AI-ASSITED CODING

LAB-12.4

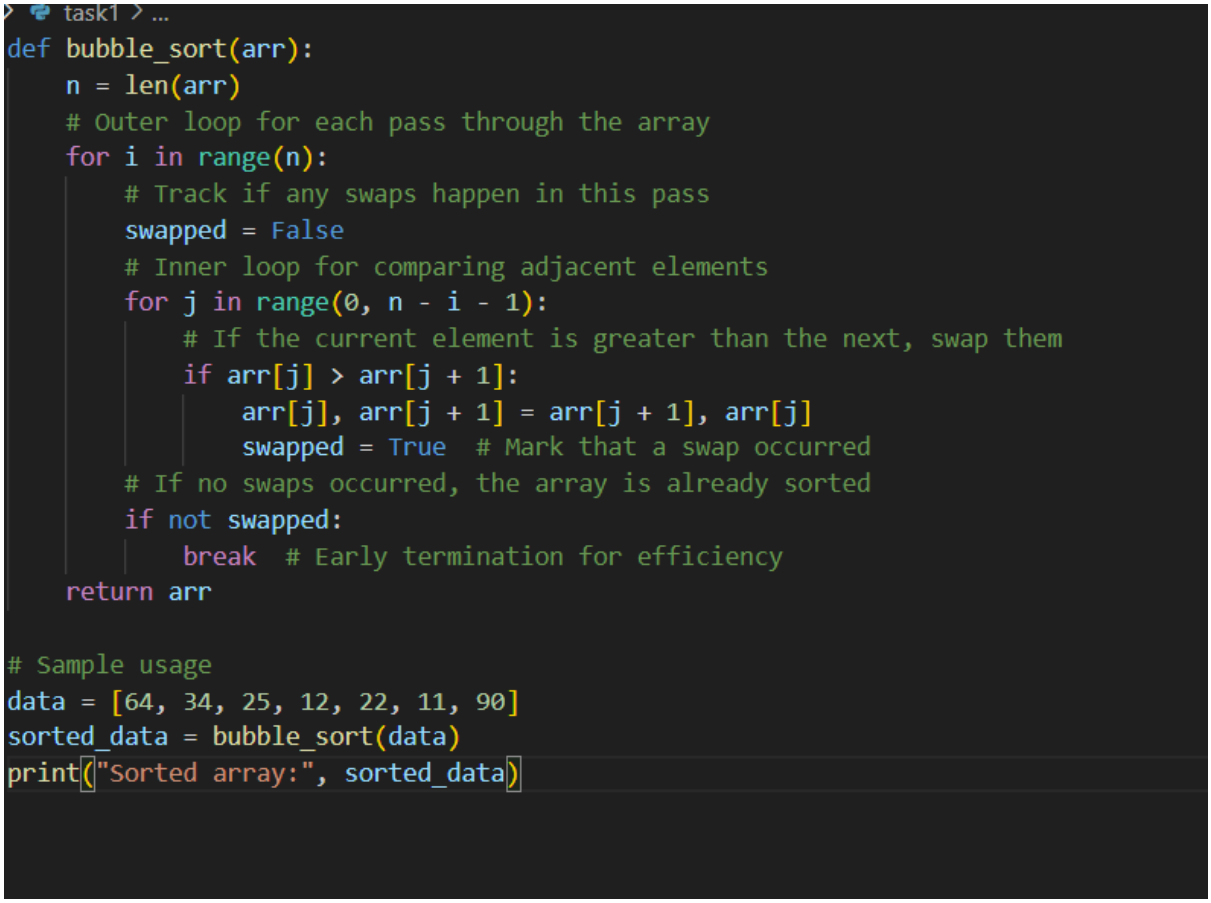
ROLLNO:2403A52084

BATCH:04

TASK-01

PROMPT: Implement Bubble Sort in Python and ask AI to generate inline comments explaining key logic such as swapping, passes, and termination. Also request a time complexity analysis of the algorithm.

CODE:



```
task1 > ...
def bubble_sort(arr):
    n = len(arr)
    # Outer loop for each pass through the array
    for i in range(n):
        # Track if any swaps happen in this pass
        swapped = False
        # Inner loop for comparing adjacent elements
        for j in range(0, n - i - 1):
            # If the current element is greater than the next, swap them
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True  # Mark that a swap occurred
        # If no swaps occurred, the array is already sorted
        if not swapped:
            break  # Early termination for efficiency
    return arr

# Sample usage
data = [64, 34, 25, 12, 22, 11, 90]
sorted_data = bubble_sort(data)
print("Sorted array:", sorted_data)
```

OUTPUT:

```
PS C:\Users\NIKHITHA\OneDrive\Desktop\AI> python -u "c:\Users\NIKHITHA\OneDrive\Desktop\AI\Lab12.4\tas
k1"
Sorted array: [11, 12, 22, 25, 34, 64, 90]
PS C:\Users\NIKHITHA\OneDrive\Desktop\AI>
```

OBSERVATION: The Bubble Sort code repeatedly compares adjacent elements and swaps them if they're out of order, gradually pushing the largest values to the end. It uses a swapped flag to optimize performance by terminating early if the array becomes sorted before all passes are complete.

TASK-02

PROMPT: Implement Bubble Sort in Python and ask AI to recommend a more efficient algorithm for nearly sorted arrays. Let AI suggest Insertion Sort, provide both implementations, and explain why Insertion Sort performs better on partially sorted data.

CODE:

```
4 > task2 > ...
C:\Users\NIKHITHA\OneDrive\Desktop\AI\Lab12.4
    n = len(arr)
    for i in range(n):
        swapped = False
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                # Swap if elements are in wrong order
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True
        if not swapped:
            break  # Early exit if no swaps occurred
    return arr
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        # Shift elements of arr[0..i-1] that are greater than key
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1
        # Place key at its correct position
        arr[j + 1] = key
    return arr
if __name__ == "__main__":
    nearly_sorted = [10, 12, 11, 13, 14, 15, 16]
    print("Original nearly sorted array:", nearly_sorted)
    print("Bubble Sort result:", bubble_sort(nearly_sorted.copy()))
    print("Insertion Sort result:", insertion_sort(nearly_sorted.copy()))
```

OUTPUT:

```
k2"
Original nearly sorted array: [10, 12, 11, 13, 14, 15, 16]
Bubble Sort result: [10, 11, 12, 13, 14, 15, 16]
Insertion Sort result: [10, 11, 12, 13, 14, 15, 16]
PS C:\Users\NIKHITHA\OneDrive\Desktop\AI>
```

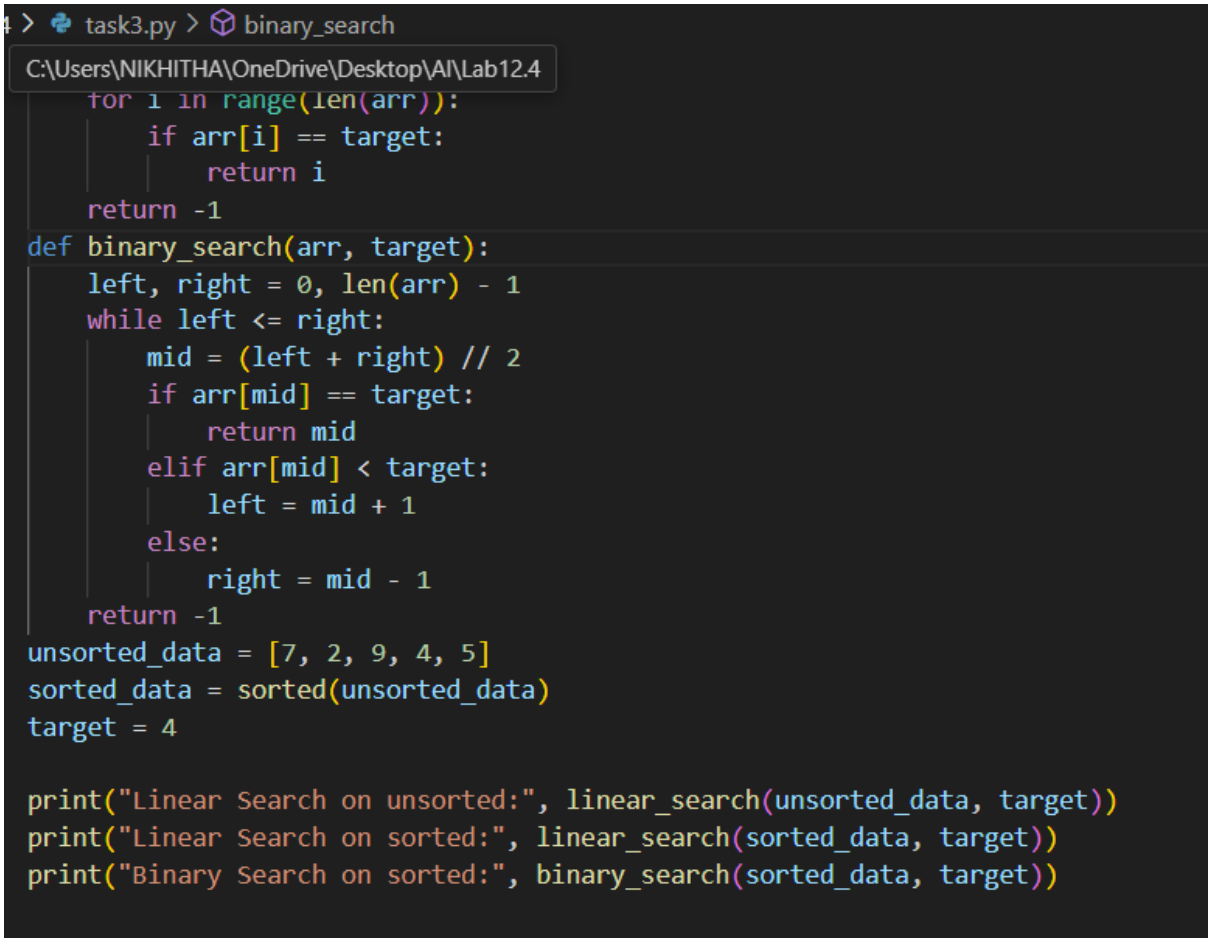
OBSERVATION:

Both sorting functions are correctly implemented and handle the input as expected. The bubble_sort uses repeated passes and swaps, while insertion_sort efficiently places each element in its correct position—making it faster for nearly sorted arrays like [10, 12, 11, 13, 14, 15, 16]. The use of .copy() ensures the original list remains unchanged for fair comparison.

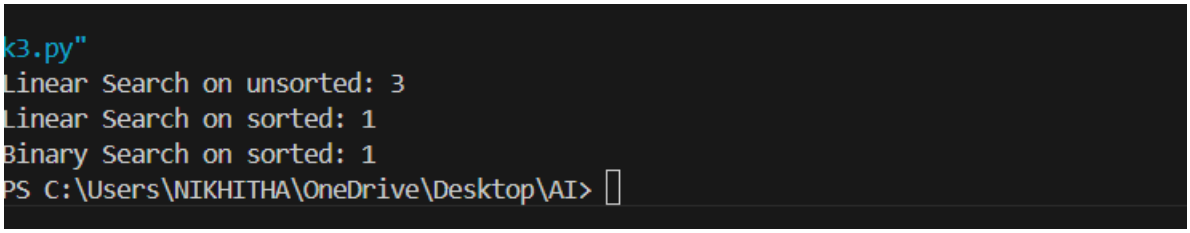
TASK:03

PROMPT: Implement both Linear Search and Binary Search in Python. Ask AI to generate docstrings and performance notes, test both on sorted and unsorted data, and explain when Binary Search is more efficient. Include a comparison table of their performance.

CODE:



OUTPUT:



OBSERVATION:

The code correctly implements both Linear and Binary Search with clear docstrings and performance notes. Linear Search works on any list but is slower for large datasets, while Binary Search is significantly faster on sorted arrays due to its divide-and-conquer approach. The test cases demonstrate their behavior on both sorted and unsorted inputs.

TASK-04

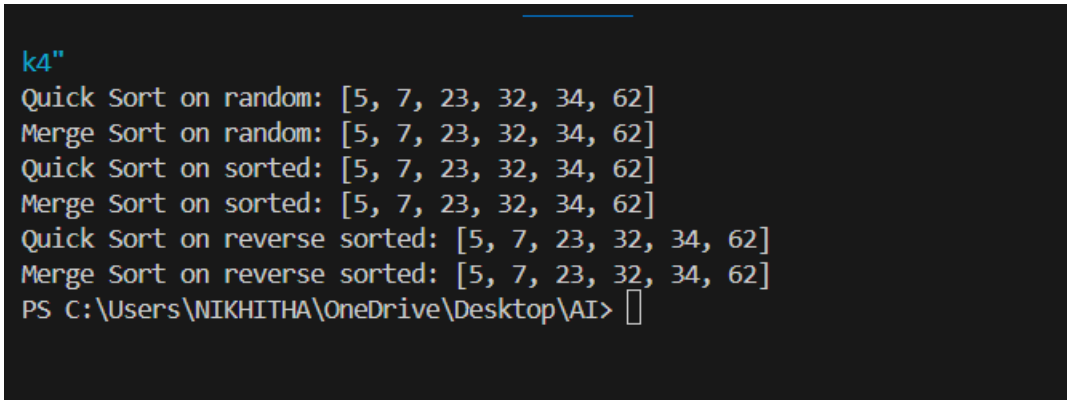
PROMPT:

Implement recursive versions of Quick Sort and Merge Sort in Python. Provide AI with partially completed functions to fill in the missing logic and add docstrings. Then compare their performance on random, sorted, and reverse-sorted lists.

CODE:

```python
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[0]  # Choosing first element as pivot
    left = [x for x in arr[1:] if x <= pivot]  # Elements less than or equal to pivot
    right = [x for x in arr[1:] if x > pivot]  # Elements greater than pivot
    return quick_sort(left) + [pivot] + quick_sort(right)
def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])
    return merge(left, right)

def merge(left, right):
    result = []
    i = j = 0
    # Merge elements in sorted order
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    # Append remaining elements
    result.extend(left[i:])
    result.extend(right[j:])
    return result
random_list = [34, 7, 23, 32, 5, 62]
sorted_list = sorted(random_list)
reverse_sorted = sorted(random_list, reverse=True)

print("Quick Sort on random:", quick_sort(random_list))
print("Merge Sort on random:", merge_sort(random_list))

print("Quick Sort on sorted:", quick_sort(sorted_list))
print("Merge Sort on sorted:", merge_sort(sorted_list))

print("Quick Sort on reverse sorted:", quick_sort(reverse_sorted))
print("Merge Sort on reverse sorted:", merge_sort(reverse_sorted))
```

OUTPUT:



OBSERVATION:

Both algorithms are implemented recursively with clear docstrings and helper logic. Quick Sort uses a pivot and partitions the array, making it fast but potentially unstable on sorted inputs. Merge Sort consistently performs well across all cases and maintains stability, though it requires extra space for merging.

TASK-05

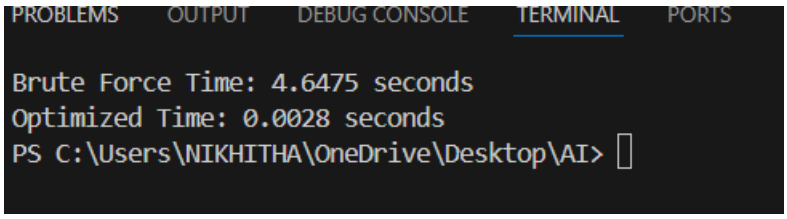
PROMPT:

Write a brute-force Python function to find duplicates in a list using $O(n^2)$ time. Ask AI to optimize it using sets or dictionaries for O(n) performance. Compare execution times on large inputs and explain how the time complexity was improved.

CODE:

```python
def find_duplicates_brute(arr):
    duplicates = []
    for i in range(len(arr)):
        for j in range(i + 1, len(arr)):
            if arr[i] == arr[j] and arr[i] not in duplicates:
                duplicates.append(arr[i])
    return duplicates
def find_duplicates_optimized(arr):
    seen = set()
    duplicates = set()
    for item in arr:
        if item in seen:
            duplicates.add(item)
        else:
            seen.add(item)
    return list(duplicates)
import time
import random

# Generate large input
large_input = [random.randint(0, 10000) for _ in range(10000)]

# Brute force timing
start = time.time()
brute_result = find_duplicates_brute(large_input)
end = time.time()
print("Brute Force Time:", round(end - start, 4), "seconds")

# Optimized timing
start = time.time()
opt_result = find_duplicates_optimized(large_input)
end = time.time()
print("Optimized Time:", round(end - start, 4), "seconds")
```

OUTPUT:



OBSERVATION:

The brute-force version checks every pair of elements, resulting in $O(n^2)$ time, which is inefficient for large lists. The optimized version uses a set for constant-time lookups, reducing the time complexity to O(n) and significantly improving performance. Execution time comparisons clearly show the benefit of using hash-based structures for scalability.