Merge Sort:

Merge sort is defined as a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array. Merge sort is a recursive algorithm that continuously splits the array in half until it cannot be further divided i.e., the array has only one element left (an array with one element is always sorted). Then the sorted subarrays are merged into one sorted array.

Algorithm:

The concept of Divide and Conquer involves three steps:

- Divide the problem into multiple subproblems.
- Solve the Sub Problems. The idea is to break down the problem into atomic subproblems, where they are actually solved.
- Combine the solutions of the subproblems to find the solution of the actual problem.

So, the merge sort working rule involves the following steps:

- Divide the unsorted array into subarray, each containing a single element.
- Take adjacent pairs of two single-element array and merge them to form an array of 2 elements.
- Repeat the process till a single sorted array is obtained.

Code:

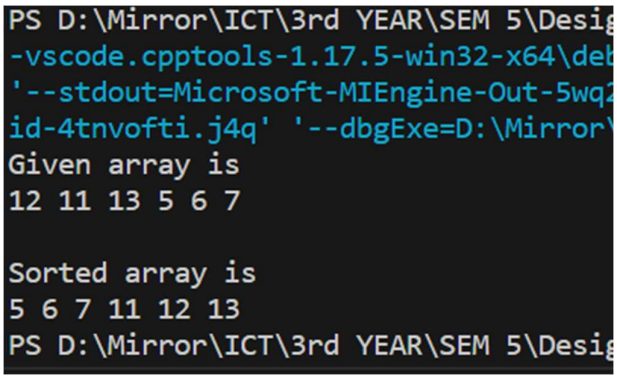
```
// C++ program for Merge Sort

#include <bits/stdc++.h>
using namespace std;

// Merges two subarrays of array[].
// First subarray is arr[begin..mid]
// Second subarray is arr[mid+1..end]
void merge(int array[], int const left, int const mid,
           int const right)
{
    int const subArrayOne = mid - left + 1;
    int const subArrayTwo = right - mid;

    // Create temp arrays
```

```
auto *leftArray = new int[subArrayOne],
        *rightArray = new int[subArrayTwo];

// Copy data to temp arrays leftArray[] and rightArray[]
for (auto i = 0; i < subArrayOne; i++)
        leftArray[i] = array[left + i];
for (auto j = 0; j < subArrayTwo; j++)
        rightArray[j] = array[mid + 1 + j];

auto indexOfSubArrayOne = 0, indexOfSubArrayTwo = 0;
int indexOfMergedArray = left;

// Merge the temp arrays back into array[left..right]
while (indexOfSubArrayOne < subArrayOne
        && indexOfSubArrayTwo < subArrayTwo) {
        if (leftArray[indexOfSubArrayOne]
                <= rightArray[indexOfSubArrayTwo]) {
                array[indexOfMergedArray]
                        = leftArray[indexOfSubArrayOne];
                indexOfSubArrayOne++;
        }
        else {
                array[indexOfMergedArray]
                        = rightArray[indexOfSubArrayTwo];
                indexOfSubArrayTwo++;
        }
        indexOfMergedArray++;
}

// Copy the remaining elements of
// left[], if there are any
while (indexOfSubArrayOne < subArrayOne) {
        array[indexOfMergedArray]
                = leftArray[indexOfSubArrayOne];
        indexOfSubArrayOne++;
        indexOfMergedArray++;
}

// Copy the remaining elements of
// right[], if there are any
```

```
        while (indexOfSubArrayTwo < subArrayTwo) {
                array[indexOfMergedArray]
                        = rightArray[indexOfSubArrayTwo];
                indexOfSubArrayTwo++;
                indexOfMergedArray++;
        }
        delete[] leftArray;
        delete[] rightArray;
}

// begin is for left index and end is right index
// of the sub-array of arr to be sorted
void mergeSort(int array[], int const begin, int const end)
{
        if (begin >= end)
                return;

        int mid = begin + (end - begin) / 2;
        mergeSort(array, begin, mid);
        mergeSort(array, mid + 1, end);
        merge(array, begin, mid, end);
}

// UTILITY FUNCTIONS
// Function to print an array
void printArray(int A[], int size)
{
        for (int i = 0; i < size; i++)
                cout << A[i] << " ";
        cout << endl;
}

// Driver code
int main()
{
        int arr[] = { 12, 11, 13, 5, 6, 7 };
        int arr_size = sizeof(arr) / sizeof(arr[0]);

        cout << "Given array is \n";
        printArray(arr, arr_size);
```

```
    mergeSort(arr, 0, arr_size - 1);

    cout << "\nSorted array is \n";
    printArray(arr, arr_size);
    return 0;
}
```

Output:



Space complexity: _____

Justification:_____
_____

Time complexity:

Best case time complexity: _____

Justification:_____
_____

Worst case time complexity: _____

Justification:_____