Ulster University

COM410 Programming Practice

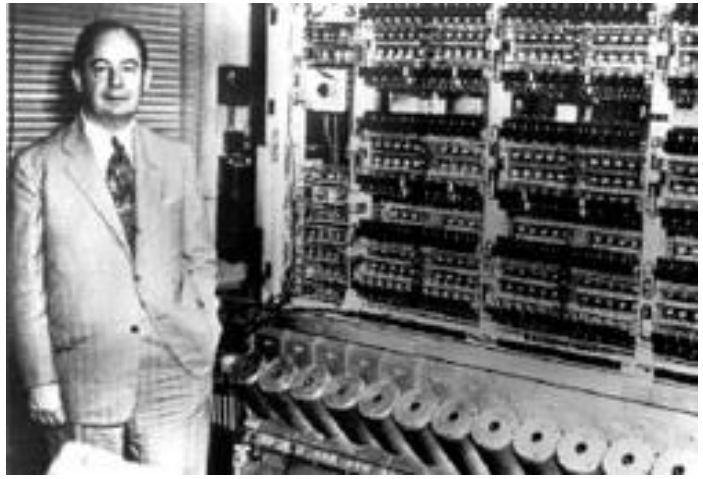
B2.2 Improved Sorting Techniques

Limitations of $O(n^2)$ methods

- Bubble sort is easy to implement and is often sufficient when you want to sort small arrays.
- One of a family of similar techniques including Selection Sort, Insertion Sort and Shell Sort
- All have an execution time related to the square of the number of elements to be sorted – so the sort time grows rapidly as the number of elements increases.
- Unsuited for application to larger datasets

Merge Sort

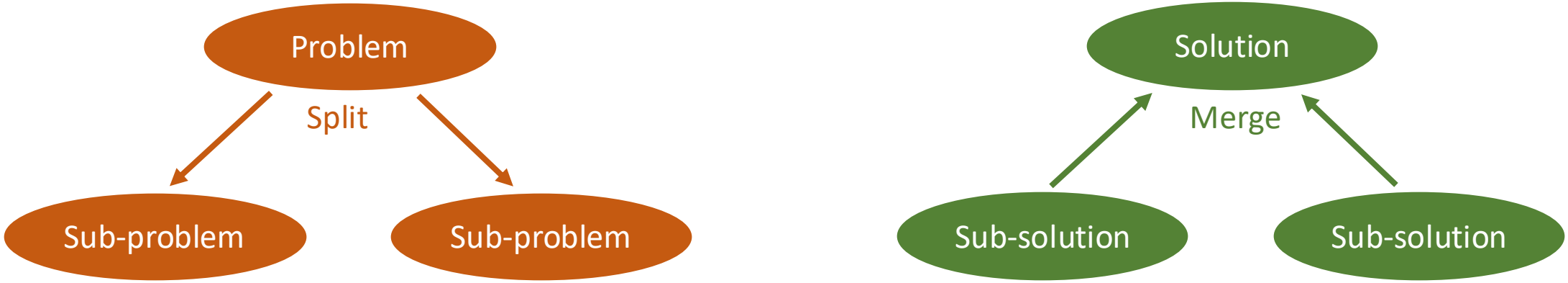
- Invented by John Von Neumann in 1945
- Basic plan is very simple – **divide and conquer**
  - Divide array into two halves
  - Sort each half
  - Merge the two halves



input	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
sort left half	E	E	G	M	O	R	R	S	T	E	X	A	M	P	L	E
sort right half	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
merge results	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

Divide and Conquer Algorithms

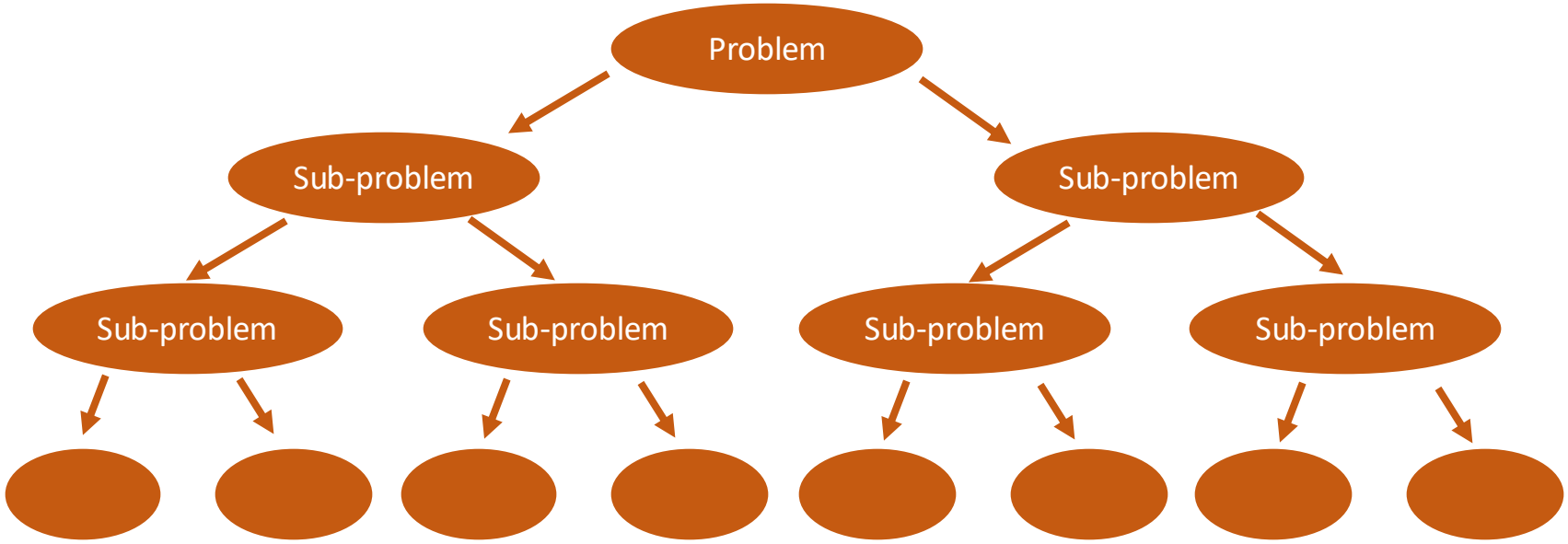
- Divide and conquer algorithm strategy (to problem solving):



  - divide a problem into two or more small but distinct problems,
    solve each new problem, then combine their solutions to solve the original problem

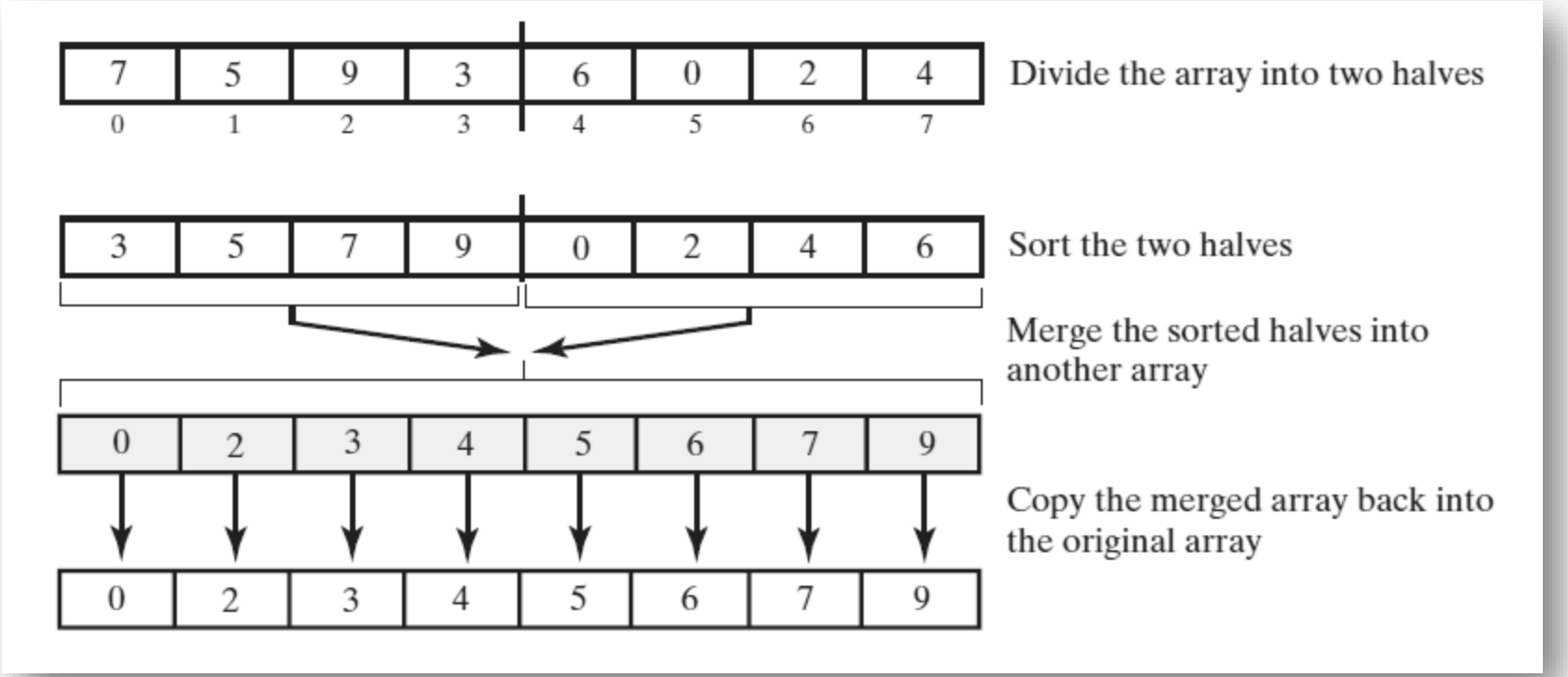
Divide and Conquer Algorithms

- Often applied in a repetitive manner – sub-problems are themselves divided until the smallest possible problem is found



Merge Sort

- Merge two sorted arrays into a temporary array, then copy the temporary array back to the original array



- How do we sort the two halves of the array? **Use a merge sort!**

Merge Sort

- Pseudocode of algorithm for Merge Sort:

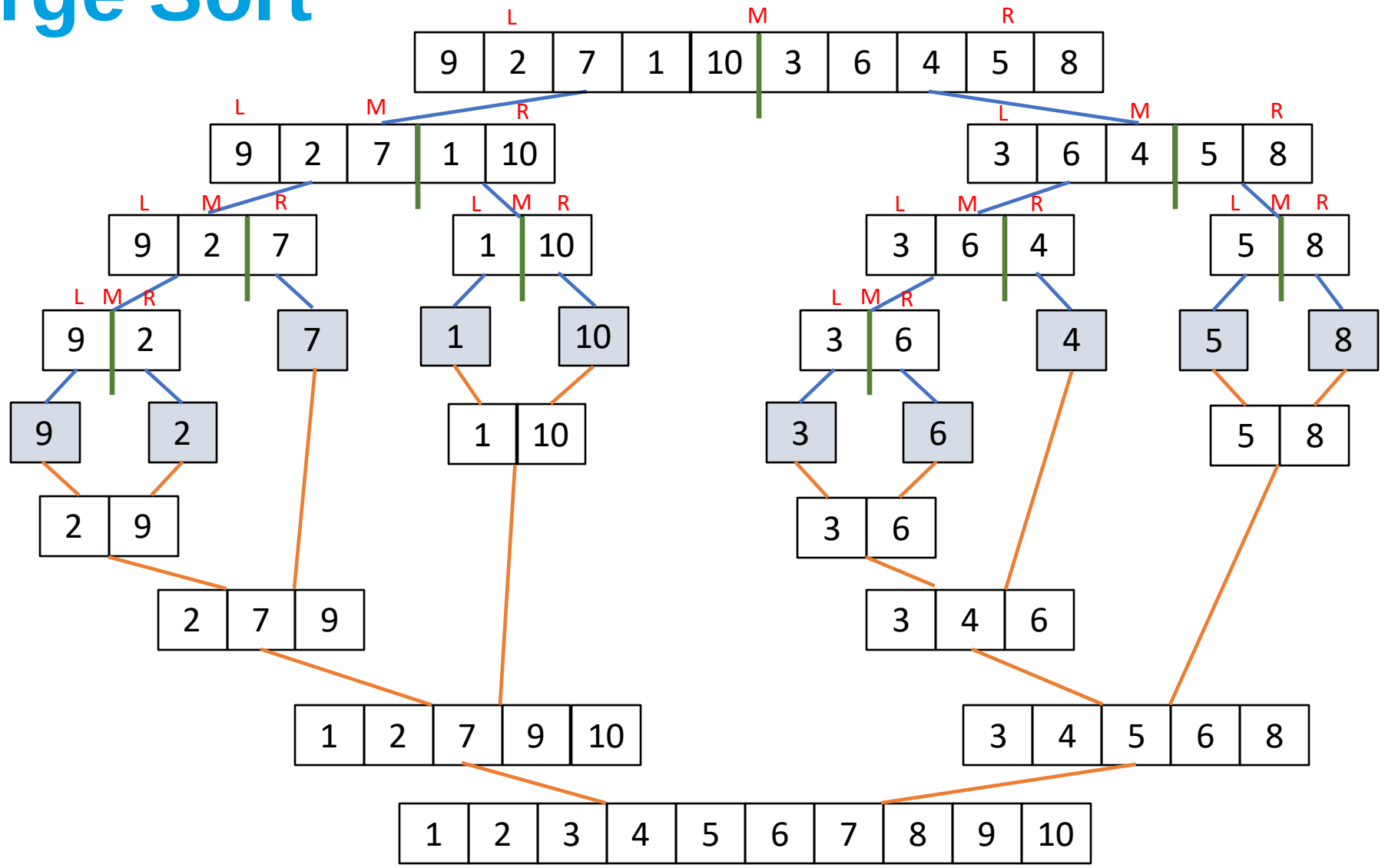
```
Algorithm mergeSort(a, first, last)
// Sorts the entries of an array a between positions first and last.

if first < last
    set mid to the approximate midpoint between first and last

    call mergeSort(a, first, mid)
    call mergeSort(a, mid + 1, last)

    merge the sorted arrays a[first] to a[mid] and a[mid + 1] to a[last]
```

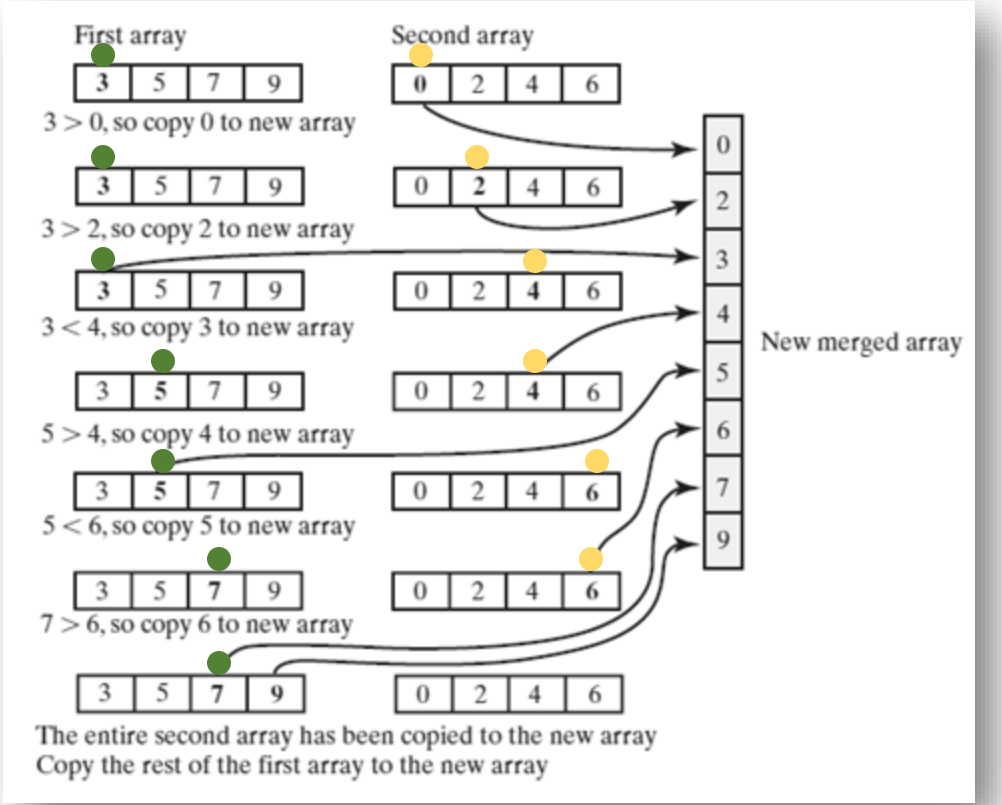
Merge Sort


L M R
9 2 7 1 10 3 6 4 5 8
L M R
9 2 7 1 10
L M R
3 6 4 5 8
L M R
9 2 7
L M R
1 10
L M R
3 6 4
L M R
5 8
L M R
9 2
7
1
10
L M R
3 6
4
5
8
9
2
1
10
3
6
5
8
2 9
3 6
2 7 9
3 4 6
1 2 7 9 10
3 4 5 6 8
1 2 3 4 5 6 7 8 9 10

Merge Sort (Merging Arrays)

- The real effort during execution of merge sort occurs during the merge step

- Merging two sorted arrays isn't difficult but does require an additional (auxiliary) array
- Processing both arrays from beginning to end:
  - Compare an entry in one array with an entry in the other array
  - Copy smaller entry to the new array
- After reaching end of one array, simply copy remaining entries from other array to the auxiliary array



Merge Sort

- Pseudocode of algorithm to merge 2 sorted arrays:

```
Algorithm merge (a, first, mid, last)
// Merge the array from a[first] to a[mid] with the array from a[mid + 1] to a[last]
// using a temporary array.

set pointers to the first elements in each of the 3 arrays (2 source and temporary)
while both source arrays have elements remaining to be processed
   if the current element in the first source array <= that from the second array
      copy from first source array to the temporary array
      increment pointers into first and temporary arrays
   else
      copy from second source array to the temporary array
      increment pointers into second and temporary arrays
copy elements from remaining source array into the temporary array
copy sorted temporary array values back into a
```

Implementation note

- The algorithm for the `merge()` operation suggests that we should create a temporary array into which we merge the values from the pair of source arrays
    - This would require that we create a new array every time that `merge()` is called – very expensive and slow!
    - A much better approach is to create the temporary array once in the main sort method and pass it to the `merge()` operation as a parameter
    - Hence the `merge()` method will have the signature

```
private static void merge(int[] arr, int[] temp,
                          int first, int mid, int last)
```

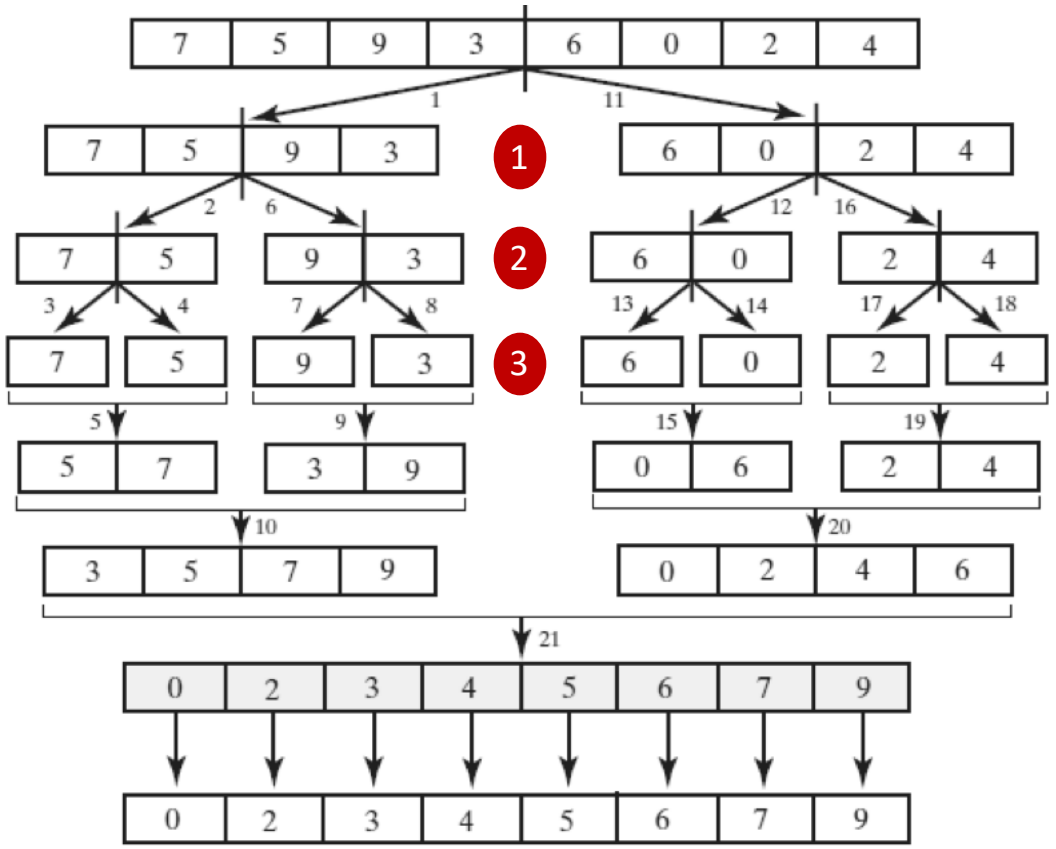
Scenario

- Add the class `MergeSort` to your `Sorting` project and implement the method `mergeSort()`
    - Test the implementation for an integer array of 10 values where

      (i) the array to be sorted is in random order, (ii) the array is already in sorted order, and (iii) the array is in reverse order
    - Repeat the timing experiment previously performed on the Bubble Sort
        - Generate random integer arrays of size 100, 200, 400, 800, 1600, 3200 and 6400 elements and generate the time required to sort each.
        - Perform 1000 timed sorts on each size of array (with new random contents each time) and take the total execution time.
        - Present the results as a list of pairs of values showing the array size and total execution time for 1000 sort operations.

Efficiency of Merge Sort

- In worst-case merge sort has an order of O(n log n)



- Assume *n* is a power of 2 (such as shown)
- In general, if *n* is 2^k, *k* levels of recursive calls occur, where $k = \log_2 n$
- At each call to the merge step, at most ***n* – 1** comparisons are required
- Each merge also requires ***n*** moves to the temporary array and ***n*** moves back to the original array
- In total each merge requires **3*n* – 1** operations

Stable Sorting Algorithms

- A sorting algorithm is **stable** if it does not change the relative order of objects that are equal
- For example, if object **x** appears before object **y** in a collection of data, and **`x.compareTo(y) = 0`**, a stable sorting algorithm will leave object **x** before object **y** after sorting the data
- Stability is important for certain applications - for example, if we sort a group of people first by name then by age, a stable sorting algorithm will ensure that people of the same age will remain in alphabetical order
- Both Bubble Sort and Merge Sort are **stable**