Data Structures and Algorithms

Chapter 5 Sorting

Advance Sort

Nor Bahiah Hj Ahmad & Dayang Norhayati A.Jawawi
School of Computing

Divide and Conquer Sorting Strategy

Merge Sort

Quick Sort

Divide and Conquer Sorting Strategy

- **Divide**
  - **Break into sub-problems** that are themselves **smaller instances** of the **same type** of problem
  - **Recursively solving** this problem
- **Conquer** (overcome)
  - The **solution to the original problem** is then formed from the **solutions to the sub-problems**.

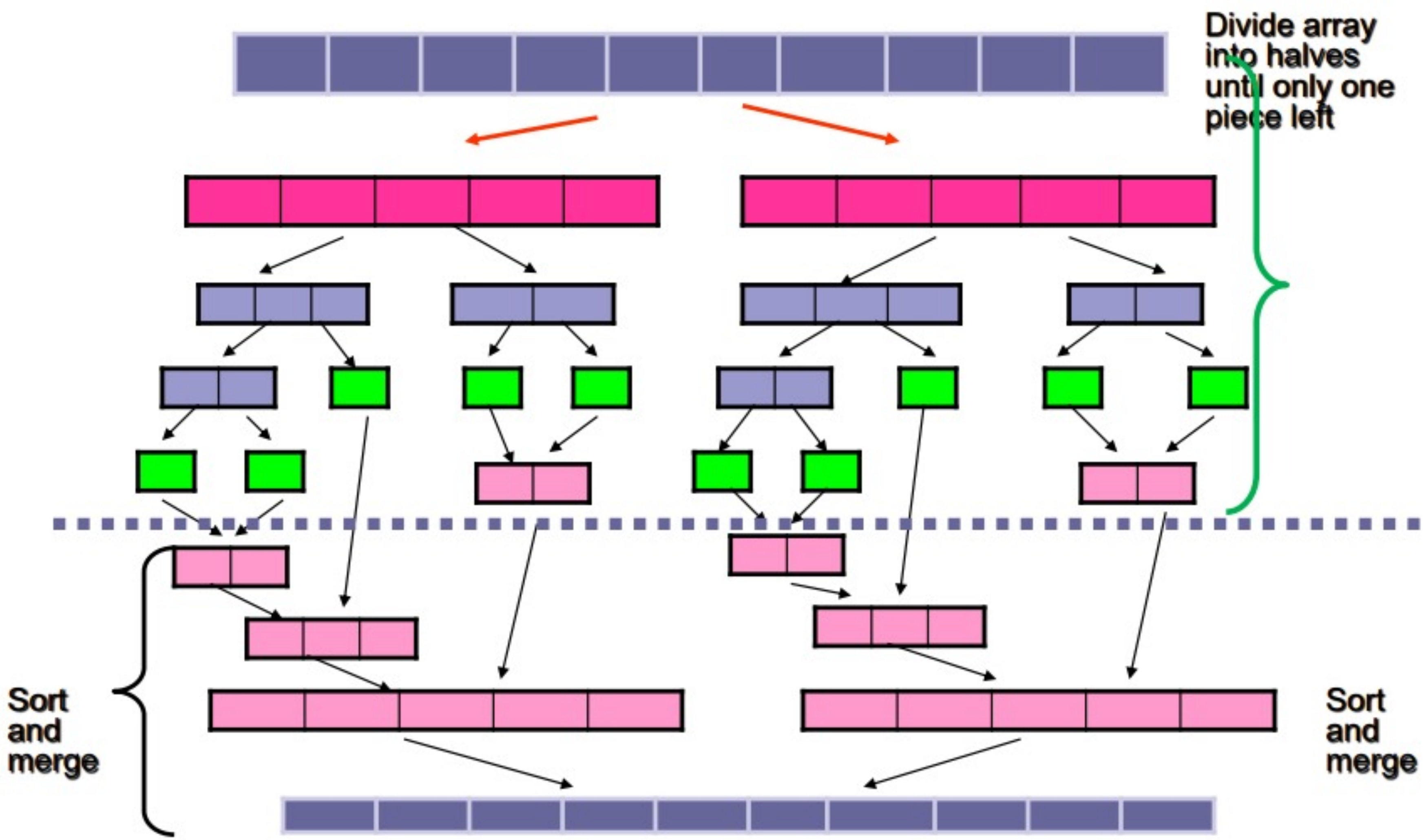
Merge Sort

6 5 3 1 8 7 2 4

Merge Sort

- Applies **divide and conquer** strategy.
- **Three main steps** in Merge Sort algorithm:
  - **Divide an array into halves**
  - **Sort each half**
  - **Merge the sorted halves into one sorted array**
- A **recursive** sorting algorithm
- Performance is **independent of the initial orde**r of the array items

Merge Sort Operation

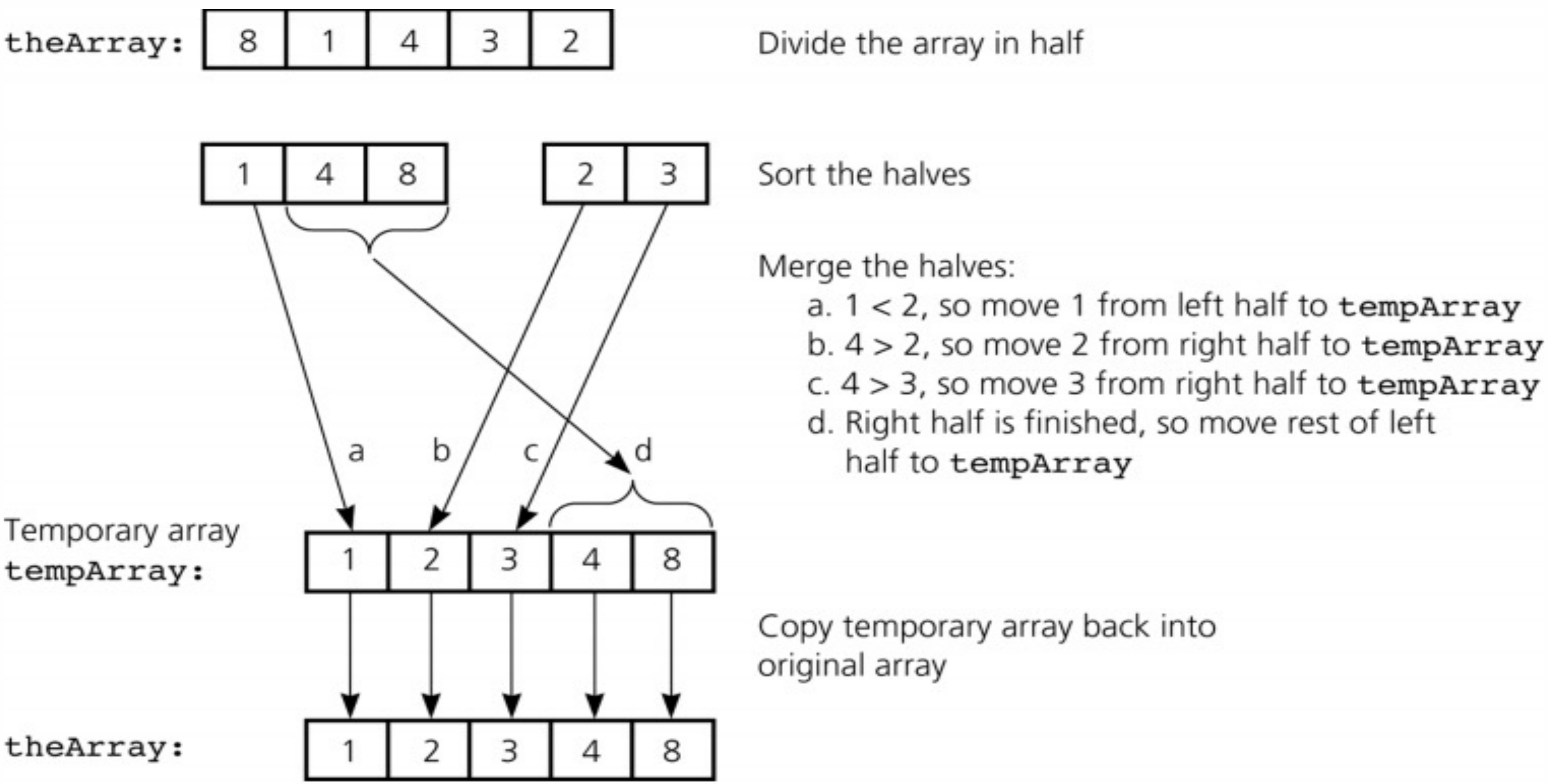


Merge Sort Implementation

Need 2 functions

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



Note: A merge sort with an auxiliary temporary array

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
```

merge() function

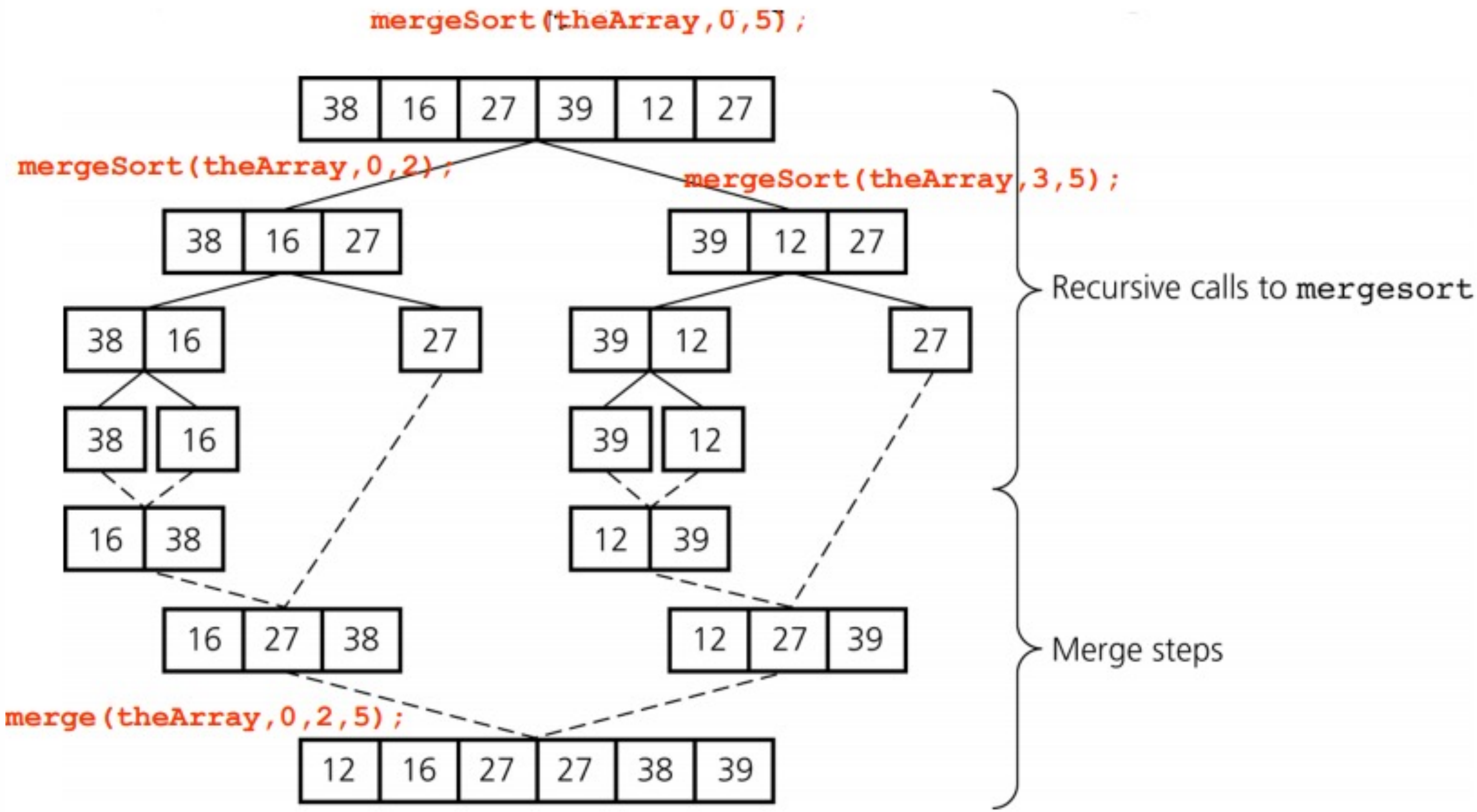
```
const int MAX_SIZE = maxNmbrItemInArry;
void merge(DataType theArray[], int first, int mid, int last)
{
        DataType tempArray[MAX_SIZE]; // temp array
        int first1 = first; // first subarray begin
        int last1 = mid; // end of first subarray
        int first2 = mid + 1; // secnd subarry begin
        int last2 = last; // end of secnd subarry
        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 for
        for (; first1 <= last1; ++first1, ++index)
            tempArray[index] = theArray[first1];
        for (; first2 <= last2; ++first2, ++index)
            tempArray[index] = theArray[first2];
        // copy the result back into the original array
        for (index = first; index <= last; ++index)
            theArray[index] = tempArray[index];
} // end merge function
```

while both **sub-arrays are not empty**, **copy** the **smaller** item into the **temporary** array

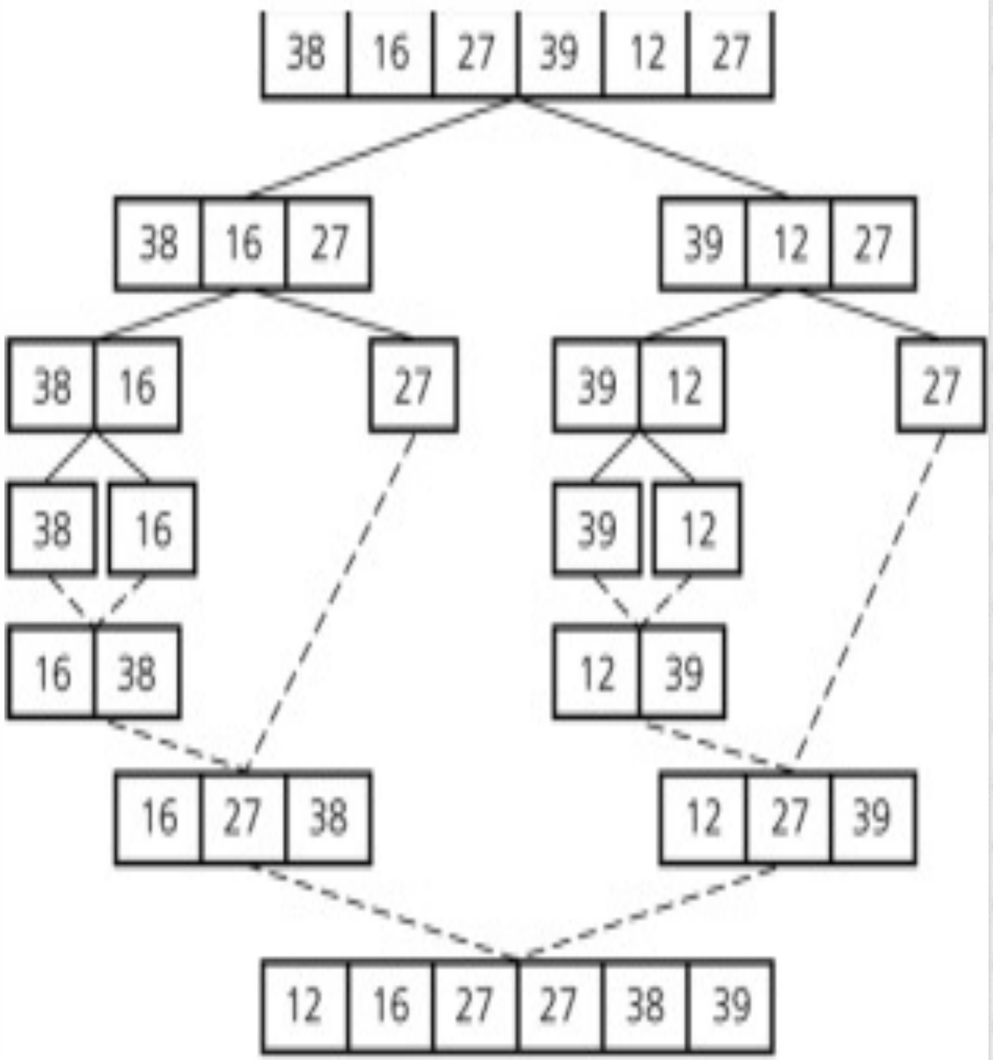
Move remaining item to **temporary** array and **finish off the second sub-array, if necessary**

copy the result from **temporary** array into the **original** array

mergeSort [38 16 27 39 12 27]



mergeSort [38 16 27 39 12 27] (continued…)




```
Content of the array before sorting :38  16  27  39  12  27
Content of sublist 1 -> 38  16  27  39  12  27
Content of sublist 2 -> 38  16  27
Content of sublist 3 -> 38  16
Content of sublist 4 -> 38
Content of sublist 5 -> 16
Content of merged list 16  38
Content of sublist 6 -> 27
Content of merged list 16  27  38
Content of sublist 7 -> 39  12  27
Content of sublist 8 -> 39  12
Content of sublist 9 -> 39
Content of sublist 10 -> 12
Content of merged list 12  39
Content of sublist 11 -> 27
Content of merged list 12  27  39
Content of merged list 12  16  27  27  38  39
Content of the array after sorting : 12  16  27  27  38  39
```

Result AFTER execution

Merge Sort Analysis

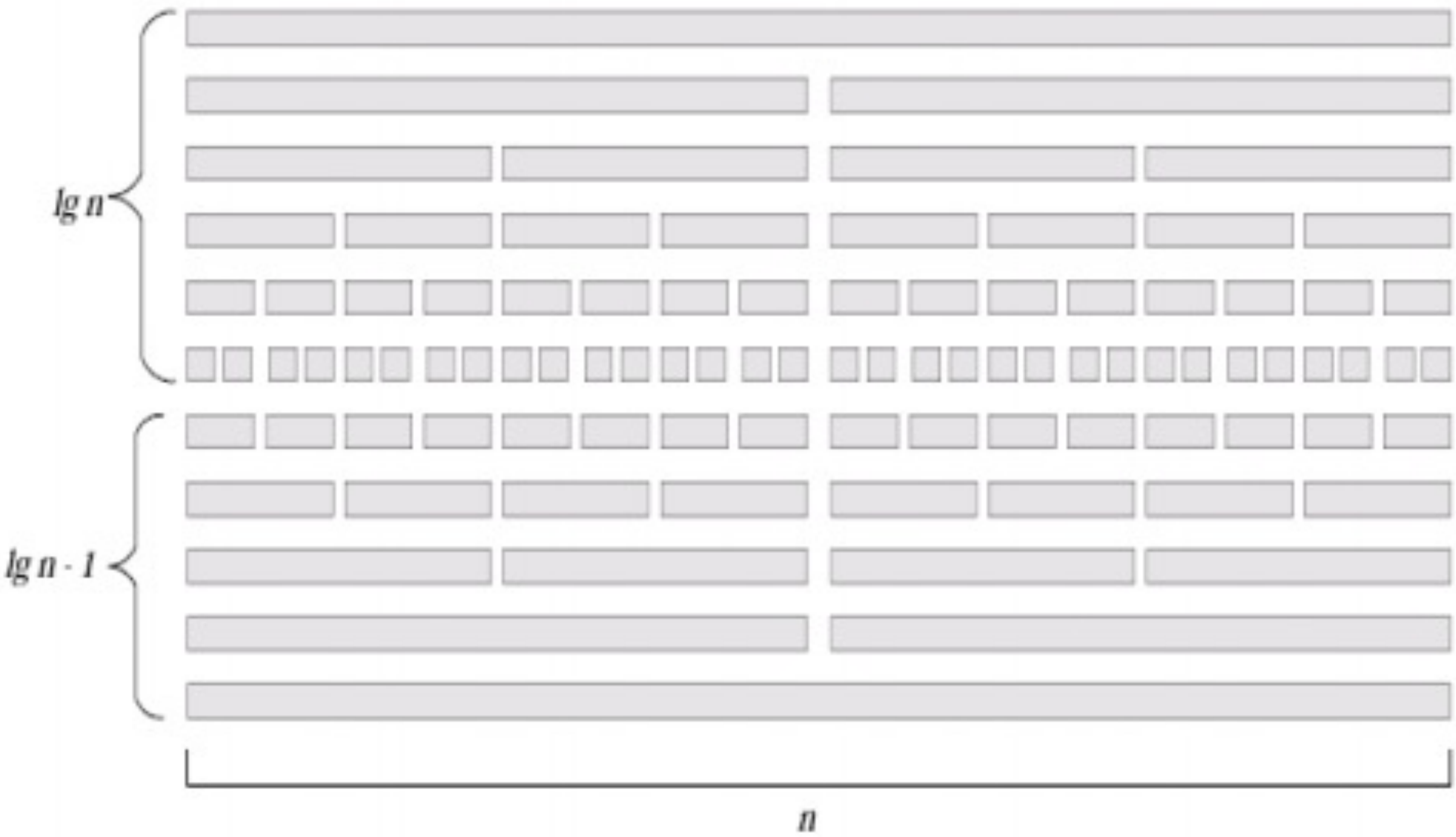
- The list is always divided into two balanced list (or almost balanced for odd size of list)
- The number of calls to repeatedly divide the list until there is one item left in the list is:

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

- If the **left** segment and the **right** segment of the list have the equal size (or **almost equal** size), then ***x ≈ lg n***. The **number of iteration** is approximately ***n lg n***.

- **The same number of repetition is needed to sort and merge the list.**
- **Thus, as a whole number of steps needed to sort data using merge sort is *2n lg n*, which is *O(n lg n)*.**

Merge Sort Analysis (continued…)



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

Quick Sort

6 5 3 1 8 7 2 4

Quick Sort Operation

- Quick sort is similar with merge sort in using divide and conquer technique.

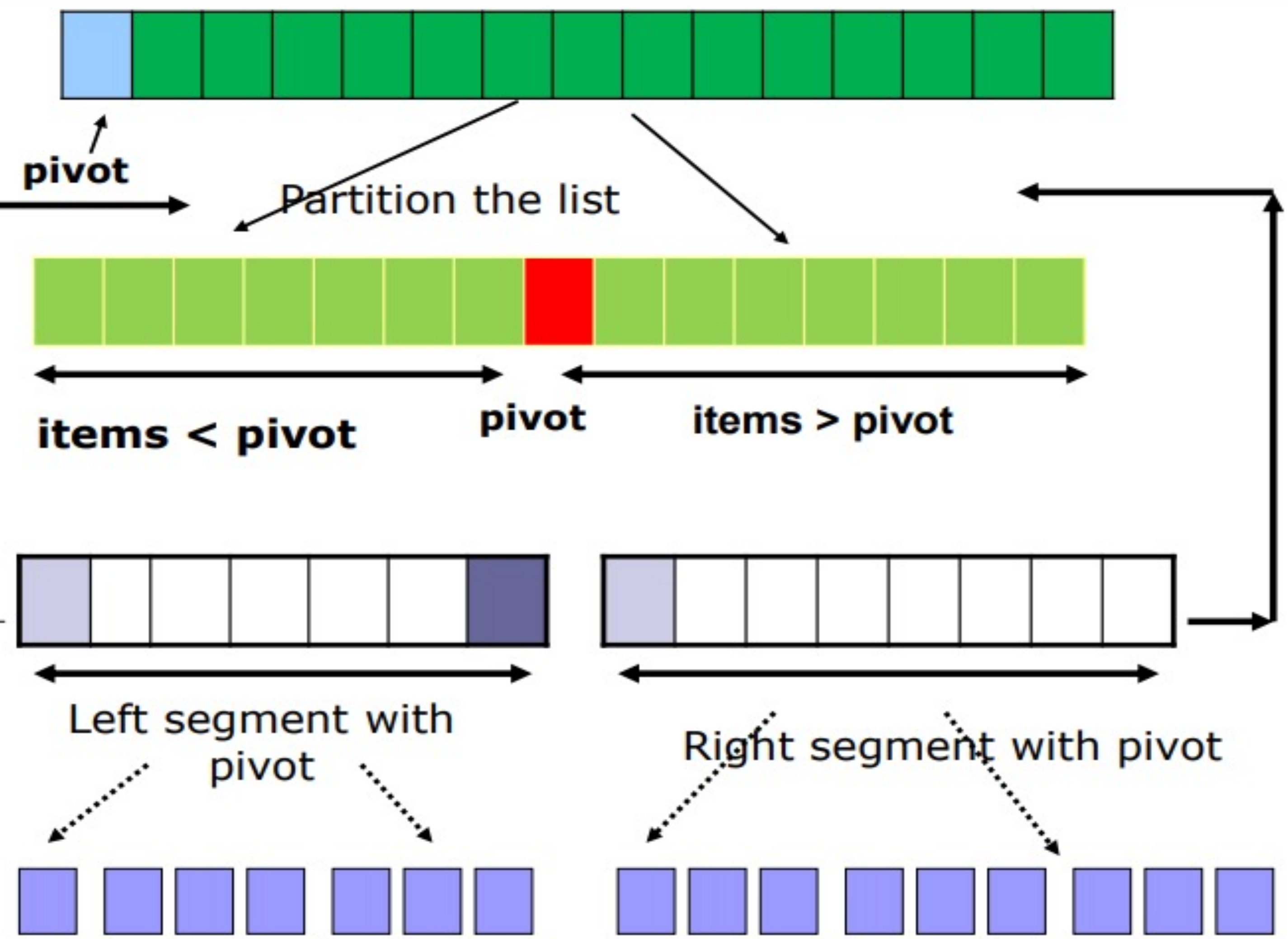
- Differences of Quick sort and Merge sort :

Quick Sort	**Merge Sort**
Partition the list based on the **pivot** value	Partition the list by **dividing the list into two**
No merge operation is needed since when there is only **one item left in the list to be sorted**, all **other items are already in sorted position**.	**Merge operation** is needed to **sort and merge** the item in the **left and right segments.**

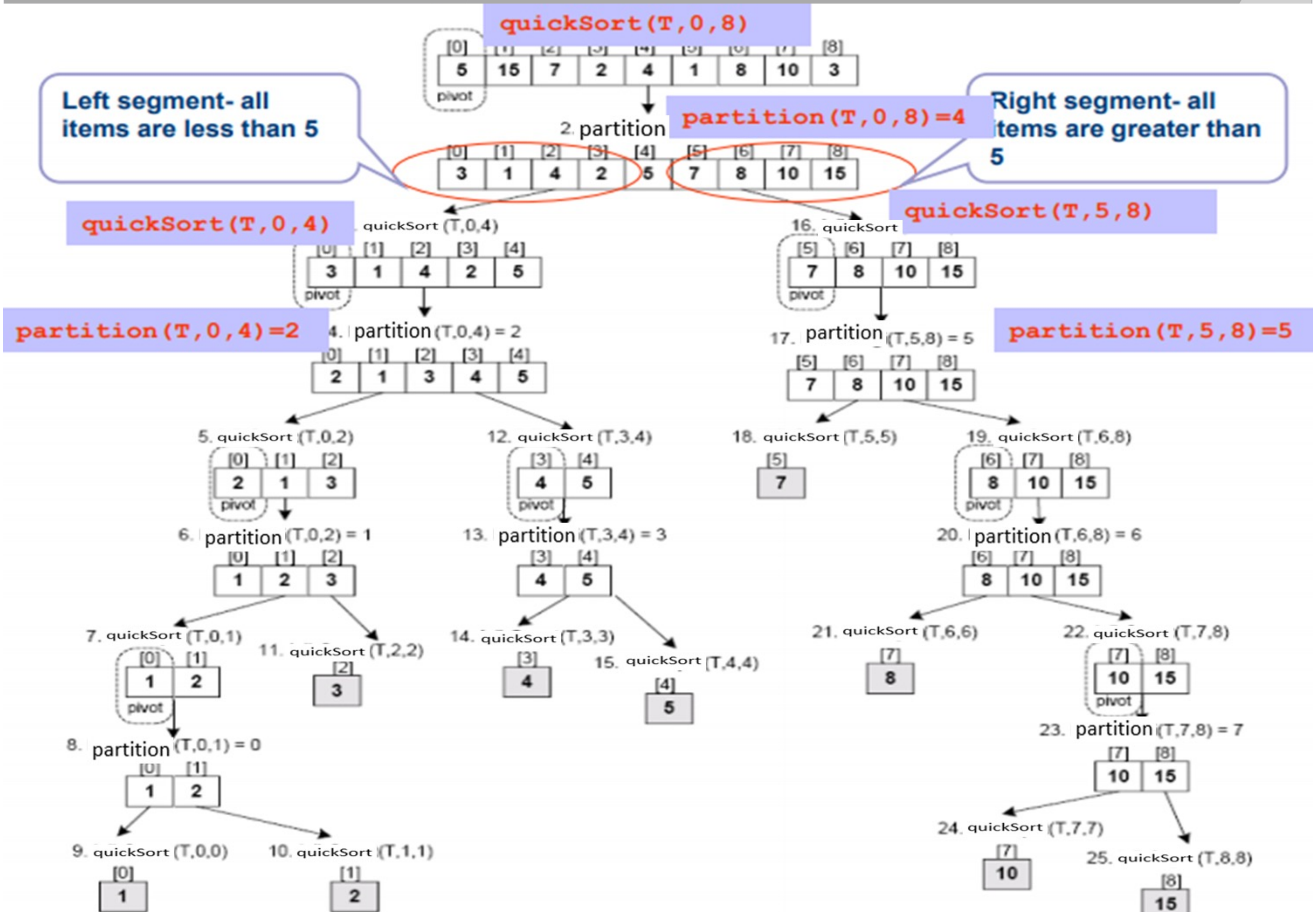
Quick Sort

- A divide-and-conquer algorithm
- Strategy
  - Choose a **pivot (first element in the array)**
  - Partition the array about the pivot
    - **items < pivot**
    - **items >= pivot**
    - **Pivot is now in correct sorted position**
- Sort the left section again until there is one item left
- Sort the right section again until there is one item left

Quick Sort Process



Partition process is repeated until there is only one item left in the list.



quickSort(T,0,8)
Left segment- all items are less than 5
partition(T,0,8)=4
Right segment- all items are greater than 5
quickSort(T,0,4)
quickSort(T,5,8)
partition(T,0,4)=2
partition(T,5,8)=5
2. partition
4. partition (T,0,4) = 2
5. quickSort (T,0,2)
6. partition (T,0,2) = 1
7. quickSort (T,0,1)
8. partition (T,0,1) = 0
9. quickSort (T,0,0)
10. quickSort (T,1,1)
11. quickSort (T,2,2)
12. quickSort (T,3,4)
13. partition (T,3,4) = 3
14. quickSort (T,3,3)
15. quickSort (T,4,4)
16. quickSort
17. partition (T,5,8) = 5
18. quickSort (T,5,5)
19. quickSort (T,6,8)
20. partition (T,6,8) = 6
21. quickSort (T,6,6)
22. quickSort (T,7,8)
23. partition (T,7,8) = 7
24. quickSort (T,7,7)
25. quickSort (T,8,8)
pivot

Quick Sort Implementation

2 functions are needed :

- **quickSort() function**
  - a **recursive** function that will **partition** the list into several sub lists **until there is one item left** in the sub list.
- **partition() function**
  - **organize** the data so that the **items with values less than pivot will be on the left of the pivot**, while the **values at the right pivot contains items that are greater or equal to pivot**.

```
void quickSort(dataType arrayT[], int first , int last)
{
        int cut;
        if (first < last)
        {
                cut = partition(T, first, last);
                quickSort(T, first, cut);
                quickSort(T, cut+1, last);
        }
}
```

partition() function

```
int partition(int T[], int first, int last)
{
 int pivot, temp;
 int loop, cutPoint, bottom, top;
 pivot=T[first]; // identify pivot
 bottom=first; top= last;
 loop=1; //always TRUE

 while (loop)
 {
       while (T[top]>pivot)
       { top--; }
       while(T[bottom]<pivot)
       { bottom++; }
       if (bottom<top)
       {
          // change pivot place
          temp=T[bottom];
          T[bottom]=T[top];
          T[top]=temp;
        }
        else
       {
             loop=0; //loop false
             cutPoint = top;
        }
  } // end while
  return cutPoint;
} // end partition()
```

Organize the data so that the items with values **less than pivot** will be on the **left** of the pivot while the values at the **right** of pivot, pivot contains items that are **greater or equal to pivot**.

Find **smaller** value than pivot from **top** array

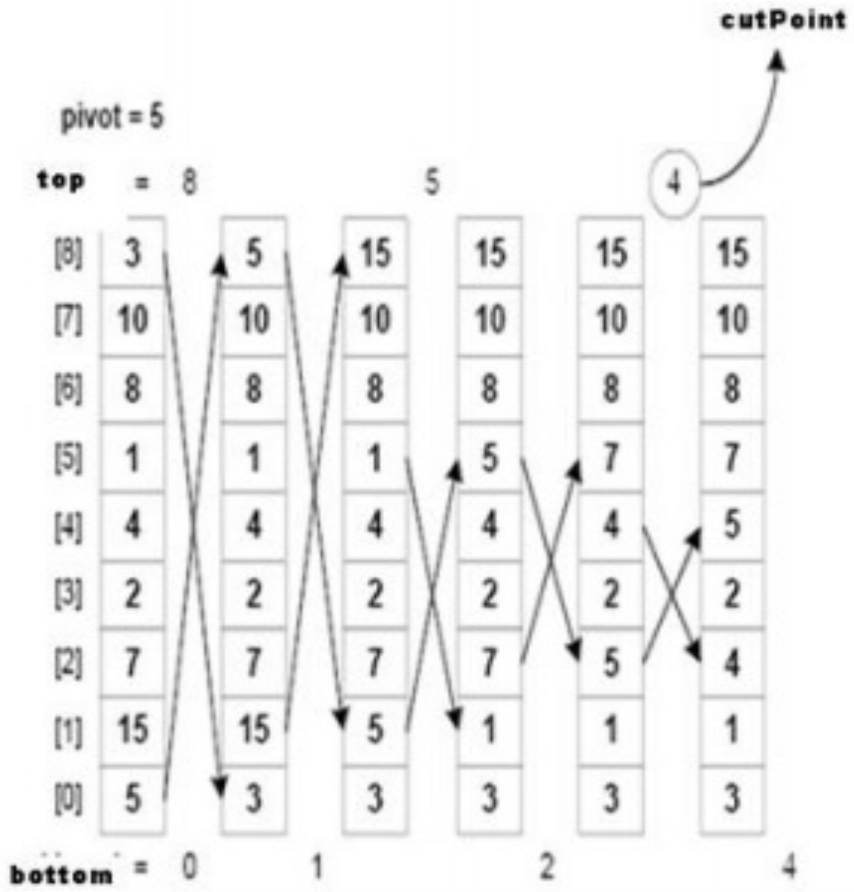
Find **larger** value than pivot from **bottom** array

change pivot's place

Partition process for array: [5 15 7 2 4 1 8 10 3]

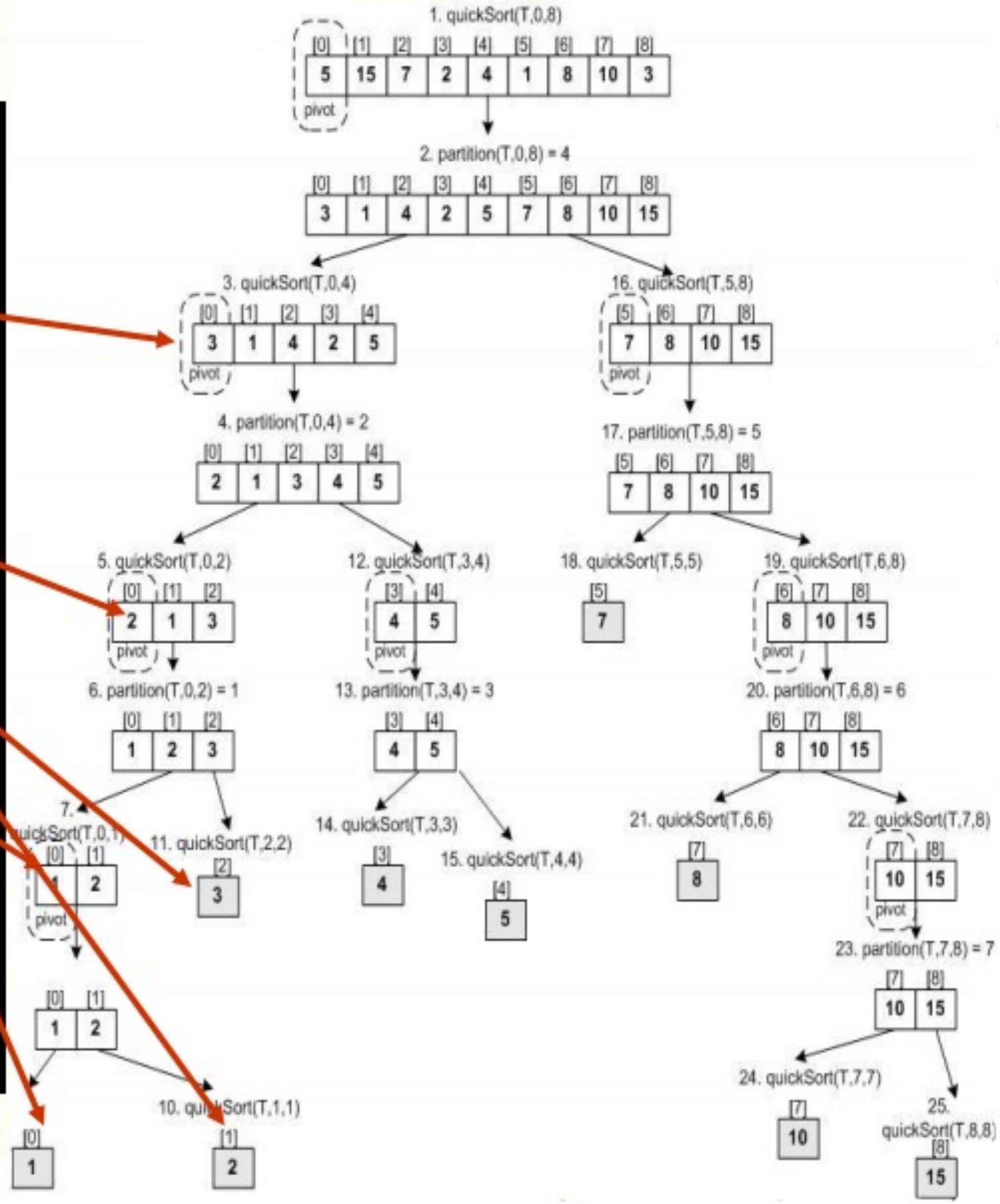
After execution of function partition(), **pivot 5** will be placed at **index 4** and the **value 4**, will be returned to function quickSort() for further partition.

```
Content of the array before sorting  :5  15  7  2  4  1  8  10  3

The sublist with pivot = 5
5  15  7  2  4  1  8  10  3

The sublist with pivot = 3
3  1  4  2  5

The sublist with pivot = 2
2  1  3

The sublist with pivot = 1
1  2

The sublist with pivot = 4
4  5

The sublist with pivot = 7
7  8  10  15

The sublist with pivot = 8
8  10  15

The sublist with pivot = 10
10  15

Content of the array after sorting  : 1  2  3  4  5  7  8  10  15 _
```



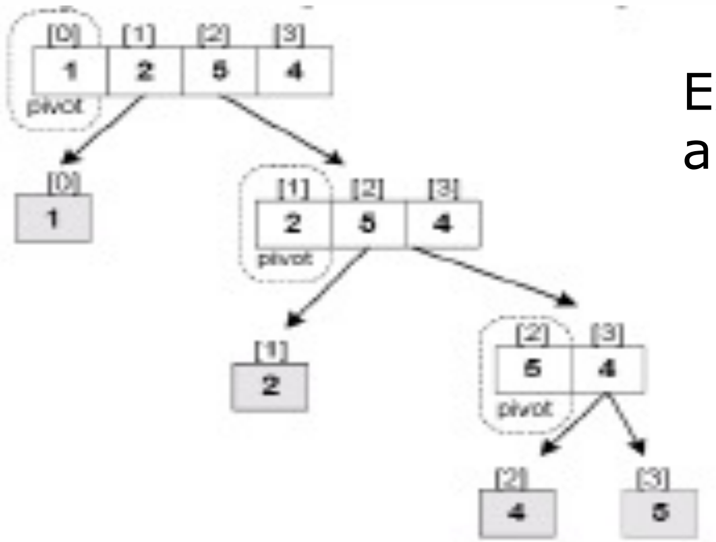
quickSort[5 15 7 2 4 1 8 10 3]

```
Content of the array before sorting :5  15  7  2  4  1  8
The sublist -> 1 with pivot = 5
5  15  7  2  4  1  8  10  3
The sublist -> 2 with pivot = 3
3  1  4  2  5
The sublist -> 3 with pivot = 2
2  1  3
The sublist -> 4 with pivot = 1
1  2
The sublist -> 5 with one piece item = 1

The sublist -> 6 with one piece item = 2

The sublist -> 7 with one piece item = 3

The sublist -> 8 with pivot = 4
4  5
The sublist -> 9 with one piece item = 4

The sublist -> 10 with one piece item = 5

The sublist -> 11 with pivot = 7
7  8  10  15
The sublist -> 12 with one piece item = 7

The sublist -> 13 with pivot = 8
8  10  15
The sublist -> 14 with one piece item = 8

The sublist -> 15 with pivot = 10
10  15
The sublist -> 16 with one piece item = 10

The sublist -> 17 with one piece item = 15
```



Quick Sort Analysis

- The **efficiency** of quick sort depends on the **pivot** value.
- This class chose the **first element in the array** as pivot value.
- However, pivot can also be chosen at **random**, or from the **last** element in the array.
- The **worse case** for quick sort occur when the **smallest item or the largest item always be chosen as pivot** value causing the left partition and the right partition **not balance**.

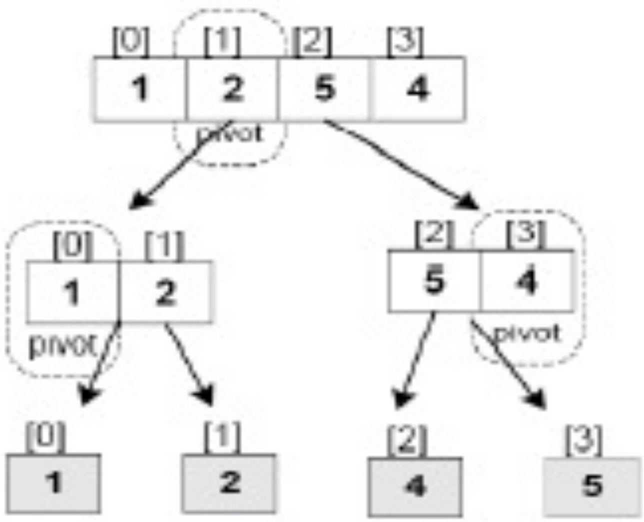


Example of worse case quick sort: sorted array [1 2 5 4] causing imbalance partition.

Quick Sort Analysis (continued…)

- The **best case** for quick sort happen when the list is partition into **balance** segment.
- Must chose the right pivot that can put other items in balance situation.
- The number of comparisons in partition process for base case situation is as follows:

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

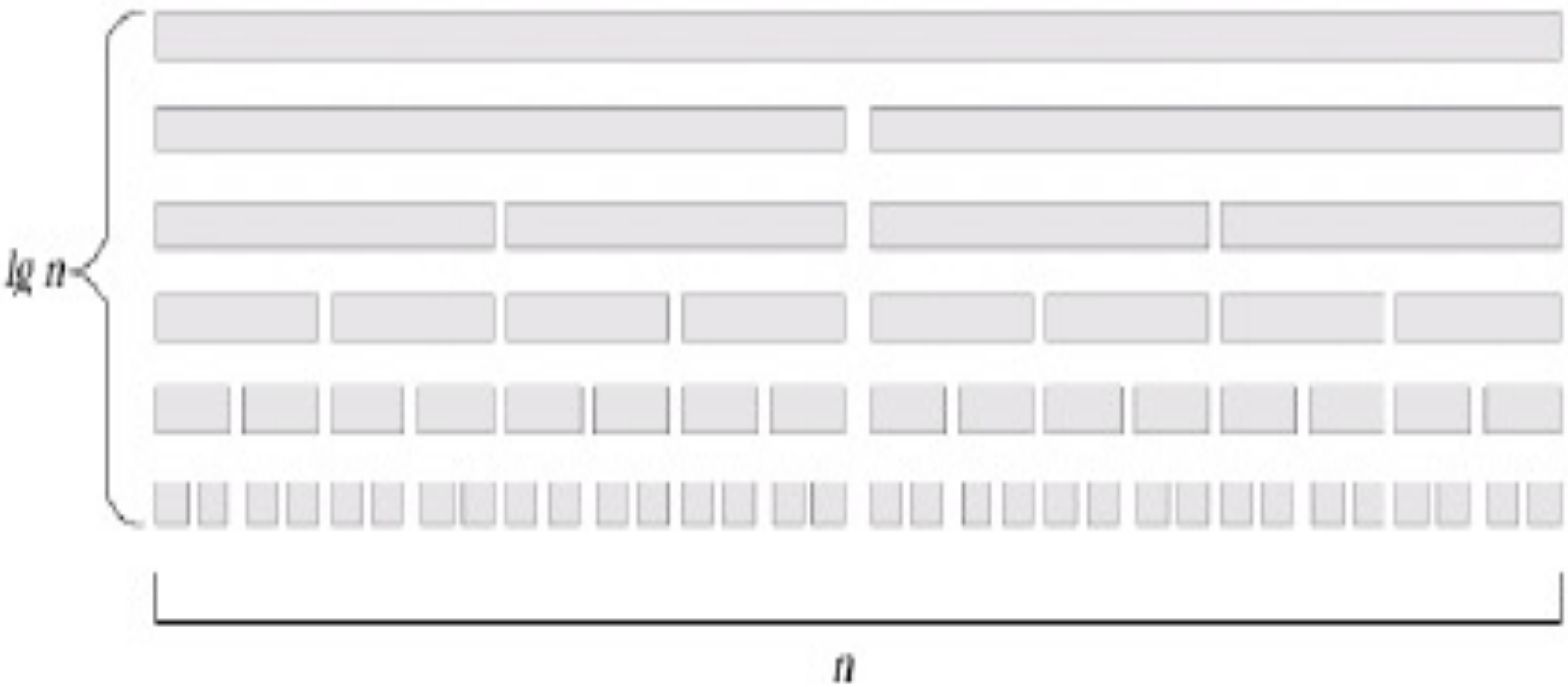


The best case for quick sort happen when the left segment and the right segment is balanced (have the same size) with value ***x* ≈ lg *n***

Example of best case quick sort: array[1 2 5 4]..

Quick Sort Analysis

- The number of steps to get the **balance segment** while partitioning the array is ***lg n*** and the number of comparisons depend on the size list, ***n***.



Quick Sort

- **Analysis**
  - **Average case: $O(n * \log_2 n)$**
  - **Worst case: $O(n^2)$**
    - When the array is already sorted, and the smallest item is chosen as the pivot
  - Quicksort is usually extremely fast in practice
  - Even if the worst case occurs, quicksort's performance is acceptable for moderately large arrays

A Comparison of Sorting Algorithms

- Approximate growth rates of time required for eight sorting algorithms

	Worst case	Average case
Selection sort	n^2	n^2
Bubble sort	n^2	n^2
Insertion sort	n^2	n^2
Mergesort	$n * \log n$	$n * \log n$
Quicksort	n^2	$n * \log n$
Radix sort	n	n
Treesort rt	n^2	$n * \log n$
	$n * \log n$	$n * \log n$

Order-of-Magnitude Analysis and Big O Notation

- A comparison of growth-rate functions shows that ***O(n log n)*** algorithm is significantly faster than ***O(n²)*** algorithm.

Function	10	100	1,000	10,000	100,000	1,000,000
1	1	1	1	1	1	1
$\log_2 n$	3	6	9	13	16	19
n	10	10^2	10^3	10^4	10^5	10^6
$n * \log_2 n$	30	664	9,965	10^5	10^6	10^7
n^2	10^2	10^4	10^6	10^8	10^{10}	10^{12}
n^3	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}
2^n	10^3	10^{30}	10^{301}	$10^{3,010}$	$10^{30,103}$	$10^{301,030}$

Summary

- Order-of-magnitude analysis and Big O notation measure an algorithm's time requirement as a function of the problem size by using a growth-rate function
- To compare the efficiency of algorithms
  - Examine growth-rate functions when problems are large
  - Consider only significant differences in growth-rate functions
- Worst-case and average-case analyses
  - Worst-case analysis considers the maximum amount of work an algorithm will require on a problem of a given size
  - Average-case analysis considers the expected amount of work that an algorithm will require on a problem of a given size
- Order-of-magnitude analysis can be the basis of your choice of an ADT implementation
- Selection sort, bubble sort, and insertion sort are all ***$O(n^2)$*** algorithms
- Quicksort and mergesort are two very fast recursive sorting algorithms

Exercise

Show how Quick Sort and Merge Sort algorithm is implemented on the following list of data:

[12 9 20 18 7 5 15 17 11 25 30 35]

Discuss the efficiency of both sorting techniques applying on the data.

Next…