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).

Space Complexity: The space complexity of the merge sort implementation is O(n), where 'n' is the length of the input array. This space complexity arises from the recursion depth and the additional space required for the temporary left and right halves during the merging process.

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 quicksort(arr):
    if len(arr) <= 1:
        return arr
    else:
        pivot = arr[0]
        less = [x for x in arr[1:] if x <= pivot]
        greater = [x for x in arr[1:] if x > pivot]
        return quicksort(less) + [pivot] + quicksort(greater)

my_list = [12, 4, 5, 6, 7, 3, 1, 15]
print("Original array:", my_list)
sorted_list = quicksort(my_list)
print("Sorted array:", sorted_list)
```

Output:

```
Original array: [12, 4, 5, 6, 7, 3, 1, 15]
Sorted array: [1, 3, 4, 5, 6, 7, 12, 15]
```

Algorithm analysis:

Best Case Time Complexity: The best-case time complexity occurs when the pivot is always chosen in such a way that it consistently divides the array into two equal halves. In this ideal scenario, each recursive call roughly halves the size of the remaining subarrays.
The best-case time complexity is O(n log n).

Worst Case Time Complexity:
The worst-case time complexity occurs when the pivot is consistently chosen poorly, leading to highly unbalanced partitions. In this situation, each recursive call may only reduce the size of one partition by one element, resulting in a recursion depth of n. The worst-case time complexity is O(n^2).

Average Case Time Complexity: On average, Quicksort performs well, and the average-case time complexity is O(n log n). The average-case analysis takes into account random or well-distributed data and assumes that the pivot selection is not consistently biased toward the worst-case scenario. Randomized algorithms, such as choosing the pivot randomly, can help achieve better average-case performance.

Space Complexity: The space complexity of the Quicksort implementation is O(n), where 'n' is the length of the input array. This space complexity arises from the recursion depth and the creation of additional lists (less and greater) during each recursive call.