LAB SESSION 12

Sorting Algorithms – Quick Sort & Merge Sort

THEORY

Sorting is the process of arranging elements in a list into a specific order, most commonly ascending or descending. It is a fundamental operation in computer science, enabling faster searches, easier data processing, and improved efficiency in algorithms that depend on ordered data.
Two highly efficient sorting algorithms are **Merge Sort** and **Quick Sort**, both of which use the **divide-and-conquer** strategy.
Merge Sort works by repeatedly splitting the array into two halves until each subarray has only one element, then merging these sorted subarrays to produce a completely sorted array. Its time complexity remains **O(n log n)** in all cases, making it predictable. However, it requires additional memory (O(n)) for the merging process, which can be a limitation for very large datasets. Merge Sort is stable — equal elements maintain their original order — and is well-suited for linked list sorting or external sorting (data stored outside of main memory).
Quick Sort operates by choosing a pivot element, partitioning the array so that elements less than the pivot appear before it and elements greater than the pivot come after it, and then recursively applying the process to each partition. Quick Sort has an average time complexity of **O(n log n)** but can degrade to **O(n^2)** in the worst case if the pivot selection is poor. By using techniques like random pivot selection or median-of-three, the worst case can be avoided in practice. Quick Sort typically sorts in-place and requires less additional memory compared to Merge Sort.
Comparison: Merge Sort is preferred when stability is important or when dealing with data that doesn't fit entirely in memory. Quick Sort is often faster in practice for in-memory arrays due to better cache performance and low overhead.

PROCEDURE

1. Define functions to perform Merge Sort and Quick Sort.
2. Input a dataset from the user or read from a file.
3. Apply both algorithms to identical datasets.
4. Display the sorted outputs for each algorithm.
5. Measure and record the execution time for each sorting method.
6. Compare results and discuss the findings.

CODE

```
#include <bits/stdc++.h>
using namespace std;

// Merge Sort
void merge(vector<int>& a, int l, int m, int r) {
   int n1 = m - l + 1, n2 = r - m;
   vector<int> L(n1), R(n2);
   for (int i=0; i<n1; i++) L[i] = a[l+i];
   for (int j=0; j<n2; j++) R[j] = a[m+1+j];
   int i=0, j=0, k=l;
```

```
    while (i<n1 && j<n2) {
        if (L[i] <= R[j]) a[k++] = L[i++];
        else a[k++] = R[j++];
    }
    while (i<n1) a[k++] = L[i++];
    while (j<n2) a[k++] = R[j++];
}

void mergeSort(vector<int>& a, int l, int r) {
    if (l >= r) return;
    int m = l + (r-l)/2;
    mergeSort(a, l, m);
    mergeSort(a, m+1, r);
    merge(a, l, m, r);
}

// Quick Sort
int partition(vector<int>& a, int l, int r) {
    int pivot = a[r];
    int i = l - 1;
    for (int j = l; j <= r-1; ++j) {
        if (a[j] <= pivot) {
            ++i; swap(a[i], a[j]);
        }
    }
    swap(a[i+1], a[r]);
    return (i+1);
}

void quickSort(vector<int>& a, int l, int r) {
    if (l < r) {
        int pi = partition(a, l, r);
        quickSort(a, l, pi-1);
        quickSort(a, pi+1, r);
    }
}

void printVec(const vector<int>& a) {
    for (int x : a) cout << x << " ";
    cout << "\n";
}

int main() {
    vector<int> arr = {38, 27, 43, 3, 9, 82, 10};

    cout << "Original: "; printVec(arr);

    auto a1 = arr;
    mergeSort(a1, 0, (int)a1.size()-1);
    cout << "Merge Sort: "; printVec(a1);
    auto a2 = arr;
    quickSort(a2, 0, (int)a2.size()-1);
    cout << "Quick Sort: "; printVec(a2);

    return 0;
}
```

EXPECTED OUTPUT

```
Original: 38 27 43 3 9 82 10
Merge Sort: 3 9 10 27 38 43 82
Quick Sort: 3 9 10 27 38 43 82
```

EXERCISE

1. Implement a randomized Quick Sort and compare its performance with the standard Quick Sort.
2. Modify Merge Sort to work with linked lists.
3. Identify which algorithm performs better for nearly sorted data and explain why.