# Sorting Algorithms

**Definitions**

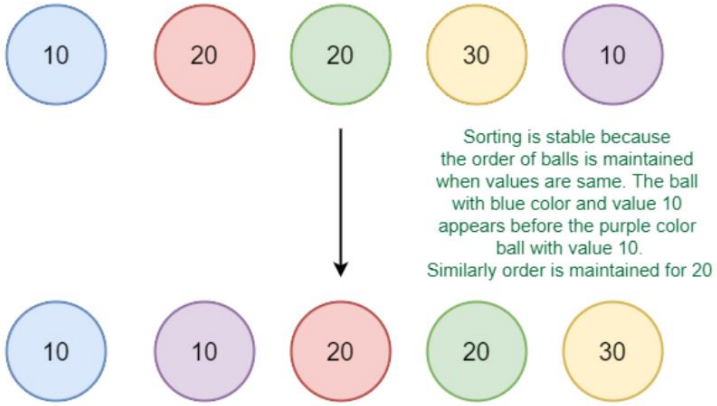
- Sorting is a process that organizes a collection of data into either ascending or descending order.

- Internal Sort – all of the data are held in primary storage during the sorting process (requires that the collection of data fit entirely in the computer's main memory).

- External sort – uses primary storage for the data currently being sorted and secondary storage for any data that will not fit in primary memory (We can use this sort when the collection of data cannot fit in the computer's main memory all at once but must reside in secondary storage such as on a disk).

- An **in-place algorithm** is an algorithm that does not need an extra space and produces an output in the same memory that contains the data by transforming the input 'in-place'. However, a small constant extra space used for variables is allowed.

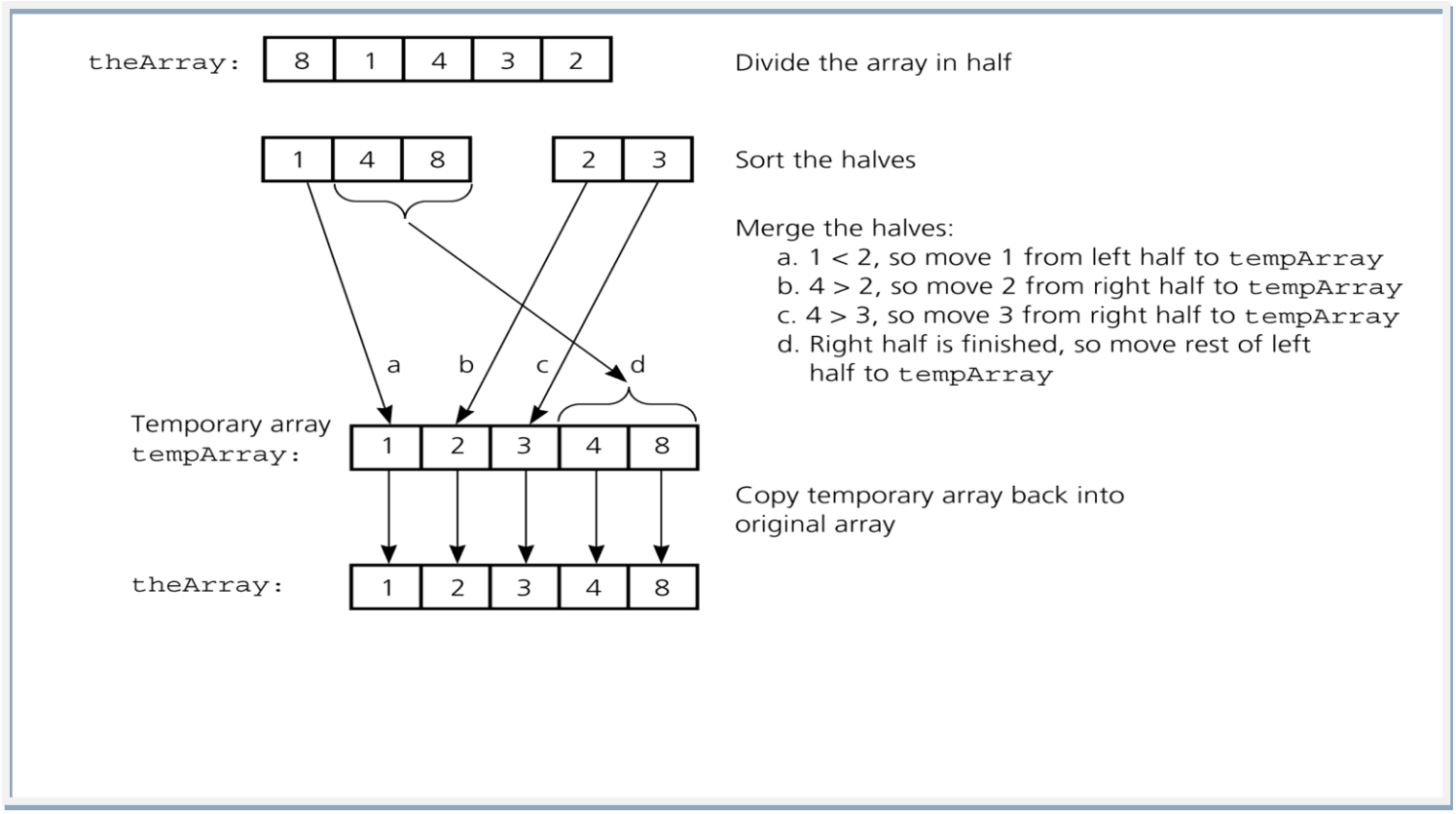
- A sorting algorithm is said to be **stable** if two objects with equal keys appear in the same order in sorted output as they appear in the input data set




*Example of stable sort*

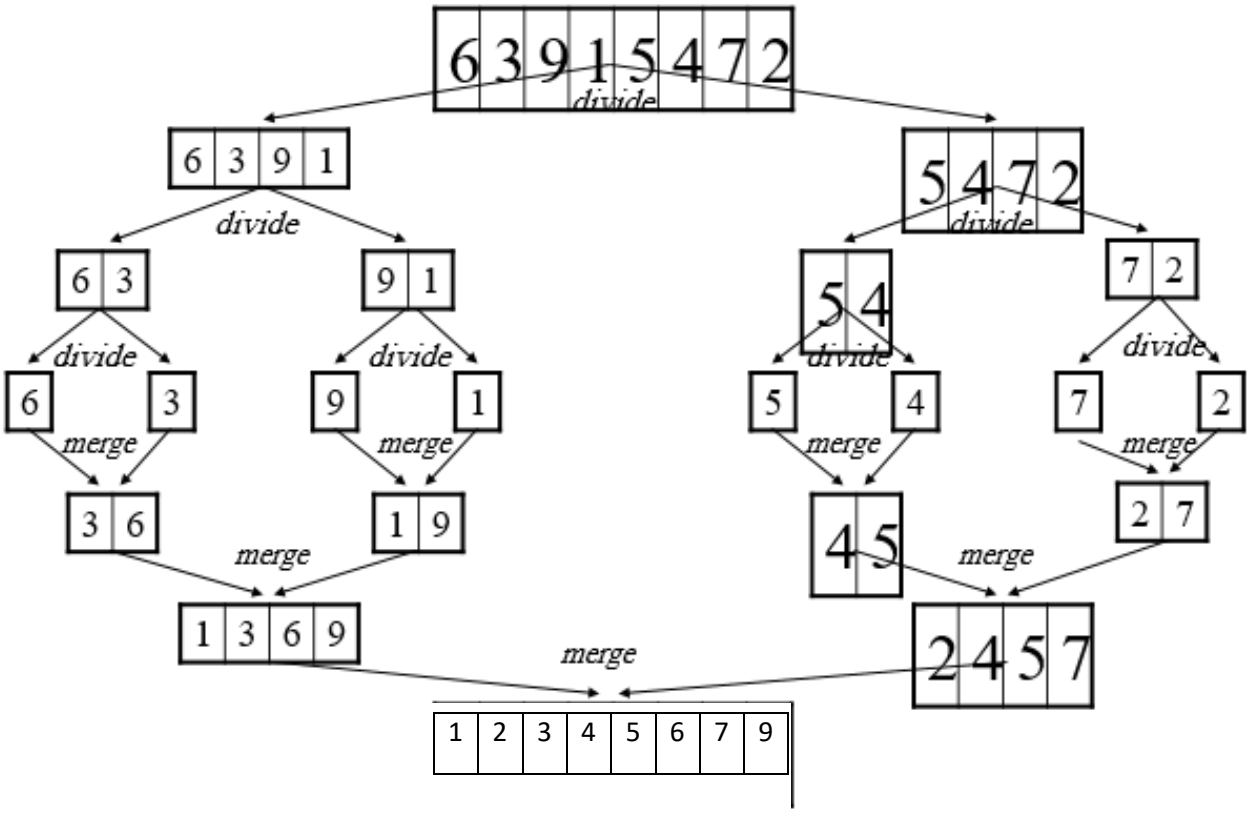
**Mergesort**

- Mergesort algorithm is one of two important divide-and-conquer sorting algorithms (the other one is quicksort).
- It is a recursive algorithm.
  - Divides the list into halves,
  - Sort each halve separately, and
  - Then merge the sorted halves into one sorted array.

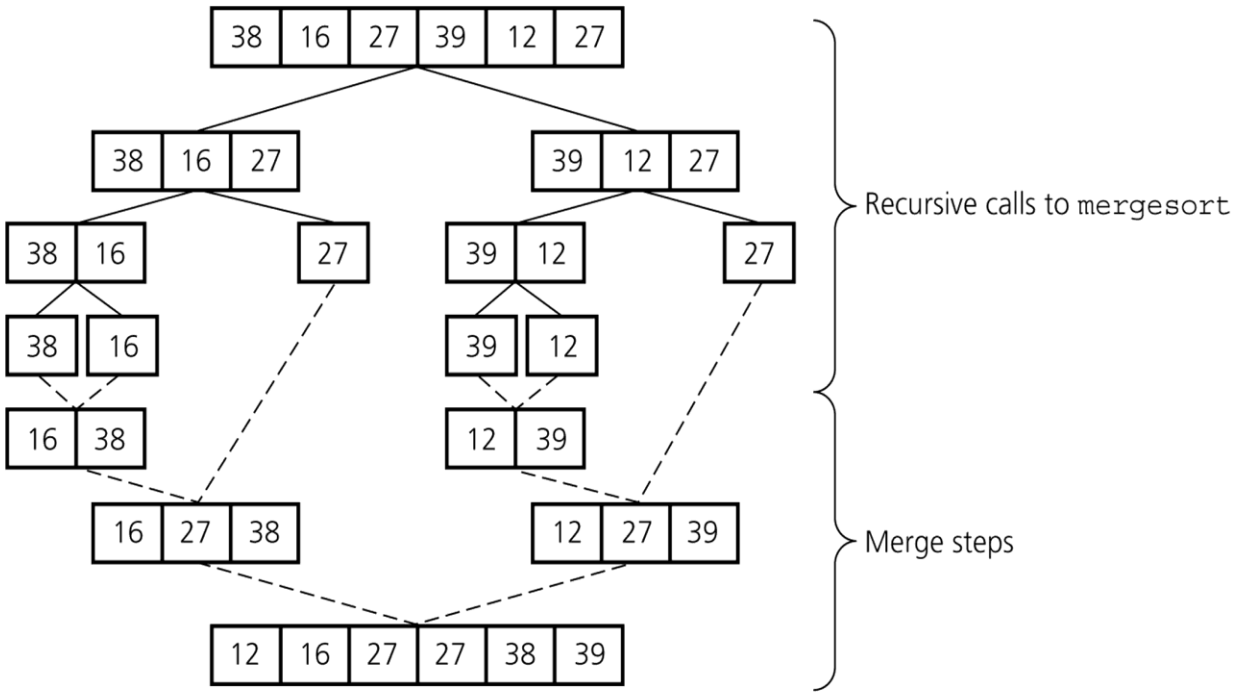


**Examples:**

**Example (1)**



**Example (2)**




**Analysis of Merge-Sort**

- The height ***h*** of the merge-sort tree is ***O***(log ***n***)
  - at each recursive call we divide in half the sequence,
- The overall amount or work done at the nodes of depth ***i*** is ***O***(***n***)
  - we partition and merge $2^i$ sequences of size $n/2^i$
  - we make $2^{i+1}$ recursive calls
- Thus, the total running time of merge-sort is ***O***(***n*** log ***n***)

**Coding:**

```
public class MergeSort {

      public static void sort(Comparable [] data, int low, int high) {

            int mid = (low + high)/2;        constant
            if (low < high) {
                  System.out.println(low+"    "+mid+"    "+high);
                  sort(data, low, mid);// T(n/2)
                  sort(data, mid+1, high);
                  merge(data, low, mid,high);// T(n)
            }
      }
      public static void merge(Comparable [] data,int low,int mid,int high) {

            int n = high - low+1;
            Comparable[] temp = new Comparable[n];
            int i = low, j = mid+1, k=0;
            while ((i <= mid) && (j <= high)){
               if (data[j].compareTo(data[i]) <= 0)
                 temp[k++] = data[j++];
               else
                 temp[k++] = data[i++];
            }
            if (i <= mid)
              for (; i<= mid;)
                 temp[k++] = data[i++];
             else
               for (; j<= high; )
                 temp[k++] = data[j++];
             for (k = 0; k < n; k++)
                 data[low + k] = temp[k];
      }
  }
```

$$T(n) = \begin{cases} a & n=1 \\ 2\,T(n/2) + cn & n>1 \end{cases} \qquad O(n \log n)$$