

ASSIGNMENT-12.4

TASK-1

Prompt: Write a Python implementation of Bubble Sort.

- Instructions:
 - o Students implement Bubble Sort normally.
 - o Ask AI to generate inline comments explaining key logic (like swapping, passes, and termination).
 - o Request AI to provide time complexity analysis.

Code with output:

```
[1] ✓ Os
def bubble_sort(arr):
    """
    Sorts a list using the Bubble Sort algorithm.

    Args:
        arr: The list to be sorted.
    """
    n = len(arr)
    # Traverse through all array elements
    for i in range(n):
        # Flag to optimize: if no two elements were swapped by inner loop, then break
        swapped = False

        # 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]
                swapped = True

        # If no two elements were swapped by inner loop, then break
        if not swapped:
            break
    return arr

# Example usage:
my_list = [64, 34, 25, 12, 22, 11, 90]
print("Original list:", my_list)
sorted_list = bubble_sort(my_list)
print("Sorted list:", sorted_list)
```

Original list: [64, 34, 25, 12, 22, 11, 90]
Sorted list: [11, 12, 22, 25, 34, 64, 90]

Explanation:

This notebook demonstrates the Bubble Sort algorithm in Python.

The code defines a function `bubble_sort(arr)` that takes a list `arr` as input and sorts it in ascending order using Bubble Sort.

How Bubble Sort works:

1. **Outer loop:** The outer loop iterates through the entire list. In each pass of the outer loop, the largest unsorted element "bubbles up" to its correct position at the end of the unsorted portion of the list.
2. **Inner loop:** The inner loop compares adjacent elements and swaps them if they are in the wrong order. This process is repeated until the end of the unsorted portion of the list is reached.
3. **Optimization:** A `swapped` flag is used to optimize the algorithm. If no swaps occur in a pass of the inner loop, it means the list is already sorted, and the algorithm can terminate early.

The example usage shows how to call the `bubble_sort` function with a sample list and prints the original and sorted lists.

The markdown cell below the code provides a time and space complexity analysis of the Bubble Sort algorithm.

TASK-2

Prompt: Optimizing Bubble Sort → Insertion Sort

- Task: Provide Bubble Sort code to AI and ask it to suggest a more efficient algorithm for partially sorted arrays.
- Instructions:
 - o Students implement Bubble Sort first.
 - o Ask AI to suggest an alternative (Insertion Sort).
 - o Compare performance on nearly sorted input.

Code with output:

1
0s

```
def insertion_sort(arr):  
    """  
    Sorts a list using the Insertion Sort algorithm.  
  
    Args:  
        arr: The list to be sorted.  
    """  
    # Traverse through 1 to len(arr)  
    for i in range(1, len(arr)):  
        key = arr[i]  
        # Move elements of arr[0..i-1], that are greater than key,  
        # to one position ahead of their current position  
        j = i - 1  
        while j >= 0 and key < arr[j]:  
            arr[j + 1] = arr[j]  
            j -= 1  
        arr[j + 1] = key  
    return arr  
  
# Example usage:  
my_list_insertion = [64, 34, 25, 12, 22, 11, 90]  
print("Original list for Insertion Sort:", my_list_insertion)  
sorted_list_insertion = insertion_sort(my_list_insertion)  
print("Sorted list using Insertion Sort:", sorted_list_insertion)
```

```
Original list for Insertion Sort: [64, 34, 25, 12, 22, 11, 90]  
Sorted list using Insertion Sort: [11, 12, 22, 25, 34, 64, 90]
```

Explanation:

This code implements the Insertion Sort algorithm. Here's a breakdown:


- **insertion_sort(arr) function:**
 - Takes a list `arr` as input.
 - The outer loop (`for i in range(1, len(arr))`) iterates from the second element of the list (`i = 1`) to the end.
 - `key = arr[i]` stores the current element being considered for insertion.
 - The inner loop (`while j >= 0 and key < arr[j]`) compares the `key` with elements to its left (`arr[j]`).
 - If an element to the left is greater than the `key`, it's shifted one position to the right (`arr[j + 1] = arr[j]`).
 - This shifting continues until the correct position for the `key` is found (either the beginning of the list or an element less than or equal to the `key`).
 - `arr[j + 1] = key` inserts the `key` into its sorted position.
 - The function returns the sorted list.
- **Example Usage:**
 - A sample list `my_list_insertion` is defined.
 - The original list is printed.
 - The `insertion_sort` function is called to sort the list.
 - The sorted list is printed.

TASK-3

Prompt: Binary Search vs Linear Search

- Task: Implement both Linear Search and Binary Search.
- Instructions:
 - Use AI to generate docstrings and performance notes.
 - Test both algorithms on sorted and unsorted data.
 - Ask AI to explain when Binary Search is preferable.

Code with output:

```
[5]  def linear_search(arr, target):  
    """  
    Performs a linear search on a list to find the index of a target value.  
  
    Args:  
        arr: The list to search within.  
        target: The value to search for.  
  
    Returns:  
        The index of the target in the list if found, otherwise -1.  
    """  
    for index, element in enumerate(arr):  
        if element == target:  
            return index  
    return -1  
  
def binary_search(arr, target):  
    """  
    Performs a binary search on a sorted list to find the index of a target value.  
  
    Args:  
        arr: The sorted list to search within.  
        target: The value to search for.  
  
    Returns:  
        The index of the target in the list if found, otherwise -1.  
    """  
    left, right = 0, len(arr) - 1  
    while left <= right:  
        mid = (left + right) // 2  
        if arr[mid] == target:  
            return mid  
        elif arr[mid] < target:  
            left = mid + 1  
        else:
```

```
        right = mid - 1
    return -1

# Example usage for Linear Search:
my_list_linear = [10, 5, 8, 12, 3, 7]
target_value_linear = 8
index_linear = linear_search(my_list_linear, target_value_linear)
print(f"Linear Search: The target value {target_value_linear} is found at index: {index_linear}")
target_value_linear = 99
index_linear = linear_search(my_list_linear, target_value_linear)
print(f"Linear Search: The target value {target_value_linear} is found at index: {index_linear}")

# Example usage for Binary Search (requires sorted list):
my_list_binary = [3, 5, 7, 8, 10, 12] # Sorted list
target_value_binary = 8
index_binary = binary_search(my_list_binary, target_value_binary)
print(f"Binary Search: The target value {target_value_binary} is found at index: {index_binary}")
target_value_binary = 99
index_binary = binary_search(my_list_binary, target_value_binary)
print(f"Binary Search: The target value {target_value_binary} is found at index: {index_binary}")
```

Linear Search: The target value 8 is found at index: 2
Linear Search: The target value 99 is found at index: -1
Binary Search: The target value 8 is found at index: 3
Binary Search: The target value 99 is found at index: -1

Explanation:

This code contains implementations of two search algorithms: Linear Search and Binary Search.

Linear Search (`linear_search` function):

- This function searches for a `target` value within a list `arr` by checking each element one by one from the beginning to the end.
- It iterates through the list using `enumerate` to get both the index and the element.
- If the current element matches the `target`, the function returns the current `index`.
- If the loop finishes without finding the `target`, it means the target is not in the list, and the function returns -1.
- Linear Search works on both sorted and unsorted lists.

Binary Search (`binary_search` function):

- This function searches for a `target` value within a **sorted** list `arr`.
- It works by repeatedly dividing the search interval in half.
- `left` and `right` variables represent the boundaries of the current search interval.
- In each iteration, it calculates the middle index (`mid`) of the interval.
- It compares the element at the `mid` index with the `target`:
 - If they are equal, the `target` is found, and the `mid` index is returned.
 - If the middle element is less than the `target`, the search continues in the right half of the interval by updating `left` to `mid + 1`.
 - If the middle element is greater than the `target`, the search continues in the left half of the interval by updating `right` to `mid - 1`.
- The loop continues until the `left` index is greater than the `right` index, indicating that the `target` is not in the list, and the function returns -1.
- **Important:** Binary Search is significantly faster than Linear Search for large lists, but it requires the input list to be sorted.

The example usage demonstrates how to call both functions with sample lists and target values, printing the results. Note that the list used for Binary Search (`my_list_binary`) is sorted.

TASK-4

Prompt: Quick Sort and Merge Sort Comparison,implement Quick Sort and Merge Sort using recursion.

- Instructions:

- o Provide AI with partially completed functions for recursion.
- o Ask AI to complete the missing logic and add docstrings.
- o Compare both algorithms on random, sorted, and reverse-sorted lists.

Code with output:

```
def quick_sort(arr):  
    """  
    Sorts a list using the recursive Quick Sort algorithm.  
  
    Args:  
        arr: The list to be sorted.  
  
    Returns:  
        The sorted list.  
    """  
    if len(arr) <= 1:  
        return arr  
  
    pivot = arr[-1]  
    less_than_pivot = []  
    greater_than_pivot = []  
  
    for i in range(len(arr) - 1):  
        if arr[i] <= pivot:  
            less_than_pivot.append(arr[i])  
        else:  
            greater_than_pivot.append(arr[i])  
  
    return quick_sort(less_than_pivot) + [pivot] + quick_sort(greater_than_pivot)  
  
def merge_sort(arr):  
    """  
    Sorts a list using the recursive Merge Sort algorithm.  
  
    Args:  
        arr: The list to be sorted.  
  
    Returns:
```

```

    """ The sorted list.
    """
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2
    left_half = arr[:mid]
    right_half = arr[mid:]

    left_half = merge_sort(left_half)
    right_half = merge_sort(right_half)

    return merge(left_half, right_half)

def merge(left, right):
    """
    Merges two sorted lists into a single sorted list.

    Args:
        left: The first sorted list.
        right: The second sorted list.

    Returns:
        A new list containing elements from both input lists in sorted order.
    """
    merged = []
    i = 0
    j = 0

    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            merged.append(left[i])
            i += 1
        else:
            merged.append(right[j])
            j += 1

    while i < len(left):
        merged.append(left[i])
        i += 1

    while j < len(right):
        merged.append(right[j])
        j += 1

    return merged

# Example usage for Quick Sort:
my_list_quick = [64, 34, 25, 12, 22, 11, 90]
print("Original list for Quick Sort:", my_list_quick)
sorted_list_quick = quick_sort(my_list_quick)
print("Sorted list using Quick Sort:", sorted_list_quick)

# Example usage for Merge Sort:
my_list_merge = [38, 27, 43, 3, 9, 82, 10]
print("Original list for Merge Sort:", my_list_merge)
sorted_list_merge = merge_sort(my_list_merge)
print("Sorted list using Merge Sort:", sorted_list_merge)

Original list for Quick Sort: [64, 34, 25, 12, 22, 11, 90]
Sorted list using Quick Sort: [11, 12, 22, 25, 34, 64, 90]
Original list for Merge Sort: [38, 27, 43, 3, 9, 82, 10]
Sorted list using Merge Sort: [3, 9, 10, 27, 38, 43, 82]
```

Explanation:

Time Complexity Analysis of Quick Sort and Merge Sort

Quick Sort:

- **Best Case:** $O(n \log n)$ - Occurs when the pivot element is always the median or close to the median, dividing the list into roughly equal halves at each step.
- **Average Case:** $O(n \log n)$ - The average performance is also very good due to the balanced partitioning that happens on average.
- **Worst Case:** $O(n^2)$ - Occurs when the pivot selection consistently results in highly unbalanced partitions (e.g., always picking the smallest or largest element as the pivot in an already sorted or reverse-sorted list).

Merge Sort:

- **Best Case:** $O(n \log n)$ - The time complexity is always $O(n \log n)$ regardless of the initial order of the elements because it always divides the list into halves and merges them.
- **Average Case:** $O(n \log n)$ - Same as the best case.
- **Worst Case:** $O(n \log n)$ - Same as the best case.

Space Complexity:

- **Quick Sort:** $O(\log n)$ on average (due to recursion stack), but can be $O(n)$ in the worst case with a naive pivot selection.
- **Merge Sort:** $O(n)$ due to the need for a temporary array during the merging process.

Comparison:

- Both Quick Sort and Merge Sort have an average time complexity of $O(n \log n)$, which is generally considered efficient for larger datasets.
- Merge Sort has a guaranteed $O(n \log n)$ time complexity in all cases (best, average, and worst), making it more predictable.
- Quick Sort's performance is highly dependent on the pivot selection strategy. While its average case is excellent, its worst-case can be significantly slower than Merge Sort.
- Merge Sort requires more extra space ($O(n)$) compared to Quick Sort ($O(\log n)$ on average).

TASK-5

Prompt: AI-Suggested Algorithm Optimization

- Task: Give AI a naive algorithm (e.g., $O(n^2)$ duplicate search).
- Instructions:
 - o Students write a brute force duplicate-finder.
 - o Ask AI to optimize it (e.g., by using sets/dictionaries with $O(n)$ time).
 - o Compare execution times with large input sizes.

Code with output:


```

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

    Args:
        arr: The list to search for duplicates.

    Returns:
        A list of unique duplicate elements found in the input list.
    """
    seen = set()
    duplicates = []
    for element in arr:
        if element in seen:
            if element not in duplicates:
                duplicates.append(element)
        else:
            seen.add(element)
    return duplicates

# Example usage:
my_list_optimized = [1, 2, 3, 4, 2, 5, 6, 3, 7, 8, 8, 1]
print("Original list:", my_list_optimized)
duplicate_elements_optimized = find_duplicates_optimized(my_list_optimized)
print("Duplicate elements (optimized):", duplicate_elements_optimized)

```

```

Original list: [1, 2, 3, 4, 2, 5, 6, 3, 7, 8, 8, 1]
Duplicate elements (optimized): [2, 3, 8, 1]

```

Explanation:

Explanation of Complexity Improvement

The naive duplicate finding algorithm has a time complexity of $O(n^2)$ because it uses nested loops. For each element in the list, it iterates through the rest of the list to check for duplicates. This means the number of operations grows quadratically with the size of the input list (n).

The optimized algorithm, which uses a set, has a time complexity of $O(n)$. It achieves this by leveraging the constant time complexity ($O(1)$ on average) of set operations (adding and checking for membership). The algorithm iterates through the list only once. For each element, it checks if it's already in the `seen` set. If it is, it's a duplicate. If not, it adds the element to the `seen` set.

Therefore, by using a set, the optimized algorithm reduces the time complexity from quadratic ($O(n^2)$) to linear ($O(n)$), making it significantly faster for large input lists.