Program 11:

Design and implement C/C++ program to sort a given set of n integer elements using Merge Sort method and compute its time complexity. Run the program for varied values of n>5000 and record the time taken to sort. Plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator.

Algorithm:

ALGORITHM *Mergesort*$(A[0..n-1])$

//Sorts array $A[0..n-1]$ by recursive mergesort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
if $n > 1$
copy $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$
copy $A[\lfloor n/2 \rfloor..n-1]$ to $C[0..\lceil n/2 \rceil - 1]$
Mergesort$(B[0..\lfloor n/2 \rfloor - 1])$
Mergesort$(C[0..\lceil n/2 \rceil - 1])$
Merge(B, C, A)

ALGORITHM *Merge*$(B[0..p-1], C[0..q-1], A[0..p+q-1])$

//Merges two sorted arrays into one sorted array
//Input: Arrays $B[0..p-1]$ and $C[0..q-1]$ both sorted
//Output: Sorted array $A[0..p+q-1]$ of the elements of B and C
$i \leftarrow 0;\ j \leftarrow 0;\ k \leftarrow 0$
while $i < p$ **and** $j < q$ **do**
if $B[i] \le C[j]$
$A[k] \leftarrow B[i];\ i \leftarrow i + 1$
else $A[k] \leftarrow C[j];\ j \leftarrow j + 1$
$k \leftarrow k + 1$
if $i = p$
copy $C[j..q-1]$ to $A[k..p+q-1]$
else copy $B[i..p-1]$ to $A[k..p+q-1]$

Code:

```
#include<stdio.h>

#include<stdlib.h>
```

```
#include<time.h>// Function prototypes
void mergeSort(int arr[], int low, int high);
void merge(int arr[], int low, int mid, int high);
double timeMergeSort(int arr[], int n);
// Main function
int main() {
srand(time(NULL));
int step = 500;
printf("n\tTime (ms)\n");
for(int n = 500; n <= 10000; n += step) {
double totalTime = 0.0;
for (int i = 0; i < 5; i++) { // Repeat 5 times and take average time
// Generate random numbers to fill the array
int *arr = (int *)malloc(n * sizeof(int));
for (int j = 0; j < n; j++) {
arr[j] = rand() % 1000;
}
totalTime += timeMergeSort(arr, n);
free(arr);
}
double averageTime = totalTime / 5.0;
printf("%d\t%.2f\n", n, averageTime);
}
return 0;
```

```
}
// Merge sort algorithm
void mergeSort(int arr[], int low, int high) {
    if (low < high) {
            int mid = (low + high) / 2;
    mergeSort(arr, low, mid);
    mergeSort(arr, mid + 1, high);
    merge(arr, low, mid, high);
    }
}
void merge(int arr[], int low, int mid, int high) {
    int n1 = mid - low + 1;
    int n2 = high - mid;
    int *L = (int *)malloc(n1 * sizeof(int));
    int *R = (int *)malloc(n2 * sizeof(int));
    for(int i = 0; i < n1; i++)
        L[i] = arr[low + i];
    for(int j = 0; j < n2; j++)
        R[j] = arr[mid + 1 + j];
    int i=0;
    int j=0;
    int k=low;
    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++;
    }
    while(j < n2) {
    arr[k] = R[j];
    j++;
    k++;
    }
    free(L);
    free(R);
}
// Timing function for Merge Sort
double timeMergeSort(int arr[], int n)
{
    clock_t start = clock();
    mergeSort(arr, 0, n - 1);
```

```
    clock_t end = clock();

    return ((double)(end - start)) * 1000.0 / CLOCKS_PER_SEC;

}
```

Output:

```
n        Time (ms)
500      0.00
1000     0.00
1500     3.00
2000     0.00
2500     0.00
3000     0.00
3500     3.20
4000     0.00
4500     3.00
5000     0.00
5500     3.20
6000     0.00
6500     3.20
7000     3.00
7500     3.20
8000     3.20
8500     0.00
9000     0.00
9500     6.20
10000    3.20
```