Merge Sort

Merge 2 sorted Arrays: Problem 01

You are given two integer arrays nums1 and nums2, sorted in non-decreasing order, and two integers m and n, representing the number of elements in nums1 and nums2 respectively.

Merge nums1 and nums2 into a single array sorted in non-decreasing order.

The final sorted array should not be returned by the function, but instead be stored inside the array nums1. To accommodate this, nums1 has a length of m + n, where the first m elements denote the elements that should be merged, and the last n elements are set to 0 and should be ignored. nums2 has a length of n.

Merge 2 sorted Arrays: Problem 01

Input: nums1 = [1,2,3,0,0,0], m = 3, nums2 = [2,5,6], n = 3
Output: [1,2,2,3,5,6]
Explanation: The arrays we are merging are [1,2,3] and [2,5,6].
The result of the merge is [1,2,2,3,5,6]

Input: nums1 = [0], m = 0, nums2 = [1], n = 1
Output: [1]
Explanation: The arrays we are merging are [] and [1].
The result of the merge is [1].
Note that because m = 0, there are no elements in nums1. The 0 is only there to ensure the merge result can fit in nums1.

Solution

```
void merge(vector<int> &nums1, int m, vector<int> &nums2, int n)
{

}
```

Solution

```
void merge(vector<int> &nums1, int m, vector<int> &nums2, int n)
{
    int i = 0;
    int j = 0;
    vector<int> arr3;
    while (i < m && j < n){
        if (nums1[i] < nums2[j])
        {
            arr3.push_back(nums1[i]);
            i++;
        }
        else
        {
            arr3.push_back(nums2[j]);
            j++;
        }
    }
    while (i < m){
        arr3.push_back(nums1[i]);
        i++;
    }
    while (j < n){
        arr3.push_back(nums2[j]);
        j++;
    }
    for (int x = 0; x < arr3.size(); x++){
        nums1[x] = arr3[x];
    }
}
```

Sort half sorted Array: Problem 02

You are given one integer array arr, it is virtually divided into two sorted arrays, such that from the start to the mid (first part is sorted) and from mid + 1 to the end (second part is sorted).

arr = [1, 3, 5, 7, 9, 2, 4, 6, 8, 10]

Your goal is to sort this complete array into ascending order.

arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Solution

```
void merge(int arr[], int start, int mid, int end)
{

}
```

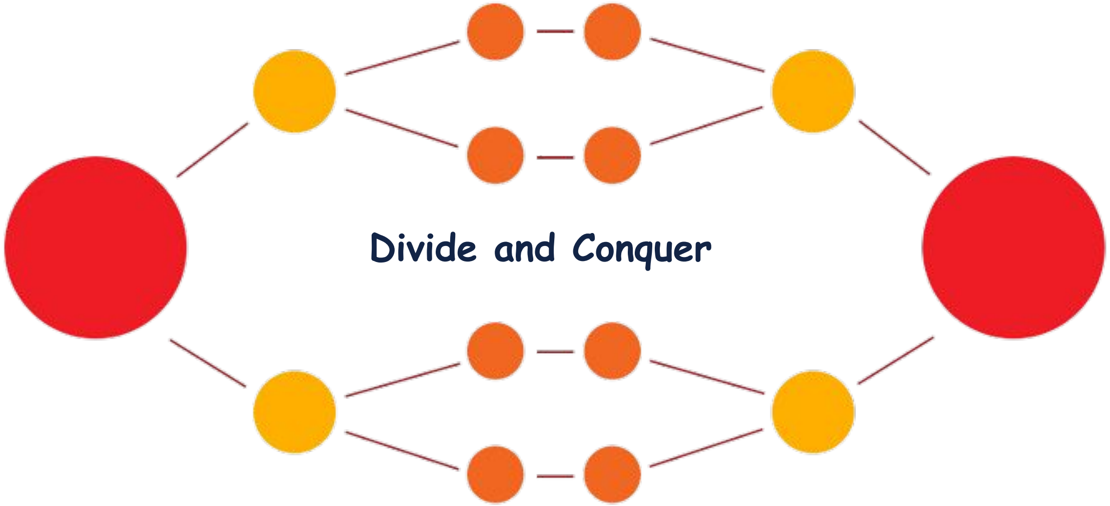
Solution

```
void merge(int arr[], int start, int mid, int end){
        int i = start;
        int j = mid + 1;
        queue<int> tempArr;
        while (i <= mid && j <= end){
            if (arr[i] < arr[j])
            {
                tempArr.push(arr[i]);
                i++;
            }
            else
            {
                tempArr.push(arr[j]);
                j++;
            }
        }
        while (i <= mid){
            tempArr.push(arr[i]);
            i++;
        }
        while (j <= end){
            tempArr.push(arr[j]);
            j++;
        }
        for (int x = start; x <= end; x++){
            arr[x] = tempArr.front();
            tempArr.pop();
        }
}
```

MergeSort: Sorting Algorithm

MergeSort uses the concept of merging two sorted Arrays.

It follows the Divide and Conquer Strategy.



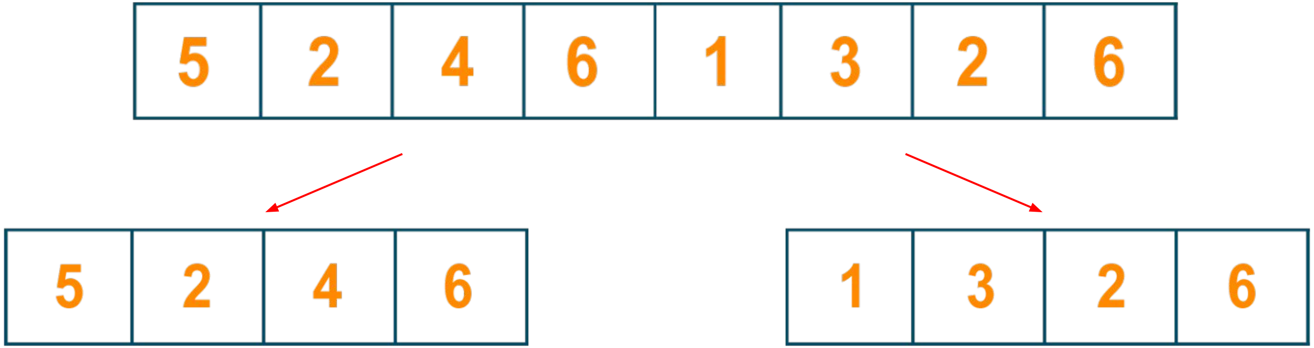
MergeSort: Divide

Consider the following Array to be sorted.

5	2	4	6	1	3	2	6

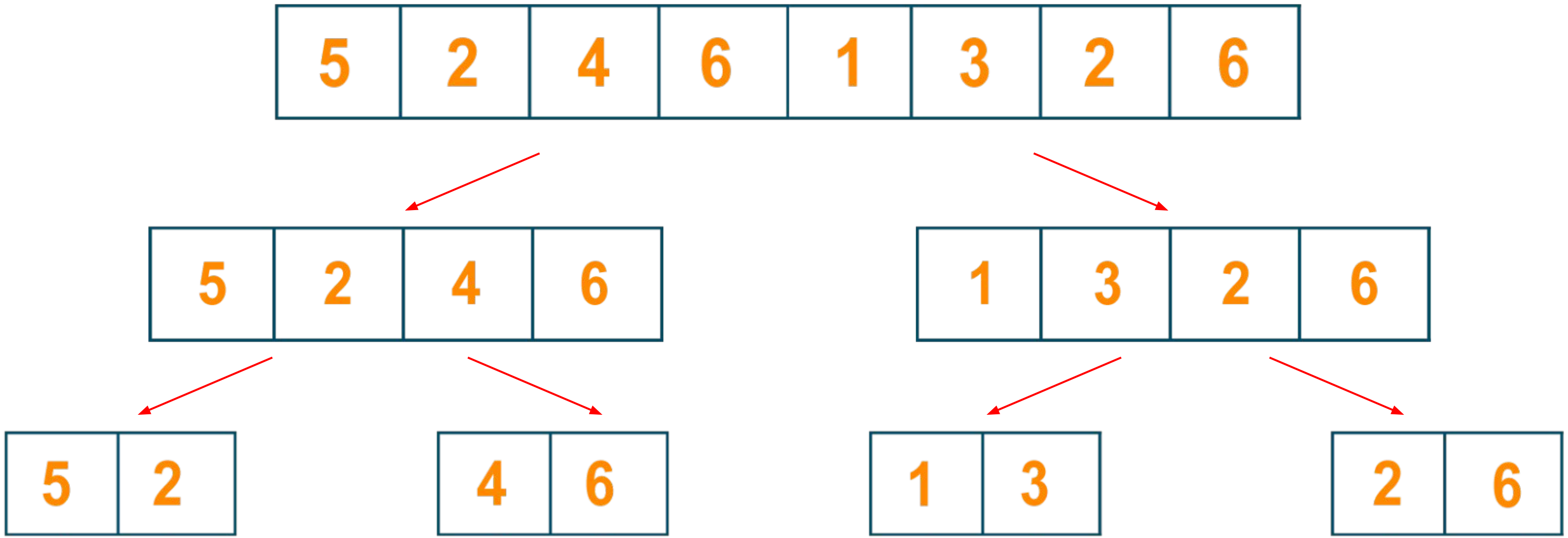
MergeSort: Divide

Let's Divide this Array into 2 halves.



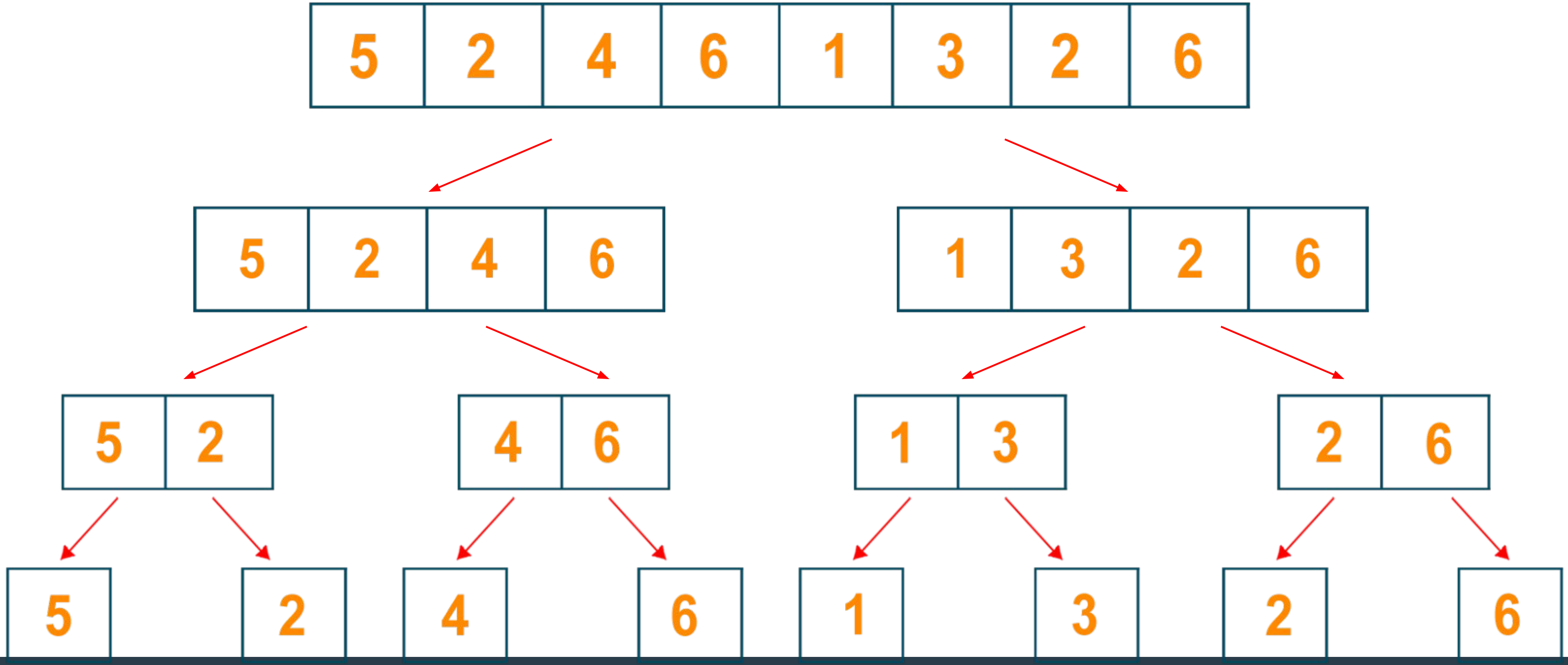
MergeSort: Divide

Let's further divide these 2 halves into 4 halves.



MergeSort: Divide

Let's further divide these 4 halves into 8 halves.



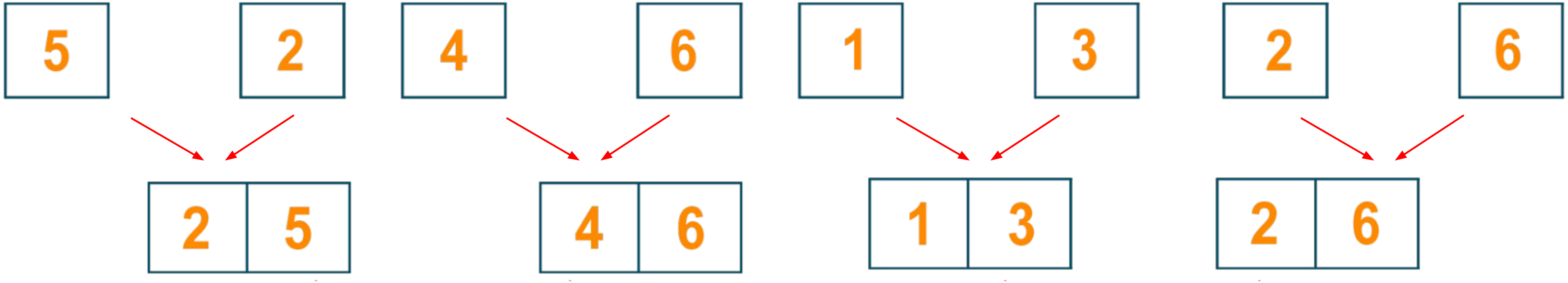
MergeSort: Conquer

Now, we will merge the divided halves into sorted array.

5	2	4	6	1	3	2	6

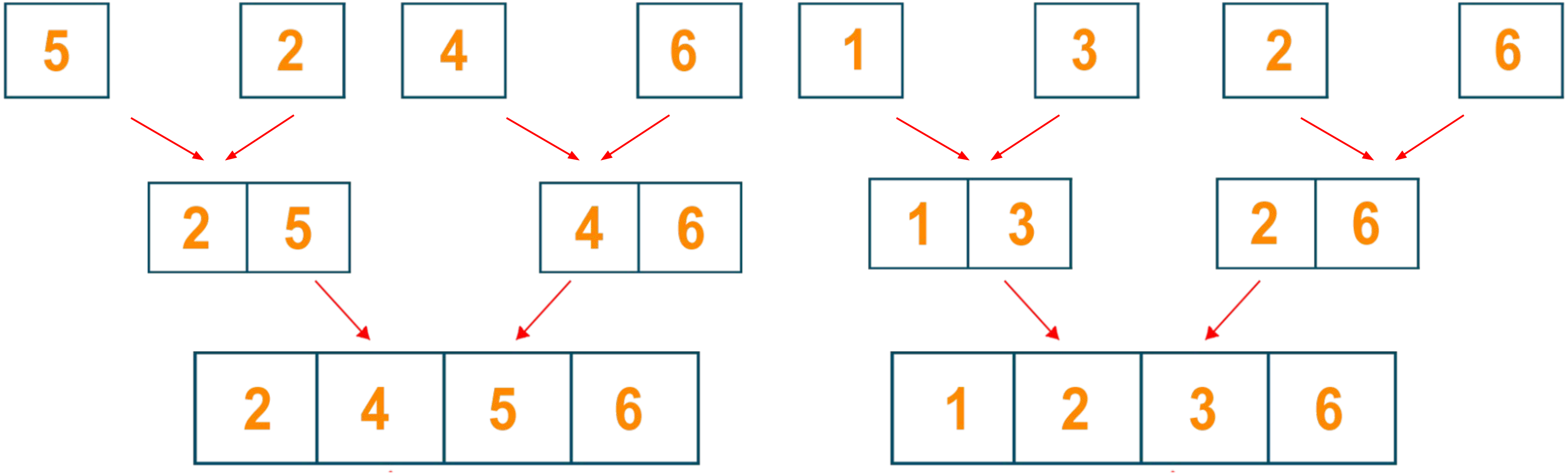
MergeSort: Conquer

Now, we will merge the divided halves into sorted array.



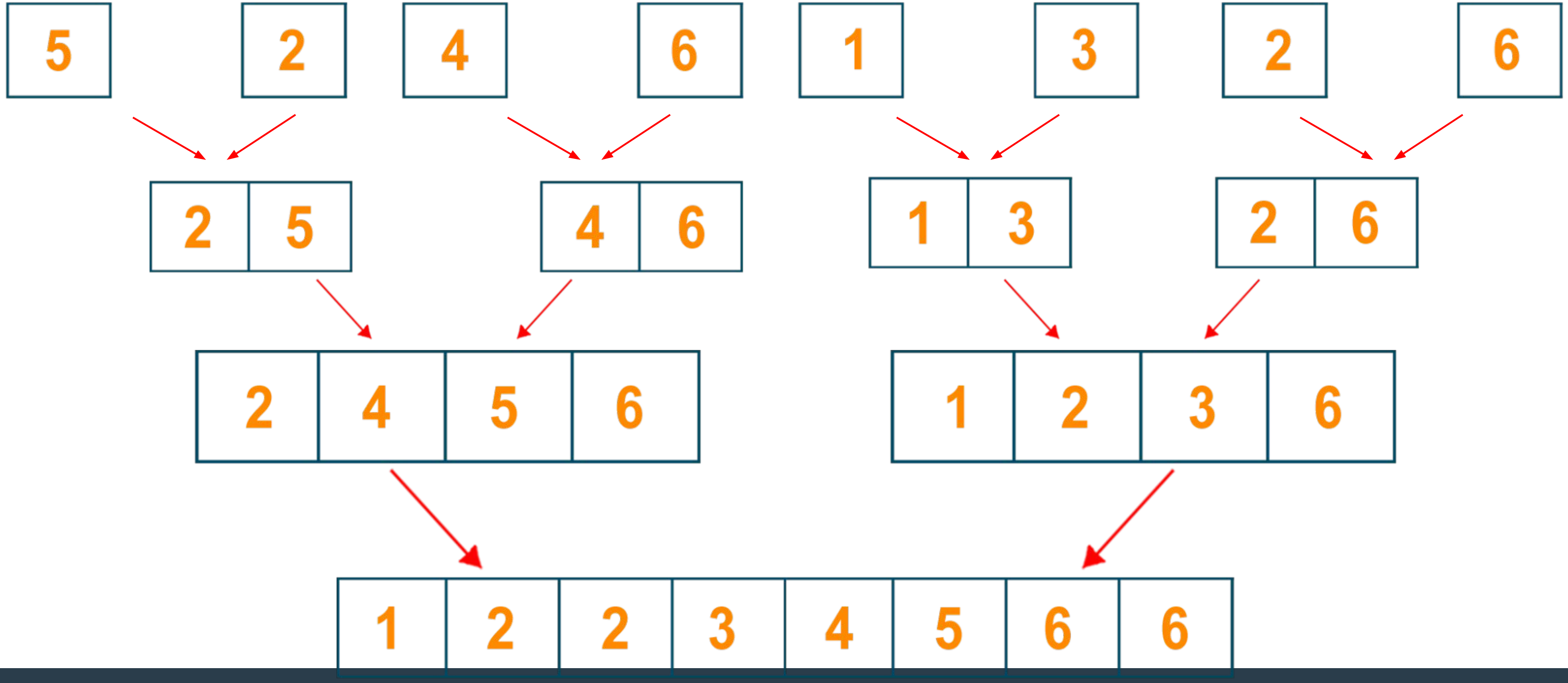
MergeSort: Conquer

Now, we will merge the divided halves into sorted array.



MergeSort: Conquer

Now, we will merge the divided halves into sorted array.



MergeSort: Sorting Algorithm

Since, the divide and conquer algorithms are based on dividing the problems into smaller problems, therefore, it is easy to do it with the help of recursion.

MergeSort: Implementation

Let's write the code for mergeSort.

MergeSort: Implementation

```
main()
{
    int arr[8] = {5, 2, 4, 6, 1, 3, 2, 6};
    mergeSort(arr, 0, 7);
    for (int x = 0; x < 8; x++)
    {
        cout << arr[x] << " ";
    }
}
```

```
void mergeSort(int arr[], int start, int end)
{
    if (start < end)
    {
        int mid = (start + end) / 2;
        mergeSort(arr, start, mid);
        mergeSort(arr, mid + 1, end);
        merge(arr, start, mid, end);
    }
}
```

MergeSort

```
void merge(int arr[], int start, int mid, int end){
        int i = start;
        int j = mid + 1;
        queue<int> tempArr;
        while (i <= mid && j <= end){
            if (arr[i] < arr[j])
            {
                tempArr.push(arr[i]);
                i++;
            }
            else
            {
                tempArr.push(arr[j]);
                j++;
            }
        }
        while (i <= mid){
            tempArr.push(arr[i]);
            i++;
        }
        while (j <= end){
            tempArr.push(arr[j]);
            j++;
        }
        for (int x = start; x <= end; x++){
            arr[x] = tempArr.front();
            tempArr.pop();
        }
}
```

MergeSort

Now the important question is: What is the Time complexity of this Algorithm?

```
void merge(int arr[], int start, int mid, int end){
        int i = start;
        int j = mid + 1;
        queue<int> tempArr;
        while (i <= mid && j <= end){
            if (arr[i] < arr[j])
            {
                tempArr.push(arr[i]);
                i++;
            }
            else
            {
                tempArr.push(arr[j]);
                j++;
            }
        }
        while (i <= mid){
            tempArr.push(arr[i]);
            i++;
        }
        while (j <= end){
            tempArr.push(arr[j]);
            j++;
        }
        for (int x = start; x <= end; x++){
            arr[x] = tempArr.front();
            tempArr.pop();
        }
}
```

MergeSort

Now the important question is: What is the Time complexity of this Algorithm?

Time required for Dividing as well as the Time required for Conquering

```
void merge(int arr[], int start, int mid, int end){
        int i = start;
        int j = mid + 1;
        queue<int> tempArr;
        while (i <= mid && j <= end){
            if (arr[i] < arr[j])
            {
                tempArr.push(arr[i]);
                i++;
            }
            else
            {
                tempArr.push(arr[j]);
                j++;
            }
        }
        while (i <= mid){
            tempArr.push(arr[i]);
            i++;
        }
        while (j <= end){
            tempArr.push(arr[j]);
            j++;
        }
        for (int x = start; x <= end; x++){
            arr[x] = tempArr.front();
            tempArr.pop();
        }
}
```

MergeSort

Now the **important** question is: What is the **Time complexity** of this Algorithm?

From the binary search we know that division stages will take $O(\log_2 n)$.

```
void merge(int arr[], int start, int mid, int end){
        int i = start;
        int j = mid + 1;
        queue<int> tempArr;
        while (i <= mid && j <= end){
            if (arr[i] < arr[j])
            {
                tempArr.push(arr[i]);
                i++;
            }
            else
            {
                tempArr.push(arr[j]);
                j++;
            }
        }
        while (i <= mid){
            tempArr.push(arr[i]);
            i++;
        }
        while (j <= end){
            tempArr.push(arr[j]);
            j++;
        }
        for (int x = start; x <= end; x++){
            arr[x] = tempArr.front();
            tempArr.pop();
        }
}
```

MergeSort

Now the important question is: What is the Time complexity of this Algorithm?

And the conquering/merging stage takes O(n) as there is no nested loop.

```
void merge(int arr[], int start, int mid, int end){
        int i = start;
        int j = mid + 1;
        queue<int> tempArr;
        while (i <= mid && j <= end){
            if (arr[i] < arr[j])
            {
                tempArr.push(arr[i]);
                i++;
            }
            else
            {
                tempArr.push(arr[j]);
                j++;
            }
        }
        while (i <= mid){
            tempArr.push(arr[i]);
            i++;
        }
        while (j <= end){
            tempArr.push(arr[j]);
            j++;
        }
        for (int x = start; x <= end; x++){
            arr[x] = tempArr.front();
            tempArr.pop();
        }
}
```

MergeSort

Now the **important** question is: What is the **Time complexity** of this Algorithm?

Worst Time Complexity $= O(n \log_2(n))$

```
void merge(int arr[], int start, int mid, int end){
        int i = start;
        int j = mid + 1;
        queue<int> tempArr;
        while (i <= mid && j <= end){
            if (arr[i] < arr[j])
            {
                tempArr.push(arr[i]);
                i++;
            }
            else
            {
                tempArr.push(arr[j]);
                j++;
            }
        }
        while (i <= mid){
            tempArr.push(arr[i]);
            i++;
        }
        while (j <= end){
            tempArr.push(arr[j]);
            j++;
        }
        for (int x = start; x <= end; x++){
            arr[x] = tempArr.front();
            tempArr.pop();
        }
}
```

MergeSort

Now another **important** question is: What is the **Space complexity** of this Algorithm?

```
void merge(int arr[], int start, int mid, int end){
        int i = start;
        int j = mid + 1;
        queue<int> tempArr;
        while (i <= mid && j <= end){
            if (arr[i] < arr[j])
            {
                tempArr.push(arr[i]);
                i++;
            }
            else
            {
                tempArr.push(arr[j]);
                j++;
            }
        }
        while (i <= mid){
            tempArr.push(arr[i]);
            i++;
        }
        while (j <= end){
            tempArr.push(arr[j]);
            j++;
        }
        for (int x = start; x <= end; x++){
            arr[x] = tempArr.front();
            tempArr.pop();
        }
}
```

Sorting Algorithms

https://www.geeksforgeeks.org/merge-sort/

https://visualgo.net/en/sorting

Sorting Algorithm	Time Complexity			Space Complexity
	Best Case	Average Case	Worst Case	Worst Case
Bubble Sort	$O(N)$	$O(N^2)$	$O(N^2)$	$O(1)$
Selection Sort	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(1)$
Insertion Sort	$O(N)$	$O(N^2)$	$O(N^2)$	$O(1)$
Merge Sort	$O(N*\log_2 N)$	$O(N*\log_2 N)$	$O(N*\log_2 N)$	$O(N)$

Sorting Algorithms

Sorting Algorithm	In-Place	Stable
Bubble Sort	Yes	Yes
Selection Sort	Yes	No
Insertion Sort	Yes	Yes
Merge Sort	No	Yes

Learning Objective

Students should be able to apply sorting using Merge Sort.

Self Assessment

To visually see the Algorithms Running

https://visualgo.net/en/sorting