Assignment 12.4

TASK_1:

Code:

```
Start coding or generate with AI.

[1]
0s
def bubble_sort(arr):
  """
  Sorts a list using the bubble sort algorithm.

  Args:
    arr: The list to be sorted.

  Returns:
    The sorted list.
  """
  n = len(arr)
  # Outer loop for passes through the list
  # The list becomes sorted from the end to the beginning with each pass
  # In each pass, the largest unsorted element "bubbles up" to its correct position
  for i in range(n):
    # Flag to optimize the sort - if no two elements are swapped
    # by inner loop, then the array is sorted
    swapped = False

    # Inner loop for comparisons and swaps
    # The range decreases with each pass because the end of the list
    # is already sorted
    for j in range(0, n - i - 1):
      # Compare adjacent elements
      if arr[j] > arr[j + 1]:
        # Swap the elements if they are in the wrong order
        # Swapping happens because the larger element needs to move to the right
        # to get to its sorted position.
        arr[j], arr[j + 1] = arr[j + 1], arr[j]
        swapped = True

    # If no two elements were swapped in inner loop, then break
    # The loop terminates when a pass is completed without any swaps,
    # indicating that the array is fully sorted.
    if swapped == False:
        break

  return arr

# Example usage:
my_list = [64, 34, 25, 12, 22, 11, 90]
print("Original list:", my_list)
sorted_list = bubble_sort(my_list)
print("Sorted list:", sorted_list)

# Time Complexity:
# Best Case: O(n) - when the array is already sorted
# Average Case: O(n^2)
# Worst Case: O(n^2) - when the array is sorted in reverse order
# Space Complexity: O(1) - it's an in-place sorting algorithm,
# using constant extra space
```

Output:

```
Original list: [64, 34, 25, 12, 22, 11, 90]
Sorted list: [11, 12, 22, 25, 34, 64, 90]
```

TASK_2:

Code:

```
def insertion_sort(arr):
  """
  Sorts a list using the insertion sort algorithm.

  Args:
    arr: The list to be sorted.

  Returns:
    The sorted list.
  """
  # Traverse through 1 to len(arr)
  for i in range(1, len(arr)):
    key = arr[i]
    # Move elements of arr[0..i-1], that are greater than key,
    # to one position ahead of their current position
    j = i - 1
    while j >= 0 and key < arr[j]:
      arr[j + 1] = arr[j]
      j -= 1
    arr[j + 1] = key
  return arr

# Example usage:
my_list_insertion = [64, 34, 25, 12, 22, 11, 90]
print("Original list for Insertion Sort:", my_list_insertion)
sorted_list_insertion = insertion_sort(my_list_insertion)
print("Sorted list using Insertion Sort:", sorted_list_insertion)

# Time Complexity:
# Best Case: O(n) - when the array is already sorted
# Average Case: O(n^2)
# Worst Case: O(n^2) - when the array is sorted in reverse order
# Space Complexity: O(1) - it's an in-place sorting algorithm
```

Output:

```
Original list for Insertion Sort: [64, 34, 25, 12, 22, 11, 90]
Sorted list using Insertion Sort: [11, 12, 22, 25, 34, 64, 90]
```

Task-3:

Code:

```python
def linear_search(arr, target):
  """
  Performs a linear search for a target value in a list.

  Args:
    arr: The list to search within.
    target: The value to search for.

  Returns:
    The index of the target in the list if found, otherwise -1.
  """
  # Iterate through each element in the list
  for i in range(len(arr)):
    # If the current element is the target, return its index
    if arr[i] == target:
      return i
  # If the target is not found after checking all elements, return -1
  return -1

# Example usage:
my_list_linear = [64, 34, 25, 12, 22, 11, 90]
target_linear = 22
index_linear = linear_search(my_list_linear, target_linear)

if index_linear != -1:
  print(f"Linear Search: Target {target_linear} found at index {index_linear}")
else:
  print(f"Linear Search: Target {target_linear} not found in the list")

# Performance Notes:
# Time Complexity:
# Best Case: O(1) - when the target is the first element
# Average Case: O(n) - on average, half the list needs to be checked
# Worst Case: O(n) - when the target is the last element or not in the list
# Space Complexity: O(1) - uses constant extra space
```

Output:

```
Linear Search: Target 22 found at index 4
```

TASK_4:

Code:

```python
def quick_sort(arr):
    """
    Sorts a list using the recursive Quick Sort algorithm.

    Args:
        arr: The list to be sorted.

    Returns:
        The sorted list.
    """
    if len(arr) <= 1:
        return arr  # Base case: an array with 0 or 1 elements is already sorted

    pivot = arr[len(arr) // 2]  # Choose a pivot element (middle element in this case)
    left = [x for x in arr if x < pivot]  # Elements less than the pivot
    middle = [x for x in arr if x == pivot] # Elements equal to the pivot
    right = [x for x in arr if x > pivot] # Elements greater than the pivot

    # Recursively sort the left and right sub-arrays and combine with the middle
    return quick_sort(left) + middle + quick_sort(right)

# Example usage:
my_list_quick = [64, 34, 25, 12, 22, 11, 90]
print("Original list for Quick Sort:", my_list_quick)
sorted_list_quick = quick_sort(my_list_quick)
print("Sorted list using Quick Sort:", sorted_list_quick)

# Performance Notes:
# Time Complexity:
# Best Case: O(n log n) - when the pivot is chosen such that it divides the array into roughly equal halves.
# Average Case: O(n log n) - generally performs well on average.
# Worst Case: O(n^2) - when the pivot choice consistently results in highly unbalanced partitions (e.g., always picking the smallest or l
# Space Complexity: O(log n) on average due to recursive call stack. In the worst case (unbalanced partitions), it can be O(n).
```

Output:

```
Original list for Quick Sort: [64, 34, 25, 12, 22, 11, 90]
Sorted list using Quick Sort: [11, 12, 22, 25, 34, 64, 90]
```

TASK _5:

code:

```python
def find_duplicates_brute_force(arr):
  """
  Finds duplicate elements in a list using a brute-force approach.

  Args:
    arr: The list to search for duplicates.

  Returns:
    A list of duplicate elements.
  """
  duplicates = []
  n = len(arr)
  # Iterate through each element
  for i in range(n):
    # Compare the current element with all subsequent elements
    for j in range(i + 1, n):
      # If a duplicate is found and it's not already in the duplicates list
      if arr[i] == arr[j] and arr[i] not in duplicates:
        duplicates.append(arr[i])
  return duplicates

# Example usage:
my_list_duplicates = [1, 2, 3, 4, 2, 5, 6, 3, 7, 8, 8]
print("Original list:", my_list_duplicates)
found_duplicates = find_duplicates_brute_force(my_list_duplicates)
print("Duplicates found (brute force):", found_duplicates)

# Performance Notes:
# Time Complexity: O(n^2) - Due to nested loops, each element is compared with every other element.
# Space Complexity: O(k) - where k is the number of unique duplicate elements found.
```

Output:

```
Original list: [1, 2, 3, 4, 2, 5, 6, 3, 7, 8, 8]
Duplicates found (brute force): [2, 3, 8]
```

*****