SCJ2013 Data Structure & Algorithms

Merge Sort

Nor Bahiah Hj Ahmad & Dayang
Norhayati A. Jawawi

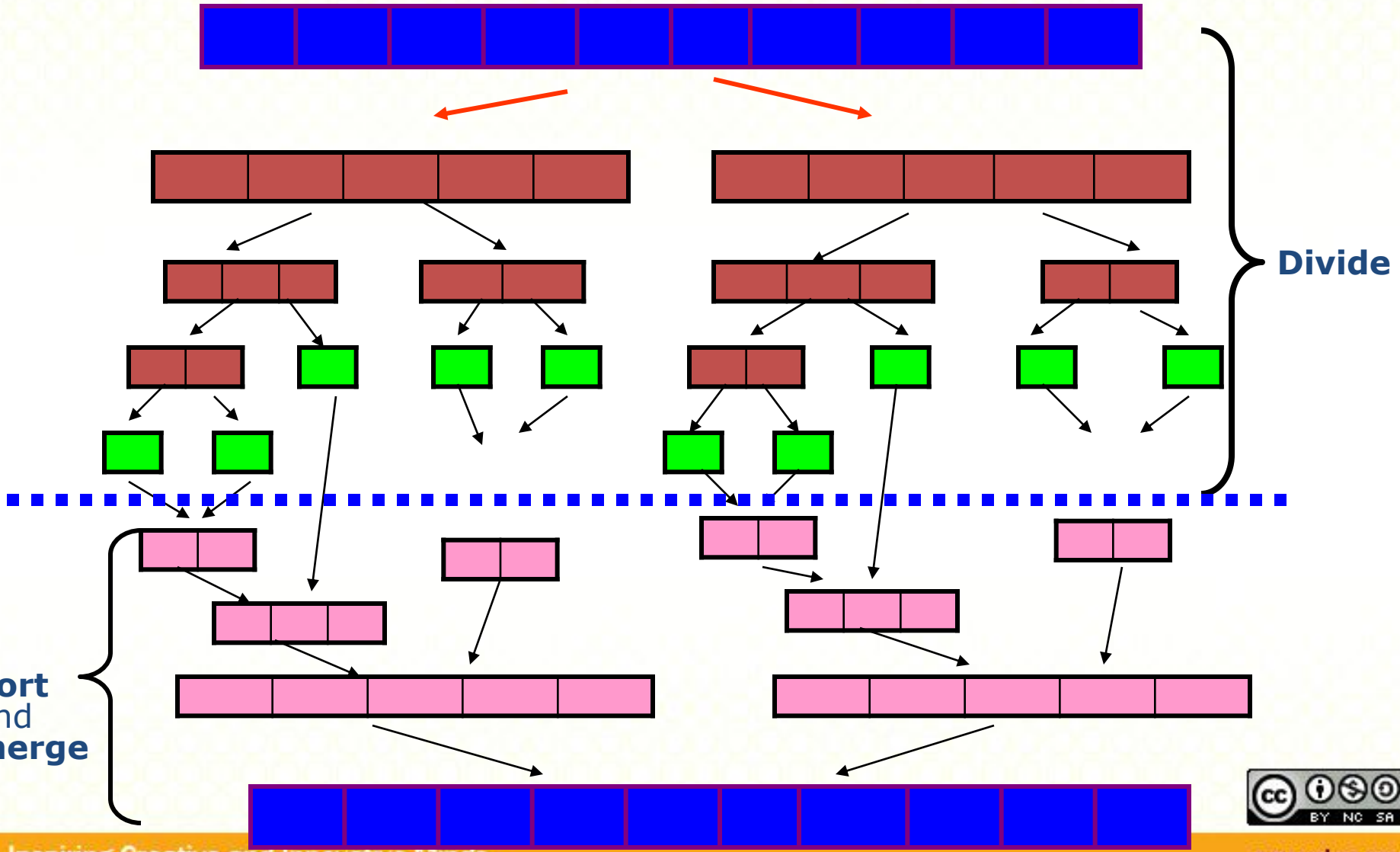
Merge Sort

- Merge Sort applies **divide** and **conquer** strategy.

- Three main steps in Merge Sort algorithm:
  - **Divide** an array into halves until only one piece left
  - **Sort** each half
  - **Merge** the sorted halves into one sorted array

- A recursive sorting algorithm
- Performance is independent of the initial order of the array items

Merge Sort Operation



Merge Sort Implementation

- `MergeSort()` function
  - A **Recursive** function that **divide** the array into pieces until each piece contain only one item.
  - The small pieces are **merged** into larger sorted pieces until one sorted array is achieved.

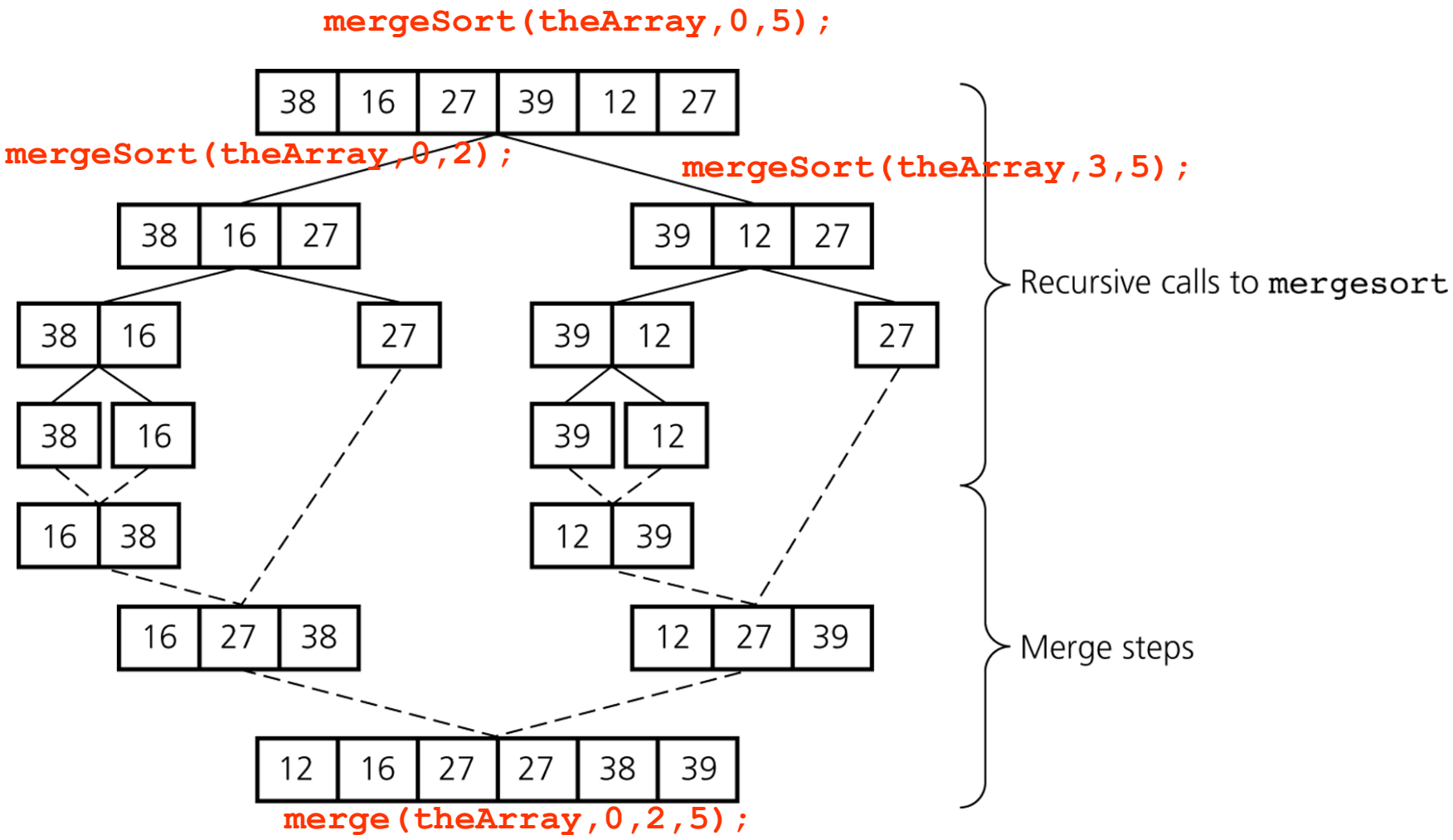
mergeSort() function

```
void mergeSort(DataType theArray[],int first,int last)
{ if (first < last)
  {  // sort each half
     int mid = (first + last)/2;
     // index of midpoint
     // sort left half theArray[first..mid]
      mergesort(theArray, first, mid);
      // sort right half theArray[mid+1..last]
      mergesort(theArray, mid+1, last);
      // merge the two halves
      merge(theArray, first, mid, last);
}  // end if
}  // end mergesort
```

divide the array into pieces

small pieces are **merged**

mergeSort [38 16 27 39 12 27]



A mergesort of an array of six integers

Merge Sort Implementation

- `Merge()` function
  - **Compares** an item into one half of the array with item in the other half of the array and
  - **Moves** the smaller item into temporary array.
  - Then, the **remaining** items are simply **moved** to the temporary array.
  - The temporary array is copied back into the original array.

merge() function

```
const int MAX_SIZE = maxNmbrItemInArry;
void merge(DataType theArray[],
           int first, int mid, int last)
{  DataType tempArray[MAX SIZE];  // temp array
   int first1 = first;  // first subarray begin
   int last1  = mid;    // end of first subarray
   int first2 = mid + 1; // secnd subarry begin
   int last2  = last;   // end of secnd subarry
```

Duplicate the positions

```
  // while both subarrays are not empty,
  // copy the smaller item into the
  // temporary array
    int index = first1;
  // next available location in tempArray
  for (; (first1 <= last1) && (first2 <=
   last2); ++index)
  {if (theArray[first1] < theArray[first2])
     {  tempArray[index] = theArray[first1];
        ++first1;      }
     else
     {  tempArray[index] = theArray[first2];
        ++first2;      }
   } // end if
```

Moves the smaller item into temporary array

merge() function

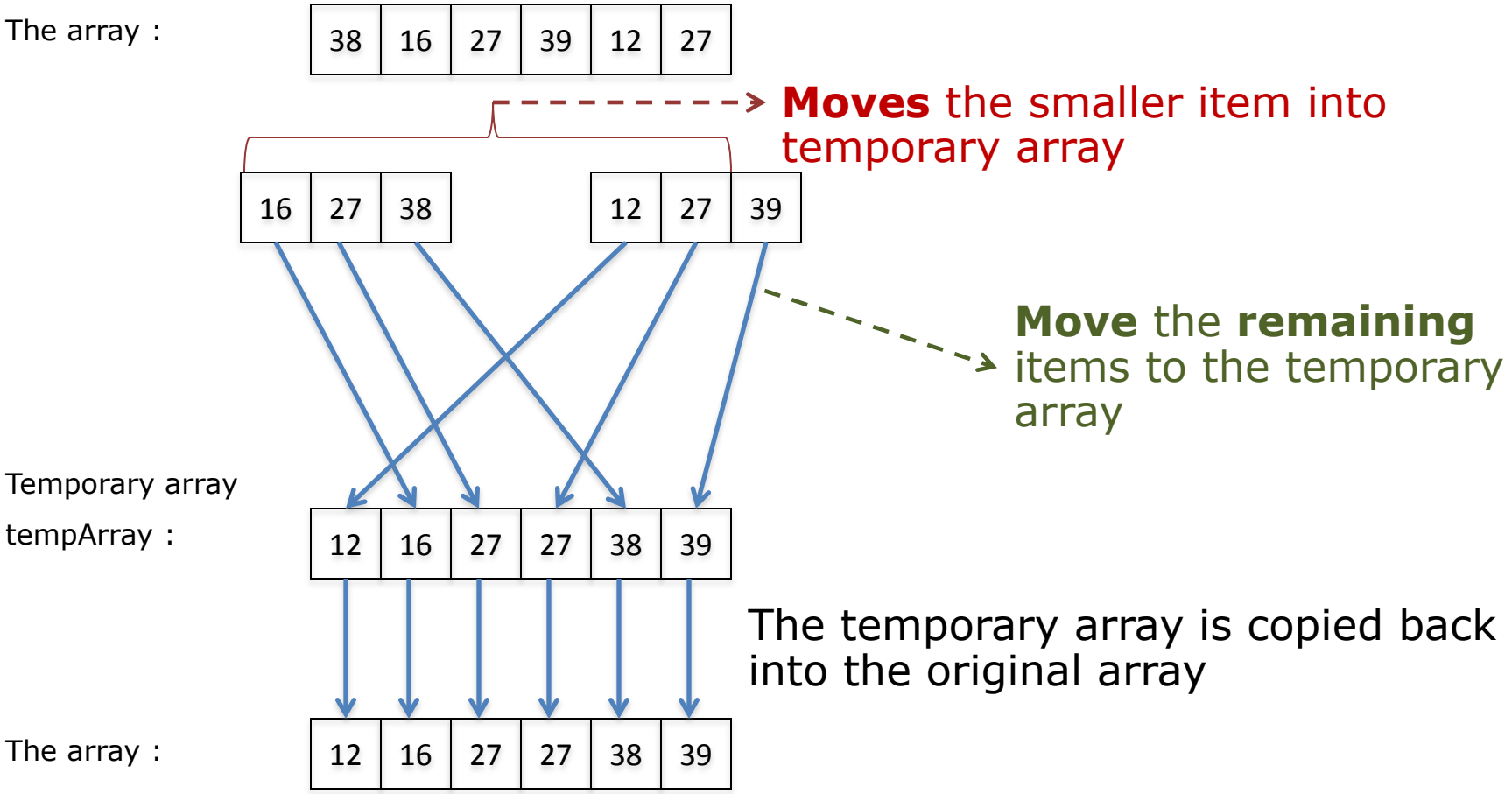
```
for (; first1 <= last1;
       ++first1, ++index)
  tempArray[index] =
        theArray[first1];
// finish off the second
// subarray, if necessary

for (; first2 <= last2;
       ++first2, ++index)
   tempArray[index] =
        theArray[first2];
```

move the **remaining** items to the temporary array

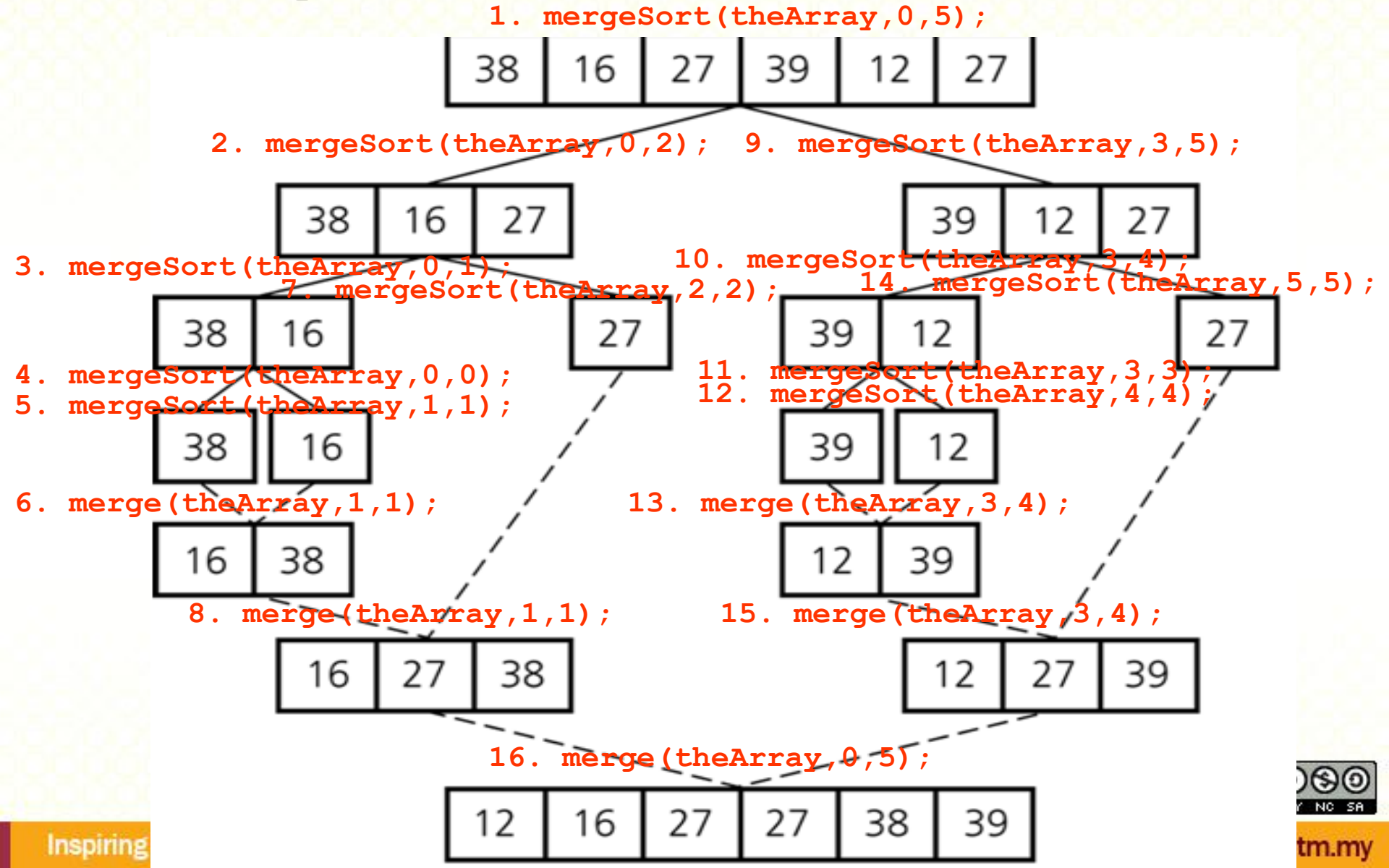
```
// copy the result back
// into the original
//  array
for (index = first;
  index <= last; ++index)
  theArray[index] =
       tempArray[index];
} // end merge function
```

The temporary array is copied back into the original array

Merge Sort Operation



mergeSort [38 16 27 39 12 27]



mergeSort [38 16 27 39 12 27]

The Execution Result

```
Unsorted data [38 16 27 39 12 27 ]

Content of divided sublist with first=0 & last=5 [38 16 27 39 12 27 ]
Content of divided sublist with first=0 & last=2 [38 16 27 ]
Content of divided sublist with first=0 & last=1 [38 16 ]
Content of divided sublist with first=0 & last=0 [38 ]
Content of divided sublist with first=1 & last=1 [16 ]
Content of merged list with first=0 & last=1 [16 38 ]
Content of divided sublist with first=2 & last=2 [27 ]
Content of merged list with first=0 & last=2 [16 27 38 ]
Content of divided sublist with first=3 & last=5 [39 12 27 ]
Content of divided sublist with first=3 & last=4 [39 12 ]
Content of divided sublist with first=3 & last=3 [39 ]
Content of divided sublist with first=4 & last=4 [12 ]
Content of merged list with first=3 & last=4 [12 39 ]
Content of divided sublist with first=5 & last=5 [27 ]
Content of merged list with first=3 & last=5 [12 27 39 ]
Content of merged list with first=0 & last=5 [12 16 27 27 38 39 ]

Sorted data [12 16 27 27 38 39 ]
Press any key to continue . . .
```

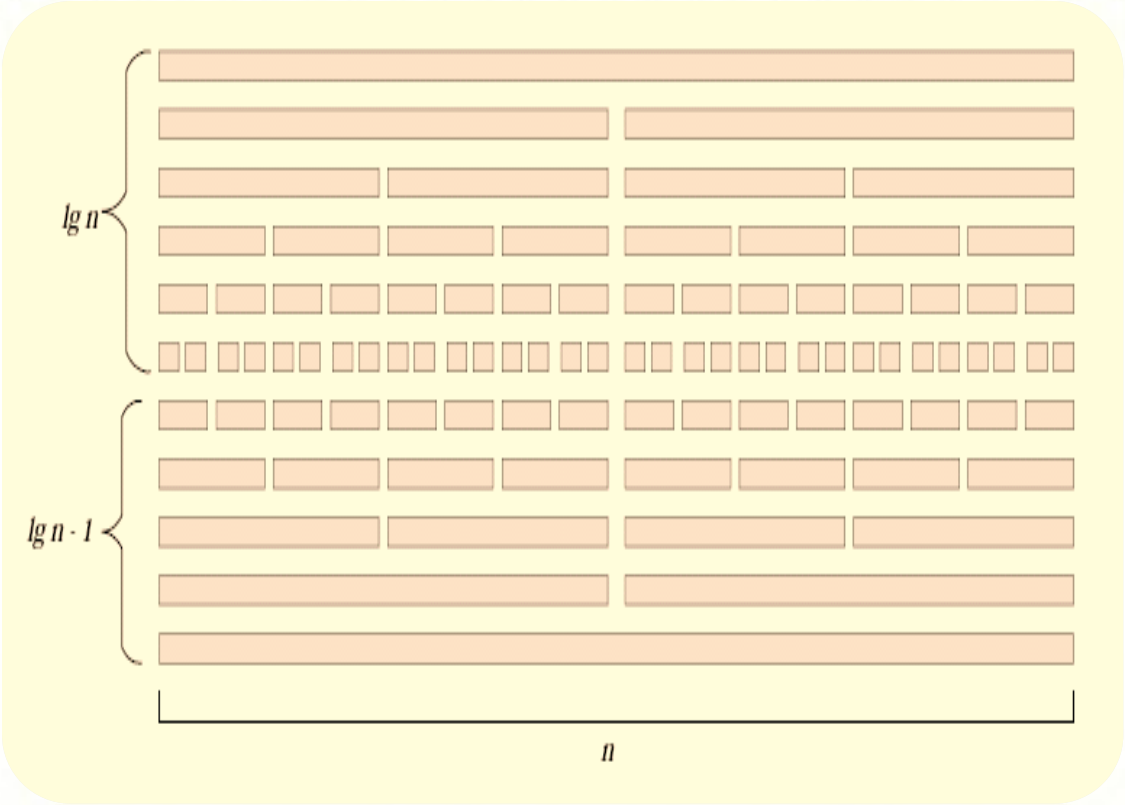
Merge Sort Analysis

- The list is **always divided into two balanced** list (or almost balanced for odd size of list)
- The number of calls to repeatedly divide the list until there is one item left in the list is:

$$n + 2\frac{n}{2} + 4\frac{n}{4} + 8\frac{n}{8} + 16\frac{n}{16} + \ldots\ldots x\frac{n}{x}$$

- Assuming that the left segment and the right segment of the list have the equal size (or almost equal size), then $x \approx \lg n$. The number of iteration is approximately $n \lg n$

Merge Sort Analysis



The same number of repetition is needed to sort and merge the list.

Thus, as a whole the number of steps needed to sort data using merge sort is $2n \lg n$, *which is*

$O(n \lg n)$.

Mergesort

- Analysis
  - Worst case: O(n * $\log_2 n$)
  - Average case: O(n * $\log_2 n$)
    - Performance is independent of the initial order of the array items
- Advantage
  - Mergesort is an extremely fast algorithm
- Disadvantage
  - Mergesort requires a second array (temporary array) as large as the original array

References

1. Frank M. Carano, Janet J Prichard. "*Data Abstraction and problem solving with C++" Walls and Mirrors*. 5th edition (2007). Addision Wesley.
2. Nor Bahiah et al. *Struktur data & algoritma menggunakan C++. Penerbit UTM, 2005.*