The Merge Sort Algorithm

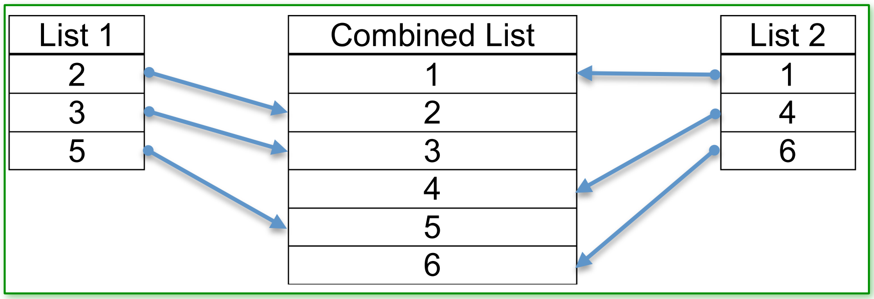
The Merge Sort algorithm is often referred to as a “Divide and Conquer” algorithm. In a sense, it is made up of two algorithms, one that divides, and the other conquers! First let us examine the second of the two algorithms and why it is so useful.

The conquer algorithm efficiently combines two already sorted lists into a single sorted list. The algorithm simply goes through each list starting with the first element, determines which list has the smaller value, and puts that element into the next spot in the combined list. It continues comparing the next element in each list until all elements have been combined into one unified ordered list.

Let's begin by looking at short example:

The algorithm to combine to lists that are already sorted is fairly simple breaks down into two primary parts. Before looking at each part, here is the brief code to set up the arrays to work with:

Lines **01** and **02** create two arrays, `list1` and `list2` with three elements each.



List 1
2
3
5

Combined List
1
2
3
4
5
6

List 2
1
4
6

```
01 int[] list1={2,3,5};
02 int[] list2={1,4,6};
03 int[] combinedList=new int[list1.length+list2.length];
04 int list1Index=0, list2Index=0;
05 int combinedListIndex=0;
```

Line **03** creates an empty array `combinedList` with 6 elements to fill by combining the two smaller lists.

Lines **04** and **05** create index variables that are initialized to 0 for each list. These variables will keep track of which element of each list will be the next to be processed as the algorithm continues.

The variables `list1Index` and `list2Index` indicate the next array element to be processed, so the loop in line **01** will continue until one of these variables indicates that the end of its related list has been reached.

```
01 while(list1Index<list1.length && list2Index<list2.length) {
02  if(list1[list1Index]<list2[list2Index]) {
03   combinedList[combinedListIndex]=list1[list1Index];
04   list1Index++;
05  } else {
06   combinedList[combinedListIndex]=list2[list2Index];
07   list2Index++;
08  }
09  combinedListIndex++;
10 }
```

During each iteration of the loop, line **02** compares the values pointed to by `list1Index` and `list2Index`. The smaller of the two is copied into the `combinedList` array at the `combinedListIndex` position by either line **03** or line **06**. Whichever list the value was copied from has it's index variable incremented so that the next comparison will be on the next value in that list by line **04** or line **07**. Finally, regardless of which list the value was copied from, line **09** increments the `combinedListIndex` so the next value copied will go into the next empty position in the `combinedList` array.

The algorithm will leave one of the two lists containing some elements that have yet to be added to the `combinedList` array so the second part of the algorithm will take care of that.

Of the two while loops on the below, only one will actually execute because after the code above has run, one of the list*N*Index variables will be equal to it's associated list's length and therefore not execute.

```
while(list1Index<list1.length) {
  combinedList[combinedListIndex]=list1[list1Index];
  list1Index++;
  combinedListIndex++;
}
while(list2Index<list2.length) {
  combinedList[combinedListIndex]=list2[list2Index];
  list2Index++;
  combinedListIndex++;
}
```

The other while loop simply copies the remaining elements in the list so that when it is done, `combinedList` represents the fully ordered and integrated combination of `list1` and `list2`.

The code below shows the complete JAVA program from the example along with a final loop to output the results to the console:

```
public class CombineLists {
  public static void main(String[] args) {
    int[] list1={2,3,5};
    int[] list2={1,4,6};
    int[] combinedList=new int[list1.length+list2.length];
    int list1Index=0, list2Index=0, combinedListIndex=0;
    while(list1Index<list1.length && list2Index<list2.length) {
      if(list1[list1Index]<list2[list2Index]) {
        combinedList[combinedListIndex]=list1[list1Index];
        list1Index++;
      } else {
        combinedList[combinedListIndex]=list2[list2Index];
        list2Index++;
      }
      combinedListIndex++;
    }
    while(list1Index<list1.length) {
      combinedList[combinedListIndex]=list1[list1Index];
      list1Index++;
      combinedListIndex++;
    }
    while(list2Index<list2.length) {
      combinedList[combinedListIndex]=list2[list2Index];
      list2Index++;
      combinedListIndex++;
    }
    for (int i=0; i<combinedList.length; i++) {
      System.out.println("["+i+"]="+combinedList[i]);
    }
  }
}
```

Console Output:

```
[0]=1
[1]=2
[2]=3
[3]=4
[4]=5
[5]=6
```

This completes the “merge” portion of the Merge Sort.

Once again, this is only one half of the overall algorithm. This sub-algorithm of the overall algorithm can itself be said to have two parts.

In part 1, two lists that are each in sorted order are integrated in order one element at a time until one of the lists runs out of elements.

At that point, part 2 takes over, copying the remaining elements from the other list in order into the combined list.

However, to be useful in the complete Merge Sort algorithm, the process will need to be performed as a method. Furthermore, it will be much more useful if it can integrate two portions of an array back into the same array.

This means that the algorithm above will operate on a source list which has within it two sub-portions, analogous to our two lists in the example above, which are each made up of elements in numerical ascending order.

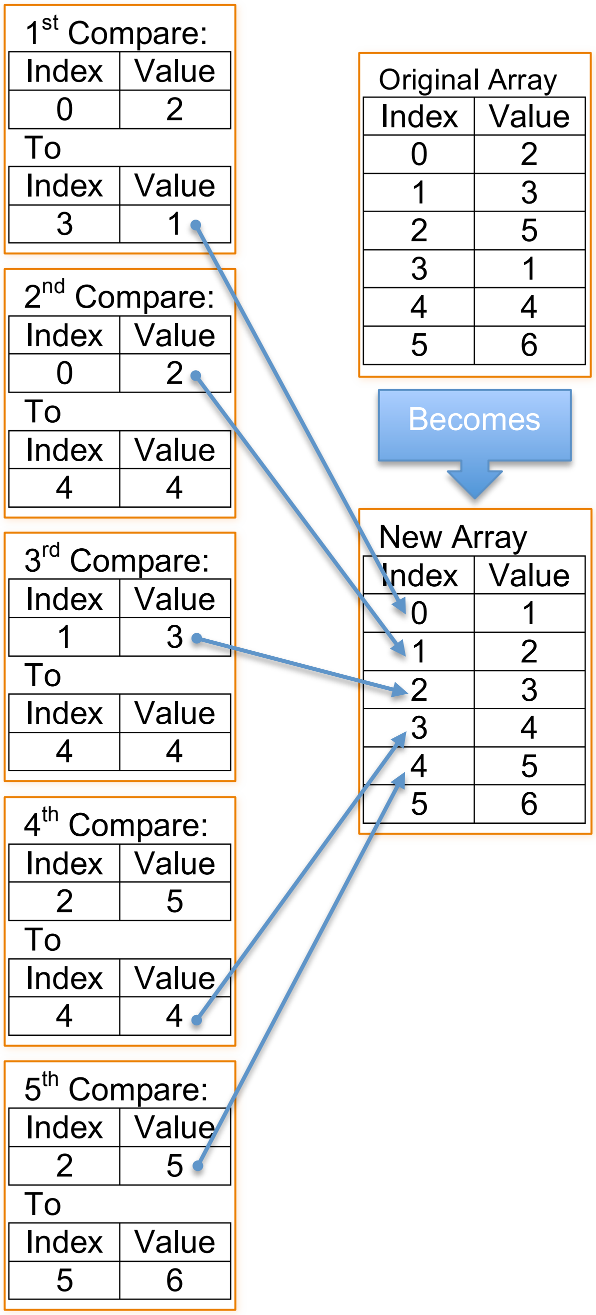
These two portions of the overall list must be integrated in order into a new temporary array and then the elements of the temporary array must be copied back into the original array.

This process is somewhat complex so let's examine each part of the overall algorithm the method must complete:

- Step 1: Create a temporary Array representing the elements of the source array to be merged.
- Step 2: Merge the elements of the temporary array back into the source array, until one of the sub-sections runs out of elements to be merged.
- Step 3: Copy all the remaining elements in the temporary array that did not run out of elements back into the source array.

The method is now complete.

Before looking at JAVA code to accomplish this, examine the process visually:



1st Compare:

Index	Value
0	2

To

Index	Value
3	1

2nd Compare:

Index	Value
0	2

To

Index	Value
4	4

3rd Compare:

Index	Value
1	3

To

Index	Value
4	4

4th Compare:

Index	Value
2	5

To

Index	Value
4	4

5th Compare:

Index	Value
2	5

To

Index	Value
5	6

Original Array

Index	Value
0	2
1	3
2	5
3	1
4	4
5	6

New Array

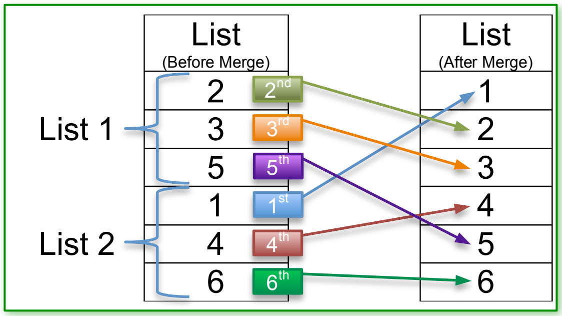
Index	Value
0	1
1	2
2	3
3	4
4	5
5	6

In this example, a single list is split into two "virtual" lists.

As you can see the first three elements starting at index 0, and the next three elements, starting at index 3 are each in ascending numerical order.

The algorithm that we are writing will integrate these two sub-lists into a single list that is then completely sorted into ascending order.

The process is the same as before, comparing the first elements in each list, then copying the lower value into the ordered list. Another way to look at this is a series of comparisons.



Step 1: In the first iteration of a loop within the algorithm the elements at index 0 and index 3 are compared. Since the element at index 3 has the lower value, it is copied into the new list in the first position.

Step 2: In the next iteration, the elements at indexes 0 and 4 are compared because the value from index 3 has already been used. The element at index 0 has the lower value so it is copied to index 1 of the new list.

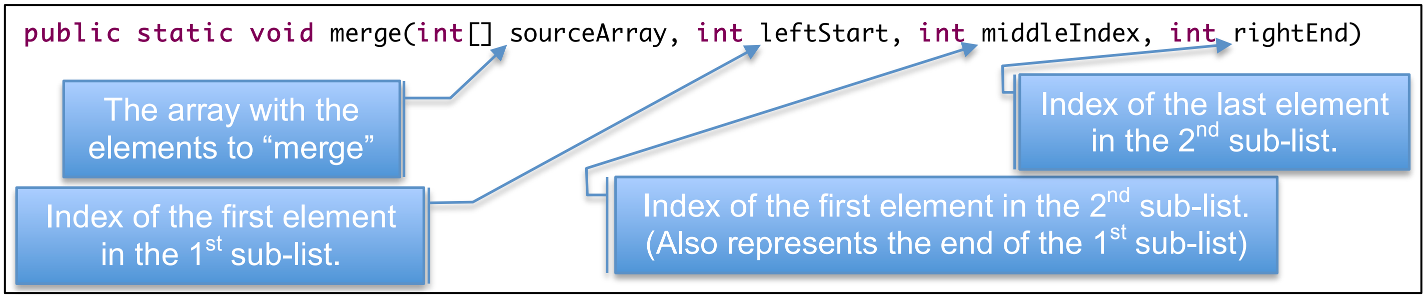
Steps 3-5: The process continues as show in the diagram, copying the next lowest element until one list has no more elements to copy.

It is at this point that the algorithm moves on to copying the remaining elements from the list that has not run out of elements.

Each of the remaining elements is copied. In this example there is only the final element at index 5 to copy to the final position.

When writing the actual code, it is important to understand the goals for the method. The result of this method will be to have the elements specified by the parameters of the method call merged **back** into the source array. It is also important to remember that the method will generally only be “merging” part of the source array, not the entire array as is illustrated in the example above.

We will begin by looking at the method header:



```
public static void merge(int[] sourceArray, int leftStart, int middleIndex, int rightEnd)
```

The array with the elements to “merge”

Index of the first element in the 1st sub-list.

Index of the first element in the 2nd sub-list. (Also represents the end of the 1st sub-list)

Index of the last element in the 2nd sub-list.

The `merge` method step-by-step:
Step 1: Input Parameters–

```
public static void merge(int[] sourceArray, int leftStart, int middleIndex, int rightEnd)
```

Part of `sourceArray` contains two lists that need to be integrated in ascending numerical order using the merge algorithm. The index of the first list is represented by `leftStart`. The index of the beginning of the second list is represented by `middleIndex`. The first list must end before the second list starts, so `middleIndex` also represents the upper limit to the first list. Finally, the index of the end of the second list is given by `rightEnd`.

```
01if (middleIndex>=sourceArray.length) {
02  return;
03}
04if (rightEnd>=sourceArray.length) {
05  rightEnd=sourceArray.length;
06}
07int[] tempArray=new int[sourceArray.length];
08for (int i=leftStart; i<rightEnd; i++) {
09  tempArray[i]=sourceArray[i];
10}
```

Step 2: Getting Ready–

In order to make sure that the method doesn't corrupt the `sourceArray` once the merge begins; a verification check is done to make sure that the second list falls within the bounds of the array in lines `01` to `03`.

If `rightEnd` falls outside the bounds of the array then lines `04` to `06` sets it to the actual end of the array.

Line `07` creates `tempArray` so that the elements to be merged from `sourceArray` can be copied in lines `08` to `10` into it using the same indexes for each element.

Step 3: The Merge Loop– Lines 01 to 03 initialize the variable leftIndex with the leftStart value, rightIndex with the middleIndex value, and the index of where to place the next lowest value back into the sourceArray into the variable currentCopyIndex from leftStart.

```
01 int leftIndex=leftStart;
02 int rightIndex=middleIndex;
03 int currentCopyIndex=leftStart;
04 while(leftIndex<middleIndex && rightIndex<rightEnd) {
05  if (tempArray[leftIndex]<tempArray[rightIndex]) {
06    sourceArray[currentCopyIndex]=tempArray[leftIndex];
07    leftIndex++;
08  } else {
09    sourceArray[currentCopyIndex]=tempArray[rightIndex];
10    rightIndex++;
11  }
12  currentCopyIndex++;
13 }
```

leftIndex represents the current index of the next position to compare from the 1st sub-list, rightIndex the current index of the next position to compare from the 2nd sub-list, and currentCopyIndex is where the next value should be placed back into the sourceArray.

The while loop on line 04 iterates so long as leftIndex is less than middleIndex (which represents the end of the 1st sub-list) and rightIndex is less than rightEnd (which represents the end of the 2nd sub-list).

Within the loop, line 05 compares the values stored in tempArray at the indexes leftIndex and rightIndex. If the value at leftIndex is lower, then line 06 copies the value from tempArray at leftIndex back into sourceArray at the index stored in currentCopyIndex, and then line 07 increments leftIndex so that the next compare happen to the next element in that part of the list. If the value at leftIndex is **not** lower, then line 09 copies the value from tempArray at rightIndex back into sourceArray at the index stored in currentCopyIndex, and then line 10 increments rightIndex so that the next compare happens to the next element in that part of the list. Finally line 12 increments currentCopyIndex. Remember that the loop in line 07 will end when either index variable reaches the end of its associated sub-list.

Step 4: Copy the Remaining Elements–

Only one of the two while loops will actually execute because the one of the index variables will have reached the end of it's associated sub-list.

The other while loop will continue to copy any remaining elements from the other sub-list from tempArray back into sourceArray.

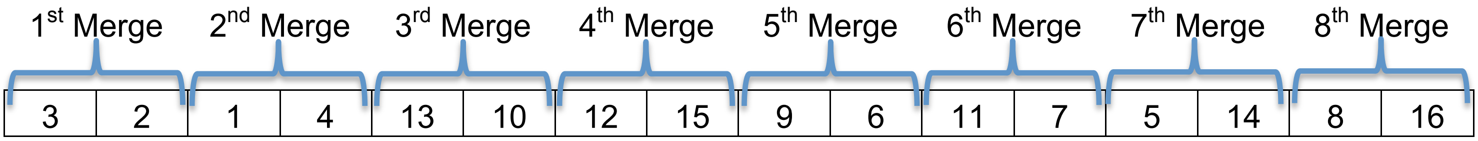
```
while (leftIndex<middleIndex) {
  sourceArray[currentCopyIndex]=tempArray[leftIndex];
  leftIndex++;
  currentCopyIndex++;
}
while (rightIndex<rightEnd) {
  sourceArray[currentCopyIndex]=tempArray[rightIndex];
  rightIndex++;
  currentCopyIndex++;
}
```

Now that we have a functional merge method (our “conquer” half of the “divide and conquer” solution) it is time to address the “divide” part of the algorithm.

Remember that the merge method only merges lists that are made up of elements that are **already** in sorted ascending order. The divide part of our algorithm will guarantee that we only merge lists that are already sorted by combining lists which start with only one element in them (by definition, these two lists must each be in ascending order of course!). The resulting list will have two elements in sorted order so that list can then be combined with another list in ever growing sizes until the whole list has been sorted successfully.

Let's examine this process in a sample list:

First the list is broken up into sub-lists of one element each, and the two lists are "merged" with the process repeated for each pair of elements in the list:

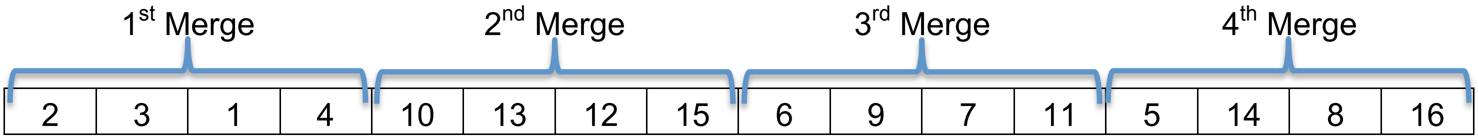


3	2	1	4	13	10	12	15	9	6	11	7	5	14	8	16

List After Merges

2	3	1	4	10	13	12	15	6	9	7	11	5	14	8	16

The process is repeated by this time each sub-list has two elements, so the resulting merged list has four elements:

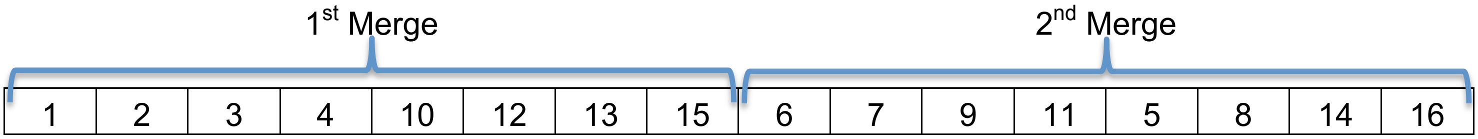


2	3	1	4	10	13	12	15	6	9	7	11	5	14	8	16

List After Merges

1	2	3	4	10	12	13	15	6	7	9	11	5	8	14	16

Notice that each iteration of the process, the size of the merged list doubles. In this next iteration each sub-list has four elements and the resulting merged list will have 8 elements:

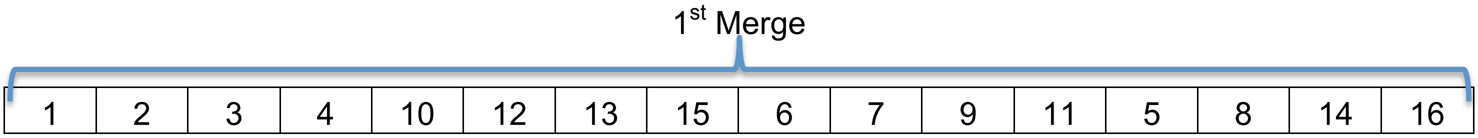


1	2	3	4	10	12	13	15	6	7	9	11	5	8	14	16

List After Merges

1	2	3	4	10	12	13	15	5	6	7	8	9	11	14	16

In the next iteration the list will be completely sorted:



1	2	3	4	10	12	13	15	6	7	9	11	5	8	14	16

List After Merges

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

And the list is sorted!

Writing the JAVA code:

This portion of the algorithm will be calling the `merge` method so recall the method signature is:

```
public static void merge(int[] sourceArray, int leftStart, int middleIndex, int rightEnd)
```

Now all that remains in to write the JAVA code that implements this portion of the Merge Sort algorithm.

Nested for loops with a call to the `merge` method will complete the algorithm.

```
for (int blockSize=1; blockSize<arr.length; blockSize*=2) {
 for (int leftIndex=0; leftIndex<arr.length; leftIndex+=2*blockSize) {
  merge(arr, leftIndex, leftIndex+blockSize, leftIndex+2*blockSize);
 }
}
```

The outer loop tracks the size of each sub-list to be merged using `blockSize` which doubles with each iteration.

The inner loop calculates the start position for each merge using `blockSize`. The merge method gets called passing the inner loop's `leftIndex` as the first parameter (`leftStart`), `leftIndex+blockSize` as the second parameter (`middleIndex`), and `leftIndex+2*blockSize` as the final parameter (`rightEnd`).

Let's examine the values for the sample run from the previous page to see how these loops would call the merge method:

blockSize	leftIndex (leftStart)	leftIndex+blockSize (middleIndex)	leftIndex+2*blockSize (rightEnd)	merge(arr, leftIndex, leftIndex+blockSize, leftIndex+2*blockSize);
1				
	0	1	2	merge(arr, 0, 1, 2);
	2	3	4	merge(arr, 2, 3, 4);
	4	5	6	merge(arr, 4, 5, 6);
	6	7	8	merge(arr, 6, 7, 8);
	8	9	10	merge(arr, 8, 9, 10);
	10	11	12	merge(arr, 10, 11, 12);
	12	13	14	merge(arr, 12, 13, 14);
	14	15	16	merge(arr, 14, 15, 16);
2				
	0	2	4	merge(arr, 0, 2, 4);
	4	6	8	merge(arr, 4, 6, 8);
	8	10	12	merge(arr, 8, 10, 12);
	12	14	16	merge(arr, 12, 14, 16);
4				
	0	4	8	merge(arr, 0, 4, 8);
	8	12	16	merge(arr, 8, 12, 16);
8				
	0	8	16	merge(arr, 0, 8, 16);

All that remains is to put it all together into one JAVA program!

Here is the complete MergeSort.java program with sample data:

```
public class MergeSort {
  public static void main(String[] args) {
    int[] arr={3,2,1,4,13,10,12,15,9,6,11,7,5,14,8,16};
    System.out.println("Before:");
    listArray(arr);
    mergeSort(arr);
    System.out.println("After:");
    listArray(arr);
  }
  public static void listArray(int[] arr) {
    System.out.print("{");
    for (int i=0; i<arr.length-1; i++) {
      System.out.print(arr[i]+", ");
    }
    System.out.println(arr[arr.length-1]+"}");
  }
  public static void mergeSort(int[] arr) {
    for (int blockSize=1; blockSize<arr.length; blockSize*=2) {
      for (int leftIndex=0; leftIndex<arr.length; leftIndex+=2*blockSize) {
        merge(arr, leftIndex, leftIndex+blockSize, leftIndex+2*blockSize);
      }
    }
  }
  public static void merge(int[] sourceArray, int leftStart, int middleIndex, int rightEnd) {
    if (middleIndex>=sourceArray.length) {
      return;
    }
    if (rightEnd>=sourceArray.length) {
      rightEnd=sourceArray.length;
    }
    int[] tempArray=new int[sourceArray.length];
    for (int i=leftStart; i<rightEnd; i++) {
      tempArray[i]=sourceArray[i];
    }
    int leftIndex=leftStart;
    int rightIndex=middleIndex;
    int currentCopyIndex=leftStart;
    while(leftIndex<middleIndex && rightIndex<rightEnd) {
      if (tempArray[leftIndex]<tempArray[rightIndex]) {
        sourceArray[currentCopyIndex]=tempArray[leftIndex];
        leftIndex++;
      } else {
        sourceArray[currentCopyIndex]=tempArray[rightIndex];
        rightIndex++;
      }
      currentCopyIndex++;
    }
    while (leftIndex<middleIndex) {
      sourceArray[currentCopyIndex]=tempArray[leftIndex];
      leftIndex++;
      currentCopyIndex++;
    }
    while (rightIndex<rightEnd) {
      sourceArray[currentCopyIndex]=tempArray[rightIndex];
      rightIndex++;
      currentCopyIndex++;
    }
  }
}
```

Console Output:

```
Before:
{3, 2, 1, 4, 13, 10, 12, 15, 9, 6, 11, 7, 5, 14, 8, 16}
After:
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16}
```