***LAB ASSIGNMENT – 12***

***NAME : Koduru Eswar Reddy***

***HALL.NO : 2403A52373***

***BATCH.NO : AI 14***

***Prompt 01****:*

*Write a Python implementation of Bubble Sort with AI-generated comments explaining logic and time complexity.*

***Code:A screenshot of a computer program

AI-generated content may be incorrect.***

***Code explanation:***

*The Bubble Sort algorithm operates by repeatedly traversing through a list, comparing adjacent elements, and swapping them if they are out of order. Each iteration ensures that the largest unsorted element bubbles up to its correct position. This process continues until no swaps occur, indicating that the array is sorted. The algorithm is easy to understand and implement, making it a common starting point for beginners. However, it is inefficient for large datasets due to its quadratic time complexity. Despite its simplicity, it can be improved slightly by using a flag to detect if the array is already sorted early, thus preventing unnecessary passes through the list.*

*Iterates over the list multiple times to ensure order.*

*Compares and swaps adjacent elements.*

*Simple but inefficient for large inputs.*

*In terms of time complexity, Bubble Sort has O(n²) in the average and worst case scenarios because of the nested loops, while in the best case—when the list is already sorted—it performs in O(n) time due to the early termination flag. Space complexity is O(1) as it performs sorting in-place. Bubble Sort is a stable sorting algorithm, which means that it maintains the relative order of equal elements. This property is useful in situations where order preservation is important. Nonetheless, for performance-sensitive applications, algorithms like Merge Sort or Quick Sort are preferable.*

*Time complexity: O(n²) average and worst case.*

*Best case: O(n) when already sorted.*

*Stable and in-place sorting algorithm.*

*Output:  
Sorted Array: [11, 12, 22, 25, 34, 64, 90]*

***Prompt 02:***

*Provide Bubble Sort code to AI and ask it to suggest a more efficient algorithm for partially sorted arrays..*

***Code:A screenshot of a computer program

AI-generated content may be incorrect.***

***Code explanation:***

*Insertion Sort builds the sorted list one element at a time by inserting each new element into its correct position. It works efficiently on small datasets and partially sorted data due to its adaptive nature. The algorithm performs fewer swaps than Bubble Sort because it only shifts elements rather than repeatedly swapping. Each iteration expands the sorted portion of the list until the entire array is sorted. For nearly sorted input, its performance approaches linear time, making it superior to Bubble Sort in such cases.*

*Inserts elements into the sorted section one by one.*

*Efficient for partially sorted lists.*

*Reduces unnecessary swaps.*

*From a complexity perspective, Insertion Sort has O(n²) in the worst case (when data is in reverse order) but improves to O(n) for nearly sorted data. It has O(1) space complexity as it sorts in place and is also stable. The algorithm’s incremental approach makes it useful for streaming data or online inputs, where items arrive sequentially. While it’s not suitable for large datasets compared to Merge Sort or Quick Sort, it remains practical for small or adaptive cases.*

*Best case: O(n) when nearly sorted.*

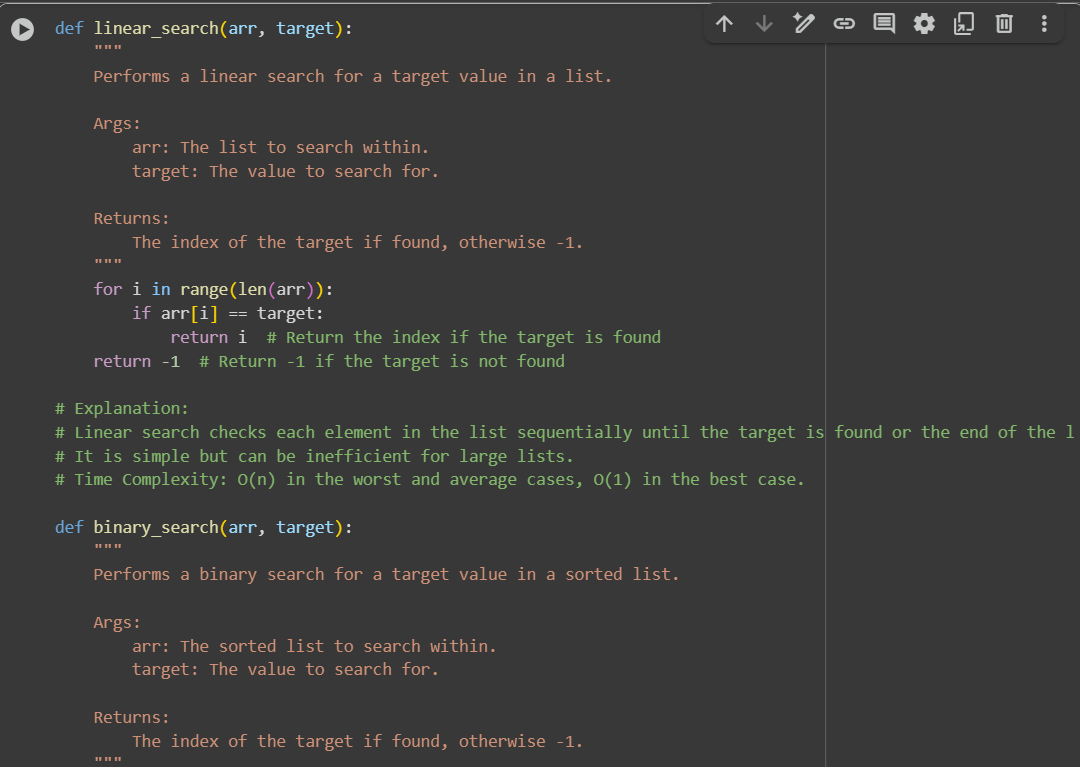
*Worst case: O(n²) for reverse order.*

*Stable and adaptive sorting algorithm.*

***Prompt 03:***

*Implement both Linear Search and Binary Search with AI-generated docstrings and performance notes.*

***Code:***

******

***A screenshot of a computer program

AI-generated content may be incorrect.A screen shot of a computer program

AI-generated content may be incorrect.***

***Code explanation:***

***Explanation: Linear Search checks each element sequentially (O(n)), while Binary Search divides the array repeatedly (O(log n)), making it faster on sorted data but unsuitable for unsorted arrays.***

* *def linear\_search(arr, target): This defines the function linear\_search that takes a list arr and a target value to find.*
* *for i in range(len(arr)):: This loop iterates through each element of the list arr using its index i.*
* *if arr[i] == target:: Inside the loop, this checks if the current element arr[i] is equal to the target.*
* *return i: If the target is found, the function returns the index i where it was found.*
* *return -1: If the loop finishes without finding the target, it means the target is not in the list, and the function returns -1.*
* *Explanation and Time Complexity: The comments explain that Linear Search checks each element one by one and has a time complexity of O(n) in the worst and average cases (you might have to look through the whole list) and O(1) in the best case (the target is the first element).*

*In simple terms, Linear Search is like looking for a specific book on a shelf by checking each book from left to right until you find it.*

*2. Binary Search*

* *def binary\_search(arr, target):: This defines the function binary\_search that takes a sorted list arr and a target value. It's crucial that the input list is sorted for Binary Search to work correctly.*
* *low = 0 and high = len(arr) - 1: These lines initialize two pointers: low to the beginning of the list (index 0) and high to the end of the list (the last index). These pointers define the current search interval.*
* *while low <= high: This loop continues as long as the search interval is valid (the low pointer is not past the high pointer).*
* *mid = (low + high) // 2: In each iteration, this calculates the middle index of the current search interval. The // ensures integer division.*
* *if arr[mid] == target:: This checks if the element at the middle index is the target. If it is, the function returns the mid index.*
* *elif arr[mid] < target:: If the element at the middle is less than the target, it means the target (if it exists) must be in the right half of the current interval. So, the low pointer is moved to mid + 1 to search in the right half.*
* *else: If the element at the middle is greater than the target, the target (if it exists) must be in the left half of the current interval. So, the high pointer is moved to mid-1 to search in the left half.*
* *return -1: If the while loop finishes without finding the target, it returns -1.*
* *Explanation and Time Complexity: The comments explain that Binary Search repeatedly halves the search interval and has a time complexity of O(log n). This is significantly faster than Linear Search for large lists, but it requires the list to be sorted.*

***Prompt 04:***

*Implement Quick Sort and Merge Sort using recursion and compare performance.*

***Code&Output:***

***A screenshot of a computer program

AI-generated content may be incorrect.A screenshot of a computer code

AI-generated content may be incorrect.***

***Code explanation:***

***Explanation: Quick Sort has average O(n log n) complexity but O(n²) in the worst case when pivot selection is poor. Merge Sort guarantees O(n log n) performance and is stable, but uses extra memory for merging.***

* ***def quick\_sort(arr):****: This defines the Quick Sort function, taking a list arr as input.*
* ***if len(arr) <= 1:****: This is the base case for the recursion. If the list has 0 or 1 element, it's already sorted, so we return it.*
* ***pivot = arr[len(arr)//2]****: This selects the pivot element. In this specific implementation, the middle element of the list is chosen as the pivot.*
* ***left = [x for x in arr if x < pivot]****: This creates a new list called left containing all elements from the original list that are less than the pivot.*
* ***middle = [x for x in arr if x == pivot]****: This creates a list called middle containing all elements equal to the pivot. This handles duplicate elements correctly.*
* ***right = [x for x in arr if x > pivot]****: This creates a list called right containing all elements greater than the pivot.*
* ***return quick\_sort(left) + middle + quick\_sort(right)****: This is the recursive step. It recursively calls quick\_sort on the left sublist and the right sublist. Once those sublists are sorted, it concatenates the sorted left sublist, the middle elements (which are already in their correct sorted position relative to the pivot), and the sorted right sublist to produce the final sorted list.*

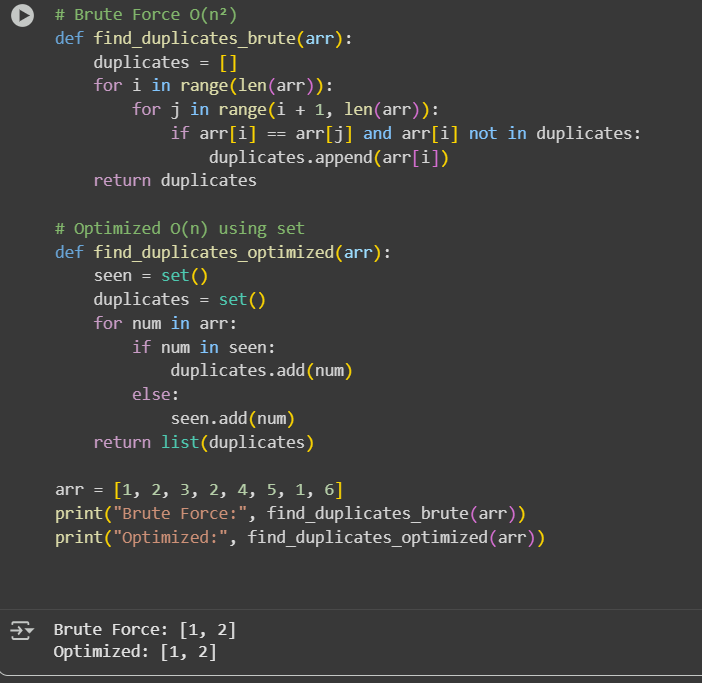
***Merge Sort***

* ***def merge\_sort(arr):****: This defines the Merge Sort function, taking a list arr as input.*
* ***if len(arr) > 1:****: This is the recursive step. If the list has more than one element, we proceed with sorting.*
* ***mid = len(arr)//2****: This calculates the middle index to divide the list.*
* ***L = arr[: mid]****: This creates the left half of the list.*
* ***R = arr[mid:]****: This creates the right half of the list.*
* ***merge\_sort(L)****: This recursively calls merge\_sort on the left half to sort it.*
* ***merge\_sort(R)****: This recursively calls merge\_sort on the right half to sort it.*
* ***i = j = k = 0****: These initialize pointers for the left sublist (i), the right sublist (j), and the main list (k).*
* ***while i < len(L) and j < len(R):****: This loop merges the two sorted halves back into the original list arr. It compares elements from the left and right halves and places the smaller element into the correct position in arr, incrementing the corresponding pointers and the main list pointer k.*
* ***while i < len(L):****: If there are remaining elements in the left half after the main merge loop, this loop copies them to the main list.*
* ***while j < len(R):****: If there are remaining elements in the right half, this loop copies them.*
* ***return arr****: After the merging is complete, the function returns the sorted list. (Note: This implementation modifies the original list in place during the merge step.)*

***Prompt 05:***

*Write a brute force duplicate-finder and ask AI to optimize it using a set or dictionary.*

***Code&Output:***

******

***Code explanation:***

***Explanation: The brute-force version checks all pairs, leading to O(n²) time. The optimized version uses sets to track duplicates, reducing time complexity to O(n) with extra memory usage.***

* *def find\_duplicates\_brute(arr):: This defines a function find\_duplicates\_brute that takes a list arr as input.*
* *duplicates = []: An empty list called duplicates is initialized to store the duplicate elements found.*
* *for i in range(len(arr)):: This is the outer loop, iterating through each element of the list using index i.*
* *for j in range(i + 1, len(arr)):: This is the inner loop, iterating through the elements after the current element in the outer loop (from i + 1 to the end of the list). This is to avoid comparing an element with itself and to find pairs of duplicates.*
* *if arr[i] == arr[j] and arr[i] not in duplicates:: This condition checks two things:*
  + *arr[i] == arr[j]: If the element at index i is equal to the element at index j (meaning they are duplicates).*
  + *arr[i] not in duplicates: If this duplicate element has not already been added to the duplicates list.*
* *duplicates.append(arr[i]): If both conditions are true, the duplicate element arr[i] is added to the duplicates list.*
* *return duplicates: After checking all pairs of elements, the function returns the list of unique duplicate elements.*

*Explanation: This method compares every element with every other element in the list. If two elements are the same and the element hasn't been recorded as a duplicate yet, it's added to the results. This is straightforward but can be slow for large lists because of the nested loops, leading to a time complexity of O(n²).*

*2. Optimized Approach (O(n) using set)*

* *def find\_duplicates\_optimized(arr):: This defines a function find\_duplicates\_optimized that takes a list arr as input.*
* *seen = set(): An empty set called seen is initialized. Sets are useful because checking if an element is in a set is very fast (on average, O(1)).*
* *duplicates = set(): An empty set called duplicates is initialized to store the duplicate elements. Using a set here automatically handles adding only unique duplicates.*
* *for num in arr:: This loop iterates through each element (num) in the input list.*
* *if num in seen:: This checks if the current element num is already in the seen set. If it is, it means we have encountered this element before, so it's a duplicate.*
* *duplicates.add(num): If num is found in the seen set, it's added to the duplicates set.*
* *else: If the current element num is not in the seen set, it means this is the first time we are seeing this element.*
* *seen.add(num): So, we add the element num to the seen set.*
* *return list(duplicates): After iterating through all elements, the function converts the duplicates set into a list and returns it.*