

AI ASSISTED CODING

LAB-12.4

NAME: A. MANIDEEPIKA

ENROLLNO:2403A52052

BATCH:03

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-i-1' 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"Original 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\ramch\OneDrive\Desktop\ai> & C:\Users\ramch\AppData\Local\Programs\Python\Python312\python.exe c:\Users\ramch\OneDrive\Desktop\ai\lab12.4\12.4.1.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\ramch\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:

```
lab124 > 1242py > 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, 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> & 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: 0.000750 seconds
Insertion Sort took: 0.000763 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:

```
lab12.4 > 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         - Space Complexity: O(1)
24         - Suitable for small lists or unsorted lists where the cost of sorting
25           would outweigh the benefits of a faster search algorithm.
26     """
27     for i in range(len(arr)):
28         if arr[i] == target:
29             return i
30     return -1
31
32 def binary_search(arr: list, target: any) -> int:
33     """
34     Performs a binary search to find the target element in a sorted list.
35
36     Binary search works by repeatedly dividing the search interval in half.
37     It requires the input list to be sorted. If the value of the search key
38     is less than the item in the middle of the interval, narrow the interval
39     to the lower half. Otherwise, narrow it to the upper half.
40
41     Args:
42         arr (list): The sorted list of elements to search through.
43         target (any): The element to search for.
44
45     Returns:
46         int: The index of the target element if found, otherwise -1.
47
48     Performance Notes:
```

```

lab124 > 1243.py > ...
32 def binary_search(arr: list, target: any) -> int:
49     - Pre-requisite: The input list 'arr' MUST be sorted.
50     - Time Complexity:
51         - Best Case: O(1) (target is the middle element)
52         - Average Case: O(log n)
53         - Worst Case: O(log n)
54     - Space Complexity: O(1) (iterative version) or O(log n) (recursive version due to call stack)
55     - Highly efficient for large, sorted datasets.
56     ---
57     low = 0
58     high = len(arr) - 1
59
60     while low <= high:
61         mid = (low + high) // 2
62         if arr[mid] == target:
63             return mid
64         elif arr[mid] < target:
65             low = mid + 1
66         else:
67             high = mid - 1
68     return -1
69
70 # --- Performance Comparison ---
71 if __name__ == "__main__":
72     # Generate data
73     LIST_SIZE = 100_000
74     sorted_data = list(range(LIST_SIZE))
75     unsorted_data = random.sample(range(LIST_SIZE), LIST_SIZE) # Unique random numbers
76
77     # Targets for search
78     target_present_start = 0
79     target_present_middle = LIST_SIZE // 2
80     target_present_end = LIST_SIZE - 1
81     target_not_present = LIST_SIZE + 100
82
83     print(f"--- Performance Comparison (List Size: {LIST_SIZE}) ---\n")
84
85     # Student Observation Table Header
86     print(f"Scenario:<30 | {'Linear Search Time (s)':<25} | {'Binary Search Time (s)':<25}")
87     print("-" * 85)
88
89     # Test 1: Linear Search on unsorted data (target present)
90     start_time = time.perf_counter()
91     linear_search(unsorted_data, target_present_middle)
92     end_time = time.perf_counter()
93     linear_time_unsorted_present = end_time - start_time
94     print(f"Unsorted (Target Present):<30 | {linear_time_unsorted_present:<25.8f} | {'N/A (Requires Sorted)':<25}")
95
96

```

```

124 > 1243.py > ...
3 linear_time_unsorted_present = end_time - start_time
4 print(f"Unsorted (Target Present):<30 | {linear_time_unsorted_present:<25.8f} | {'N/A (Requires Sorted)':<25}")
5
6 # Test 2: Linear Search on unsorted data (target not present)
7 start_time = time.perf_counter()
8 linear_search(unsorted_data, target_not_present)
9 end_time = time.perf_counter()
10 linear_time_unsorted_not_present = end_time - start_time
11 print(f"Unsorted (Target Not Present):<30 | {linear_time_unsorted_not_present:<25.8f} | {'N/A (Requires Sorted)':<25}")
12
13 # Test 3: Linear Search on sorted data (target present)
14 start_time = time.perf_counter()
15 linear_search(sorted_data, target_present_middle)
16 end_time = time.perf_counter()
17 linear_time_sorted_present = end_time - start_time
18
19 # Test 4: Binary Search on sorted data (target present)
20 start_time = time.perf_counter()
21 binary_search(sorted_data, target_present_middle)
22 end_time = time.perf_counter()
23 binary_time_sorted_present = end_time - start_time
24 print(f"Sorted (Target Present):<30 | {linear_time_sorted_present:<25.8f} | {binary_time_sorted_present:<25.8f}")
25
26 # Test 5: Linear Search on sorted data (target not present)
27 start_time = time.perf_counter()
28 linear_search(sorted_data, target_not_present)
29 end_time = time.perf_counter()
30 linear_time_sorted_not_present = end_time - start_time
31
32 # Test 6: Binary Search on sorted data (target not present)
33 start_time = time.perf_counter()
34 binary_search(sorted_data, target_not_present)
35 end_time = time.perf_counter()
36 binary_time_sorted_not_present = end_time - start_time
37 print(f"Sorted (Target Not Present):<30 | {linear_time_sorted_not_present:<25.8f} | {binary_time_sorted_not_present:<25.8f}")
38
39 print("\nNote: Binary Search times for unsorted data are marked 'N/A' as it requires a sorted list.")
40 print("If the data is initially unsorted, the time to sort it must be added to Binary Search's total time.")

```

OUTPUT:

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS C:\Users\ramch\OneDrive\Desktop\ai> & C:\Users\ramch\AppData\Local\Programs\Python\Python312\python.exe c:/Users/ramch/OneDrive/Desktop/ai/lab12.4/12.4.3.py
--- Performance Comparison (List Size: 100000) ---

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) | 0.00224120 | 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.
PS C:\Users\ramch\OneDrive\Desktop\ai>
```

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:

```
lab12.4 > 12.4.4.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-COMPLETED 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])
49             i += 1
50         else:
51             sorted_list.append(right[j])
52             j += 1
53     sorted_list.extend(left[i:] if i < len(left) else right[j:])
54     return sorted_list
```

```

lab124 > 12.44py merge_sort
42 def _merge(left: list, right: list) -> list:
43     i = 1
44     else:
45         sorted_list.append(right[j])
46         j += 1
47     # Append remaining elements
48     sorted_list.extend(left[i:])
49     sorted_list.extend(right[j:])
50     return sorted_list
51
52 def quick_sort(arr: list):
53     """
54     Sorts a list in-place in ascending order using the Quick Sort algorithm.
55
56     Quick Sort is a divide-and-conquer algorithm. It works by selecting a
57     'pivot' element from the array and partitioning the other elements into
58     two sub-arrays, according to whether they are less than or greater than
59     the pivot. The sub-arrays are then sorted recursively. This implementation
60     modifies the list in-place.
61
62     Args:
63         arr (list): The list of elements to be sorted.
64
65     Returns:
66         None: The list is sorted in-place.
67
68     Performance Notes:
69         - Time Complexity:
70             - Best Case:  $O(n \log n)$  (pivot is always the median)
71             - Average Case:  $O(n \log n)$ 
72             - Worst Case:  $O(n^2)$  (pivot is always the smallest or largest element,
73               which occurs with already sorted or reverse-sorted data).
74         - Space Complexity:  $O(\log n)$  on average (due to recursion stack),
75            $O(n)$  in the worst case.
76     """
77     # --- AI-COMPLETED LOGIC ---
78     _quick_sort_recursive(arr, 0, len(arr) - 1)
79
80 def _quick_sort_recursive(arr, low, high):
81     """Helper function for recursive calls."""
82     if low < high:
83         partition_index = _partition(arr, low, high)
84         _quick_sort_recursive(arr, low, partition_index - 1)
85         _quick_sort_recursive(arr, partition_index + 1, high)
86
87 def _partition(arr, low, high):
88     """Partitions the array and returns the pivot's final index."""
89     pivot = arr[high]
90

```

```

93 def _partition(arr, low, high):
94     """Partitions the array and returns the pivot's final index."""
95     pivot = arr[high]
96     i = low
97     for j in range(low, high):
98         if arr[j] <= pivot:
99             i += 1
100             arr[i], arr[j] = arr[j], arr[i]
101     arr[i], arr[high] = arr[high], arr[i]
102     return i
103
104 # --- Performance Comparison ---
105 if __name__ == "__main__":
106     LIST_SIZE = 1000
107
108     # Generate data
109     random_data = [random.randint(0, LIST_SIZE) for _ in range(LIST_SIZE)]
110     sorted_data = list(range(LIST_SIZE))
111     reverse_sorted_data = list(range(LIST_SIZE, 0, -1))
112
113     datasets = {
114         "Random": random_data,
115         "Sorted": sorted_data,
116         "Reverse-Sorted": reverse_sorted_data
117     }
118
119     print(f"--- Sorting Algorithm Performance Comparison (List Size: {LIST_SIZE}) ---\n")
120     print(f"{'Data Type':<20} | {'Quick Sort Time (s)':<25} | {'Merge Sort Time (s)':<25}")
121     print("-" * 75)
122
123     for name, data in datasets.items():
124         # Time Quick Sort
125         # We pass a copy because quick_sort sorts in-place
126         qs_data = data.copy()
127         start_time = time.perf_counter()
128         quick_sort(qs_data)
129         end_time = time.perf_counter()
130         qs_time = end_time - start_time
131
132         # Time Merge Sort
133         # We pass a copy to be consistent, although merge_sort returns a new list
134         ms_data = data.copy()
135         start_time = time.perf_counter()
136         merge_sort(ms_data)
137         end_time = time.perf_counter()
138         ms_time = end_time - start_time
139
140         print(f"{'name':<20} | {'qs_time':<25.8f} | {'ms_time':<25.8f}")
141
142     print("\nNote: Quick Sort's  $O(n^2)$  worst-case on sorted data is clearly visible.")
143     print("Merge Sort's  $O(n \log n)$  performance is consistent across all data types.")
144

```

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/lab12.4/12.4.4.py
--- Sorting Algorithm Performance Comparison (List Size: 1000) ---

Data Type      | Quick Sort Time (s) | Merge Sort Time (s)
-----
Random         | 0.00176790          | 0.00289380
Sorted         | 0.03095380          | 0.00082480
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:

```

lab12.4 > 12.4.5.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.
49 """

```

```

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

```

OUTPUT:


```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS Python + -
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: 0.000501 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.