**LAB Assignment Submission**

for

**Data Structures and Algorithms**

**Course Code: CSE2711**

**B.Tech CSE-VII/ECOM**

**Batch 2024**

**Name:** Vedansh Mathur

**Enrollment Number:** 240686

**Problem Statement-**

Problem 1: Merge Sort

i. Write a program to implement Merge Sort step by step (data stored in an 1D array).

o Show how the array is recursively divided into subarrays.

o Print the intermediate states of the array at each merge step.

ii. Demonstrate the recurrence relation of Merge Sort T(n) = 2T(n/2) + O(n) by counting

the number of recursive calls.

iii. Test your program on arrays of different sizes (n = 8, 16, 32, 64).

Learning Outcome:

> How divide-and-conquer breaks a big problem into smaller subproblems.

> How merging gives the sorted output.

> Time complexity analysis using recurrence.

Problem 2: Quick Sort

i. Implement Quick Sort with three pivot strategies (data stored in an 1D array):

o First element as pivot.

o Last element as pivot.

o Median-of-three as pivot.

ii. For each pivot strategy:

o Show the partitioning steps (print subarrays at each recursive stage).

o Count the number of comparisons and swaps.

iii. Compare the performance of the three strategies on:

o A sorted array (best/worst case).

o A reverse-sorted array.

o A random array.

Learning Outcome:

> Role of pivot selection in divide-and-conquer efβiciency.

> Why Quick Sort’s best/average case is O(n log n) but worst case is O(n2).

> Practical impact of input data distribution.

Problem 3: Merge Sort vs. Quick Sort: A Comparative Study

Apply both algorithms and analyze efβiciency using divide-and-conquer principles.

i. Implement both Merge Sort and Quick Sort in the same program.

ii. Generate test arrays of increasing size (n = 1000, 3000, 5000, 8000).

iii. Measure and compare:

o Execution time.

o Number of recursive calls.

o Memory usage (for Merge Sort vs. Quick Sort).

iv. Write a short report answering:

o Which algorithm uses more memory and why?

o Which algorithm performs better on random vs. nearly-sorted data?

o How divide-and-conquer ensures scalability of these algorithms.

Learning Outcome:

 Trade-offs between the two algorithms.

 Deep understanding of divide-and-conquer efβiciency.

 Realization that practical performance depends on both algorithm design and data

characteristics.

**Solution –**

**A1->**

#include <iostream>

using namespace std;

int callCount = 0;   // To count recursive calls

// Function to print array

void printArray(int arr[], int start, int end) {

    cout << "[ ";

    for (int i = start; i <= end; i++) cout << arr[i] << " ";

    cout << "]";

}

// Merge two subarrays

void merge(int arr[], int l, int m, int r) {

    int n1 = m - l + 1;

    int n2 = r - m;

    int L[n1], R[n2];

    for (int i = 0; i < n1; i++) L[i] = arr[l + i];

    for (int j = 0; j < n2; j++) R[j] = arr[m + 1 + j];

    int i = 0, j = 0, k = l;

    while (i < n1 && j < n2) {

        if (L[i] <= R[j]) arr[k++] = L[i++];

        else arr[k++] = R[j++];

    }

    while (i < n1) arr[k++] = L[i++];

    while (j < n2) arr[k++] = R[j++];

    cout << "Merging: ";

    printArray(arr, l, r);

    cout << endl;

}

// Recursive Merge Sort

void mergeSort(int arr[], int l, int r) {

    callCount++;  // Counting recursive calls

    if (l < r) {

        int m = (l + r) / 2;

        cout << "Divide: ";

        printArray(arr, l, r);

        cout << " -> Left: ";

        printArray(arr, l, m);

        cout << " | Right: ";

        printArray(arr, m + 1, r);

        cout << endl;

        mergeSort(arr, l, m);

        mergeSort(arr, m + 1, r);

        merge(arr, l, m, r);

    }

}

int main() {

    int n;

    cout << "Enter size of array (n=8,16,32,64): ";

    cin >> n;

    int arr[n];

    cout << "Enter " << n << " elements: ";

    for (int i = 0; i < n; i++) cin >> arr[i];

    cout << "\nOriginal Array: ";

    for (int i = 0; i < n; i++) cout << arr[i] << " ";

    cout << endl << endl;

    mergeSort(arr, 0, n - 1);

    cout << "\nFinal Sorted Array: ";

    for (int i = 0; i < n; i++) cout << arr[i] << " ";

    cout << endl;

    cout << "\nTotal Recursive Calls (T(n)): " << callCount << endl;

    return 0;

}

A2->

#include <iostream>

using namespace std;

int comp = 0, swp = 0;

void swap(int &a, int &b) {

    int t = a;

    a = b;

    b = t;

    swp++;

}

void print(int arr[], int n) {

    for (int i = 0; i < n; i++) cout << arr[i] << " ";

    cout << endl;

}

int choosePivot(int arr[], int l, int h, int type) {

    if (type == 1) return l;          // first element

    if (type == 2) return h;          // last element

    int m = (l + h) / 2;              // median of three

    if ((arr[l] <= arr[m] && arr[m] <= arr[h]) || (arr[h] <= arr[m] && arr[m] <= arr[l])) return m;

    if ((arr[m] <= arr[l] && arr[l] <= arr[h]) || (arr[h] <= arr[l] && arr[l] <= arr[m])) return l;

    return h;

}

int partition(int arr[], int l, int h, int type) {

    int p = choosePivot(arr, l, h, type);

    swap(arr[l], arr[p]);   // pivot ko left me lao

    int pivot = arr[l];

    int i = l + 1, j = h;

    while (true) {

        while (i <= h && arr[i] <= pivot) { i++; comp++; }

        while (j >= l+1 && arr[j] > pivot) { j--; comp++; }

        if (i >= j) break;

        swap(arr[i], arr[j]);

    }

    swap(arr[l], arr[j]);

    print(arr, h+1);

    return j;

}

void quickSort(int arr[], int l, int h, int type) {

    if (l < h) {

        int p = partition(arr, l, h, type);

        quickSort(arr, l, p - 1, type);

        quickSort(arr, p + 1, h, type);

    }

}

int main() {

    int arr[] = {29, 10, 14, 37, 13};

    int n = 5;

    cout << "Original array: ";

    print(arr, n);

    for (int type = 1; type <= 3; type++) {

        int temp[5];

        for (int i = 0; i < n; i++) temp[i] = arr[i];

        comp = swp = 0;

        cout << "\nQuick Sort using ";

        if (type == 1) cout << "First element as pivot\n";

        else if (type == 2) cout << "Last element as pivot\n";

        else cout << "Median-of-three as pivot\n";

        quickSort(temp, 0, n - 1, type);

        cout << "Sorted array: ";

        print(temp, n);

        cout << "Comparisons = " << comp << ", Swaps = " << swp << "\n";

    }

    return 0;

}

A3->

#include <iostream>

#include <vector>

#include <chrono>

using namespace std;

using namespace std::chrono;

// Global counters for recursive calls

int mergeCalls = 0;

int quickCalls = 0;

// ---------------- Merge Sort ----------------

void merge(int arr[], int l, int m, int r) {

    int n1 = m - l + 1;

    int n2 = r - m;

    int L[n1], R[n2]; // extra memory used

    for (int i = 0; i < n1; i++) L[i] = arr[l+i];

    for (int i = 0; i < n2; i++) R[i] = arr[m+1+i];

    int i=0, j=0, k=l;

    while(i<n1 && j<n2){

        if(L[i]<=R[j]) arr[k++] = L[i++];

        else arr[k++] = R[j++];

    }

    while(i<n1) arr[k++] = L[i++];

    while(j<n2) arr[k++] = R[j++];

}

void mergeSort(int arr[], int l, int r) {

    mergeCalls++;

    if(l<r){

        int m = (l+r)/2;

        mergeSort(arr,l,m);

        mergeSort(arr,m+1,r);

        merge(arr,l,m,r);

    }

}

// ---------------- Quick Sort ----------------

int partition(int arr[], int l, int h){

    int pivot = arr[h]; // last element pivot

    int i = l-1;

    for(int j=l;j<h;j++){

        if(arr[j]<=pivot){

            i++;

            swap(arr[i], arr[j]);

        }

    }

    swap(arr[i+1], arr[h]);

    return i+1;

}

void quickSort(int arr[], int l, int h){

    quickCalls++;

    if(l<h){

        int pi = partition(arr,l,h);

        quickSort(arr,l,pi-1);

        quickSort(arr,pi+1,h);

    }

}

// ---------------- Helper Function ----------------

void generateArray(int arr[], int n, int type){

    // type=0: random, type=1: sorted, type=2: reverse

    for(int i=0;i<n;i++){

        if(type==0) arr[i] = rand()%10000;

        else if(type==1) arr[i] = i+1;

        else arr[i] = n-i;

    }

}

int main(){

    vector<int> sizes = {1000,3000,5000,8000};

    srand(time(0));

    for(int n : sizes){

        int arr1[n], arr2[n];

        cout << "\nArray size = " << n << "\n";

        generateArray(arr1,n,0); // random

        for(int i=0;i<n;i++) arr2[i]=arr1[i]; // same array for Quick Sort

        // Merge Sort

        mergeCalls=0;

        auto start = high\_resolution\_clock::now();

        mergeSort(arr1,0,n-1);

        auto end = high\_resolution\_clock::now();

        auto duration = duration\_cast<milliseconds>(end-start).count();

        cout << "Merge Sort -> Time: " << duration << " ms, Recursive Calls: " << mergeCalls << endl;

        // Quick Sort

        quickCalls=0;

        start = high\_resolution\_clock::now();

        quickSort(arr2,0,n-1);

        end = high\_resolution\_clock::now();

        duration = duration\_cast<milliseconds>(end-start).count();

        cout << "Quick Sort -> Time: " << duration << " ms, Recursive Calls: " << quickCalls << endl;

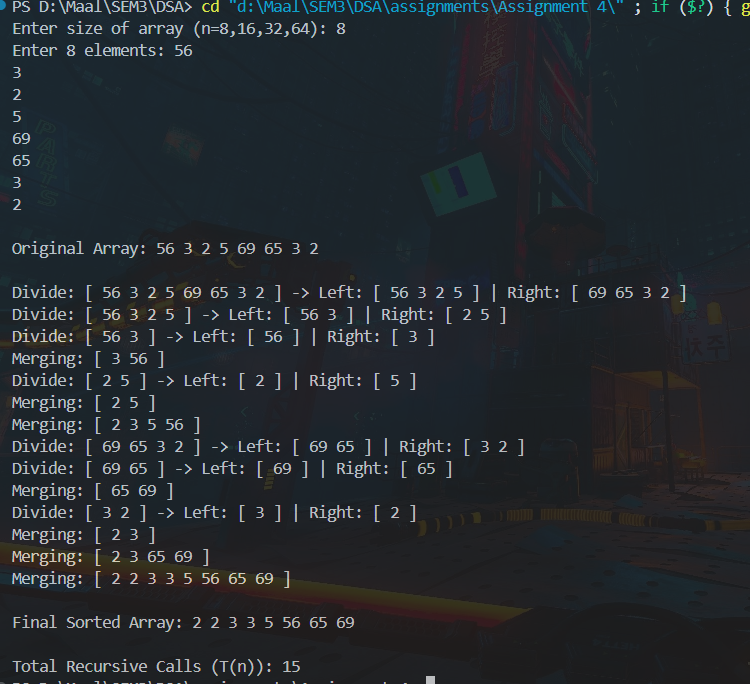
    }

    return 0;

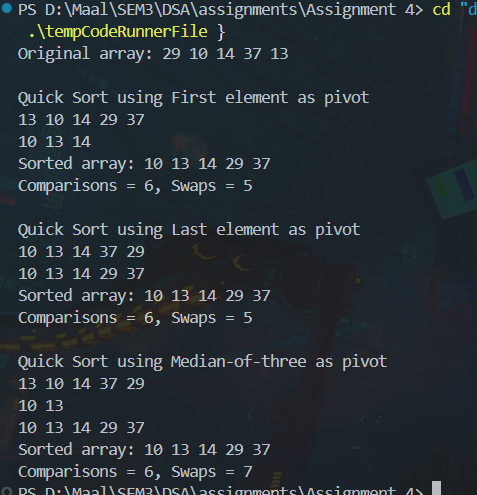
}

**Output –**

O/P 1->

****

O/P 2->

****

O/P 3->

