LAB ASSIGNMENT – 12

NAME : M.SHIVA

HALL.NO : 2403A52377

BATCH : AI 14

AI USED : PERPLEXITY.AI , GEMINI

PROMPT 01 :

Write a Python function for the Bubble Sort algorithm. Please add inline comments to explain the key logic, including the purpose of the passes, the comparison and swapping of elements, and how the algorithm terminates. After the code, provide a brief analysis of its time and space complexity

CODE :

```
def bubble_sort(arr):
    n = len(arr)
    # Traverse through all array elements. This outer loop represents
each pass.
    for i in range(n):
        # A flag to optimize the sort. If no swaps occur in a pass, the
array is already sorted.
        swapped = False
        # The last i elements are already in place, so the inner loop
range can be reduced.
        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] # This is the
swapping logic.
                swapped = True # Set the swapped flag to True to
indicate a swap occurred.

        # If no two elements were swapped by inner loop, then break.
The list is sorted.
        if not swapped:
```

```
            break
    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]
```

CODE EXPLANATION :

- The function bubble_sort takes a list arr as input.
- The outer loop (for i in range(n)) iterates through the list for each pass. With each pass, the next largest element "bubbles up" to its correct position at the end of the unsorted portion.
- The inner loop (for j in range(0, n-i-1)) compares adjacent elements arr[j] and arr[j+1].
- If arr[j] is greater than arr[j+1], their positions are swapped.
- A swapped flag is used for optimization. If an entire pass is completed without any swaps, it means the list is already sorted, and the function can terminate early.

PROMPT 02 :

Here is my Python implementation of Bubble Sort. For a list that is already partially sorted, is there a more efficient sorting algorithm? Please provide the code for that alternative algorithm and explain why it performs better than Bubble Sort on nearly sorted data

CODE :

```
def bubble_sort(arr):
    n = len(arr)
    # Outer loop for passes
    for i in range(n):
        swapped = False
```

```
        # Inner loop for comparisons and swaps
        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 no swaps in a pass, array is sorted
        if not swapped:
            break
    return arr

# Example with a partially sorted array
partially_sorted = [2, 1, 4, 6, 5, 8]
sorted_list = bubble_sort(partially_sorted.copy())
print(f"Bubble Sort Result: {sorted_list}")
```

OUTPUT :

```
Bubble Sort Result: [1, 2, 4, 5, 6, 8]
```

CODE EXPLANATION :

Bubble Sort Code Explanation

- def bubble_sort(arr):: Defines a function named bubble_sort that takes one argument, a list arr, which it will sort.
- n = len(arr): Gets the total number of elements in the list and stores it in the variable n.
- for i in range(n):: This is the outer loop that controls the number of "passes." The algorithm needs at most n passes to sort the list completely.
- swapped = False: A flag is created at the start of each pass. It's used to check if any swaps were made during this pass. This is an optimization.
- for j in range(0, n-i-1):: This is the inner loop where adjacent elements are compared. The range decreases with each outer loop pass (n-i-1) because the largest elements are already moved to the end of the list.
- if arr[j] > arr[j+1]:: This condition checks if the current element is greater than the one next to it.

- arr[j], arr[j+1] = arr[j+1], arr[j]: If the condition is true, this line performs the swap, putting the smaller element first.
- swapped = True: The flag is set to True to record that a swap has occurred in this pass.
- if not swapped: break: After the inner loop completes, this checks the flag. If no swaps were made (swapped is False), it means the list is already sorted, and the break statement exits the outer loop early to save time.
- return arr: The function returns the final sorted list.

Insertion Sort Code Explanation

- def insertion_sort(arr):: Defines a function named insertion_sort that accepts a list arr to be sorted.
- for i in range(1, len(arr)):: This loop iterates from the second element (index 1) to the end of the list. It treats the portion of the list before index i as the "sorted" part.
- key = arr[i]: The current element to be sorted is selected and stored in the key variable. The goal is to insert this key into its correct place within the sorted part of the list (arr[0...i-1]).
- j = i - 1: A new variable j is initialized to the index just before the key. This j will be used to scan backwards through the sorted part of the list.
- while j >= 0 and key < arr[j]:: This while loop runs as long as two conditions are met:
    1. j >= 0: We haven't reached the beginning of the list.
    2. key < arr[j]: The key element is smaller than the element at position j in the sorted part.
- arr[j + 1] = arr[j]: While the loop runs, it shifts the larger element (arr[j]) one position to the right to make space for the key.
- j -= 1: The index j is decremented to compare the key with the next element to the left.
- arr[j + 1] = key: Once the while loop finishes, j+1 is the correct position for the key. This line inserts the key into the empty spot created by the shifting.
- return arr: After iterating through all elements, the function returns the fully sorted list.

PROMPT 03 :

Please provide Python implementations for both Linear Search and Binary Search. For each function, include a detailed docstring explaining what the function does, its parameters, what it returns, and a 'Performance Notes' section within the docstring that describes its time complexity. Also, explain the conditions under which Binary Search is preferable to Linear Search.

CODE AND OUTPUT 1:

```
def linear_search(arr, target):
    """
    Performs a linear search to find the index of a target element in a
list.

    This function iterates through each element of the list one by one
until it
    finds the target or reaches the end of the list.

    Parameters:
    arr (list): The list to be searched. It can be sorted or unsorted.
    target: The element to search for.

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

    Performance Notes:
    - Time Complexity: O(n), where 'n' is the number of elements in the
list.
      In the worst-case scenario, it has to scan the entire list.
    """
    for i in range(len(arr)):
        if arr[i] == target:
            return i  # Target found, return its index
    return -1  # Target not found in the list
```

CODE AND OUTPUT 2:

```
def binary_search(arr, target):
    """
    Performs a binary search to find the index of a target element in a
sorted list.

    This function repeatedly divides the search interval in half. If
the value of the
    search key is less than the item in the middle of the interval, it
narrows
    the interval to the lower half. Otherwise, it narrows it to the
upper half.

    Parameters:
    arr (list): The list to be searched. IMPORTANT: This list must be
sorted.
    target: The element to search for.

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

    Performance Notes:
    - Time Complexity: O(log n), where 'n' is the number of elements in
the list.
      This is significantly faster than linear search for large lists
because it
      eliminates half of the remaining elements with each comparison.
    """
    low = 0
    high = len(arr) - 1

    while low <= high:
        mid = (low + high) // 2  # Find the middle index

        # Check if target is present at mid
        if arr[mid] == target:
            return mid

        # If target is greater, ignore the left half
        elif arr[mid] < target:
            low = mid + 1

        # If target is smaller, ignore the right half
        else:
            high = mid - 1

    return -1  # Target is not present in the array
```

CODE EXPLANATION :

Linear Search

- def linear_search(arr, target):: Defines a function that takes a list arr and a target value to find.
- for i in range(len(arr)):: This loop iterates through every index i of the list, from the beginning to the end.
- if arr[i] == target:: At each position, it checks if the element at the current index i matches the target.
- return i: If a match is found, the function immediately stops and returns the index i.
- return -1: If the loop finishes without finding the target, it means the element is not in the list, so the function returns -1.

Binary Search

- def binary_search(arr, target):: Defines a function that takes a sorted list arr and a target value.
- low = 0, high = len(arr) - 1: Initializes two pointers: low at the start of the list and high at the end. These define the current search space.
- while low <= high:: The search continues as long as the low pointer is not past the high pointer.
- mid = (low + high) // 2: Calculates the middle index of the current search space.
- if arr[mid] == target:: Checks if the element at the middle index is the target. If yes, the index is returned.
- elif arr[mid] < target:: If the middle element is less than the target, it means the target must be in the right half of the search space. So, low is updated to mid + 1.
- else:: If the middle element is greater than the target, it means the target must be in the left half. So, high is updated to mid - 1.
- return -1: If the while loop finishes, the low pointer has crossed the high pointer, meaning the target was not found. The function returns -1.

PROMPT 04:

I have the following partially completed recursive functions for Quick Sort and Merge Sort. Please complete the missing logic inside the functions. Also, add a detailed docstring to each function that explains its purpose, parameters, return value, and its average, best, and worst-case time complexities. Finally, provide a brief comparison of how they perform on random, sorted, and reverse-sorted lists

CODE AND OUTPUT 01 :

```
def quick_sort(arr):
    """
    Sorts a list using the Quick Sort algorithm via recursion.

    It works by selecting a 'pivot' element and partitioning the other
    elements into two sub-lists according to whether they are less than
    or greater than the pivot. The sub-lists are then sorted
recursively.

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

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

    Complexity Analysis:
    - Average Case: O(n log n) - Highly efficient for random data.
    - Best Case: O(n log n) - When partitions are perfectly balanced.
    - Worst Case: O(n^2) - Occurs with sorted or reverse-sorted data if
the pivot is chosen poorly (e.g., always the first or last element),
leading to unbalanced partitions.
    """
    if len(arr) <= 1:
        return arr  # Base case: a list of 0 or 1 elements is already
sorted.

    # Choose the middle element as the pivot to improve performance on
sorted lists
    pivot = arr[len(arr) // 2]

    # Partition the list into three parts
    left = [x for x in arr if x < pivot]    # Elements smaller than the
pivot
    middle = [x for x in arr if x == pivot] # Elements equal to the
pivot
    right = [x for x in arr if x > pivot]   # Elements larger than the
pivot

    # Recursively sort the left and right parts and combine them
    return quick_sort(left) + middle + quick_sort(right)
```

CODE AND OUTPUT 02 :

```
def merge_sort(arr):
    """
    Sorts a list using the Merge Sort algorithm via recursion.

    It is a divide-and-conquer algorithm that divides the list into two
    halves, recursively sorts them, and then merges the sorted halves.

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

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

    Complexity Analysis:
    - Average, Best, and Worst Case: O(n log n) - Its performance is
very
      consistent because the list is always split into equal halves.
    """
    if len(arr) <= 1:
        return arr  # Base case: a list of 0 or 1 elements is already
sorted.

    # 1. Divide
    mid = len(arr) // 2
    left_half = arr[:mid]
    right_half = arr[mid:]

    # 2. Conquer (Recursively sort the halves)
    left_sorted = merge_sort(left_half)
    right_sorted = merge_sort(right_half)

    # 3. Combine (Merge the sorted halves)
    return merge(left_sorted, right_sorted)

def merge(left, right):
    """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])
            i += 1
        else:
            sorted_list.append(right[j])
```

```
            j += 1

    # Append any remaining elements
    sorted_list.extend(left[i:])
    sorted_list.extend(right[j:])
    return sorted_list
```

CODE EXPLANATION :

Quick Sort

- Base Case: The recursion stops if the list has 0 or 1 element, as it's already considered sorted.
- Pivot Selection: An element is chosen as the "pivot." Here, we use the middle element, which helps avoid the worst-case scenario on sorted lists.
- Partitioning: The list is divided into three smaller lists: left (elements smaller than the pivot), middle (elements equal to the pivot), and right (elements larger than the pivot).
- Recursive Call: The quick_sort function is called again on the left and right lists.
- Combine: The final sorted list is created by combining the sorted left list, the middle list, and the sorted right list.

Merge Sort

- Base Case: The recursion stops when a list has 0 or 1 element.
- Divide: The list is split into two halves: left_half and right_half.
- Conquer (Recursive Call): merge_sort is called recursively on both halves until the base case is reached. This breaks the original list down into lists of single elements.
- Combine (Merge): The merge helper function takes two sorted lists and combines them into one larger sorted list by repeatedly comparing the first elements of each and appending the smaller one to the result.

PROMPT 05 :

"I have written a brute-force Python function to find duplicate elements in a list. It works, but it's very slow on large inputs. Can you provide a more optimized version of this algorithm that can find the duplicates more efficiently? Please explain how your optimized version improves the time complexity and compare their performance

CODE AND OUTPUT 1 :

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

CODE AND OUTPUT 2 :

```
def find_duplicates_optimized(arr):
    """
    Finds duplicate elements in a list using an optimized, O(n)
approach.

    It uses a set to keep track of seen elements, achieving linear time
complexity.
    """
    seen = set()
    duplicates = set()
    for element in arr:
        # If the element has been seen before, it's a duplicate
        if element in seen:
            duplicates.add(element)
        else:
            # Otherwise, add it to the set of seen elements
            seen.add(element)
    return sorted(list(duplicates))
```

CODE EXPLANATION :

Brute-Force Algorithm

- for i in range(n):: The outer loop that selects an element.
- for j in range(i + 1, n):: The inner loop that compares the selected element with all subsequent elements.
- if arr[i] == arr[j]...: Checks if two elements are identical.
- ... and arr[i] not in duplicates:: This is the slow part. To avoid adding the same duplicate multiple times, it performs a linear search on the duplicates list.

Optimized Algorithm

- seen = set(): Initializes an empty set to store unique numbers we have encountered so far.
- duplicates = set(): Initializes a set to store the duplicate numbers found (using a set avoids storing the same duplicate multiple times automatically).
- for element in arr:: A single loop that iterates through the list just once.
- if element in seen:: This is the fast part. It checks if the current element is already in the seen set. This check is, on average, an O(1) operation.
- duplicates.add(element): If the element was already in seen, it's a duplicate, so it's added to the duplicates set.
- seen.add(element): If the element is new, it's added to seen for future checks.