ASSIGNMENT-12.4

## Task 1: Implementing Bubble Sort with AI Comments

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

Code:def bubble\_sort(arr):  
 for i in range(len(arr) - 1):  
 swapped = False  
 for j in range(len(arr) - 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  
  
data = [64, 34, 25, 12, 22, 11, 90]  
print("Sorted Array:", bubble\_sort(data))

Explanation:

Bubble Sort compares each pair of adjacent elements and swaps them if they are in the wrong order. After each full pass, the largest element moves to the end, like a bubble rising in water. This keeps happening until the list becomes sorted. It’s simple to code and easy to understand, making it good for beginners. The algorithm uses a flag to detect if the list is already sorted and stops early to save time. Even though it’s basic, it helps in understanding sorting logic.

* Compares and swaps adjacent items.
* Largest element moves to the end in each pass.
* Stops early if no swaps happen.

The time complexity of Bubble Sort is O(n²) in the worst and average cases because it compares many elements repeatedly. In the best case, when the list is already sorted, it can finish in O(n). It doesn’t need extra space since sorting happens in place. However, it’s not suitable for big datasets because it’s slow compared to Quick Sort or Merge Sort. Still, it’s a good example of how optimization and early exit checks can make simple algorithms smarter.

* Time complexity: O(n²) worst, O(n) best.
* Works in-place with no extra memory.
* Not efficient for large data sets.

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

## Task 2: Optimizing Bubble Sort → Insertion Sort

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

Code:def insertion\_sort(arr):  
 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  
 return arr  
  
data = [11, 12, 22, 25, 34, 64, 90]  
print("Optimized Sorted Array:", insertion\_sort(data))

Explanation:

Insertion Sort works by dividing the list into sorted and unsorted parts. It picks each element from the unsorted part and places it at the right position in the sorted part. It performs well for small or nearly sorted data. Each pass reduces disorder and keeps the list closer to sorted. It shifts items rather than swapping, so fewer operations occur compared to Bubble Sort. This makes it more efficient for inputs that are already almost in order.

* Builds sorted list step by step.
* Shifts elements to insert the new one correctly.
* Good for small or sorted data.

Its time complexity is O(n²) in the worst case, but for nearly sorted data, it performs close to O(n). It is also stable and sorts in-place, using O(1) extra memory. Insertion Sort is useful when data comes one by one, like streaming or incremental input. Although slower than Quick Sort on large data, it’s practical when working with smaller datasets.

* Best case O(n), worst O(n²).
* Stable and in-place.
* Useful for incremental sorting.

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

## Task 3: Binary Search vs Linear Search

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

Code:def linear\_search(arr, target):  
 for i in range(len(arr)):  
 if arr[i] == target:  
 return i  
 return -1  
  
def binary\_search(arr, target):  
 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  
  
arr = [11, 12, 22, 25, 34, 64, 90]  
print("Linear Search Result:", linear\_search(arr, 25))  
print("Binary Search Result:", binary\_search(arr, 25))

Explanation:

Linear Search checks each element one by one until it finds the target or reaches the end. It’s simple to implement and works for any data, sorted or not. However, it becomes slow as the list grows because it may check every item. Binary Search, on the other hand, only works on sorted data. It repeatedly divides the list in half, checking if the target is greater or smaller than the middle element.

* Linear Search: checks all elements.
* Binary Search: divides the range in half.
* Binary Search needs sorted input.

Because of this halving approach, Binary Search is much faster with time complexity O(log n), while Linear Search is O(n). Binary Search is best when data is sorted and static. In contrast, Linear Search is flexible but inefficient for large datasets. Both are useful depending on whether sorting is possible beforehand.

* Linear: O(n), Binary: O(log n).
* Binary is faster on sorted lists.
* Linear is simple but slower on large data.

Output:  
Linear Search Result: 3  
Binary Search Result: 3

## Task 4: Quick Sort and Merge Sort Comparison

Prompt: Implement Quick Sort and Merge Sort using recursion and compare performance.

Code:def quick\_sort(arr):  
 if len(arr) <= 1:  
 return arr  
 pivot = arr[len(arr)//2]  
 left = [x for x in arr if x < pivot]  
 middle = [x for x in arr if x == pivot]  
 right = [x for x in arr if x > pivot]  
 return quick\_sort(left) + middle + quick\_sort(right)  
  
def merge\_sort(arr):  
 if len(arr) > 1:  
 mid = len(arr)//2  
 L = arr[:mid]  
 R = arr[mid:]  
 merge\_sort(L)  
 merge\_sort(R)  
 i = j = k = 0  
 while i < len(L) and j < len(R):  
 if L[i] < R[j]:  
 arr[k] = L[i]  
 i += 1  
 else:  
 arr[k] = R[j]  
 j += 1  
 k += 1  
 while i < len(L):  
 arr[k] = L[i]  
 i += 1  
 k += 1  
 while j < len(R):  
 arr[k] = R[j]  
 j += 1  
 k += 1  
 return arr  
  
data = [64, 34, 25, 12, 22, 11, 90]  
print("Quick Sort:", quick\_sort(data))  
print("Merge Sort:", merge\_sort(data))

Explanation:

Quick Sort works by selecting a pivot element and splitting the list into smaller and larger parts. It then recursively sorts these parts. When the pivot is chosen well, it’s very fast. Merge Sort splits the list into halves until single elements remain, then merges them back in sorted order. Both are efficient divide-and-conquer algorithms that perform much better than Bubble or Insertion Sort on large data.

* Quick Sort: splits using pivot.
* Merge Sort: splits then merges.
* Both use recursion and divide-and-conquer.

Quick Sort’s average time complexity is O(n log n), but it can go up to O(n²) if the pivot is poor. Merge Sort always performs O(n log n), but it needs extra memory for merging. Quick Sort is faster for in-memory sorting, while Merge Sort is better for linked lists or large files.

* Quick Sort: O(n log n) avg, O(n²) worst.
* Merge Sort: O(n log n) always.
* Merge Sort uses extra memory.

Output:  
Quick Sort: [11, 12, 22, 25, 34, 64, 90]  
Merge Sort: [11, 12, 22, 25, 34, 64, 90]

## Task 5: AI-Suggested Algorithm Optimization

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

Code: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 num in arr:  
 if num in seen:  
 duplicates.add(num)  
 else:  
 seen.add(num)  
 return list(duplicates)  
  
arr = [1, 2, 3, 2, 4, 5, 1, 6]  
print("Brute Force:", find\_duplicates\_brute(arr))  
print("Optimized:", find\_duplicates\_optimized(arr))

Explanation:

The brute-force method checks every pair of elements to find duplicates, which takes a lot of time when the list is large. It compares each element to all others, leading to many unnecessary checks. The optimized version uses a set to track seen elements, making it much faster by checking membership in constant time. It also keeps a separate set for duplicates to avoid repetition.

* Brute force: checks all pairs.
* Optimized: uses set for faster lookup.
* Set operations make it quicker.

The brute-force version has O(n²) time complexity, while the optimized one runs in O(n). The trade-off is small extra memory for the set, but it’s worth it for speed. This approach shows how small changes in data structure choice can improve performance greatly.

* Brute force O(n²), optimized O(n).
* Uses extra memory but saves time.
* Efficient for large datasets.

Output:  
Brute Force: [1, 2]  
Optimized: [1, 2]