DESIGN AND ANALYSIS OF ALGORITHMS

LAB WORKBOOK WEEK-4

NAME : M. Charitha Varshini

ROLL.NO: CH.SC.U4CSE24128

DEPARTMENT: CSE-B

MergeSort:

Code:

```c
#include <stdio.h>
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 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];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
```

```c
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

void mergeSort(int 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);
    }
}

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

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

```c
    int arr[n];

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

    printf("Given array: \n");
    printArray(arr, n);

    mergeSort(arr, 0, n - 1);

    printf("\nSorted array: \n");
    printArray(arr, n);

    return 0;
}
```

Output:

```
charitha@ubuntu-charitha:~$ nano mergesort.c
charitha@ubuntu-charitha:~$ gcc -o mergesort mergesort.c
charitha@ubuntu-charitha:~$ ./mergesort
Enter the number of elements: 12
Enter the elements of the array:
157   110 147 122 111 149 151 141 123 112 117 133
Given array:
157 110 147 122 111 149 151 141 123 112 117 133

Sorted array:
110 111 112 117 122 123 133 141 147 149 151 157
charitha@ubuntu-charitha:~$
```

Time complexity:

The array is repeatedly divided into two halves.

The division continues until single elements remain.

At each level, the merge function compares and combines all n elements.

The number of division levels is log n.

Total number of operations = n * log n.

Time Complexity = O(n log n)

Space Complexity:

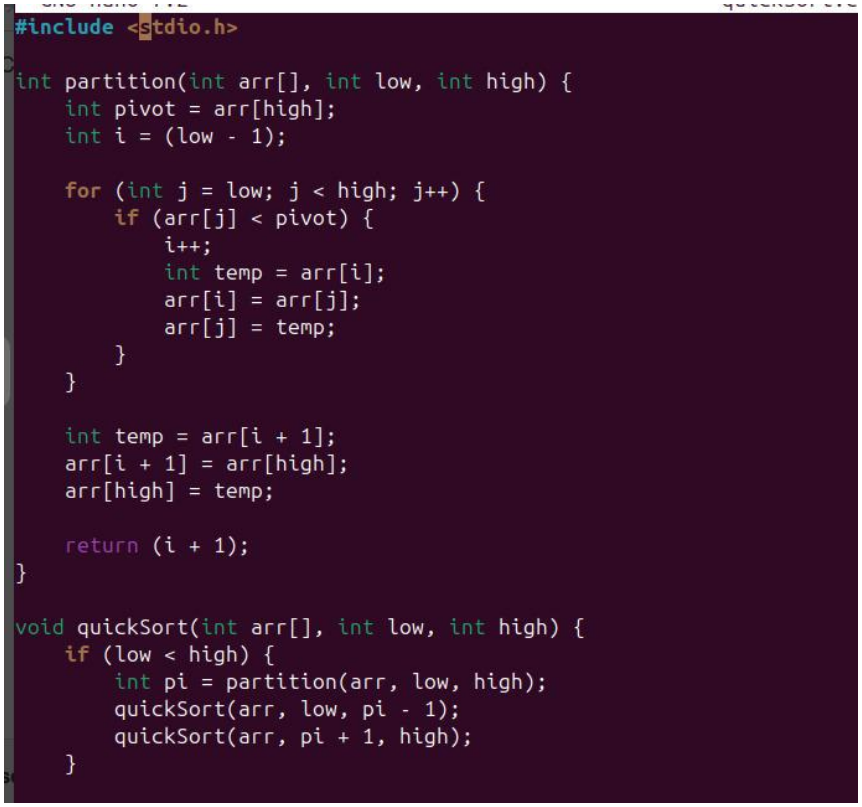
int l, m, r, i, j, k -- 6 × 4 = 24 bytes

Temporary arrays L[] and R[] -- n elements

Space Complexity = O(n)

QuickSort:

Code:



```c
#include <stdio.h>

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++;
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }

    int temp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = temp;

    return (i + 1);
}

void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
```

```c
}
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}
int main() {
    int n;
    printf("Enter the number of elements: ");
    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]);
    }
    printf("Given array: \n");
    printArray(arr, n);

    quickSort(arr, 0, n - 1);
    printf("\nSorted array: \n");
    printArray(arr, n);
    return 0;
}
```

Output:

```
charitha@ubuntu-charitha:~$ nano quicksort.c
charitha@ubuntu-charitha:~$ gcc -o quicksort quicksort.c
charitha@ubuntu-charitha:~$ ./quicksort
Enter the number of elements: 12
Enter the elements of the array:
157 110 147 122 111 149 151 141 123 112 117 133
Given array:
157 110 147 122 111 149 151 141 123 112 117 133

Sorted array:
110 111 112 117 122 123 133 141 147 149 151 157
charitha@ubuntu-charitha:~$
```

Time Complexity:

Pivot element is selected

All the elements are compared with the pivot element

The partition loop runs n times

recursion depth is log n.

Total number of operations = n * log n.

Average & Best Case Time Complexity = O(n log n)

Worst Case:

If the pivot is always the smallest or largest element,

recursion depth becomes n.

Total comparisons =n^2.

Worst Case Time Complexity = O(n^2)

Space Complexity:

int low, high, i, j, pivot, temp → 6 × 4 = 24 bytes

No additional arrays are used.

Only recursive function calls

Space Complexity = O(log n)