Data Structures : Algorithms and Applications(Lab)

EXPERIMENT No: 14

AIM: Implementation of Menu driven Merge Sort and Quick Sort

THEORY:

Merge Sort-

Merge Sort is a divide-and-conquer sorting algorithm that works by recursively dividing an array into two halves, sorting each half, and then merging the sorted halves back together. The main steps are:

1. Divide: Split the unsorted array into two approximately equal halves.
2. Conquer: Recursively sort the two halves.
3. Combine: Merge the sorted halves back into one sorted array.

Complexity:

- Time Complexity: O(n log n) in all cases (best, average, and worst) because the array is always divided in half.
- Space Complexity: O(n) due to the temporary arrays used for merging.

Quick Sort-

Quick Sort is another divide-and-conquer algorithm that selects a 'pivot' element from the array and partitions the other elements into two sub-arrays according to whether they are less than or greater than the pivot. The steps are:

1. Partition: Rearrange the array so that all elements less than the pivot come before it and all elements greater come after it. The pivot is now in its final position.
2. Recursively sort the sub-arrays.

Complexity:

- Time Complexity:
    - Average and best case: O(n log n)
    - Worst case: $O(n^2)$ (occurs when the smallest or largest element is always chosen as the pivot).
- Space Complexity: O(log n) for the recursive stack space in the average case.

CODE:

```
#include <stdio.h>

// Function prototypes
void merge(int arr[], int left, int mid, int right);
void mergeSort(int arr[], int left, int right);
int partition(int arr[], int low, int high);
void quickSort(int arr[], int low, int high);
void display(int arr[], int n);
void swap(int* a, int* b);

int main() {
   int n, choice;
   printf("Enter the number of elements: ");
   scanf("%d", &n);

   int arr[n];
   for (int i = 0; i < n; i++) {
      printf("Enter element %d: ", i + 1);
      scanf("%d", &arr[i]);
   }

   while (1) {
      printf("\n1. Merge Sort\n2. Quick Sort\n3. Display Array\n4. Exit\n");
      printf("Enter your choice: ");
      scanf("%d", &choice);

      switch (choice) {
         case 1:
            mergeSort(arr, 0, n - 1);
            printf("Sorted using Merge Sort.\n");
            display(arr, n);  // Display sorted array
            break;
         case 2:
            quickSort(arr, 0, n - 1);
            printf("Sorted using Quick Sort.\n");
            display(arr, n);  // Display sorted array
            break;
         case 3:
            display(arr, n);
            break;
         case 4:
            return 0;
```

```
            default:
                printf("Invalid choice. Select 1-4.\n");
        }
    }
}

void merge(int arr[], int left, int mid, int right) {
    int sizeFirst = mid - left + 1, sizeSecond = right - mid;
    int firstArr[sizeFirst], secondArr[sizeSecond];

    for (int i = 0; i < sizeFirst; i++) firstArr[i] = arr[left + i];
    for (int j = 0; j < sizeSecond; j++) secondArr[j] = arr[mid + 1 + j];

    for (int i = 0, j = 0, k = left; k <= right; k++) {
        if (i < sizeFirst && (j >= sizeSecond || firstArr[i] <= secondArr[j])) {
            arr[k] = firstArr[i++];
        } else {
            arr[k] = secondArr[j++];
        }
    }
}

void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}

int partition(int arr[], int low, int high) {
    int pivot = arr[high], i = low - 1;
    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            swap(&arr[++i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return i + 1;
}

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

```
    if (low < high) {
        int pivot = partition(arr, low, high);
        quickSort(arr, low, pivot - 1);
        quickSort(arr, pivot + 1, high);
    }
}

void display(int arr[], int n) {
    printf("Array: ");
    for (int i = 0; i < n; i++) printf("%d ", arr[i]);
    printf("\n");
}

void swap(int* a, int* b) {
    int temp = *a; *a = *b; *b = temp;
}
```

OUTPUT:

```
Enter the number of elements: 4
Enter element 1: 2
Enter element 2: 3
Enter element 3: 4
Enter element 4: 6

1. Merge Sort
2. Quick Sort
3. Display Array
4. Exit
Enter your choice: 1
Sorted using Merge Sort.
Array: 2 3 4 6

1. Merge Sort
2. Quick Sort
3. Display Array
4. Exit
Enter your choice: 2
Sorted using Quick Sort.
Array: 2 3 4 6

1. Merge Sort
2. Quick Sort
3. Display Array
4. Exit
Enter your choice: 3
Array: 2 3 4 6

1. Merge Sort
2. Quick Sort
3. Display Array
4. Exit
Enter your choice: 4


...Program finished with exit code 0
Press ENTER to exit console.
```

CONCLUSION:

In conclusion, both Merge Sort and Quick Sort are efficient sorting algorithms with distinct advantages: Merge Sort guarantees O(n log n) performance and stability, while Quick Sort is generally faster in practice with in-place sorting. The choice between them depends on the specific needs of the application, including dataset size and memory constraints.