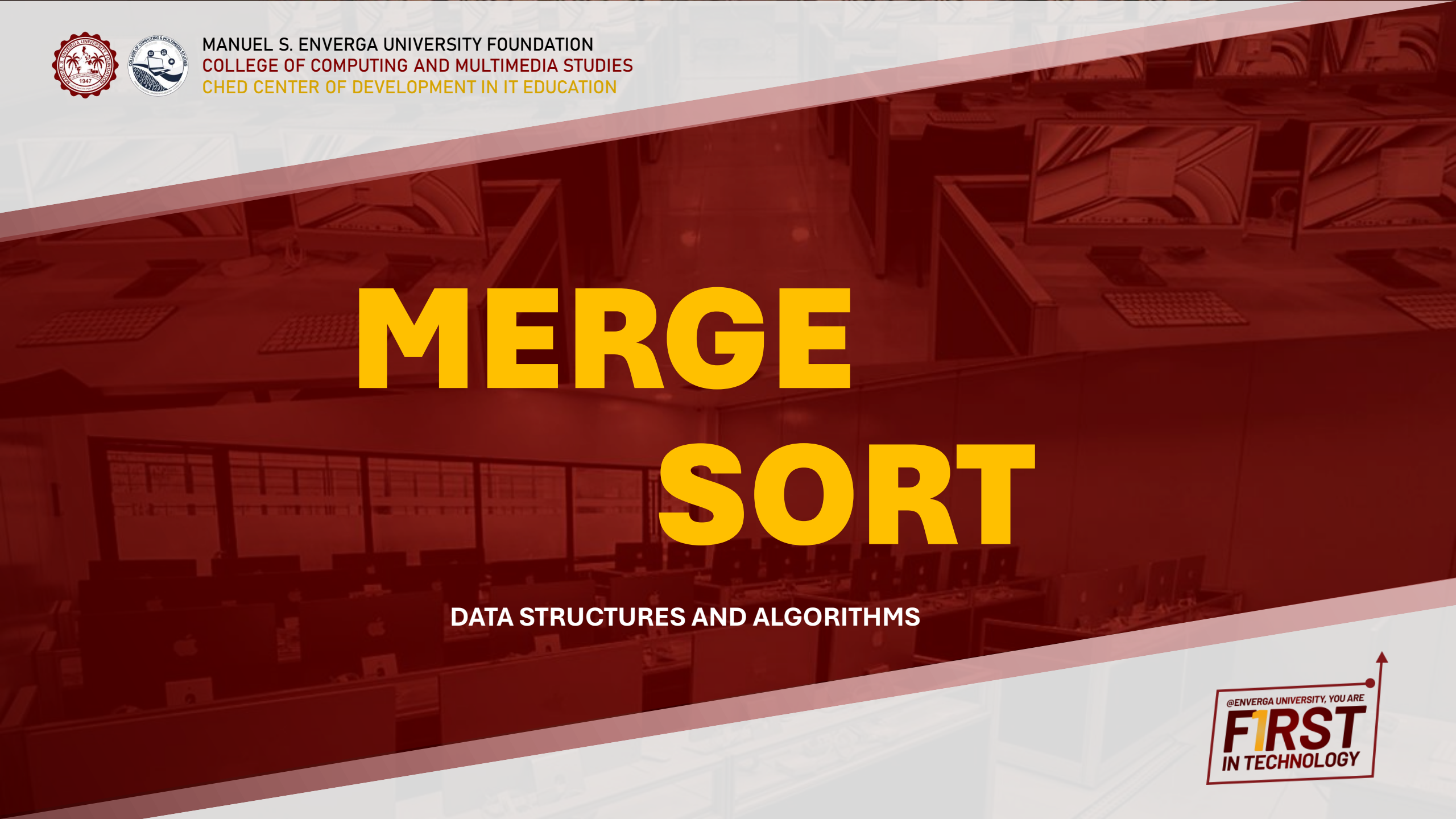
MANUEL S. ENVERGA UNIVERSITY FOUNDATION
COLLEGE OF COMPUTING AND MULTIMEDIA STUDIES
CHED CENTER OF DEVELOPMENT IN IT EDUCATION



MERGE SORT

DATA STRUCTURES AND ALGORITHMS

GROUP 5

MEMBERS

ROOSC ZANO

NEO MEDRANO

ROMINA VICTORIA SALAPARE

ROENTGEN COMPRA

SEAN MIKAEL CID

@ENVERGA UNIVERSITY, YOU ARE FIRST IN TECHNOLOGY

Learning Outcomes

- **Understanding Merge Sort**: Learn how it splits, sorts, and merges subarrays using the divide-and-conquer approach.
- **Recognizing Limitations**: Understand its space overhead and why it's less ideal for in-place sorting.
- **Comparison with Algorithms**: Compare its efficiency and stability with Quick Sort and Bubble Sort.
- **Implementation and Application**: Write functions for sorting arrays and use it in projects requiring stable sorting.
- **Parallel Processing**: Recognize its suitability for parallel processing due to independent subarray sorting.
- Implement search functionality in a real-world project

What is **Merge Sort?**

- **Merge sort** is a popular sorting algorithm known for its efficiency and stability. It follows the **divide-and-conquer** approach to sort a given array of elements. It works by recursively dividing the input array into smaller subarrays and sorting those subarrays then merging them back together to obtain the sorted array.

- In simple terms, we can say that the process of merge sort is to divide the array into two halves, sort each half, then merge the sorted halves back together. This process is repeated until the entire array is sorted.

Applications of Merge Sort

Sorting Large Datasets:

- Merge Sort is widely used for **external sorting**, where datasets are too large to fit entirely in memory. For example, in systems that process large files or databases stored on disk, Merge Sort can split the data into smaller chunks, sort them individually, and then merge the sorted chunks efficiently.

Example: You have a list of numbers: `[8, 1, 3, 7, 2, 6]`. Merge Sort splits it into smaller parts, sorts each part, and then combines them back together into a sorted list: `[1, 2, 3, 6, 7, 8]`.

Applications of Merge Sort

Sorting Linked Lists:

- Merge Sort is particularly suitable for sorting linked lists because it does not require random access to elements like Quick Sort. Its ability to split the list in O(n)O(n)O(n) time and merge sorted sublists efficiently makes it an ideal choice for such data structures.

Example: A linked list 4 -> 2 -> 6 -> 1 is split into 4 -> 2 and 6 -> 1, sorted separately, and then merged to give 1 -> 2 -> 4 -> 6.

Applications of Merge Sort

Stable Sorting Needs:

- As a **stable sorting algorithm**, Merge Sort preserves the relative order of elements with equal keys. This makes it valuable in applications like **database management** systems, where maintaining the stability of records with similar fields is critical for consistency.

Example: Sorting a list of tuples `[("A", 3), ("B", 1), ("C", 3)]`. Merge Sort keeps the order of `"A"` and `"C"` the same as they have the same value (3): `[("B", 1), ("A", 3), ("C", 3)]`.

Applications of Merge Sort

Parallel Processing:

- The divide-and-conquer nature of Merge Sort lends itself well to **parallel processing**. Subarrays can be sorted independently on different processors, significantly improving performance for large-scale data in distributed systems or multi-core environments.

Example: A list `[9, 7, 3, 5]` is split into `[9, 7]` and `[3, 5]`. Both halves are sorted in parallel, then merged to give `[3, 5, 7, 9]`.

Applications of Merge Sort

Merge Algorithms and Multiway Merging:

- Merge Sort is often used as a foundation for algorithms that require merging sorted data. For example, in **multiway merging**, sorted files or streams from different sources are combined into one sorted output, which is a common requirement in data integration systems and streaming platforms.

Example: Multiway merging is used to combine multiple sorted lists, like merging `[1, 4]`, `[2, 5]`, and `[3, 6]` into a single sorted list `[1, 2, 3, 4, 5, 6]`.

Advantages of Merge Sort

- It is quicker for larger lists because unlike insertion and bubble sort it doesn't go through the whole list seveal times.
- It has a consistent running time, carries out different bits with similar times in a stage.

Disadvantages of Linear Search

- Slower comparative to the other sort algorithms for smaller tasks.
- Goes through the whole process even if the list is sorted before.
- Uses more memory space to store the sub elements of the initial split list.

Pseudocode Algorithm

The MERGESORT algorithm for sorting a list $L = (a_0, \ldots, a_{n-1})$ of n items goes as follows.

if $n = 1$ **then** return L
else
  $m = \lfloor (n-1)/2 \rfloor$
  $L_1 = \text{MERGESORT}(a_0, \ldots, a_m)$
  $L_2 = \text{MERGESORT}(a_{m+1}, \ldots, a_{n-1})$
  return $\text{MERGE}(L_1, L_2)$

where the function MERGE for two sorted input lists $L_1 = (a_0, \ldots, a_{r-1})$, $L_2 = (b_0, \ldots, b_{s-1})$ is defined by

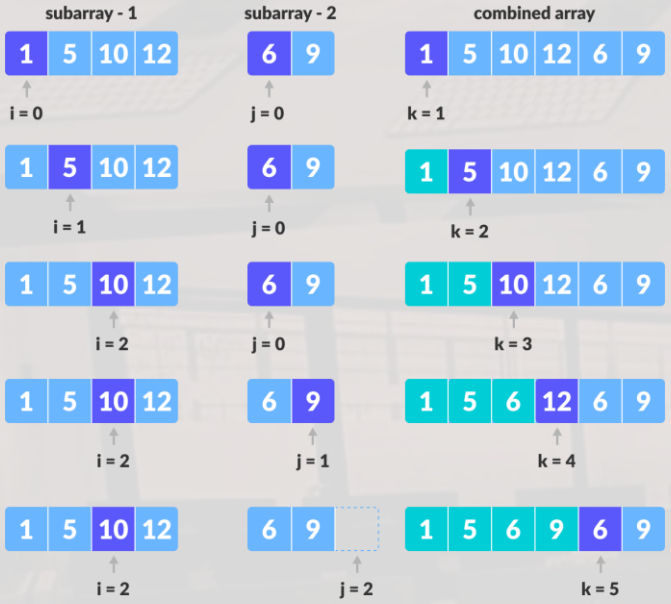
if L_1 is empty **then** return L_2
elseif L_2 is empty **then** return L_1
else // i.e., both lists are not empty
  if $a_0 \leqslant b_0$ **then**
    $L' = \text{MERGE}((a_1, \ldots, a_{r-1}), (b_0, \ldots, b_{s-1}))$
    return (a_0, L')
  else // i.e., $b_0 < a_0$
    $L' = \text{MERGE}((a_0, \ldots, a_{r-1}), (b_1, \ldots, b_{s-1}))$
    return (b_0, L')
  return $\text{MERGE}(L_1, L_2)$

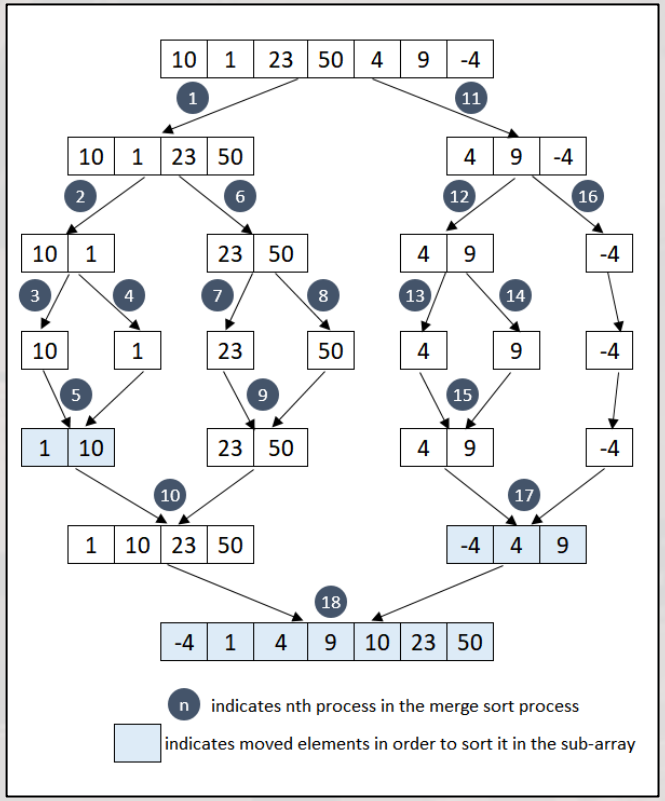
```
MERGE(A, p, q, r)
1.  n1 = q − p + 1
2.  n2 = r − q
3.  let L[1..n1 + 1] and R[1..n2 + 1] be new arrays
4.  for i = 1 to n1
5.      L[i] = A[p + i − 1]
6.  for j = 1 to n2
7.      R[j] = A[q + j]
8.  L[n1 + 1] = ∞
9.  R[n2 + 1] = ∞
10. i = 1
11. j = 1
12. for k = p to r
13.     if L[i] ≤ R[j]
14.         A[k] = L[i]
15.         i = i + 1
16.     else A[k] = R[j]
17.         j = j + 1
```

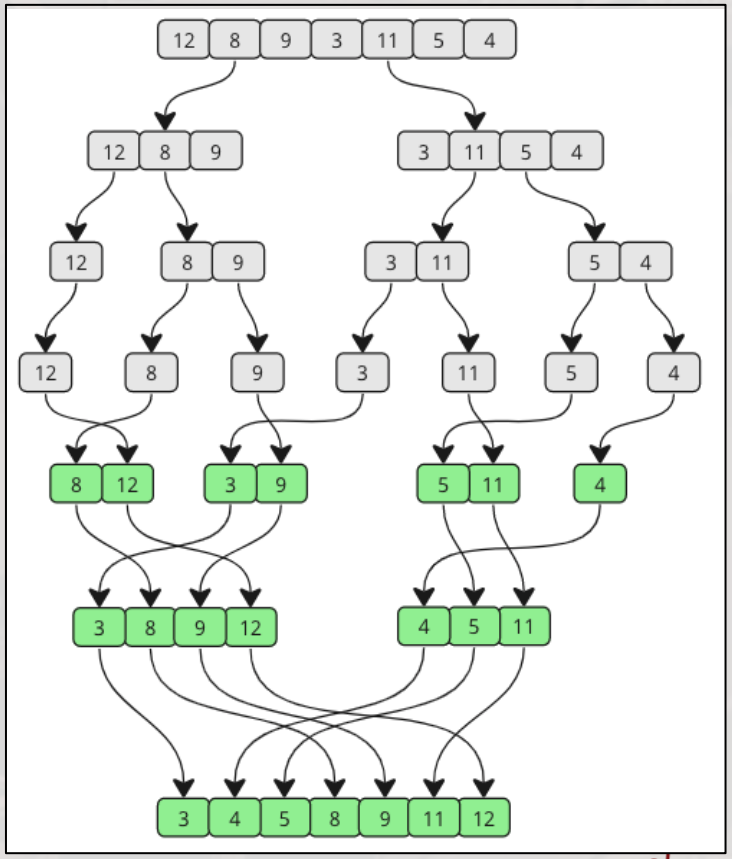
Algorithm 1: Sort-Merge Join: $R \bowtie_{R.r=S.s} S$

```
// sorting step
Sort the relation R on the attribute r;
Sort the relation S on the attribute s;
// merging step
R = first tuple in R;
S = first tuple in S;
S' = first tuple in S;
while R ≠ eof and S' ≠ eof do
    while R.r < S'.s do
        R = next tuple in R after R;
    end
    while R.r > S'.s do
        S' = next tuple in S after S';
    end
    S = S';
    while R.r == S'.s do
        S = S';
        while R.r == S.s do
            add {R, S} to result;
            S = next tuple in S after S;
        end
        R = next tuple in R after R;
    end
    S' = S;
end
```

Examples of Merge Sort







Source Code Implementation

```cpp
#include <iostream>
#include <vector>

// Function to merge two halves into a sorted array
void merge(std::vector<int>& arr, int left, int mid, int right) {
    // Calculate the size of the two subarrays to be merged
    int n1 = mid - left + 1;
    int n2 = right - mid;

    // Create temporary arrays
    std::vector<int> leftArr(n1), rightArr(n2);

    // Copy data to temp arrays leftArr[] and rightArr[]
    for (int i = 0; i < n1; i++) {
        leftArr[i] = arr[left + i];
    }
    for (int j = 0; j < n2; j++) {
        rightArr[j] = arr[mid + 1 + j];
    }

    // Merge the temp arrays back into arr[left..right]
    int i = 0; // Initial index of first subarray
    int j = 0; // Initial index of second subarray
    int k = left; // Initial index of merged subarray

    while (i < n1 && j < n2) {
        if (leftArr[i] <= rightArr[j]) {
            arr[k] = leftArr[i];
            i++;
        } else {
            arr[k] = rightArr[j];
            j++;
        }
        k++;
    }

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

    // Copy any remaining elements of rightArr[]
    while (j < n2) {
        arr[k] = rightArr[j];
        j++;
        k++;
    }
}

// Function to divide the array into two halves and sort each half
void mergeSort(std::vector<int>& arr, int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;

        // Recursively sort the two halves
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);

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

// Function to print the array
void printArray(const std::vector<int>& arr) {
    for (int num : arr) {
        std::cout << num << " ";
    }
    std::cout << std::endl;
}

int main() {
    std::vector<int> arr = {38, 27, 43, 3, 9, 82, 10};

    std::cout << "Original array: ";
    printArray(arr);

    mergeSort(arr, 0, arr.size() - 1);

    std::cout << "Sorted array: ";
    printArray(arr);

    return 0;
}
```

THANK YOU!

@ENVERGA UNIVERSITY, YOU ARE FIRST IN TECHNOLOGY