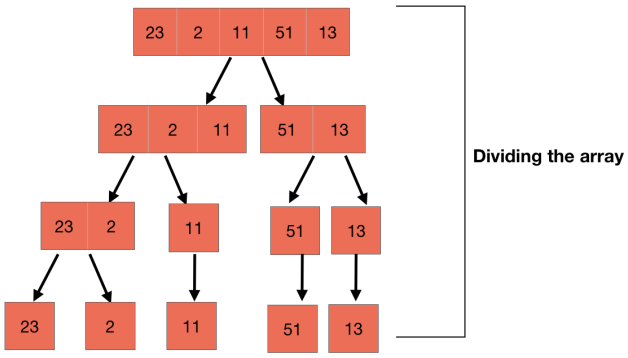
Merge Sort

Merge sort is the first algorithm we are going to study in Divide and Conquer. According to Divide and Conquer, it first divides an array into smaller subarrays and then merges them together to get a sorted array. The dividing part is the same as what we did in the previous chapter.

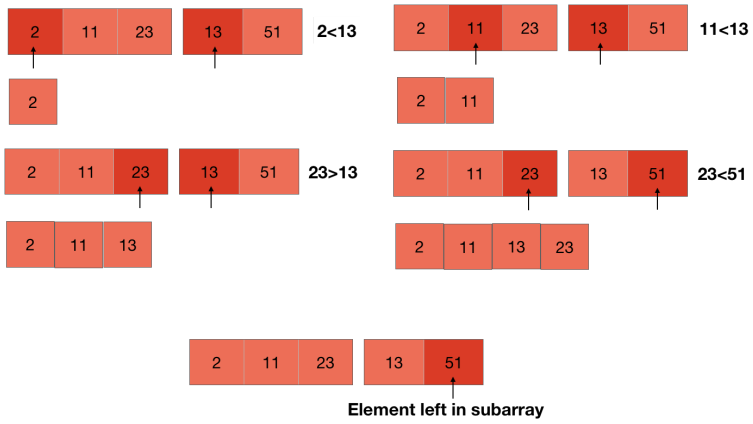


The real thing lies in the combining part. We combine the arrays back together in such a way that we get the sorted array. So, we will make two functions, one to break the arrays into subarrays and another to merge the smaller arrays together to get a sorted array. Let's start by making the first function which is similar to the function we made in the previous chapter.

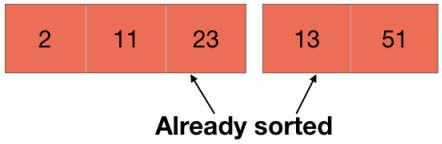
```
MERGE-SORT(A, start, end)
  if start < right
    middle = floor((start+end)/2)
    MERGE-SORT(A, start, middle)
    MERGE-SORT(A, middle+1, end)
```

With this code, we will be able to break our array into smaller subarrays, but now we want the subarrays to combine together and make a sorted array. For this purpose, let's focus on writing the merge function.

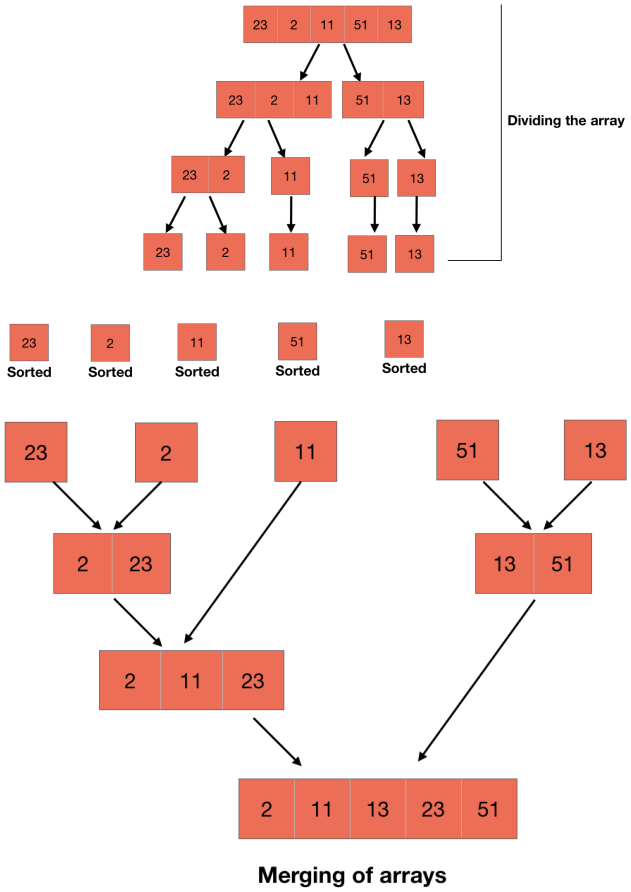
To merge the smaller arrays back together, we will start iterating over the arrays and putting the smaller element first to get a sorted array. This will be clear from the picture given below.



From the above picture, you can see that we are iterating over both the arrays simultaneously and comparing the elements and then putting the smaller element in the final array and then continuing the iteration. But one doubt can come in anyone's mind that both the arrays shown in the above picture are themselves sorted and then we are merging them together to get the final sorted array, so how are we going to get the two smaller arrays sorted?



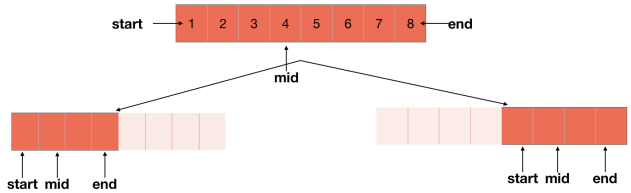
The answer is simple, we are breaking our arrays into smaller subarrays until the size of each subarray becomes 1 and any array with just a single element is already sorted in itself. So, we will start merging them together to get bigger sorted arrays.



Since we are clear with the entire idea of Merge Sort, so let's write code for the merge function.

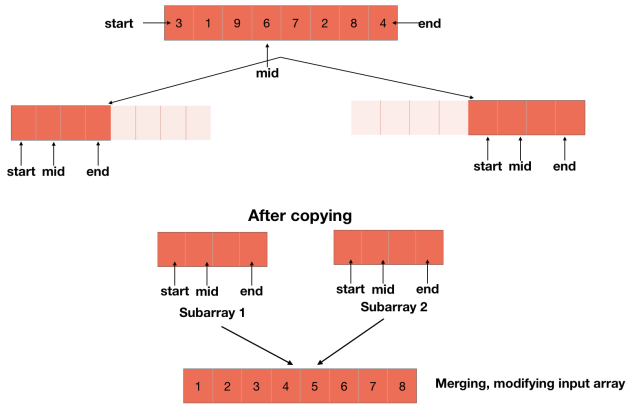
Code for Merge Sort

As shown in the picture above, the merge function is going to merge two arrays which are already sorted. However, these two arrays will not be entirely different arrays but a single array which will be separated by a variable into two arrays because this is how we are breaking our arrays.



So, our function to merge the arrays will take an array, starting index, ending index and the middle index from which the array is separated i.e., `MERGE(A, start, middle, end)`.

Now, we have to iterate over both the subarrays, compare the elements and put the smaller one into the final array. But we don't have any final array to work with, all we have is a single array and the indices. So, let's copy the subarrays into two temporary arrays and modify the initial array 'A' to make it sorted.



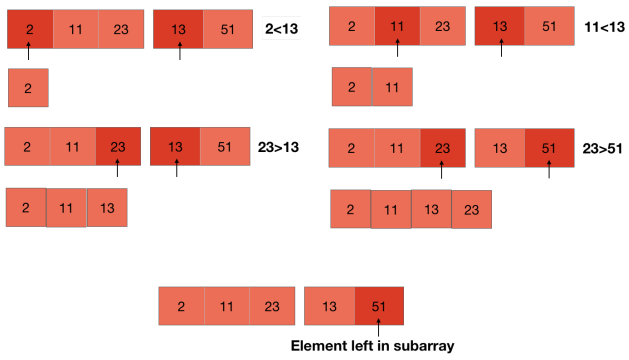
So, our first task is to make two temporary arrays.

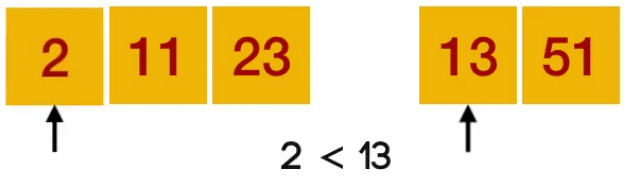
```
temp1 = A[start, middle]
temp2 = A[middle+1, end]
```

Now, we have to iterate over these arrays and compare the elements and then put the smaller elements into the bigger array.

```
i = 1
j = 1
k = start
while i <= temp1.length and j <= temp2.length
  if temp1[i] < temp2[j]
    A[k] = temp1[i]
    i=i+1
  else
    A[k] = temp2[j]
    j=j+1
  k=k+1
```

We are iterating over both the arrays with `while i <= temp1.length and j <= temp2.length` and then comparing the elements - `if temp1[i] < temp2[j]`. After that, we are putting the smaller element into the array first - `A[k] = temp1[i]` (if `temp1 has smaller element`) or `A[k] = temp2[j]` (if `temp2 has smaller element`) and then we are continuing our iteration with `i = i+1` or `j = j+1`, which ever element was put to the array. At last, `k = k+1` is making the increase for the iteration on the original array. The illustration for the same is given below.





One thing you can note from the above picture is that after this process, it might be possible that we are left with few elements in any of the subarrays. To handle this case, let's just copy the remaining elements to the array. Because these elements are already sorted and only the larger elements are going to be left, so direct copying is going to work for us.

```
while i < temp1.length
  A[k] = temp[i]
  i = i+1

while j < temp2.length
  A[k] = temp[j]
  j = j+1
```

The first `while` loop is iterating over the temp1 array and copying any leftover of the array. The second loop is doing the same thing but with the array temp2.

So, now we are ready to write the merge function.

```
MERGE(A, start, middle, end)
  temp1 = A[start, middle]
  temp2 = A[middle+1, end]
  i = 1
  j = 1
  k = start
  while i <= temp1.length and j <= temp2.length
    if temp1[i] < temp2[j]
      A[k] = temp1[i]
      i=i+1
    else
      A[k] = temp2[j]
      j=j+1
    k=k+1

  while i < temp1.length
    A[k] = temp[i]
    i = i+1

  while j < temp2.length
    A[k] = temp[j]
    j = j+1
```

Now, we have to use this MERGE function inside the MERGE-SORT function which is basically dividing the array into subarrays. Till now, we have developed the MERGE-SORT function as:

```
MERGE-SORT(A, start, end)
  if start < right
    middle = floor((start+end)/2)
    MERGE-SORT(A, start, middle)
    MERGE-SORT(A, middle+1, end)
```

We know that after the completion of the MERGE-SORT function, the function will sort any array passed to it and this is the entire point of this function. So, `MERGE-SORT(A, start, middle)` and `MERGE-SORT(A, middle+1, end)` are going to **sort the arrays** `A[start, middle]` and `A[middle+1, end]` respectively. And now we have to merge them together to get the bigger sorted array.

So, placing `MERGE(A, start, middle, end)` right after `MERGE-SORT(A, middle+1, end)` will complete the function for us.

```
MERGE-SORT(A, start, end)
  if start < right
    ...
    MERGE-SORT(A, middle+1, end)
    MERGE(A, start, middle, end)
```

But wait, we have basically assumed that `MERGE-SORT(A, start, middle)` will give us a sorted array `A[start, middle]` when the function was not even complete and our entire completion of the function is dependent on this assumption only. So, if the assumption is correct then the entire function is going to work correctly.

In other words, if we show that `MERGE-SORT` gives us a sorted array in the base case, then we can use the MERGE function afterward to generate bigger sorted arrays.

And we have already discussed that in the base case, MERGE-SORT is going to break the array into size of 1 and any array with single element is sorted in itself and thus MERGE-SORT is going to give us a sorted array in base case and after that, the MERGE function is going to combine them into bigger sorted arrays.

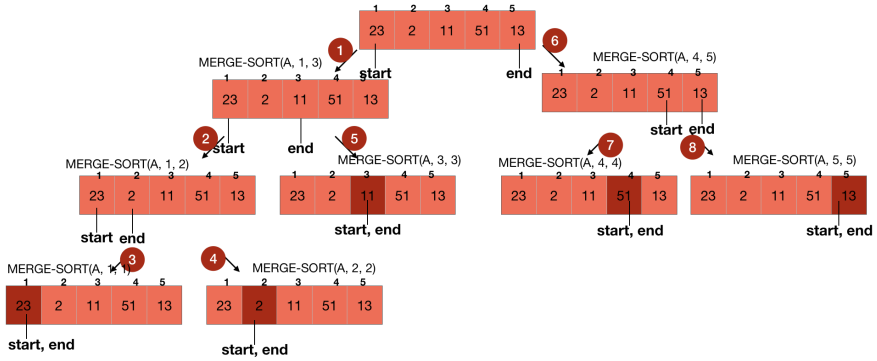
Sorted arrays in base case

Thus, the code for MERGE-SORT can be written as:

```
MERGE-SORT(A, start, end)
  if start < right
    middle = floor((start+end)/2)
    MERGE-SORT(A, start, middle)
    MERGE-SORT(A, middle+1, end)
    MERGE(A, start, middle, end)
```

C **Python** **Java**

```
#include <stdio.h>

void merge(int a[], int start, int middle, int end) {
  int size_of_temp1, size_of_temp2, i, j, k;

  //temporary arrays to copy the elements of subarray
  size_of_temp1 = (middle-start)+1;
  size_of_temp2 = (end-middle);

  int temp1[size_of_temp1], temp2[size_of_temp1];

  for(i=0; i<size_of_temp1; i++) {
    temp1[i] = a[start+i];
  }

  for(i=0; i<size_of_temp2; i++) {
    temp2[i] = a[middle+1+i];
  }

  i=0;
  j=0;
  k=start;

  while (i < size_of_temp1 && j < size_of_temp2) {
    if (temp1[i] < temp2[j]) {
      // filling the main array with the smaller element
      a[k] = temp1[i];
      i++;
    }
    else {
      // filling the main array with the smaller element
      a[k] = temp2[j];
      j++;
    }
    k++;
  }

  // copying leftovers
  while (i<size_of_temp1) {
    a[k] = temp1[i];
    k++;
    i++;
  }

  while (j<size_of_temp2) {
    a[k] = temp2[j];
    k++;
    j++;
  }
}

void merge_sort(int a[], int start, int end) {
  if (start < end) {
    int middle = (start+end)/2;
    merge_sort(a, start, middle);
    merge_sort(a, middle+1, end);
    merge(a, start, middle, end);
  }
}

int main() {
  int a[] = {4, 8, 1, 3, 10, 9, 2, 11, 5, 6};
  merge_sort(a, 0, 9);

  //printing array
  int i;
  for(i=0; i<10; i++) {
    printf("%d ",a[i]);
  }
  printf("\n");
  return 0;
}
```



Analysis of Merge Sort

Let's have a look at the MERGE function first. It just iterates over the arrays and the arrays can have at most size of n. In the rest of the part, we are just comparing and assigning values and they are constant time processes. Thus, the MERGE function has a running time of $\Theta(n)$.

Now, the analysis of the MERGE-SORT function is simple and we have already done this kind of analysis in previous chapters. The MERGE-SORT function is breaking the problem size of n into two subproblems of size $\frac{n}{2}$ each. The comparison (`if start < right`) and calculation of middle (`middle = floor((start+end)/2)`) are constant time taking processes. Also, we deduced that the MERGE function is $\Theta(n)$. So, we can write the overall running time of MERGE-SORT function as:

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) + \Theta(1)$$

We have already dealt with the same function in Recurrence, Let's Iterate, Now the Recursion and Master's Theorem chapters and we know that it is going to take $\Theta(n \lg n)$ time.

So, this was a pure implementation of divide and conquer technique into an algorithm. Let's proceed to the further chapters and learn about more algorithms which use this technique.

> “ Luck is great, but most of life is hard work. ”
>
> - Iain Duncan Smith

PREV (/course/algorithms-divide-and-conquer/) (/course/algorithms-quicksort/) NEXT

Further Readings

- Merge Sort (/blog/article/merge-sort/)