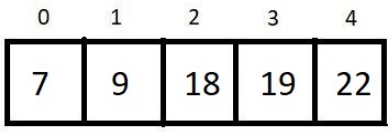
Merge Sort Algorithm

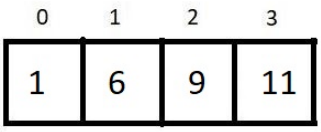
In this algorithm, we divide the arrays into subarrays and subarrays into more subarrays until the size of each subarray becomes 1. Since arrays with a single element are always considered sorted, this is where we merge. Merging two sorted subarrays creates another sorted subarray. I'll show you first how merging two sorted subarrays works.

Merging Procedure:

Suppose we have two sorted arrays, A and B, of sizes 5 and 4, respectively.

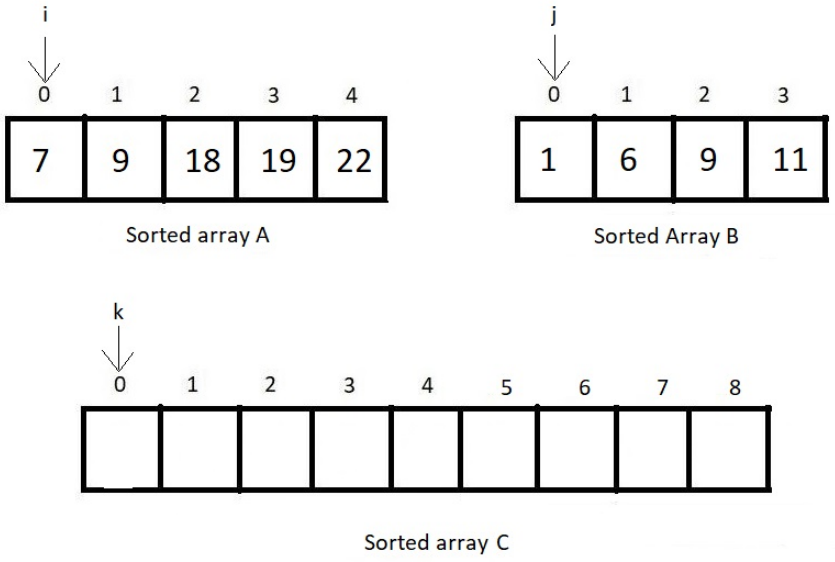


Sorted array A

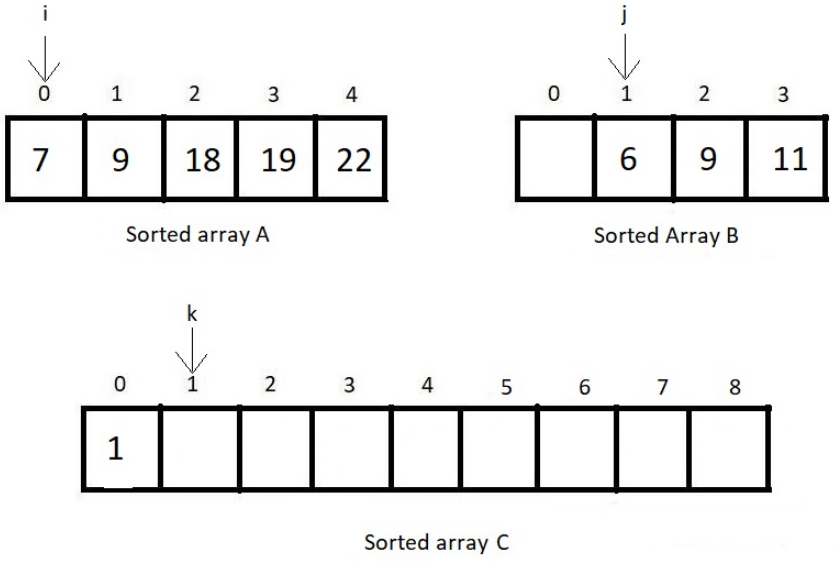


Sorted Array B

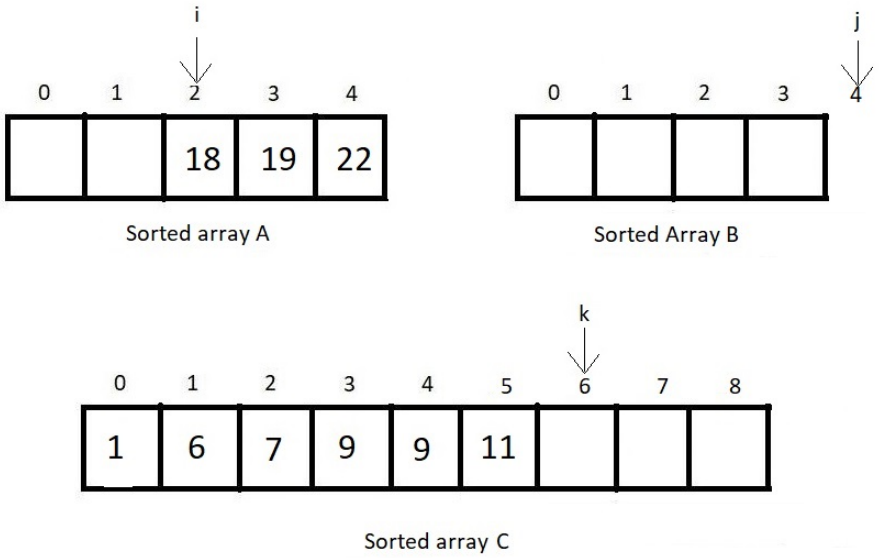
1. And we apply merging on them. Then the first job would be to create another array C with size being the sum of both the raw arrays' sizes. Here the sizes of A and B are 5 and 4, respectively. So, the size of array C would be 9.
2. Now, we maintain three index variables i, j, and k. i looks after the first element in array A, j looks after the first element in array B, and k looks after the position in array C to place the next element in.



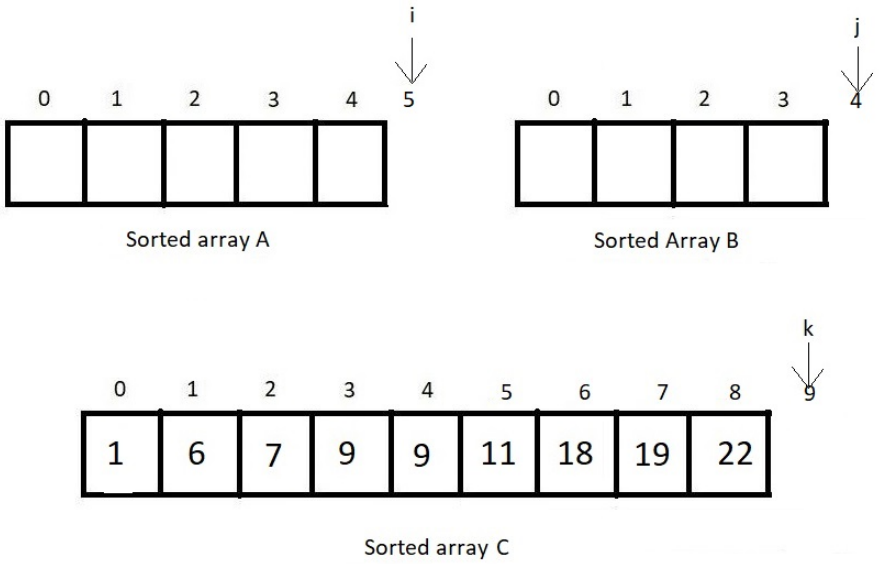
1. Initially, all i, j, and k are equal to 0.
2. Now, we compare the elements at index *i* of array A and index *j* of array B and see which one is smaller. Fill in the smaller element at index *k* of array C and increment *k* by 1. Also, increment the index variable of the array we fetched the smaller element from.
3. Here, A[i] is greater than B[j]. Hence we fill C[k] with B[j] and increase k and j by 1.



1. We continue doing step 5 until one of the arrays, A or B, gets empty.

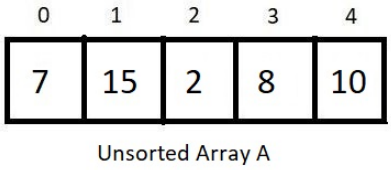


Here, array B inserted all its elements in the merged array C. Since we are only left with the elements of element A, we simply put them in the merged array as they are. This will result in our final merged array C.



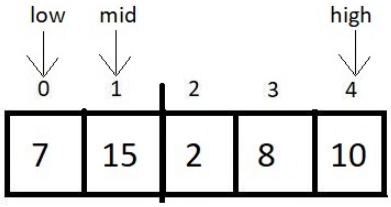
Now, this would quite not be our situation when sorting an array using the merge sort. We wouldn't have two different arrays A and B, rather a single array having two sorted subarrays. Now, I'd show you how to merge two sorted subarrays of a single array in the array itself.

Suppose there is an array A of 5 elements and contains two sorted subarrays of length 2 and 3 in itself.

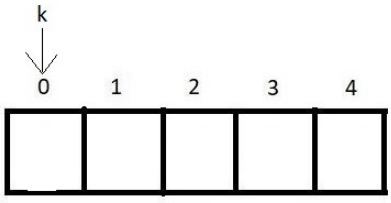


Unsorted Array A

To merge both the sorted subarrays and produce a sorted array of length 5, we will create an auxiliary array B of size 5. Now the process would be more or less the same, and the only change we would need to make is to consider the first index of the first subarray as *low* and the last index of the second subarray as *high*. And mark the index prior to the first index of the first subarray as *mid.*

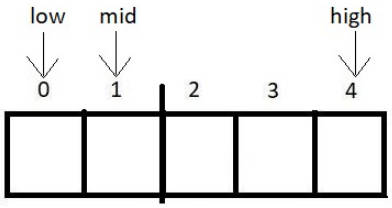


Unsorted Array A

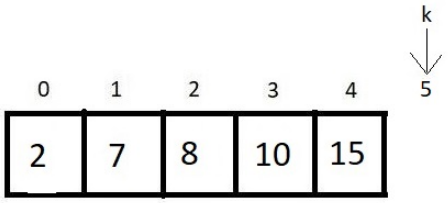


Sorted array B

Previously we had index variables *i, j, and k* initialised with 0 of their respective arrays. But here, i gets initialised with *low,* j gets initialised with *mid+1,* and k gets initialised with *low* only. And similar to what we did earlier, i runs from *low to mid*, j runs from *mid+1 to high*, and until and unless they both get all their elements merged, we continue filling elements in array B.



Unsorted Array A



Sorted array B

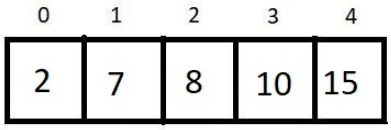
After all the elements get filled in array C, we revert back to our original array A and fill the sorted elements again from low to high, making our array merge-sorted.

```
void merge(int A[], mid,  low, high)
{
   int i, j, k , B[high+1];
   i = low;
   j = mid + 1;
   k = low;
   while (i <= mid && j <= high){
      if (A[i] < A[j]){
         B[k] = A[i];
         i++;
         k++;
      }
      else{
         B[k] = A[j];
         j++;
         k++;
      }
   }
   while (i <= mid){
      B[k] = A[i];
      k++;
      i++;
   }
   while (j <= high){
      B[k] = A[j];
      k++;
      j++;
   }
   for (int i = low; i <= high; i++)
      A[i] = B[i];
```

Copying all remaining elements from A to C

Copying all remaing elements from B to C

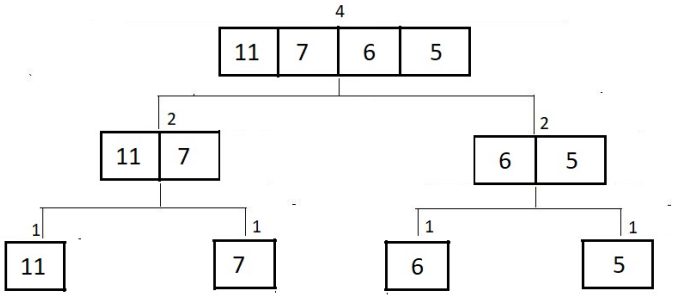
Copying elements back to A from B



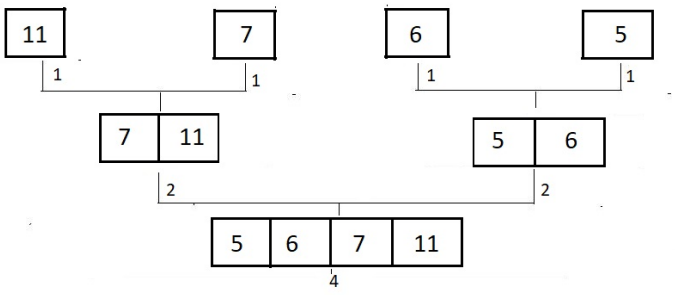
Merge sorted Array A

So that was our merging segment.

Whenever you receive an unsorted array, you break the array into fragments till the size of each subarray becomes 1. Let this be clearer via an illustration.



So, we divided the array until there are all subarrays of just length 1. Since any array/subarray of length 1 is always sorted, we just need to merge all these singly-sized subarrays into a single entity. Visit the merging procedure below.



And this is how our array got merge sorted. To achieve this divided merging and sorting, we create a recursive function merge sort and pass our array and the *low and high* index into it. This function divides the array into two parts: one from *low* to *mid* and another from *mid+1* to *high.* Then, it recursively calls itself passing these divided subarrays. And the resultant subarrays are sorted. In the next step, it just merges them. And that's it. Our array automatically gets sorted. Pseudocode for the merge sort function is illustrated below.

```
void MS(A[], low, high){
   int mid;
   if(low<high){
      mid = (low + high) /2;
      MS(A, low, mid);
      MS(A, mid+1, high);
      Merge(A, mid, low, high);
   }
}
```

- **Best Case**: O(nlogn)

  Condition: Merge Sort always divides the array into halves and merges them back together in O(nlogn) time, regardless of the initial arrangement of the data.

- **Worst Case**: O(nlogn)

  Condition: The algorithm still follows the divide-and-conquer approach, resulting in the same time complexity even in the worst case, such as when the array is sorted in reverse or random order.

- **Average Case**: O(nlogn)

  Condition: For random input arrays, the algorithm performs consistently in O(nlogn) time, as it is independent of the initial arrangement of the elements.