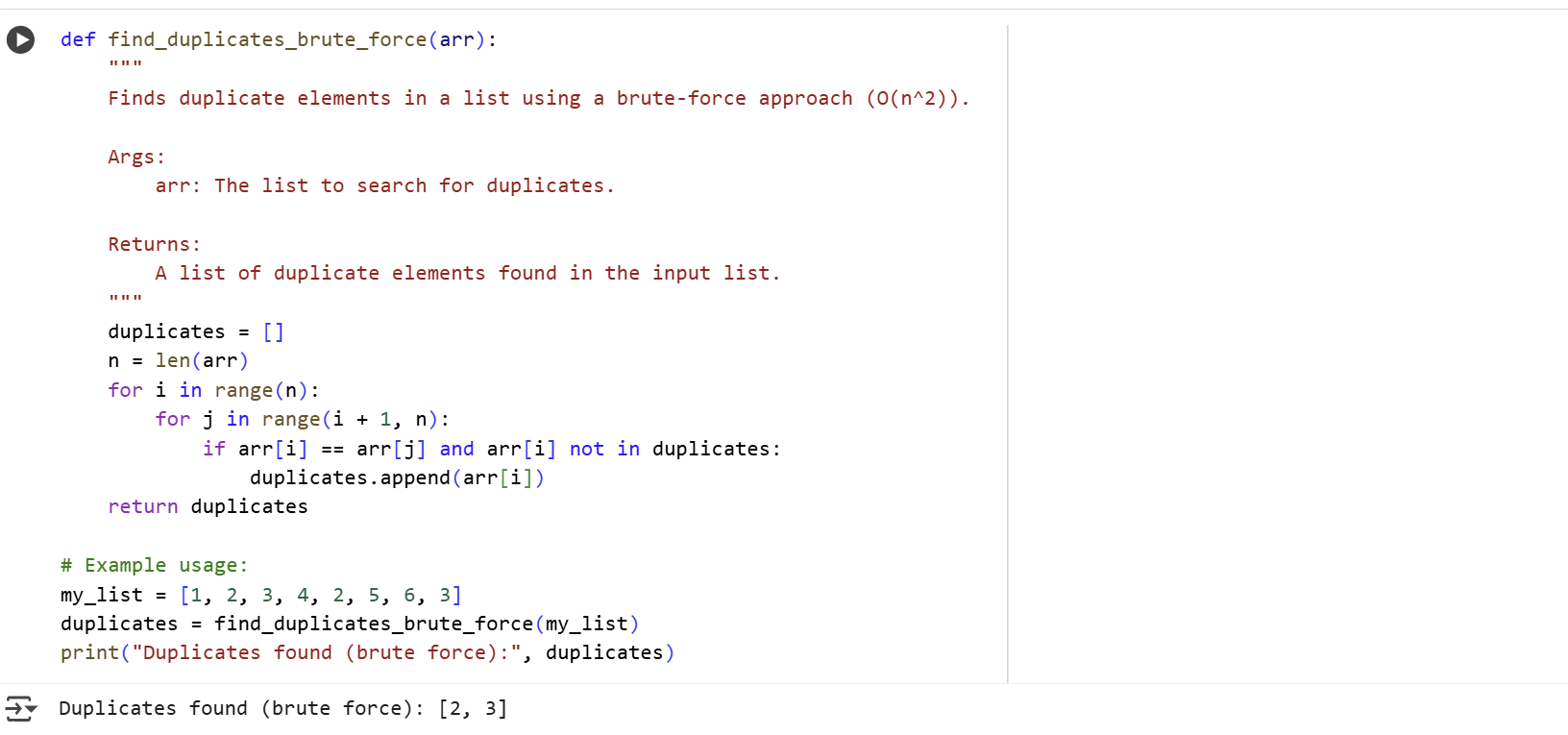


* **Explanation:**
* **def merge\_sort(arr):**: This line defines a function named merge\_sort that takes one argument, arr, which is the list to be sorted.
* **""" ... """**: This is a docstring that explains what the function does, its arguments, what it returns, and its performance characteristics.
* **if len(arr) <= 1:**: This is the base case for the recursion. If the length of the list arr is less than or equal to 1, it means the list is already considered sorted.
* **return arr**: If the base case is true, the function returns the list as is.
* **mid = len(arr) // 2**: This line calculates the middle index of the list arr using integer division (//).
* **left\_half = arr[:mid]**: This creates a new list left\_half containing elements from the beginning of arr up to (but not including) the mid index.
* **right\_half = arr[mid:]**: This creates a new list right\_half containing elements from the mid index to the end of arr.
* **left\_half = merge\_sort(left\_half)**: This line recursively calls merge\_sort on the left\_half. The result of this recursive call (the sorted left half) is then assigned back to left\_half.
* **right\_half = merge\_sort(right\_half)**: This line recursively calls merge\_sort on the right\_half. The result of this recursive call (the sorted right half) is then assigned back to right\_half.
* **return merge(left\_half, right\_half)**: This line calls the merge function with the sorted left\_half and right\_half and returns the result, which is the fully sorted merged list.
* **def merge(left, right):**: This line defines a helper function named merge that takes two sorted lists, left and right, as arguments.
* **"""Merges two sorted lists into a single sorted list."""**: This is a docstring for the merge function.
* **merged\_list = []**: This initializes an empty list called merged\_list to store the merged sorted elements.
* **left\_index = 0**: This initializes an index left\_index to 0 for the left list.
* **right\_index = 0**: This initializes an index right\_index to 0 for the right list.
* **while left\_index < len(left) and right\_index < len(right):**: This loop continues as long as there are elements to compare in both the left and right lists.
* **if left[left\_index] <= right[right\_index]:**: This checks if the current element in the left list is less than or equal to the current element in the right list.
* **merged\_list.append(left[left\_index])**: If the element in left is smaller or equal, it's appended to merged\_list.
* **left\_index += 1**: The left\_index is incremented to move to the next element in the left list.
* **else:**: If the element in the right list is smaller.
* **merged\_list.append(right[right\_index])**: The element in right is appended to merged\_list.
* **right\_index += 1**: The right\_index is incremented to move to the next element in the right list.
* **merged\_list.extend(left[left\_index:])**: After the while loop, this line appends any remaining elements in the left list (if any) to merged\_list.
* **merged\_list.extend(right[right\_index:])**: This line appends any remaining elements in the right list (if any) to merged\_list.
* **return merged\_list**: The function returns the fully merged and sorted list.
* **# Example usage:**: This is a comment indicating the start of the example usage.
* **my\_list = [64, 34, 25, 12, 22, 11, 90]**: This creates a sample list named my\_list to be sorted.
* **sorted\_list = merge\_sort(my\_list)**: This calls the merge\_sort function with my\_list and stores the returned sorted list in sorted\_list.
* **print("Sorted array is:")**: This line prints a descriptive string to the console.
* **print(sorted\_list)**: This line prints the contents of the sorted\_list to the console.

Quick sort:

1. **def quick\_sort(arr):**: This defines the function quick\_sort that takes a list arr as input.
2. **""" ... """**: This is the docstring, explaining the function's purpose, arguments, return value, and performance characteristics.
3. **if len(arr) <= 1:**: This is the base case for the recursion. If the list has 0 or 1 element, it's already sorted, so it's returned as is.
4. **else:**: If the list has more than one element, the sorting process begins.
5. **pivot = arr[0]**: The first element of the list is chosen as the "pivot". Other pivot selection strategies exist, but this is a simple one.
6. **less = [x for x in arr[1:] if x <= pivot]**: This creates a new list called less containing all elements from the rest of the original list (arr[1:]) that are less than or equal to the pivot. This is done using a list comprehension.
7. **greater = [x for x in arr[1:] if x > pivot]**: This creates another new list called greater containing all elements from the rest of the original list (arr[1:]) that are greater than the pivot. This also uses a list comprehension.
8. **return quick\_sort(less) + [pivot] + quick\_sort(greater)**: This is the core recursive step.
   * quick\_sort(less): The quick\_sort function is recursively called on the less list to sort it.
   * quick\_sort(greater): The quick\_sort function is recursively called on the greater list to sort it.
   * [pivot]: A list containing only the pivot element is created.
   * The three parts (quick\_sort(less), [pivot], and quick\_sort(greater)) are concatenated together in this order. Since less is sorted, then the pivot, and then greater is sorted, the resulting combined list is the fully sorted version of the original input list.

**Task-5:**



**Explanation:**

* **def find\_duplicates\_brute\_force(arr):**: This line defines a function named find\_duplicates\_brute\_force that takes one argument, arr, which is the list to search for duplicates within.
* **""" ... """**: This is a docstring that explains what the function does, its arguments, what it returns, and its time complexity.
* **duplicates = []**: This line initializes an empty list called duplicates. This list will be used to store the unique duplicate elements found in the input list.
* **n = len(arr)**: This gets the length of the input list arr and stores it in the variable n. This is used for the loop bounds.
* **for i in range(n):**: This is the outer loop. It iterates through the list from the first element (index 0) up to the last element (index n-1). The variable i represents the index of the current element being considered.
* **for j in range(i + 1, n):**: This is the inner loop. For each element at index i in the outer loop, this loop iterates through the rest of the list, starting from the element *after* the current element (i + 1) up to the last element. The variable j represents the index of the element being compared with arr[i].
* **if arr[i] == arr[j] and arr[i] not in duplicates:**: This is the core of the brute-force check.
  + arr[i] == arr[j]: This checks if the element at index i is equal to the element at index j. If they are equal, it means a potential duplicate has been found.
  + arr[i] not in duplicates: This additional check ensures that if a duplicate is found, it's only added to the duplicates list once. It checks if the element arr[i] is already present in the duplicates list.
* **duplicates.append(arr[i])**: If both conditions in the if statement are true (a duplicate is found and it hasn't been added to the duplicates list yet), the duplicate element arr[i] is appended to the duplicates list.
* **return duplicates**: After the nested loops complete, the function returns the duplicates list, which contains all the unique duplicate elements found in the input list.
* **# Example usage:**: This is a comment indicating the start of the example usage.
* **my\_list = [1, 2, 3, 4, 2, 5, 6, 3]**: This creates a sample list named my\_list to test the function.
* **duplicates = find\_duplicates\_brute\_force(my\_list)**: This calls the find\_duplicates\_brute\_force function with my\_list and stores the returned list of duplicates in the duplicates variable.
* **print("Duplicates found (brute force):", duplicates)**: This line prints a descriptive string and the contents of the duplicates list to the console.

This brute-force approach is straightforward but has a time complexity of O(n^2) because it involves nested loops, where for each element, it compares it with every other element in the remaining part of the list. For larger lists, this approach can become very slow.