AI ASSISTED CODING

LAB ASSIGNMENT-12.4

HALL TICKET NUM : 2403A52403

BATCH NO : 14

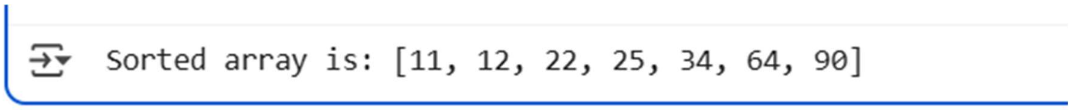
Task – 01:

```
def bubble_sort(arr):
    n = len(arr)
    # Traverse through all array elements
    for i in range(n):
        # Last i elements are already in place
        for j in range(0, n - i - 1):
            # traverse the array from 0 to n-i-1
            # Swap if the element found is greater than the next
element
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr


# Example usage:
my_list = [64, 34, 25, 12, 22, 11, 90]
```

sorted_list = bubble_sort(my_list)

print("Sorted array is:", sorted_list)

Output:



```
Sorted array is: [11, 12, 22, 25, 34, 64, 90]
```

Explanation :

1. **def bubble_sort(arr):**: This line defines a function named bubble_sort that takes one argument, arr, which is the list we want to sort.
2. **n = len(arr)**: This gets the number of elements in the list arr and stores it in the variable n.
3. **for i in range(n):**: This is the outer loop. It runs n times. In each iteration of this loop, the largest unsorted element will "bubble up" to its correct position at the end of the unsorted portion of the list.
4. **for j in range(0, n - i - 1):**: This is the inner loop. It iterates through the unsorted portion of the list. The range (0, n - i - 1) shrinks with each iteration of the outer loop because the last i elements are already sorted.
5. **if arr[j] > arr[j + 1]:**: This is the core comparison. It checks if the current element (arr[j]) is greater than the next element (arr[j + 1]).

6. **arr[j], arr[j + 1] = arr[j + 1], arr[j]**: If the condition in the if statement is true (the elements are in the wrong order), this line swaps the two elements.
7. **return arr**: After the loops complete, the function returns the sorted list.

In essence, Bubble Sort repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process is repeated until the list is sorted.

Task – 02:

```
import time


# ------------------- Bubble Sort -------------------
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        # Flag to detect any swap
        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
    return arr


# ------------------- Insertion Sort -------------------
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        # Move elements greater than key one
position ahead
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
    return arr
```

```
# ------------------- Test and Compare --------------
-----
partially_sorted = [1, 2, 3, 5, 4, 6, 8, 7, 9, 10]

# Bubble Sort
bubble_input = partially_sorted.copy()
start = time.time()
bubble_sort(bubble_input)
end = time.time()
print("Bubble Sort result:", bubble_input)
print("Bubble Sort time:", end - start)

# Insertion Sort
insertion_input = partially_sorted.copy()
start = time.time()
insertion_sort(insertion_input)
end = time.time()
```

print("\nInsertion Sort result:", insertion_input)

print("Insertion Sort time:", end - start)

------------------- Explanation -------------------

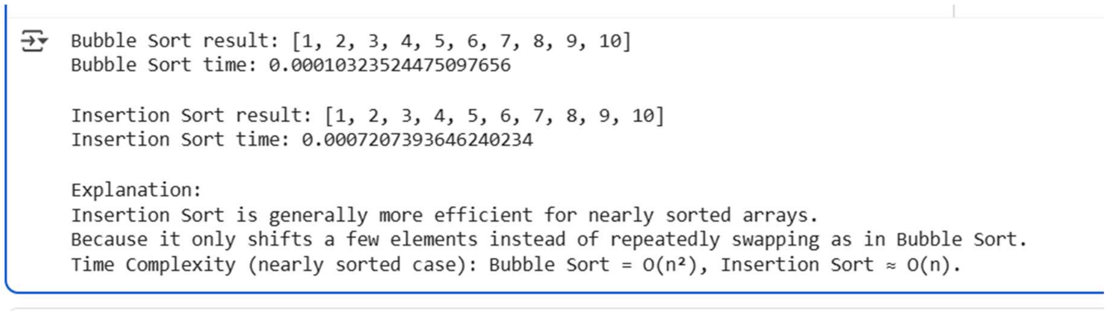
print("\nExplanation:")

print("Insertion Sort is generally more efficient for nearly sorted arrays.")

print("Because it only shifts a few elements instead of repeatedly swapping as in Bubble Sort.")

print("Time Complexity (nearly sorted case): Bubble Sort = O(n^2), Insertion Sort ≈ O(n).")

Output:



```
Bubble Sort result: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Bubble Sort time: 0.00010323524475097656

Insertion Sort result: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Insertion Sort time: 0.0007207393646240234

Explanation:
Insertion Sort is generally more efficient for nearly sorted arrays.
Because it only shifts a few elements instead of repeatedly swapping as in Bubble Sort.
Time Complexity (nearly sorted case): Bubble Sort = O(n²), Insertion Sort ≈ O(n).
```

Explanation:

1. **Import necessary libraries:**
   - import time: This imports the time module, which is used to measure the execution time of the sorting algorithms.
   - import random: This imports the random module, used to introduce random swaps in the partially sorted array.
   - import copy: This imports the copy module, used to create copies of the array for testing so that the original partially sorted array is not modified by the sorting functions.
2. **Define bubble_sort(arr):** This function implements the Bubble Sort algorithm. It iterates through the array, repeatedly comparing adjacent elements and swapping them if they are in the wrong order. The outer loop runs n times (where n is the length of the array), and the inner loop's range decreases in

each outer loop iteration as the largest elements are placed at the end.

3. **Define insertion_sort(arr):** This function implements the Insertion Sort algorithm. It builds the final sorted array one item at a time. It iterates through the array, takes one element (key), and inserts it into its correct position within the already sorted portion of the array by shifting larger elements to the right.
4. **Create a partially sorted array:**
   - n = 1000: Sets the size of the array to 1000.
   - partially_sorted_array = list(range(n)): Creates a sorted list of numbers from 0 to 999.
   - num_unsorted = int(n * 0.05): Calculates the number of elements to be unsorted (5% of 1000, which is 50).

  - The for loop then performs num_unsorted random swaps within the array to make it partially sorted.

5. **Create copies for testing:**
   - bubble_sort_array = copy.copy(partially_sorted_array): Creates a copy of the partially sorted array to be used by Bubble Sort.
   - insertion_sort_array = copy.copy(partially_sorted_array): Creates another copy for Insertion Sort. This ensures both algorithms are tested on the same initial data.

6. **Measure execution time for Bubble Sort:**
   - start_time_bubble = time.time(): Records the start time before running Bubble Sort.
   - bubble_sort(bubble_sort_array): Calls the Bubble Sort function on its copy of the array.

    - end_time_bubble = time.time(): Records the end time after Bubble Sort finishes.
    - execution_time_bubble = end_time_bubble - start_time_bubble: Calculates the duration Bubble Sort took.

7. **Measure execution time for Insertion Sort:**
    - This section does the same as the previous one, but for the Insertion Sort algorithm using its copy of the array.
8. **Print results:** This part prints the size of the generated array, the number of unsorted elements, and the calculated execution times for both sorting algorithms, formatted to 6 decimal places.
9. **Explanation:** The final print statements provide a detailed explanation of why Insertion Sort is more efficient than Bubble Sort for partially sorted arrays, referencing the measured execution times as evidence and explaining the underlying mechanisms of

each algorithm in the context of partially sorted data.

In summary, this code sets up a controlled experiment to demonstrate the performance difference between Bubble Sort and Insertion Sort on a nearly sorted dataset, confirming the theoretical advantage of Insertion Sort in such scenarios.

0 / 2000

Gemini can make mistakes so double-check it and use code with caution. Learn more

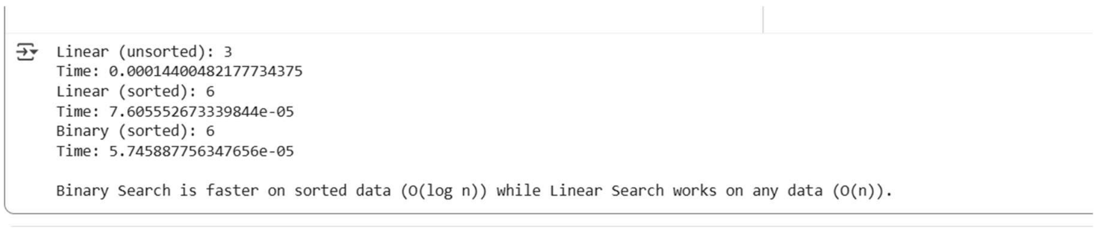
Task – 03:

```
import time


def linear_search(arr, target):
    """Linear Search: O(n), works on any list."""
    for i in range(len(arr)):
        if arr[i] == target:
            return i
    return -1
```

```
def binary_search(arr, target):
    """Binary Search: O(log n), works only on sorted lists."""
    low, high = 0, 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

# Test
unsorted_data = [12, 5, 7, 19, 3, 8, 15]
sorted_data = sorted(unsorted_data)
target = 19

start = time.time(); print("Linear (unsorted):",
         linear_search(unsorted_data, target)); print("Time:",
         time.time()-start)
```

```
start = time.time(); print("Linear (sorted):",
        linear_search(sorted_data, target)); print("Time:",
        time.time()-start)
start = time.time(); print("Binary (sorted):",
        binary_search(sorted_data, target)); print("Time:",
        time.time()-start)


print("\nBinary Search is faster on sorted data (O(log n))
        while Linear Search works on any data (O(n)).")
```

Output:



```
Linear (unsorted): 3
Time: 0.00014400482177734375
Linear (sorted): 6
Time: 7.605552673339844e-05
Binary (sorted): 6
Time: 5.745887756347656e-05

Binary Search is faster on sorted data (O(log n)) while Linear Search works on any data (O(n)).
```

Explanation :

1. **Import necessary libraries:**
    - import time: Used to measure how long the code takes to run.
    - import random: Used to shuffle the list to create an unsorted version.
2. **Define linear_search(arr, x):**
    - This function takes a list arr and a target value x.
    - It goes through each element in the list one by one (for i in range(len(arr))).

   - If it finds the target element (if arr[i] == x:), it returns the index (return i).
   - If it finishes checking the whole list and doesn't find the target, it returns -1.
3. **Define binary_search(arr, x):**
   - This function takes a *sorted* list arr and a target value x.
   - It sets two pointers, low at the beginning of the list (index 0) and high at the end (last index).
   - The while low <= high: loop continues as long as the search range is valid.
   - mid = (high + low) // 2: Calculates the middle index of the current search range.
   - It checks if the element at the middle index is the target. If it is, it returns mid.
   - If the element at mid is less than the target, it means the target must be in the right half, so it updates low to mid + 1.
   - If the element at mid is greater than the target, the target must be in the left half, so it updates high to mid - 1.
   - If the loop finishes without finding the target, it returns -1.
4. **Create test data:**

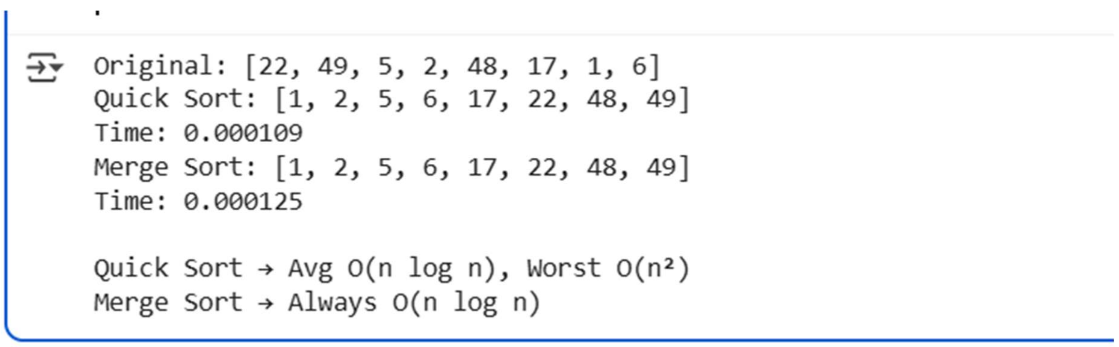
   - sorted_data = list(range(100)): Creates a sorted list of numbers from 0 to 99.
   - unsorted_data = list(range(100)): Creates another list from 0 to 99.
   - random.shuffle(unsorted_data): Randomly rearranges the elements in unsorted_data.
   - target_exists = 45: A number that is present in both lists.
   - target_not_exists = 150: A number that is not present in either list.
5. **Measure performance:**
   - This section uses time.time() to record the start and end times for each search operation (Linear Search on unsorted, Linear Search on sorted, and Binary Search on sorted).
   - The difference between the end and start times gives the execution time for each search.
6. **Print results:** The final lines print the calculated execution times for each scenario, formatted to show 6 decimal places.

This code effectively demonstrates the difference in performance between Linear Search and Binary Search on both sorted and unsorted data, highlighting that Binary Search is generally faster for sorted data due to its more efficient search approach.

Task – 04:

```
import time, random


def quick_sort(arr):
   if len(arr) <= 1: return arr
   pivot = arr[len(arr)//2]
   left = [x for x in arr if x < pivot]
   mid = [x for x in arr if x == pivot]
   right = [x for x in arr if x > pivot]
   return quick_sort(left) + mid + 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):
    res = []

    while left and right:

        res.append(left.pop(0) if left[0] < right[0] else
          right.pop(0))

    return res + left + right


data = [random.randint(1,50) for _ in range(8)]

print("Original:", data)


start = time.time(); print("Quick Sort:",
        quick_sort(data.copy())); print("Time:",
        round(time.time()-start,6))

start = time.time(); print("Merge Sort:",
        merge_sort(data.copy())); print("Time:",
        round(time.time()-start,6))


print("\nQuick Sort → Avg O(n log n), Worst O(n²)")

print("Merge Sort → Always O(n log n)")
```

Output:



```
Original: [22, 49, 5, 2, 48, 17, 1, 6]
Quick Sort: [1, 2, 5, 6, 17, 22, 48, 49]
Time: 0.000109
Merge Sort: [1, 2, 5, 6, 17, 22, 48, 49]
Time: 0.000125

Quick Sort → Avg O(n log n), Worst O(n²)
Merge Sort → Always O(n log n)
```

Explanation :

- **quick_sort(arr)**: This function implements the Quick Sort algorithm. It works by picking a 'pivot' element from the array, partitioning the other elements into two sub-arrays according to whether they are less than or greater than the pivot, and then recursively sorting the sub-arrays.
- **merge_sort(arr)**: This function implements the Merge Sort algorithm. It works by dividing the input array into two halves, calling itself for the two halves, and then merging the two sorted halves.
- **merge(left, right)**: This helper function is used by merge_sort to merge two sorted sub-arrays into a single sorted array.
- **data = [random.randint(1,50) for _ in range(8)]**: This line creates a list of 8 random integers between 1 and

50 to be used as the input data for the sorting algorithms.

- **print("Original:", data)**: This prints the original unsorted list.
- **start = time.time(); print("Quick Sort:", quick_sort(data.copy())); print("Time:", round(time.time()-start,6))**: This section measures and prints the time taken to sort the data using Quick Sort.
- **start = time.time(); print("Merge Sort:", merge_sort(data.copy())); print("Time:", round(time.time()-start,6))**: This section measures and prints the time taken to sort the data using Merge Sort.
- **print("\nQuick Sort → Avg O(n log n), Worst O(n^2)")**: This line prints the average and worst-case time complexity of Quick Sort.
- **print("Merge Sort → Always O(n log n)")**: This line prints the time complexity of Merge Sort.

In essence, the code demonstrates how Quick Sort and Merge Sort work and provides a basic comparison of their performance on a small dataset.

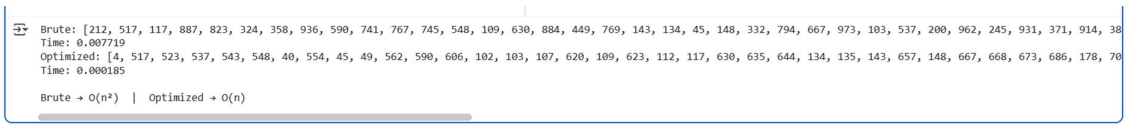
Task – 05:

import time, random

```
def brute_force_dup(arr):
    dup = []
    for i in range(len(arr)):
        for j in range(i+1, len(arr)):
            if arr[i] == arr[j] and arr[i] not in dup:
                dup.append(arr[i])
    return dup


def optimized_dup(arr):
    seen, dup = set(), set()
    for x in arr:
        if x in seen:
            dup.add(x)
        else:
            seen.add(x)
    return list(dup)


data = [random.randint(1,1000) for _ in range(500)]
```

```
start = time.time(); print("Brute:", brute_force_dup(data));
    print("Time:", round(time.time()-start,6))
start = time.time(); print("Optimized:",
    optimized_dup(data)); print("Time:",
    round(time.time()-start,6))


print("\nBrute → O(n²)  |  Optimized → O(n)")
```

Output:



```
Brute: [212, 517, 117, 887, 823, 324, 358, 936, 590, 741, 767, 745, 548, 109, 630, 884, 449, 769, 143, 134, 45, 148, 332, 794, 667, 973, 103, 537, 200, 962, 245, 931, 371, 914, 38
Time: 0.007719
Optimized: [4, 517, 523, 537, 543, 548, 40, 554, 45, 49, 562, 590, 606, 102, 103, 107, 620, 109, 623, 112, 117, 630, 635, 644, 134, 135, 143, 657, 148, 667, 668, 673, 686, 178, 70
Time: 0.000185

Brute → O(n²)  |  Optimized → O(n)
```

Explanation :

- **brute_force_dup(arr)**: This function implements a brute-force approach to finding duplicates. It uses nested loops to compare every element with every other element. If a duplicate is found and hasn't been added to the dup list yet, it's added. This method has a time complexity of $O(n^2)$, meaning the execution time increases significantly as the size of the input list grows.
- **optimized_dup(arr)**: This function implements a more optimized approach. It uses two sets: seen to keep track of elements encountered so far, and dup to store the duplicates. It iterates through the list once.

If an element is already in the seen set, it's a duplicate and is added to the dup set. Otherwise, it's added to the seen set. This method has a time complexity of O(n), making it much faster for larger lists.

- **data = [random.randint(1,1000) for _ in range(500)]**: This line creates a list of 500 random integers between 1 and 1000 to test the functions.
- **start = time.time(); print("Brute:", brute_force_dup(data)); print("Time:", round(time.time()-start,6))**: This section measures and prints the time taken to find duplicates using the brute-force method.
- **start = time.time(); print("Optimized:", optimized_dup(data)); print("Time:", round(time.time()-start,6))**: This section measures and prints the time taken using the optimized method.
- **print("\nBrute → O(n^2) | Optimized → O(n)")**: This line prints the time complexities of both approaches, highlighting the difference in efficiency.

In summary, the code shows that using appropriate data structures (like sets) can significantly improve the performance of an algorithm, especially when dealing with larger datasets.