|  |  |  |  |
| --- | --- | --- | --- |
| **Academic Year** | **2025 - 26** | **Experiment No.** | **2** |
| **Course & Semester** | **S.E. – Sem. III** | **Subject Name** | **Analysis of Algorithm** |
| **Experiment Type** | **Software Performance** | **Subject Code** | **25PCC12CS05** |

|  |  |  |  |
| --- | --- | --- | --- |
| **Name of Student:** | Atharva Dharmendra Jagtap | **Roll No.:** | 10937 |
| **Date of Performance:** |  | **Date of Submission:** |  |
| **LO Mapping** | 25PCC12CS05.1: Analyze the time and space complexity of algorithms.  25PCC12CS05.2: Apply divide and conquer strategy to solve a problem. | | |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  | | --- | --- | --- | --- | | **Indicator** | **Poor** | **Average** | **Good** | | Timeline Maintains submission deadline (3) | Submission not done (0) | One or More than One week late (1-2) | Maintains deadline (3) | | Completion and Organization (3) | N/A | Document is just acceptable (1-2) | Completed whole document and neatly organized (3) | | Program Performance (2) | Could not perform at all (0) | Implemented few parts (1) | Full implementation (2) | | Knowledge In depth knowledge of the Experiment (2) | Unable to answer questions (0) | Unable to answer few questions (1) | Able to answer all questions (2) | |
| **Assessment Marks:**   |  |  | | --- | --- | | Timeline |  | | Completion and Organization |  | | Program Performance |  | | Knowledge |  | |
| Total: (Out of 10) |
| Teacher’s Sign: Student Sign: |

**Experiment No. 2**

**AIM:** To Implement and analyze time and space complexity of Quick and Merge sort to display records of an employee working in any organization based on their work experience.

**THEORY:** Sorting is a crucial operation in data processing, enabling faster search, organization, and presentation of data. In this experiment, we use two efficient divide-and-conquer sorting algorithms—Quick Sort and Merge Sort—to sort and display employee records in descending order based on their work experience (typically measured in years).

Each employee record consists of information such as name, employee ID, and years of experience. Sorting these records helps organizations identify the most experienced employees for promotions, project leadership, or awards.

Quick Sort:

Quick Sort is a divide-and-conquer algorithm that works by selecting a pivot element and partitioning the array into two sub-arrays—one with elements greater than the pivot and one with elements less. It then recursively sorts the sub-arrays.

* Best and Average Case Time Complexity: O(n log n)
* Worst Case Time Complexity: O(n²) (occurs when pivot is poorly chosen)
* Space Complexity: O(log n) (due to recursive calls, in-place sorting)

Merge Sort:

Merge Sort is also a divide-and-conquer algorithm. It divides the array into two halves, recursively sorts each half, and then merges the sorted halves.

* Time Complexity (All Cases): O(n log n)
* Space Complexity: O(n) (requires additional space for merging)

Relevance to Employee Records:

In this context, sorting employees by experience helps in better human resource management. By implementing Quick and Merge Sort, we can analyze their efficiency and suitability for sorting large datasets such as organizational employee databases.

Understanding the time and space trade-offs of each algorithm also helps in selecting the most appropriate one based on application constraints like memory usage and expected data size.

**ALGORITHM:**

**Step 1: Start**

**Step 2: Define a structure**

* Create a structure named Employee with:
  + name (string)
  + experience (integer)

**Step 3: Input the number of employees**

* Read the total number of employees n.

**Step 4: Input employee details**

* For each employee from 1 to n:
  + Read the employee's name
  + Read the employee's experience in years
  + Store the data in an array employees[]

**Step 5: Copy original data**

* Copy all records from employees[] to another array temp[] for sorting (to preserve the original data).

**Step 6: Display sorting options**

* Show the user the sorting methods:
  + 1 → Quick Sort
  + 2 → Merge Sort

**Step 7: Get user's choice**

* Read the user's choice.

**Step 8: Sort the data**

* If choice is 1, apply **Quick Sort** on temp[] (in descending order based on experience)
* If choice is 2, apply **Merge Sort** on temp[] (also in descending order)
* If invalid choice, display an error and exit.

**Step 9: Display the sorted employee records**

* Print the employee records in descending order of experience from temp[].

**Step 10: End**

**CODE:**

#include <stdio.h>

struct Employee {

    char name[50];

    int experience;

};

void quickSort(struct Employee arr[], int low, int high) {

    if (low < high) {

        int pivot = arr[high].experience, i = low - 1;

        for (int j = low; j < high; j++) {

            if (arr[j].experience >= pivot) {

                i++;

                struct Employee temp = arr[i]; arr[i] = arr[j]; arr[j] = temp;

            }

        }

        struct Employee temp = arr[i + 1]; arr[i + 1] = arr[high]; arr[high] = temp;

        int pi = i + 1;

        quickSort(arr, low, pi - 1);

        quickSort(arr, pi + 1, high);

    }

}

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

    int n1 = m - l + 1, n2 = r - m;

    struct Employee 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].experience >= R[j].experience) 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(struct Employee arr[], int l, int r) {

    if (l < r) {

        int m = l + (r - l) / 2;

        mergeSort(arr, l, m);

        mergeSort(arr, m + 1, r);

        merge(arr, l, m, r);

    }

}

int main() {

    int n, choice;

    printf("Enter number of employees: ");

    scanf("%d", &n);

    if (n <= 0) return 1;

    struct Employee employees[n], temp[n];

    for (int i = 0; i < n; i++) {

        printf("Enter name of employee %d: ", i + 1);

        scanf("%49s", employees[i].name);

        printf("Enter experience of employee %d: ", i + 1);

        scanf("%d", &employees[i].experience);

    }

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

    printf("\nChoose sorting method:\n");

    printf("1. Quick Sort\n");

    printf("2. Merge Sort\n");

    printf("Enter choice: ");

    scanf("%1d", &choice);

    if (choice == 1) quickSort(temp, 0, n - 1);

    else if (choice == 2) mergeSort(temp, 0, n - 1);

    else {

        printf("Invalid choice.\n");

        return 1;

    }

    printf("\nSorted Employee Records (Descending by Experience):\n");

    printf("-----------------------------------------------------------\n");

    printf("|\tRank\t|\tName\t\t|\tExperience\t|\n");

    printf("-----------------------------------------------------------\n");

    for (int i = 0; i < n; i++) {

        printf("|\t%2d\t\t|\t%-10s\t|\t%3d years\t|\n", i + 1, temp[i].name, temp[i].experience);

    }

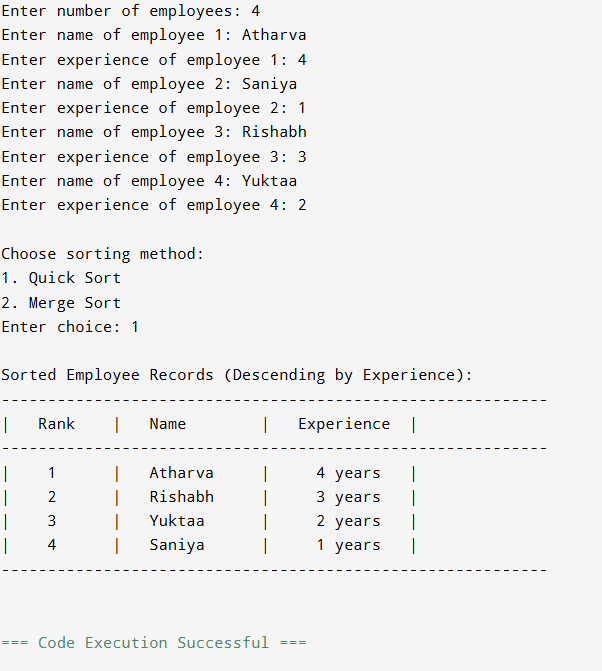
    printf("-----------------------------------------------------------\n");

    return 0;

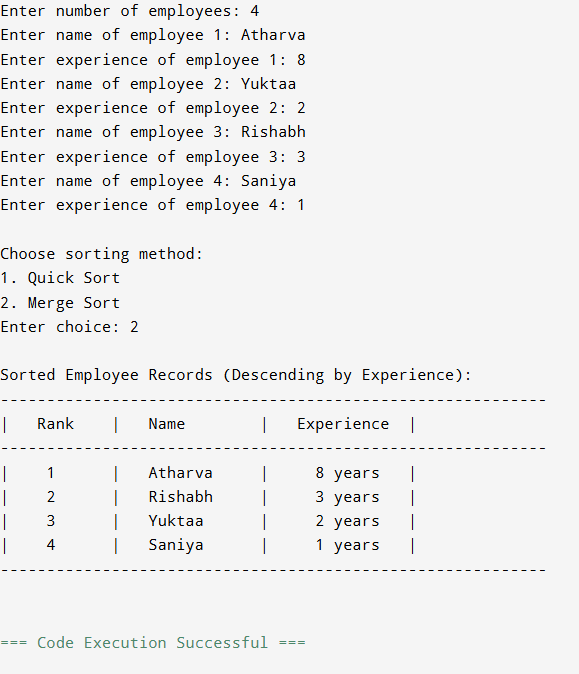
}

**OUTPUT:**

1. Quick Sort

****

1. Merge Sort

****

**POST LAB QUESTIONS**

1. How do Quick Sort and Merge Sort differ in their approach to divide-and-conquer? Where do you see these differences reflected in your implementation?

**Ans.** Quick Sort partitions the array around a pivot; Merge Sort splits the array exactly in half.Quick Sort recursively sorts subarrays in place; Merge Sort recursively sorts and then merges. Quick Sort sorts in-place, Merge Sort uses extra space for merging. Quick Sort shows pivot-based swaps; Merge Sort explicitly creates L[] and R[] arrays for merging.

1. If the employee data is nearly sorted by work experience, which algorithm shows better runtime? Justify with theoretical reasoning.

**Ans.** Quick Sort performs close to O(n) if a good pivot strategy is used (like median or randomized pivot). On the other hand, Merge Sort always O(n log n) regardless of input order. Quick Sort leverages existing order, reducing swaps and comparisons. So Quick Sort shows better runtime as it outperforms Merge Sort for nearly sorted data.

1. Why does Quick Sort sometimes take longer to run than Merge Sort?

**Ans.** Quick Sort sometimes take longer to run than Merge Sort because of the following reasons:

1. Bad Pivot: Poor pivot choices (like always picking last element in sorted data) cause O(n²) complexity.
2. Extra Comparisons: Unbalanced partitions lead to deeper recursion and more comparisons.
3. Cache Behavior: Merge Sort’s sequential access benefits CPU cache; Quick Sort may scatter accesses.
4. Stability: Merge Sort is stable; Quick Sort may do extra work when handling duplicate values.

**CONCLUSION:**

Quick Sort and Merge Sort both utilize divide-and-conquer, but they differ in performance and execution. Quick Sort partitions the array around a pivot, risking a worst-case complexity of O(n²) with poor pivot choices. Merge Sort, on the other hand, consistently splits and merges subarrays, ensuring a stable O(n log n) runtime. While Quick Sort often runs faster on average, Merge Sort offers more predictable and reliable performance, especially on datasets that could cause inefficiencies in Quick Sort, such as nearly sorted data.