Merge Sort:

Aim:
The aim of this lab practical is to implement and analyze the efficiency of the MergeSort algorithm in sorting a collection of elements in ascending order.

Theory:
Merge sort is a divide-and-conquer sorting algorithm that recursively divides an array into halves until base cases are reached. It then merges the sorted halves back together. The key step is the merge operation, which combines two sorted arrays into a single sorted array. Known for its stability and consistent O(n log n) time complexity, merge sort is efficient for large datasets. Its main drawback is the additional space requirement for temporary arrays during the merging process.

Code:
```
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        left_half = arr[:mid]
        right_half = arr[mid:]

        merge_sort(left_half)
        merge_sort(right_half)

        i = j = k = 0
        while i < len(left_half) and j < len(right_half):
            if left_half[i] < right_half[j]:
                arr[k] = left_half[i]
                i += 1
            else:
                arr[k] = right_half[j]
                j += 1
            k += 1

        while i < len(left_half):
            arr[k] = left_half[i]
            i += 1
            k += 1

        while j < len(right_half):
            arr[k] = right_half[j]
            j += 1
            k += 1
```

```
my_list = [22, 11, 33, 44, 66, 77]
print("Original array:", my_list)
merge_sort(my_list)
print("Sorted array:", my_list)
```

Output:

```
Original array: [22, 11, 33, 44, 66, 77]
Sorted array: [11, 22, 33, 44, 66, 77]
```

Algorithm Analysis:

Divide (Splitting): The array is recursively divided into halves until each subarray contains only one element. This process takes O(log n) time, as it involves repeatedly dividing the array in half.

Conquer (Merging): The merging step occurs during the recursion's backtracking phase. At each level, the merging of two sorted subarrays takes linear time O(n), where 'n' is the total number of elements across both subarrays.

Since the depth of the recursion tree is log n, and at each level, the merging step takes O(n) time, the overall time complexity is O(n log n).

Quick Sort:

Aim:
The aim of this lab practical is to implement and analyze the efficiency of the Quick Sort algorithm in sorting a collection of elements in ascending order.

Theory:
Quicksort is a widely used sorting algorithm that employs a divide-and-conquer strategy. It selects a pivot element, partitions the array into elements less than the pivot and elements greater than the pivot, and recursively sorts each partition. Quicksort has an average time complexity of O(n log n), making it efficient for large datasets. However, its performance can degrade to O(n^2) in the worst case if poorly chosen pivots. Efficient implementations often use randomized or median-of-three pivot selection to mitigate this issue.

Code:
```
def merge_sort(arr):
    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):
    merged = []
    left_index, right_index = 0, 0

    while left_index < len(left) and right_index < len(right):
        if left[left_index] < right[right_index]:
            merged.append(left[left_index])
            left_index += 1
        else:
            merged.append(right[right_index])
            right_index += 1
```

```
    merged.extend(left[left_index:])
    merged.extend(right[right_index:])

    return merged

arr = [66, 33, 22, 11, 44, 55, 77]
sorted_arr = merge_sort(arr)
print("Sorted array:", sorted_arr)
```

Output:

```
Original array: [66, 33, 22, 11, 44, 55, 77]
Sorted array: [11, 22, 33, 44, 55, 66, 77]
```

Algorithm analysis:

Time Complexity:

The divide step takes O(1) time as it involves only slicing the array to obtain the mid-point.
The conquer step involves recursively solving two subproblems of half the size, leading to a time complexity of T(n/2).
The merge step takes O(n) time, where n is the total number of elements in the array.
The overall time complexity can be expressed as T(n) = 2 * T(n/2) + O(n), which is O(n log n) in the worst and average case.

Space Complexity:

The space complexity of merge sort is O(n) due to the additional space required for the temporary arrays during the merging process.
Each recursive call creates a new set of temporary arrays, and the space required is proportional to the size of the input.