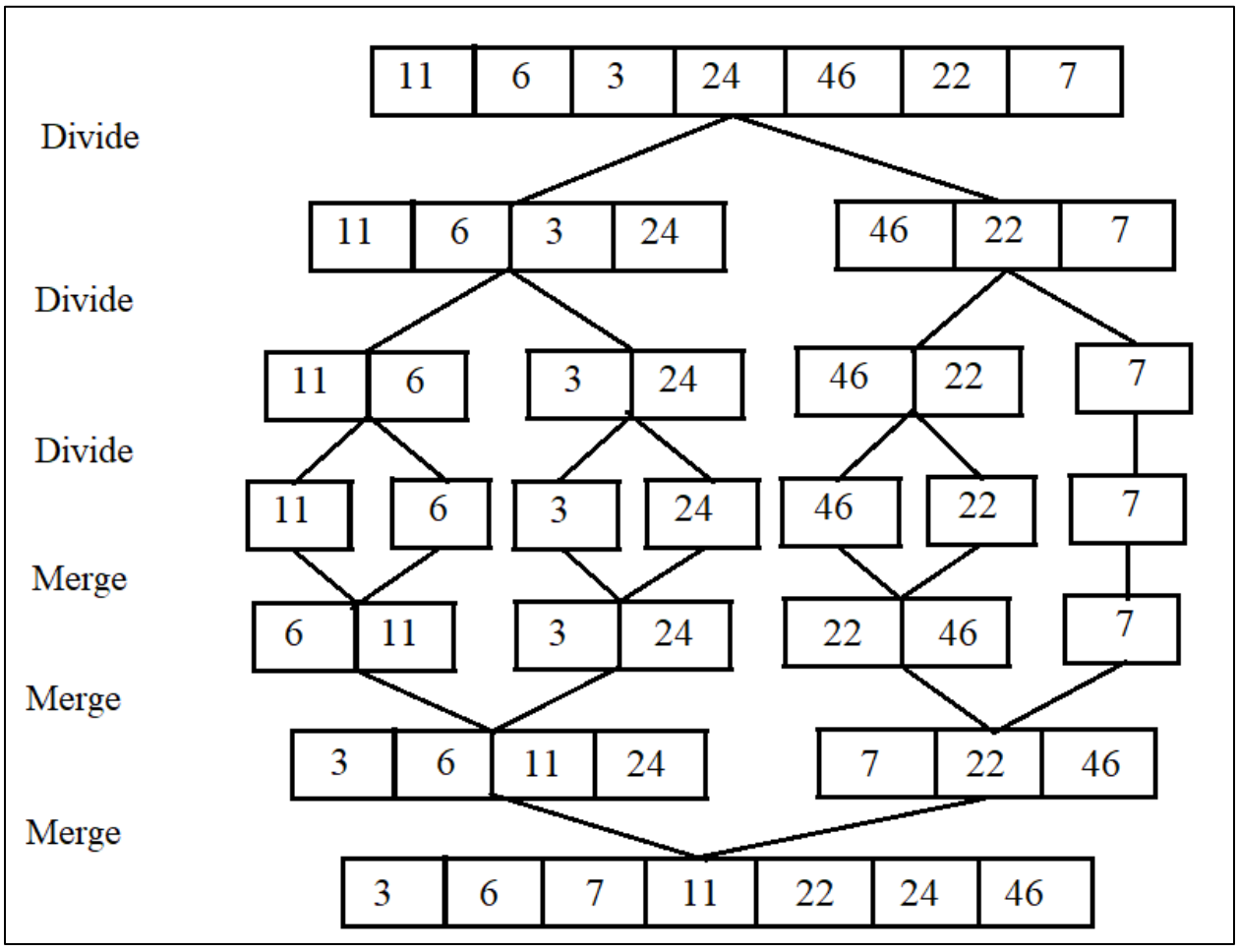
ADVANCE SOTING TECHNIQUE: MERGE SORT

Merge Sort is a classic and efficient sorting algorithm based on the Divide and Conquer strategy. It works by recursively dividing the array into two halves, sorting each half, and then merging them back together into a sorted array. It is particularly well-suited for large datasets.

STEPS IN MERGE SORT

1. **Divide:** Recursively divide the array into two halves until each sub-array contains a single element (which is trivially sorted).
2. **Conquer**: Recursively sort the two halves.
3. **Merge:** Merge the two sorted halves into a single sorted array.



Time Complexity

Worst, Average, and Best Case: O(n log n)

The algorithm always divides the array into two halves and then merges them back, which results in a logarithmic number of levels of recursion. Each level requires linear time to merge the sub-arrays.

ADVANTAGES

- Stable Sort: Merge sort maintains the relative order of equal elements.
- Predictable Performance: Merge sort consistently has a time complexity of O(n log n), making it suitable for large datasets.
- External Sorting: It's useful in scenarios where data is stored on external storage (e.g., disk) and needs to be sorted in memory-constrained environments.

DISADVANTAGES

- Extra Space Requirement: The additional space needed for temporary arrays can be a downside, especially in memory-constrained systems.
- Slower than In-Place Algorithms: Algorithms like quick sort (on average) or heap sort are in-place and often faster for smaller datasets.

CODE

```
#include <iostream>

using namespace std;



// Function to merge two subarrays

void merge(int arr[], int left, int mid, int right) {

    int n1 = mid - left + 1;

    int n2 = right - mid;
```

```
// Create temporary arrays
int* leftArr = new int[n1];
int* rightArr = new int[n2];

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

// Merge the temporary arrays back into arr[]
int i = 0, j = 0, k = left;

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

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

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

    // Free the temporary arrays
    delete[] leftArr;
    delete[] rightArr;
}

// Recursive function to perform merge sort
void mergeSort(int arr[], int left, int right) {
    if (left < right) {
```

```
        int mid = left + (right - left) / 2;

        // Recursively sort first and second halves
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);

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

// Helper function to print the array
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}

int main() {
    int arr[] = {38, 27, 43, 3, 9, 82, 10};
    int arrSize = sizeof(arr) / sizeof(arr[0]);

    cout << "Given array: ";
```

```
    printArray(arr, arrSize);

    mergeSort(arr, 0, arrSize - 1);

    cout << "Sorted array: ";
    printArray(arr, arrSize);

    return 0;
}
```

Output:

```
Given array: 38 27 43 3 9 82 10
Sorted array: 3 9 10 27 38 43 82
```