

AI ASSISTED CODING

LAB-12.4

NAME:SD.Nayaz

ENROLLNO:2403A52076

BATCH:04

TASK-01:

Implementing Bubble Sort with AI Comments.

PROMPT:

Generate a python code to implement bubble sort.

CODE:

```
def bubble_sort(arr):
    """
    Sorts a list in ascending order using the Bubble Sort algorithm.

    :param arr: A list of comparable elements.
    """
    n = len(arr)

    # Outer loop to iterate through the list. This represents the number of passes.
    # After each pass 'i', the i-th largest element will be in its correct final position.
    for i in range(n):
        # A flag to optimize the sort. If no swaps occur in a full pass,
        # the list is already sorted, and we can terminate early.
        swapped = False

        # Inner loop for comparing adjacent elements.
        # The range is 'n-1-i' because the last 'i' elements are already sorted
        # and in their correct final positions.
        for j in range(0, n - i - 1):
            # Compare the current element with the next one.
            if arr[j] > arr[j+1]:
                # If the current element is greater than the next, they are out of order.
                # Perform the swap to move the larger element to the right.
                arr[j], arr[j+1] = arr[j+1], arr[j]

                # Since a swap was made, set the swapped flag to True.
                swapped = True

        # Termination check: If the "swapped" flag is still False after a full
        # inner loop pass, it means no elements were swapped. The list is
        # sorted, so we can break out of the outer loop.
        if not swapped:
            break

    # --- Example Usage ---
if __name__ == "__main__":
    my_list = [64, 34, 25, 12, 22, 11, 90]
    print(f"Original list: {my_list}")
    bubble_sort(my_list)
    print(f"Sorted list: {my_list}")

    # Example with an already sorted list to show optimization
    sorted_list = [1, 2, 3, 4, 5]
    print(f"\nOriginal list (already sorted): {sorted_list}")
    bubble_sort(sorted_list) # This will terminate after the first pass
    print(f"Sorted list: {sorted_list}")
```

OUTPUT:

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS C:\Users\ranch\OneDrive\Desktop\ai> & C:/Users/ranch/AppData/Local/Programs/Python/Python312/python.exe c:/Users/ranch/OneDrive/Desktop/ai/lab12_4_1.py
Original list: [64, 34, 25, 12, 22, 11, 90]
Sorted list: [11, 12, 22, 25, 34, 64, 90]

Original list (already sorted): [1, 2, 3, 4, 5]
Sorted list: [1, 2, 3, 4, 5]
PS C:\Users\ranch\OneDrive\Desktop\ai>
```

OBSERVATION:

Bubble Sort repeatedly compares and swaps adjacent elements if they are in the wrong order, moving the largest element to the end in each pass — like bubbles rising to the top.

TASK-02:

Optimizing Bubble Sort → Insertion Sort

PROMPT:

Convert the following bubble sort code to the insertion sort.

CODE:

```
lab12_4 > 12_4_2.py > bubble_sort
 1 import time
 2
 3 def bubble_sort(arr):
 4     """Sorts a list using the Bubble Sort algorithm."""
 5     n = len(arr)
 6     for i in range(n):
 7         swapped = False
 8         for j in range(0, n - i - 1):
 9             if arr[j] > arr[j+1]:
10                 arr[j], arr[j+1] = arr[j+1], arr[j]
11                 swapped = True
12         if not swapped:
13             break
14
15 def insertion_sort(arr):
16     """Sorts a list using the Insertion Sort algorithm."""
17     for i in range(1, len(arr)):
18         key = arr[i]
19         j = i - 1
20         while j >= 0 and key < arr[j]:
21             arr[j + 1] = arr[j]
22             j -= 1
23         arr[j + 1] = key
24
25 # --- Performance Comparison ---
26 if __name__ == "__main__":
27     # Create a large, nearly sorted list
28     # (e.g., 10,000 elements, with just the last two swapped)
29     nearly_sorted_list = list(range(10000))
30     nearly_sorted_list[-1], nearly_sorted_list[-2] = nearly_sorted_list[-2], nearly_sorted_list[-1]
31
32 # --- Time Bubble Sort ---
33 list_for_bubble = nearly_sorted_list.copy()
34 start_time = time.perf_counter()
35 bubble_sort(list_for_bubble)
36 end_time = time.perf_counter()
37 print(f"\"Bubble Sort took: {(end_time - start_time):.6f} seconds")
38
39 # --- Time Insertion Sort ---
40 list_for_insertion = nearly_sorted_list.copy()
41 start_time = time.perf_counter()
42 insertion_sort(list_for_insertion)
43 end_time = time.perf_counter()
44 print(f"\"Insertion Sort took: {(end_time - start_time):.6f} seconds")
45
46
```

OUTPUT:

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS C:\Users\ranch\OneDrive\Desktop\ai> & C:/Users/ranch/AppData/Local/Programs/Python/Python312/python.exe c:/Users/ranch/OneDrive/Desktop/ai/lab12_4/12_4_1.py
● Original list: [64, 34, 25, 12, 22, 11, 99]
Sorted list: [11, 12, 22, 25, 34, 64, 99]

Original list (already sorted): [3, 2, 3, 4, 5]
Sorted list: [1, 2, 3, 4, 5]
PS C:\Users\ranch\OneDrive\Desktop\ai> & C:/Users/ranch/AppData/Local/Programs/Python/Python312/python.exe c:/Users/ranch/OneDrive/Desktop/ai/lab12_4/12_4_2.py
Bubble Sort took: 0.0000750 seconds
Insertion Sort took: 0.0000763 seconds
PS C:\Users\ranch\OneDrive\Desktop\ai>
```

OBSERVATION:

Optimization from Bubble Sort to Insertion Sort: Instead of repeatedly swapping adjacent elements like in Bubble Sort, **Insertion Sort** shifts elements to insert each item directly into its correct position. This reduces unnecessary swaps and makes it faster, especially for nearly sorted data.

TASK-03:

Binary Search vs Linear Search

PROMPT:

Implement the linear search and binary search with comments.

CODE:

```

lab124 > ⚡ 12.4.3.py > ...
1 import time
2 import random
3
4 def linear_search(arr: list, target: any) -> int:
5     """
6         Performs a linear search to find the target element in a list.
7
8         Linear search sequentially checks each element of the list until a match
9         is found or the entire list has been searched.
10
11    Args:
12        arr (list): The list of elements to search through.
13        target (any): The element to search for.
14
15    Returns:
16        int: The index of the target element if found, otherwise -1.
17
18    Performance Notes:
19        - Time Complexity:
20            - Best Case: O(1) (target is the first element)
21            - Average Case: O(n)
22            - Worst Case: O(n) (target is the last element or not present)
23
24        - Space Complexity: O(1)
25
26        - Suitable for small lists or unsorted lists where the cost of sorting
27            would outweigh the benefits of a faster search algorithm.
28
29    for i in range(len(arr)):
30        if arr[i] == target:
31            return i
32    return -1
33
34 def binary_search(arr: list, target: any) -> int:
35     """
36         Performs a binary search to find the target element in a sorted list.
37
38         Binary search works by repeatedly dividing the search interval in half.
39         It requires the input list to be sorted. If the value of the search key
40         is less than the item in the middle of the interval, narrow the interval
41         to the lower half. Otherwise, narrow it to the upper half.
42
43    Args:
44        arr (list): The sorted list of elements to search through.
45        target (any): The element to search for.
46
47    Returns:
48        int: The index of the target element if found, otherwise -1.
49
50    Performance Notes:

```

Ln 107, Col 55 Spaces:4 UTF-8 CR LF ⓘ Python 89

```

lab124 > ⚡ 12.4.3.py > ...
32 def binary_search(arr: list, target: any) -> int:
49     """
50         - Pre-requisite: The input list 'arr' MUST be sorted.
51         - Time Complexity:
52             - Best Case: O(1) (target is the middle element)
53             - Average Case: O(log n)
54             - Worst Case: O(log n)
55         - Space Complexity: O(1) (iterative version) or O(log n) (recursive version due to call stack)
56         - Highly efficient for large, sorted datasets.
57
58     low = 0
59     high = len(arr) - 1
60
61     while low <= high:
62         mid = (low + high) // 2
63         if arr[mid] == target:
64             return mid
65         elif arr[mid] < target:
66             low = mid + 1
67         else:
68             high = mid - 1
69     return -1
70
71     # --- Performance Comparison ---
72 if __name__ == "__main__":
73     # Generate data
74     LIST_SIZE = 100_000
75     sorted_data = list(range(LIST_SIZE))
76     unsorted_data = random.sample(range(LIST_SIZE), LIST_SIZE) # Unique random numbers
77
78     # Targets for search
79     target_present_start = 0
80     target_present_middle = LIST_SIZE // 2
81     target_present_end = LIST_SIZE - 1
82     target_not_present = LIST_SIZE + 100
83
84     print(f"\n--- Performance Comparison (List Size: {LIST_SIZE}) ---\n")
85
86     # Student Observation Table Header
87     print(f"{'Scenario':<30} | {'Linear Search Time (s)':<25} | {'Binary Search Time (s)':<25}")
88     print("-" * 85)
89
90     # Test 1: Linear Search on unsorted data (target present)
91     start_time = time.perf_counter()
92     linear_search(unsorted_data, target_present_middle)
93     end_time = time.perf_counter()
94     linear_time_unsorted_present = end_time - start_time
95     print(f"{'Unsorted (Target Present)':<30} | {linear_time_unsorted_present:<25.8f} | {'N/A (Requires Sorted)':<25}")

```

Ln 107, Col 55 Spaces:4 UTF-8 CR LF ⓘ

```

12.4 > 12.4.3.py
1 linear_time_unsorted_present = end_time - start_time
2 print(f"{'Unsorted (Target Present)':<30} | {linear_time_unsorted_present:<25.8f} | {'N/A (Requires Sorted)':<25}")
3
4 # Test 2: Linear Search on unsorted data (target not present)
5 start_time = time.perf_counter()
6 linear_search(unsorted_data, target_not_present)
7 end_time = time.perf_counter()
8 linear_time_unsorted_not_present = end_time - start_time
9 print(f"{'Unsorted (Target Not Present)':<30} | {linear_time_unsorted_not_present:<25.8f} | {'N/A (Requires Sorted)':<25}")
10
11 # Test 3: Linear Search on sorted data (target present)
12 start_time = time.perf_counter()
13 linear_search(sorted_data, target_present_middle)
14 end_time = time.perf_counter()
15 linear_time_sorted_present = end_time - start_time
16
17 # Test 4: Binary Search on sorted data (target present)
18 start_time = time.perf_counter()
19 binary_search(sorted_data, target_present_middle)
20 end_time = time.perf_counter()
21 binary_time_sorted_present = end_time - start_time
22 print(f"{'Sorted (Target Present)':<30} | {linear_time_sorted_present:<25.8f} | {binary_time_sorted_present:<25.8f}")
23
24 # Test 5: Linear Search on sorted data (target not present)
25 start_time = time.perf_counter()
26 linear_search(sorted_data, target_not_present)
27 end_time = time.perf_counter()
28 linear_time_sorted_not_present = end_time - start_time
29
30 # Test 6: Binary Search on sorted data (target not present)
31 start_time = time.perf_counter()
32 binary_search(sorted_data, target_not_present)
33 end_time = time.perf_counter()
34 binary_time_sorted_not_present = end_time - start_time
35 print(f"{'Sorted (Target Not Present)':<30} | {linear_time_sorted_not_present:<25.8f} | {binary_time_sorted_not_present:<25.8f}")
36
37 print("\nNote: Binary Search times for unsorted data are marked 'N/A' as it requires a sorted list.")
38 print("If the data is initially unsorted, the time to sort it must be added to Binary Search's total time.")

```

OUTPUT:

Scenario	Linear Search Time (s)	Binary Search Time (s)
Unsorted (Target Present)	0.00364970	N/A (Requires Sorted)
Unsorted (Target Not Present)	0.00297820	N/A (Requires Sorted)
Sorted (Target Present)	0.00111910	0.00000770
Sorted (Target Not Present)	0.00234120	0.00000340

Note: Binary Search times for unsorted data are marked 'N/A' as it requires a sorted list.
If the data is initially unsorted, the time to sort it must be added to Binary Search's total time.

OBSERVATION:

Linear Search: Checks each element one by one until the target is found or the list ends. Works on **unsorted** data but is **slow ($O(n)$)**.

Binary Search: Repeatedly divides a **sorted** list in half to find the target. Much **faster ($O(\log n)$)**, but requires the data to be sorted.

TASK-04: Quick Sort and Merge Sort Comparison

PROMPT:

Implement the quick sort and merge sort using recursion.

CODE:

```
lab124 > 1244.py > merge_sort
1  import time
2  import random
3  import sys
4
5  # Increase recursion limit for large datasets, especially for Quick Sort's worst case.
6  sys.setrecursionlimit(2000)
7
8  def merge_sort(arr: list) -> list:
9      """
10         Sorts a list in ascending order using the Merge Sort algorithm.
11
12         Merge Sort is a divide-and-conquer algorithm. It works by recursively
13         dividing the input list into two halves, calling itself for the two halves,
14         and then merging the two sorted halves.
15
16         Args:
17             arr (list): The list of elements to be sorted.
18
19         Returns:
20             list: A new list containing the sorted elements.
21
22         Performance Notes:
23             - Time Complexity:
24                 - Best Case: O(n log n)
25                 - Average Case: O(n log n)
26                 - Worst Case: O(n log n)
27             Merge Sort's performance is very consistent regardless of the initial
28             order of the input data.
29             - Space Complexity: O(n)
30             Requires additional space to hold the merged sub-arrays.
31
32     # --- AI-COMPLETE LOGIC ---
33     if len(arr) <= 1:
34         return arr
35
36     mid = len(arr) // 2
37     left_half = merge_sort(arr[:mid])
38     right_half = merge_sort(arr[mid:])
39
40     return _merge(left_half, right_half)
41
42 def _merge(left: list, right: list) -> list:
43     """Helper function to merge two sorted lists."""
44     sorted_list = []
45     i = j = 0
46     while i < len(left) and j < len(right):
47         if left[i] < right[j]:
48             sorted_list.append(left[i])
```

© In 17 Col 55 Spaces: 4 LRU

```

lab124 ✘ 1244py7 merge_sort
 42     def _merge(left: list, right: list) -> list:
 43         i = 0
 44         j = 0
 45         sorted_list = []
 46         for i in range(len(left)):
 47             if left[i] < right[j]:
 48                 sorted_list.append(left[i])
 49             else:
 50                 sorted_list.append(right[j])
 51                 j += 1
 52         # Append remaining elements
 53         sorted_list.extend(left[i+1:])
 54         sorted_list.extend(right[j+1:])
 55         return sorted_list
 56
 57     def quick_sort(arr: list):
 58         """
 59             Sorts a list in-place in ascending order using the Quick Sort algorithm.
 60
 61             Quick Sort is a divide-and-conquer algorithm. It works by selecting a
 62             'pivot' element from the array and partitioning the other elements into
 63             two sub-arrays, according to whether they are less than or greater than
 64             the pivot. The sub-arrays are then sorted recursively. This implementation
 65             modifies the list in-place.
 66
 67             Args:
 68                 arr (list): The list of elements to be sorted.
 69
 70             Returns:
 71                 None: The list is sorted in-place.
 72
 73             Performance Notes:
 74                 - Time Complexity:
 75                     - Best Case: O(n log n) (pivot is always the median)
 76                     - Average Case: O(n log n)
 77                     - Worst Case: O(n^2) (pivot is always the smallest or largest element,
 78                         which occurs with already sorted or reverse-sorted data).
 79                     - Space Complexity: O(log n) on average (due to recursion stack),
 80                         O(n) in the worst case.
 81
 82             """
 83             # --- AI-COMPLETE LOGIC ---
 84             _quick_sort_recursive(arr, 0, len(arr) - 1)
 85
 86     def _quick_sort_recursive(arr, low, high):
 87         """
 88             Helper function for recursive calls.
 89
 90             If low < high:
 91                 partition_index = _partition(arr, low, high)
 92                 _quick_sort_recursive(arr, low, partition_index - 1)
 93                 _quick_sort_recursive(arr, partition_index + 1, high)
 94
 95     def _partition(arr, low, high):
 96         """
 97             Partitions the array and returns the pivot's final index.
 98
 99             pivot = arr[high]
 100
 101             i = low
 102             for j in range(low, high):
 103                 if arr[j] <= pivot:
 104                     arr[i], arr[j] = arr[j], arr[i]
 105                     i += 1
 106             arr[i], arr[high] = arr[high], arr[i]
 107             return i
 108
 109             # --- Performance Comparison ---
 110             if __name__ == "__main__":
 111                 LIST_SIZE = 1000
 112
 113                 # Generate data
 114                 random_data = [random.randint(0, LIST_SIZE) for _ in range(LIST_SIZE)]
 115                 sorted_data = list(range(LIST_SIZE))
 116                 reverse_sorted_data = list(range(LIST_SIZE, 0, -1))
 117
 118                 datasets = {
 119                     "Random": random_data,
 120                     "Sorted": sorted_data,
 121                     "Reverse-Sorted": reverse_sorted_data
 122                 }
 123
 124                 print("---- Sorting Algorithm Performance Comparison (List Size: {LIST_SIZE}) ----\n")
 125                 print(f"\t{{'Data Type':<20} | {{'Quick Sort Time (s)':<25}} | {{'Merge Sort Time (s)':<25}}")
 126                 print("-" * 75)
 127
 128                 for name, data in datasets.items():
 129                     # Time Quick Sort
 130                     # We pass a copy because quick_sort sorts in-place
 131                     qs_data = data.copy()
 132                     start_time = time.perf_counter()
 133                     quick_sort(qs_data)
 134                     end_time = time.perf_counter()
 135                     qs_time = end_time - start_time
 136
 137                     # Time Merge Sort
 138                     # We pass a copy to be consistent, although merge_sort returns a new list
 139                     ms_data = data.copy()
 140                     start_time = time.perf_counter()
 141                     merge_sort(ms_data)
 142                     end_time = time.perf_counter()
 143                     ms_time = end_time - start_time
 144
 145                     print(f"\t{name:<20} | {qs_time:<25.8f} | {ms_time:<25.8f}")
 146
 147                 print("\nNote: Quick Sort's O(n^2) worst-case on sorted data is clearly visible.")
 148                 print("Merge Sort's O(n log n) performance is consistent across all data types.")

```

Q | Ln 17, Col 55 | Spaces: 4 | UTF-8 | CR/LF

```

93     def _partition(arr, low, high):
94         """
95             if arr[j] <= pivot:
96                 i += 1
97                 arr[i], arr[j] = arr[j], arr[i]
98             arr[i + 1], arr[high] = arr[high], arr[i + 1]
99             return i + 1
100
101             # --- Performance Comparison ---
102             if __name__ == "__main__":
103                 LIST_SIZE = 1000
104
105                 # Generate data
106                 random_data = [random.randint(0, LIST_SIZE) for _ in range(LIST_SIZE)]
107                 sorted_data = list(range(LIST_SIZE))
108                 reverse_sorted_data = list(range(LIST_SIZE, 0, -1))
109
110                 datasets = {
111                     "Random": random_data,
112                     "Sorted": sorted_data,
113                     "Reverse-Sorted": reverse_sorted_data
114                 }
115
116                 print("---- Sorting Algorithm Performance Comparison (List Size: {LIST_SIZE}) ----\n")
117                 print(f"\t{{'Data Type':<20} | {{'Quick Sort Time (s)':<25}} | {{'Merge Sort Time (s)':<25}}")
118                 print("-" * 75)
119
120                 for name, data in datasets.items():
121                     # Time Quick Sort
122                     # We pass a copy because quick_sort sorts in-place
123                     qs_data = data.copy()
124                     start_time = time.perf_counter()
125                     quick_sort(qs_data)
126                     end_time = time.perf_counter()
127                     qs_time = end_time - start_time
128
129                     # Time Merge Sort
130                     # We pass a copy to be consistent, although merge_sort returns a new list
131                     ms_data = data.copy()
132                     start_time = time.perf_counter()
133                     merge_sort(ms_data)
134                     end_time = time.perf_counter()
135                     ms_time = end_time - start_time
136
137                     print(f"\t{name:<20} | {qs_time:<25.8f} | {ms_time:<25.8f}")
138
139                 print("\nNote: Quick Sort's O(n^2) worst-case on sorted data is clearly visible.")
140                 print("Merge Sort's O(n log n) performance is consistent across all data types.")

```

Q | Ln 17, Col 55

OUTPUT:

The screenshot shows a Jupyter Notebook terminal window. The command run is `PS C:\Users\ranch\OneDrive\Desktop\ai> & C:/Users/ranch/AppData/Local/Programs/Python/Python312/python.exe c:/Users/ranch/OneDrive/Desktop/ai/lab12.4/12.4.4.py`. The output displays a table comparing Quick Sort and Merge Sort times for three data types: Random, Sorted, and Reverse-Sorted, with a note about the worst-case scenario for Quick Sort.

Data Type	Quick Sort Time (s)	Merge Sort Time (s)
Random	0.00176790	0.00280380
Sorted	0.03095380	0.00082400
Reverse-Sorted	0.02159580	0.00080990

Note: Quick Sort's $O(n^2)$ worst-case on sorted data is clearly visible.
Merge Sort's $O(n \log n)$ performance is consistent across all data types.

```
> PS C:\Users\ranch\OneDrive\Desktop\ai>
```

OBSERVATION:

Quick Sort: Uses a **pivot** to partition the array into smaller and larger elements, then sorts each part recursively. It's **faster on average** ($O(n \log n)$) but may degrade to $O(n^2)$ in the worst case.

Merge Sort: Divides the array into halves, sorts them, and then **merges** them. It always runs in **$O(n \log n)$** time but uses **extra memory** for merging.

TASK-05:AI-Suggested Algorithm Optimization

PROMPT:

Generate the python code which implements the duplicate search.

CODE:

```
lab124 > 1245.py > ...
1 import time
2 import random
3
4 def find_duplicates_brute_force(nums: list) -> list:
5     """
6         Finds duplicate numbers in a list using a brute-force, O(n^2) approach.
7
8         This algorithm compares each element with every other element to find duplicates.
9         It then ensures that each duplicate is added only once to the result list.
10
11     Args:
12         nums (list): A list of numbers.
13
14     Returns:
15         list: A list containing the unique duplicate numbers found in the input list.
16
17     Performance Notes:
18         - Time Complexity: O(n^2)
19             - The nested loops lead to quadratic time complexity, as for each
20                 element, it potentially iterates through the rest of the list.
21             - The "if num in duplicates" check within the loop can add another
22                 O(k) operation where k is the number of duplicates found so far,
23                 making it even worse in practice for many duplicates.
24         - Space Complexity: O(k) where k is the number of unique duplicates.
25         - Not suitable for large lists due to its high time complexity.
26     """
27     duplicates = []
28     n = len(nums)
29     for i in range(n):
30         for j in range(i + 1, n):
31             if nums[i] == nums[j]:
32                 if nums[i] not in duplicates: # Avoid adding the same duplicate multiple times
33                     duplicates.append(nums[i])
34     return duplicates
35
36 def find_duplicates_optimized(nums: list) -> list:
37     """
38         Finds duplicate numbers in a list efficiently using sets.
39
40         This algorithm uses two sets: one to keep track of numbers seen so far,
41         and another to store the unique duplicates found. This reduces the
42         lookup time to O(1) on average.
43
44     Args:
45         nums (list): A list of numbers.
46
47     Returns:
48         list: A list containing the unique duplicate numbers found in the input list.

```

Q Ln 1 Col 1 Sp

```

36 def find_duplicates_optimized(nums: list) -> list:
37     """
38         Performance Notes:
39             - Time Complexity: O(n) on average
40                 - Each element is processed once. Set insertion and lookup operations
41                     take O(1) time on average.
42             - Space Complexity: O(n) in the worst case
43                 - Both 'seen' and 'duplicates' sets could potentially store up to
44                     n/2 elements (if all elements are unique or all are duplicates).
45             - Highly efficient for large lists.
46     """
47
48     seen = set()
49     duplicates = set()
50     for num in nums:
51         if num in seen:
52             duplicates.add(num)
53         else:
54             seen.add(num)
55     return list(duplicates)
56
57
58 # --- Performance Comparison ---
59 if __name__ == "__main__":
60     LIST_SIZE = 5000 # Adjust for larger lists to see the difference more clearly
61     MAX_VALUE = LIST_SIZE // 2 # Ensures a good number of duplicates
62
63     # Generate a list with many duplicates
64     test_list = [random.randint(0, MAX_VALUE) for _ in range(LIST_SIZE)]
65
66     print(f"--- Duplicate Finder Performance Comparison (List Size: {LIST_SIZE}) ---\n")
67
68     # Test Brute-Force Version
69     start_time = time.perf_counter()
70     brute_force_duplicates = find_duplicates_brute_force(test_list)
71     end_time = time.perf_counter()
72     brute_force_time = end_time - start_time
73     print(f"Brute-Force Algorithm:")
74     print(f"    Time taken: {brute_force_time:.6f} seconds")
75     print(f"    Found {len(brute_force_duplicates)} unique duplicates.")
76
77     print("-" * 50)
78
79     # Test Optimized Version
80     start_time = time.perf_counter()
81     optimized_duplicates = find_duplicates_optimized(test_list)
82     end_time = time.perf_counter()
83     optimized_time = end_time - start_time
84     print(f"Optimized Algorithm (using sets):")
85     print(f"    Time taken: {optimized_time:.6f} seconds")
86     print(f"    Found {len(optimized_duplicates)} unique duplicates.")
87
88     print("-" * 50)
89
90
91
92
93
94
95
96

```

Q | Line 1, Col 1 | Spaces: 4 | UTF-8

OUTPUT:

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS Python + v
PS C:\Users\ranch\OneDrive\Desktop\ai> & C:/Users/ranch/AppData/Local/Programs/Python/Python312/python.exe c:/Users/ranch/OneDrive/Desktop/ai/lab12.4/12.4.5.py
--- Duplicate Finder Performance Comparison (List Size: 5000) ---

Brute-Force Algorithm:
Time taken: 0.413785 seconds
Found 1461 unique duplicates.

Optimized Algorithm (using sets):
Time taken: 0.0009501 seconds
Found 1461 unique duplicates.

Observation: The optimized version is significantly faster for large lists.
Speedup: 826.58x
PS C:\Users\ranch\OneDrive\Desktop\ai>

```

OBSERVATION:

The task involves first writing a naive duplicate-finding algorithm using nested loops, which has $O(n^2)$ complexity. Then, AI can optimize it by using a set or dictionary to track seen elements, reducing the complexity to $O(n)$. Students compare execution times on large inputs and explain that the optimization improves efficiency by avoiding repeated comparisons.

