AI ASSISTED CODING

LAB-12.4

NAME:V.KARTHIK REDDY

ENROLLNO:2403A52092

BATCH:04

TASK-01:

Implementing Bubble Sort with AI Comments.

PROMPT:

Generate a python code to implement bubble sort.

CODE:

```python
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"\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\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:

```python
lab12.4 > 12.4.2.py > bubble_sort
import time

def bubble_sort(arr):
    """Sorts a list using the Bubble Sort algorithm."""
    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
        if not swapped:
            break

def insertion_sort(arr):
    """Sorts a list using the Insertion Sort algorithm."""
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key

# --- Performance Comparison ---
if __name__ == "__main__":
    # Create a large, nearly sorted list
    # (e.g., 10,000 elements, with just the last two swapped)
    nearly_sorted_list = list(range(10000))
    nearly_sorted_list[-1], nearly_sorted_list[-2] = nearly_sorted_list[-2], nearly_sorted_list[-1]

    # --- Time Bubble Sort ---
    list_for_bubble = nearly_sorted_list.copy()
    start_time = time.perf_counter()
    bubble_sort(list_for_bubble)
    end_time = time.perf_counter()
    print(f"Bubble Sort took: {end_time - start_time:.6f} seconds")

    # --- Time Insertion Sort ---
    list_for_insertion = nearly_sorted_list.copy()
    start_time = time.perf_counter()
    insertion_sort(list_for_insertion)
    end_time = time.perf_counter()
    print(f"Insertion Sort took: {end_time - start_time:.6f} seconds")
```

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> & C:/Users/ramch/AppData/Local/Programs/Python/Python312/python.exe c:/Users/ramch/OneDrive/Desktop/ai/lab12.4/12.4.2.py
Bubble Sort took: 0.000750 seconds
Insertion Sort took: 0.000763 seconds
PS C:\Users\ramch\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:

```python
lab12.4 > 12.4.3.py > ...
import time
import random

def linear_search(arr: list, target: any) -> int:
    """
    Performs a linear search to find the target element in a list.

    Linear search sequentially checks each element of the list until a match
    is found or the entire list has been searched.

    Args:
        arr (list): The list of elements to search through.
        target (any): The element to search for.

    Returns:
        int: The index of the target element if found, otherwise -1.

    Performance Notes:
        - Time Complexity:
            - Best Case: O(1) (target is the first element)
            - Average Case: O(n)
            - Worst Case: O(n) (target is the last element or not present)
        - Space Complexity: O(1)
        - Suitable for small lists or unsorted lists where the cost of sorting
          would outweigh the benefits of a faster search algorithm.
    """
    for i in range(len(arr)):
        if arr[i] == target:
            return i
    return -1

def binary_search(arr: list, target: any) -> int:
    """
    Performs a binary search to find the target element in a sorted list.

    Binary search works by repeatedly dividing the search interval in half.
    It requires the input list to be sorted. If the value of the search key
    is less than the item in the middle of the interval, narrow the interval
    to the lower half. Otherwise, narrow it to the upper half.

    Args:
        arr (list): The sorted list of elements to search through.
        target (any): The element to search for.

    Returns:
        int: The index of the target element if found, otherwise -1.

    Performance Notes:
0 ⚠ 0                                    Ln 107, Col 55   Spaces: 4   UTF-8   CRLF   {} Python
```

```python
def binary_search(arr: list, target: any) -> int:
        - Pre-requisite: The input list `arr` MUST be sorted.
        - Time Complexity:
            - Best Case: O(1) (target is the middle element)
            - Average Case: O(log n)
            - Worst Case: O(log n)
        - Space Complexity: O(1) (iterative version) or O(log n) (recursive version due to call stack)
        - Highly efficient for large, sorted datasets.
    """
    low = 0
    high = len(arr) - 1

    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1
    return -1

# --- Performance Comparison ---
if __name__ == "__main__":
    # Generate data
    LIST_SIZE = 100_000
    sorted_data = list(range(LIST_SIZE))
    unsorted_data = random.sample(range(LIST_SIZE), LIST_SIZE) # Unique random numbers

    # Targets for search
    target_present_start = 0
    target_present_middle = LIST_SIZE // 2
    target_present_end = LIST_SIZE - 1
    target_not_present = LIST_SIZE + 100

    print(f"--- Performance Comparison (List Size: {LIST_SIZE}) ---\n")

    # Student Observation Table Header
    print(f"{'Scenario':<30} | {'Linear Search Time (s)':<25} | {'Binary Search Time (s)':<25}")
    print("-" * 85)

    # Test 1: Linear Search on unsorted data (target present)
    start_time = time.perf_counter()
    linear_search(unsorted_data, target_present_middle)
    end_time = time.perf_counter()
    linear_time_unsorted_present = end_time - start_time
    print(f"{'Unsorted (Target Present)':<30} | {linear_time_unsorted_present:<25.8f} | {'N/A (Requires Sorted)':<25}")
```

```python
    linear_time_unsorted_present = end_time - start_time
    print(f"{'Unsorted (Target Present)':<30} | {linear_time_unsorted_present:<25.8f} | {'N/A (Requires Sorted)':<25}")

    # Test 2: Linear Search on unsorted data (target not present)
    start_time = time.perf_counter()
    linear_search(unsorted_data, target_not_present)
    end_time = time.perf_counter()
    linear_time_unsorted_not_present = end_time - start_time
    print(f"{'Unsorted (Target Not Present)':<30} | {linear_time_unsorted_not_present:<25.8f} | {'N/A (Requires Sorted)':<25}")

    # Test 3: Linear Search on sorted data (target present)
    start_time = time.perf_counter()
    linear_search(sorted_data, target_present_middle)
    end_time = time.perf_counter()
    linear_time_sorted_present = end_time - start_time

    # Test 4: Binary Search on sorted data (target present)
    start_time = time.perf_counter()
    binary_search(sorted_data, target_present_middle)
    end_time = time.perf_counter()
    binary_time_sorted_present = end_time - start_time
    print(f"{'Sorted (Target Present)':<30} | {linear_time_sorted_present:<25.8f} | {binary_time_sorted_present:<25.8f}")

    # Test 5: Linear Search on sorted data (target not present)
    start_time = time.perf_counter()
    linear_search(sorted_data, target_not_present)
    end_time = time.perf_counter()
    linear_time_sorted_not_present = end_time - start_time

    # Test 6: Binary Search on sorted data (target not present)
    start_time = time.perf_counter()
    binary_search(sorted_data, target_not_present)
    end_time = time.perf_counter()
    binary_time_sorted_not_present = end_time - start_time
    print(f"{'Sorted (Target Not Present)':<30} | {linear_time_sorted_not_present:<25.8f} | {binary_time_sorted_not_present:<25.8f}")

    print("\nNote: Binary Search times for unsorted data are marked 'N/A' as it requires a sorted list.")
    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
import time
import random
import sys

# Increase recursion limit for large datasets, especially for Quick Sort's worst case.
sys.setrecursionlimit(2000)

def merge_sort(arr: list) -> list:
    """
    Sorts a list in ascending order using the Merge Sort algorithm.

    Merge Sort is a divide-and-conquer algorithm. It works by recursively
    dividing the input list into two halves, calling itself for the two halves,
    and then merging the two sorted halves.

    Args:
        arr (list): The list of elements to be sorted.

    Returns:
        list: A new list containing the sorted elements.

    Performance Notes:
        - Time Complexity:
            - Best Case: O(n log n)
            - Average Case: O(n log n)
            - Worst Case: O(n log n)
          Merge Sort's performance is very consistent regardless of the initial
          order of the input data.
        - Space Complexity: O(n)
          Requires additional space to hold the merged sub-arrays.
    """
    # --- AI-COMPLETED LOGIC ---
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2
    left_half = merge_sort(arr[:mid])
    right_half = merge_sort(arr[mid:])

    return _merge(left_half, right_half)

def _merge(left: list, right: list) -> list:
    """Helper function to merge two sorted lists."""
    sorted_list = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            sorted_list.append(left[i])
```

lab12.4 > 12.4.4.py > merge_sort

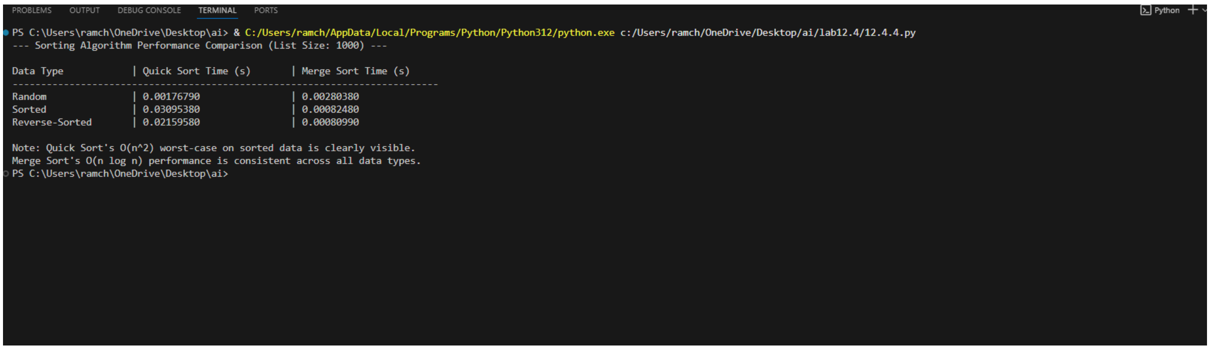
```python
def _merge(left: list, right: list) -> list:
            i += 1
        else:
            sorted_list.append(right[j])
            j += 1
    # Append remaining elements
    sorted_list.extend(left[i:])
    sorted_list.extend(right[j:])
    return sorted_list

def quick_sort(arr: list):
    """
    Sorts a list in-place in ascending order using the Quick Sort algorithm.

    Quick Sort is a divide-and-conquer algorithm. It works by selecting a
    'pivot' element from the array and partitioning the other elements into
    two sub-arrays, according to whether they are less than or greater than
    the pivot. The sub-arrays are then sorted recursively. This implementation
    modifies the list in-place.

    Args:
        arr (list): The list of elements to be sorted.

    Returns:
        None: The list is sorted in-place.

    Performance Notes:
        - Time Complexity:
            - Best Case: O(n log n) (pivot is always the median)
            - Average Case: O(n log n)
            - Worst Case: O(n^2) (pivot is always the smallest or largest element,
              which occurs with already sorted or reverse-sorted data).
        - Space Complexity: O(log n) on average (due to recursion stack),
          O(n) in the worst case.
    """
    # --- AI-COMPLETED LOGIC ---
    _quick_sort_recursive(arr, 0, len(arr) - 1)

def _quick_sort_recursive(arr, low, high):
    """Helper function for recursive calls."""
    if low < high:
        partition_index = _partition(arr, low, high)
        _quick_sort_recursive(arr, low, partition_index - 1)
        _quick_sort_recursive(arr, partition_index + 1, high)

def _partition(arr, low, high):
    """Partitions the array and returns the pivot's final index."""
    pivot = arr[high]
```

0 ⚠ 0 Ln 17, Col 55 Spaces: 4 UTF-8 CRLF

```python
def _partition(arr, low, high):
        if arr[j] <= pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1

# --- Performance Comparison ---
if __name__ == "__main__":
    LIST_SIZE = 1000

    # Generate data
    random_data = [random.randint(0, LIST_SIZE) for _ in range(LIST_SIZE)]
    sorted_data = list(range(LIST_SIZE))
    reverse_sorted_data = list(range(LIST_SIZE, 0, -1))

    datasets = {
        "Random": random_data,
        "Sorted": sorted_data,
        "Reverse-Sorted": reverse_sorted_data
    }

    print(f"--- Sorting Algorithm Performance Comparison (List Size: {LIST_SIZE}) ---\n")
    print(f"{'Data Type':<20} | {'Quick Sort Time (s)':<25} | {'Merge Sort Time (s)':<25}")
    print("-" * 75)

    for name, data in datasets.items():
        # Time Quick Sort
        # We pass a copy because quick_sort sorts in-place
        qs_data = data.copy()
        start_time = time.perf_counter()
        quick_sort(qs_data)
        end_time = time.perf_counter()
        qs_time = end_time - start_time

        # Time Merge Sort
        # We pass a copy to be consistent, although merge_sort returns a new list
        ms_data = data.copy()
        start_time = time.perf_counter()
        merge_sort(ms_data)
        end_time = time.perf_counter()
        ms_time = end_time - start_time

        print(f"{name:<20} | {qs_time:<25.8f} | {ms_time:<25.8f}")

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

0 ⚠ 0 Ln 17, Col 55

OUTPUT:



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(n²) 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:

```python
import time
import random

def find_duplicates_brute_force(nums: list) -> list:
    """
    Finds duplicate numbers in a list using a brute-force, O(n^2) approach.

    This algorithm compares each element with every other element to find duplicates.
    It then ensures that each duplicate is added only once to the result list.

    Args:
        nums (list): A list of numbers.

    Returns:
        list: A list containing the unique duplicate numbers found in the input list.

    Performance Notes:
        - Time Complexity: O(n^2)
            - The nested loops lead to quadratic time complexity, as for each
              element, it potentially iterates through the rest of the list.
            - The `if num in duplicates` check within the loop can add another
              O(k) operation where k is the number of duplicates found so far,
              making it even worse in practice for many duplicates.
        - Space Complexity: O(k) where k is the number of unique duplicates.
        - Not suitable for large lists due to its high time complexity.
    """
    duplicates = []
    n = len(nums)
    for i in range(n):
        for j in range(i + 1, n):
            if nums[i] == nums[j]:
                if nums[i] not in duplicates: # Avoid adding the same duplicate multiple times
                    duplicates.append(nums[i])
    return duplicates

def find_duplicates_optimized(nums: list) -> list:
    """
    Finds duplicate numbers in a list efficiently using sets.

    This algorithm uses two sets: one to keep track of numbers seen so far,
    and another to store the unique duplicates found. This reduces the
    lookup time to O(1) on average.

    Args:
        nums (list): A list of numbers.

    Returns:
        list: A list containing the unique duplicate numbers found in the input list.
```

```python
def find_duplicates_optimized(nums: list) -> list:
    Performance Notes:
        - Time Complexity: O(n) on average
            - Each element is processed once. Set insertion and lookup operations
              take O(1) time on average.
        - Space Complexity: O(n) in the worst case
            - Both `seen` and `duplicates` sets could potentially store up to
              n/2 elements (if all elements are unique or all are duplicates).
        - Highly efficient for large lists.
    """
    seen = set()
    duplicates = set()
    for num in nums:
        if num in seen:
            duplicates.add(num)
        else:
            seen.add(num)
    return list(duplicates)

# --- Performance Comparison ---
if __name__ == "__main__":
    LIST_SIZE = 5000  # Adjust for larger lists to see the difference more clearly
    MAX_VALUE = LIST_SIZE // 2 # Ensures a good number of duplicates

    # Generate a list with many duplicates
    test_list = [random.randint(0, MAX_VALUE) for _ in range(LIST_SIZE)]

    print(f"--- Duplicate Finder Performance Comparison (List Size: {LIST_SIZE}) ---\n")

    # Test Brute-Force Version
    start_time = time.perf_counter()
    brute_force_duplicates = find_duplicates_brute_force(test_list)
    end_time = time.perf_counter()
    brute_force_time = end_time - start_time
    print(f"Brute-Force Algorithm:")
    print(f"  Time taken: {brute_force_time:.6f} seconds")
    print(f"  Found {len(brute_force_duplicates)} unique duplicates.")

    print("-" * 50)

    # Test Optimized Version
    start_time = time.perf_counter()
    optimized_duplicates = find_duplicates_optimized(test_list)
    end_time = time.perf_counter()
    optimized_time = end_time - start_time
    print(f"Optimized Algorithm (using sets):")
    print(f"  Time taken: {optimized_time:.6f} seconds")
    print(f"  Found {len(optimized_duplicates)} unique duplicates.")
```

OUTPUT:

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.