**Python: As a scripting language**

**Subject - Unix Operating System**

**Name – Hemant Sharma**

**PRN – 22610001 Class – TYIT**

**Assignment No – 10(h)**

**Title-** Write a python function for merge/quick sort for integer list as parameter to it.

**Objective:**

1. To learn about python as scripting option.

**Theory:**

Sorting algorithms are essential in computer science for organizing data efficiently. Two widely used sorting algorithms are Merge Sort and Quick Sort, both of which are divide-and-conquer algorithms.

1. **Merge Sort**

* It divides the list into two halves recursively until each sub list has one element.
* Then, it merges the sorted sub lists back together in order.
* It has a time complexity of **O (n log n)** in all cases.

1. **Quick Sort**

* It selects a **pivot** element and partitions the list into two parts: elements less than the pivot and elements greater than the pivot.
* It then recursively sorts the two partitions.
* It has an **average-case** time complexity of **O (n log n)** but a worst case of **O(n²)** (when poorly chosen pivots lead to unbalanced partitions).

These algorithms efficiently handle large datasets and are widely used in practical applications.

**Program:**

# Merge Sort Implementation

def merge\_sort(arr):

    if len(arr) <= 1:

        return arr

    mid = len(arr) // 2

    left\_half = merge\_sort(arr[:mid])

    right\_half = merge\_sort(arr[mid:])

    return merge(left\_half, right\_half)

def merge(left, right):

    sorted\_list = []

    i = j = 0

    while i < len(left) and j < len(right):

        if left[i] < right[j]:

            sorted\_list.append(left[i])

            i += 1

        else:

            sorted\_list.append(right[j])

            j += 1

    sorted\_list.extend(left[i:])

    sorted\_list.extend(right[j:])

    return sorted\_list

# Quick Sort Implementation

def quick\_sort(arr):

    if len(arr) <= 1:

        return arr

    pivot = arr[-1]

    left = [x for x in arr[:-1] if x <= pivot]

    right = [x for x in arr[:-1] if x > pivot]

    return quick\_sort(left) + [pivot] + quick\_sort(right)

# Example usage

if \_\_name\_\_ == "\_\_main\_\_":

    arr = [100, 32, 223, 352, 5, 652]

    # Using Merge Sort

    sorted\_arr\_merge = merge\_sort(arr)

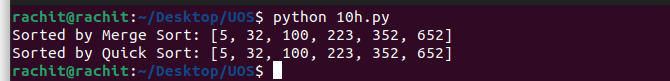
    print("Sorted by Merge Sort:", sorted\_arr\_merge)

    # Using Quick Sort

    sorted\_arr\_quick = quick\_sort(arr)

    print("Sorted by Quick Sort:", sorted\_arr\_quick)

**Output:**

****

**Conclusion:**

Both **Merge Sort** and **Quick Sort** provide efficient ways to sort an integer list using a divide-and-conquer approach.

* **Merge Sort** is stable and guarantees **O(n log n)** time complexity but uses extra space for merging.
* **Quick Sort** is often faster in practice due to in-place partitioning, but in the worst case, it can degrade to **O(n²)** if not optimized with a good pivot selection.

These sorting techniques are fundamental in data structures and algorithms, playing a crucial role in efficient data processing.