Insertion Sort:

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

Theory:
Insertion sort is a simple sorting algorithm that iterates through an array, comparing and inserting each element into its correct position. It maintains a sorted and an unsorted subarray, gradually expanding the sorted portion. The algorithm is efficient for small datasets and works well when elements are mostly in order. Its time complexity is O(n^2) on average, making it less suitable for large datasets compared to more advanced sorting algorithms like quicksort or mergesort.

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

if __name__ == "__main__":
    my_array = [22, 11, 33, 66, 77]
    print("Original Array:", my_array)
    insertion_sort(my_array)
    print("Sorted Array:", my_array)
```

Output:
```
Original Array: [22, 11, 33, 66, 77]
Sorted Array: [11, 22, 33, 66, 77]
```

Algorithm Analysis:

Worst-case Time Complexity: O(n^2)
The worst-case occurs when the array is in reverse order, and for each element, it needs to be compared and moved to the beginning of the sorted section.

Best-case Time Complexity: O(n)
The best-case occurs when the array is already sorted. In this case, only comparisons are needed, and no element movement is required.

Average-case Time Complexity: O(n^2)
On average, the algorithm performs quadratic time operations as it involves nested loops. However, it tends to perform better than the worst-case scenario for partially sorted arrays.

Space Complexity: O(1)
Insertion sort operates in-place, using a constant amount of additional space regardless of the input size. It only requires a small amount of extra memory for variables like key, i, and j.

Selection Sort:

Aim:

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

Theory:

Selection Sort is a straightforward sorting algorithm based on comparisons. It partitions the input array into two segments: a sorted segment and an unsorted segment. The algorithm iteratively identifies the smallest (or largest, depending on the desired order) element in the unsorted segment and exchanges it with the first unsorted element. This cycle continues until the entire array is arranged in the desired order.

Code:

```
def selection_sort(arr):
    n = len(arr)

    for i in range(n-1):
        min_index = i
        for j in range(i+1, n):
            if arr[j] < arr[min_index]:
                min_index = j
                arr[i], arr[min_index] = arr[min_index], arr[i]

arr = [66, 22, 11, 33, 77]
selection_sort(arr)
print("Sorted array:", arr)
```

Output:

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

Algorithm Analysis:

Worst-Case Time Complexity: the time complexity of Selection Sort is O(n^2), where "n" is the number of elements in the array. The algorithm consists of two nested loops. The outer loop iterates through each element in the array, and for each iteration, the inner loop finds the minimum element in the remaining unsorted part of the array. This results in a quadratic time complexity.

Best-Case Time Complexity: The best-case time complexity is also O(n^2). Even if the array is partially sorted or already sorted, Selection Sort still performs the same number of comparisons and swaps in the worst-case manner.

Average-Case Time Complexity: The average-case time complexity is O(n^2), as the algorithm does not gain any advantage from partially sorted input.

Space Complexity: Selection Sort is an in-place sorting algorithm with a space complexity of O(1). It only requires a constant amount of additional memory for storing variables like loop indices and temporary variables during the swapping process.