Experiment 14

Aim:

To implement a menu-driven program that allows the user to choose between Merge Sort and Quick Sort for sorting an array.

Theory:

- **Merge Sort:**
  Merge Sort is a divide-and-conquer algorithm. It divides the array into two halves, recursively sorts each half, and then merges the sorted halves to produce the final sorted array.
    - **Time Complexity:** O(n log n)
    - **Space Complexity:** O(n)
- **Quick Sort:**
  Quick Sort is also a divide-and-conquer algorithm. It picks an element as a pivot and partitions the array into two halves such that all elements smaller than the pivot are on the left and all elements greater than the pivot are on the right. It then recursively sorts the partitions.
    - **Time Complexity:** O(n log n) on average, $O(n^2)$ in the worst case
    - **Space Complexity:** O(log n) due to recursion

Algorithm Steps:

1. **Merge Sort:**
    - Divide the array into two halves.
    - Recursively sort each half.
    - Merge the two sorted halves.
2. **Quick Sort:**
    - Choose a pivot element.
    - Partition the array around the pivot such that all elements less than the pivot are on the left, and all elements greater than the pivot are on the right.
    - Recursively sort the left and right partitions.

C Code for Menu-driven Merge Sort and Quick Sort

```c
#include <stdio.h>
```

```
// Function to merge two halves for merge sort
void merge(int arr[], int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    int L[n1], R[n2];

    // Copy data to temporary arrays L[] and R[]
    for (int i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (int i = 0; i < n2; i++)
        R[i] = arr[mid + 1 + i];

    // Merge the temporary arrays back into arr[left..right]
    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    // Copy the remaining elements of L[], if any
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    // Copy the remaining elements of R[], if any
    while (j < n2) {
        arr[k] = R[j];
        j++;
```

```
        k++;
    }
}

// Function for merge sort
void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;

        // Recursively sort first and second halves
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);

        // Merge the sorted halves
        merge(arr, left, mid, right);
    }
}

// Function to partition the array for quick sort
int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = low - 1;

    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            i++;
            // Swap arr[i] and arr[j]
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }

    // Swap arr[i + 1] and arr[high] (or pivot)
    int temp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = temp;
```

```
    return i + 1;
}

// Function for quick sort
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        // Partition the array
        int pi = partition(arr, low, high);

        // Recursively sort the partitions
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

// Function to print the array
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

// Driver code for menu-driven sorting
int main() {
    int n, choice;

    printf("Enter the number of elements in the array: ");
    scanf("%d", &n);

    int arr[n];
    printf("Enter the elements of the array:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    while (1) {
        printf("\nSorting Menu:\n");
```

```
        printf("1. Merge Sort\n");
        printf("2. Quick Sort\n");
        printf("3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        // Create a copy of the original array to avoid modifying it
after sorting
        int tempArr[n];
        for (int i = 0; i < n; i++) {
            tempArr[i] = arr[i];
        }

        switch (choice) {
            case 1:
                mergeSort(tempArr, 0, n - 1);
                printf("Array after Merge Sort:\n");
                printArray(tempArr, n);
                break;
            case 2:
                quickSort(tempArr, 0, n - 1);
                printf("Array after Quick Sort:\n");
                printArray(tempArr, n);
                break;
            case 3:
                printf("Exiting...\n");
                return 0;
            default:
                printf("Invalid choice! Please try again.\n");
        }
    }

    return 0;
}
```

Explanation of Code:

1. **Merge Sort:**

    - The `mergeSort()` function recursively divides the array into halves and merges them using the `merge()` function.
2. **Quick Sort:**
    - The `quickSort()` function partitions the array using the `partition()` function and recursively sorts the partitions.
3. **Menu:**
    - The program provides a menu where the user can choose between Merge Sort and Quick Sort. The user can input their choice, and the program will apply the selected sorting algorithm and display the sorted array.

Sample Input and Output:

```
Enter the number of elements in the array: 5
Enter the elements of the array:
64 25 12 22 11

Sorting Menu:
1. Merge Sort
2. Quick Sort
3. Exit
Enter your choice: 1
Array after Merge Sort:
11 12 22 25 64

Sorting Menu:
1. Merge Sort
2. Quick Sort
3. Exit
Enter your choice: 2
Array after Quick Sort:
11 12 22 25 64

Sorting Menu:
1. Merge Sort
2. Quick Sort
3. Exit
Enter your choice: 3
Exiting...
```

Conclusion:

In this practical, a menu-driven program was successfully implemented to sort an array using Merge Sort and Quick Sort. Both algorithms are efficient with a time complexity of O(n log n), making them suitable for large datasets. The program allows the user to choose the sorting method and interactively sorts the array based on the user's selection.