Sorting Algorithms

Introduction

- Rearranging elements of an array in some order.
- Various types of sorting are:
  - Bubble Sort
  - Selection Sort
  - Insertion Sort
  - Shell Sort
  - Quick Sort
  - Merge Sort
  - Counting Sort
  - Radix Sort
  - Bucket Sort

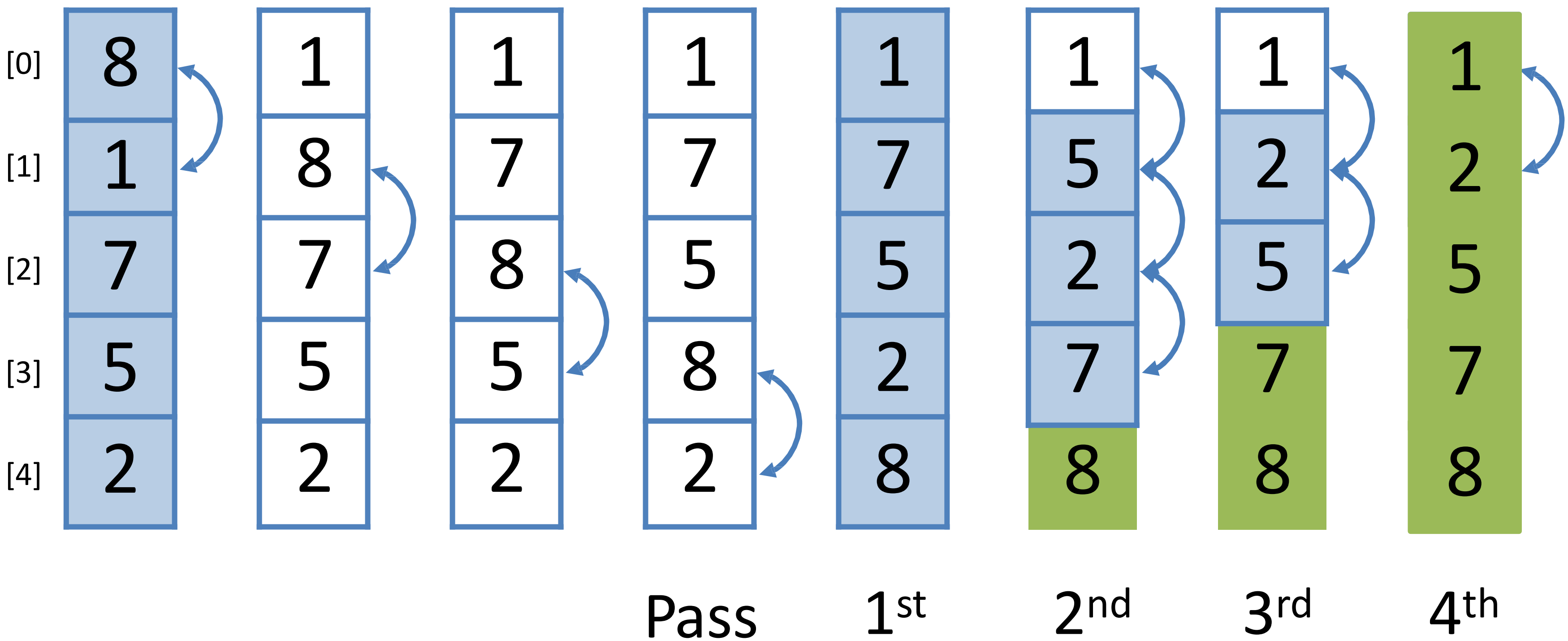
- In place.
- Stable.
- Online.

10	30	10	50	70	40	10	20

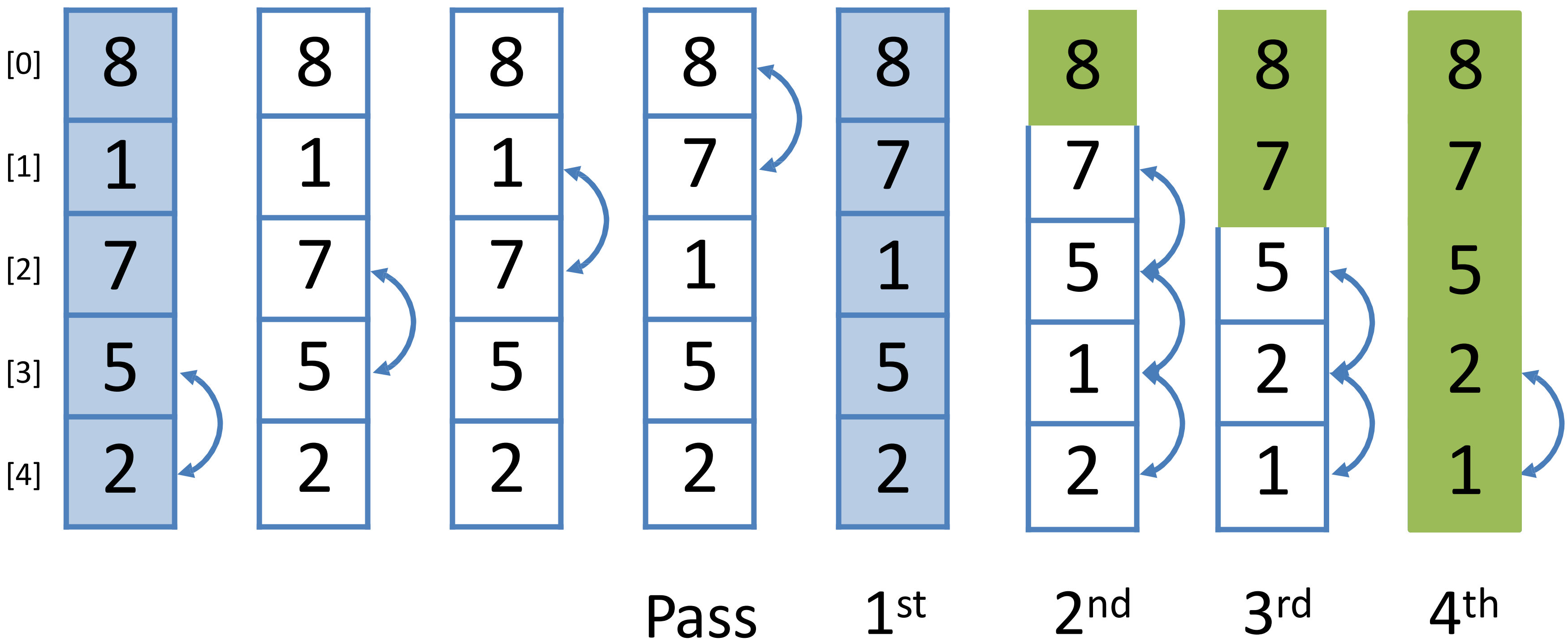
10	10	10	20	30	40	50	70

Bubble Sort

Bubble Sort – Ascending



Bubble Sort – Descending



Algorithm – Bubble Sort

Algorithm bubbleSort(A,n)

Input: An array **A** containing **n** integers.

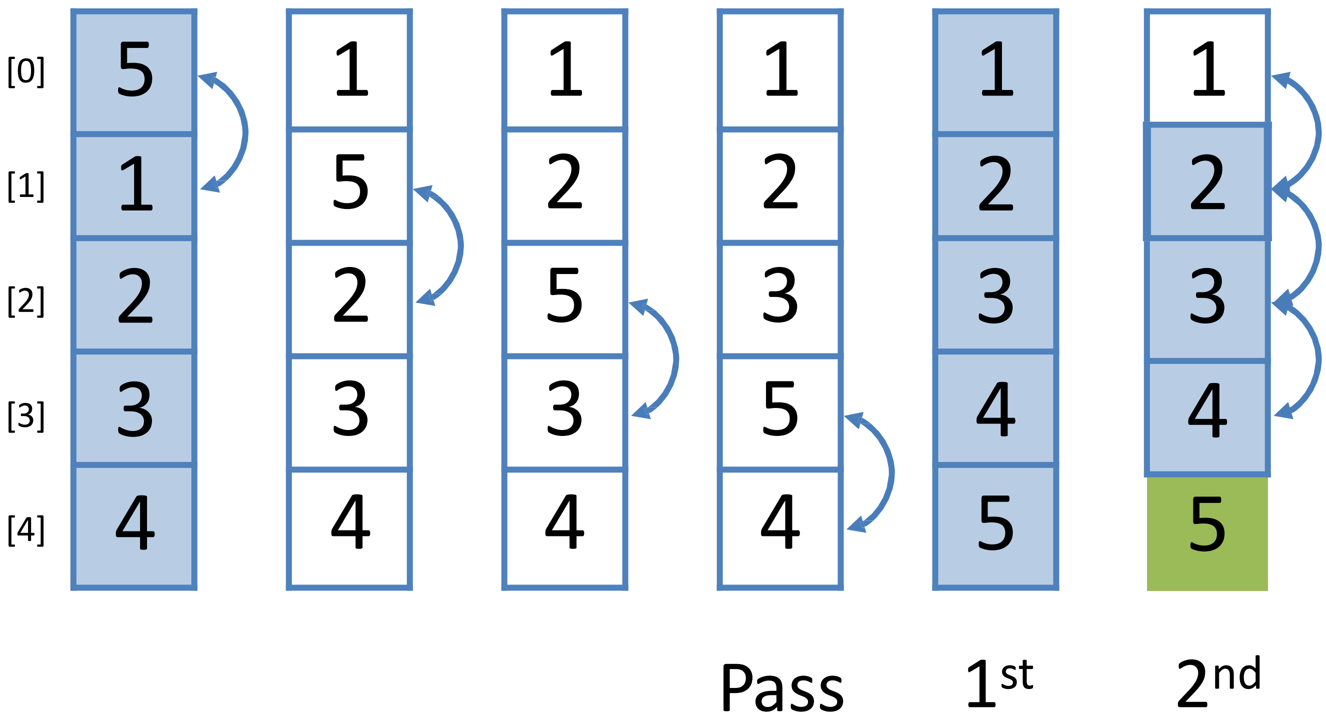
Output: The elements of **A** get sorted in increasing order.

1. **for** i = 1 to n – 1 **do**
2.   **for** j = 0 to n – i – 1 **do**
3.   **if** A[j] > A[j + 1]
4.   Exchange A[j] with A[j+1]

Time Complexity – Bubble Sort

- Best Time Complexity: **Ω** (n^2)
- Average Time Complexity: **Θ** (n^2)
- Worst Time Complexity: **O** (n^2)

Optimized Bubble Sort?



Algorithm – Optimized Bubble Sort

Algorithm bubbleSortOpt(A,n)
Input: An array **A** containing **n** integers.
Output: The elements of **A** get sorted in increasing order.

```
for  i = 1  to  n – 1
    flag = true
    for j = 0  to  n – i – 1  do
        if A[j] > A[j + 1]
            flag = false
            Exchange A[j] with A[j+1]
    if  flag == true
        break;
```

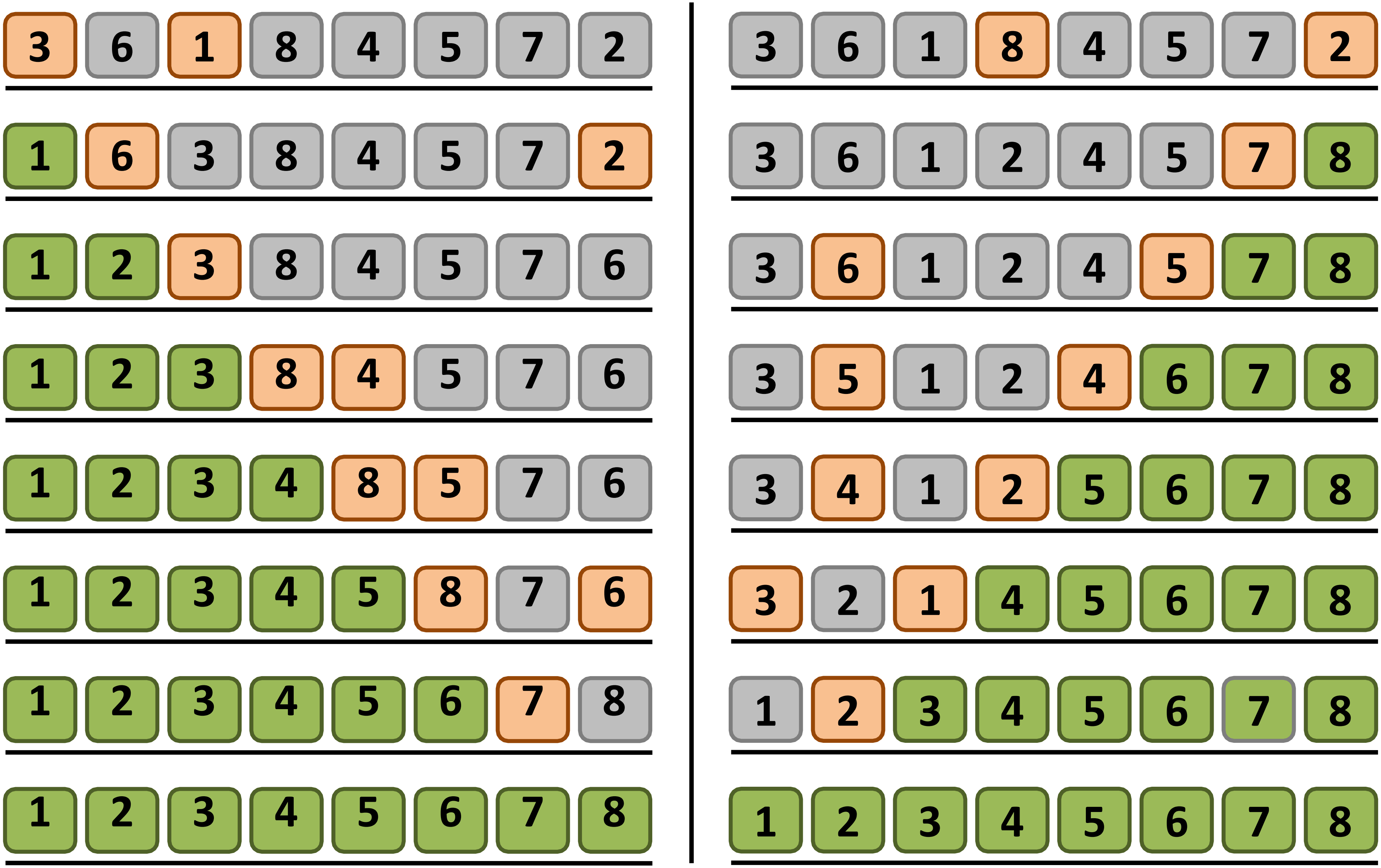
The best case complexity reduces to the order of n, but the worst and average is still n^2. So, overall the complexity is of the order of n^2 again.

Selection Sort

Selection Sort

- In-place comparison-based algorithm.
- Divides the list into two parts
  - The sorted part, which is built up from left to right at the front (left) of the list, and
  - The unsorted part, that occupy the rest of the list at the right end.
- The algorithm proceeds by
  - Finding the smallest (or the largest) element in the unsorted array
  - Swapping it with the leftmost (or the rightmost) unsorted element
  - Moving the boundary one element to the right.
  - This process continues till the array gets sorted.
- Not suitable for large data sets.

Example


3 6 1 8 4 5 7 2
1 6 3 8 4 5 7 2
1 2 3 8 4 5 7 6
1 2 3 8 4 5 7 6
1 2 3 4 8 5 7 6
1 2 3 4 5 8 7 6
1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8
3 6 1 8 4 5 7 2
3 6 1 2 4 5 7 8
3 6 1 2 4 5 7 8
3 5 1 2 4 6 7 8
3 4 1 2 5 6 7 8
3 2 1 4 5 6 7 8
1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8

Algorithm

- **Algorithm selectionSort(a[], n)**
- Input: An array **a** containing **n** elements.
- Output: The elements of **a** get sorted in increasing order.

```
for i = 0 to n – 2
        min = i
        for j = i+1 to n – 1
                if a[j] < a[min]
                        min = j
        if min != i
                Exchange a[min] with a[i]
```

Time Complexity – Selection Sort

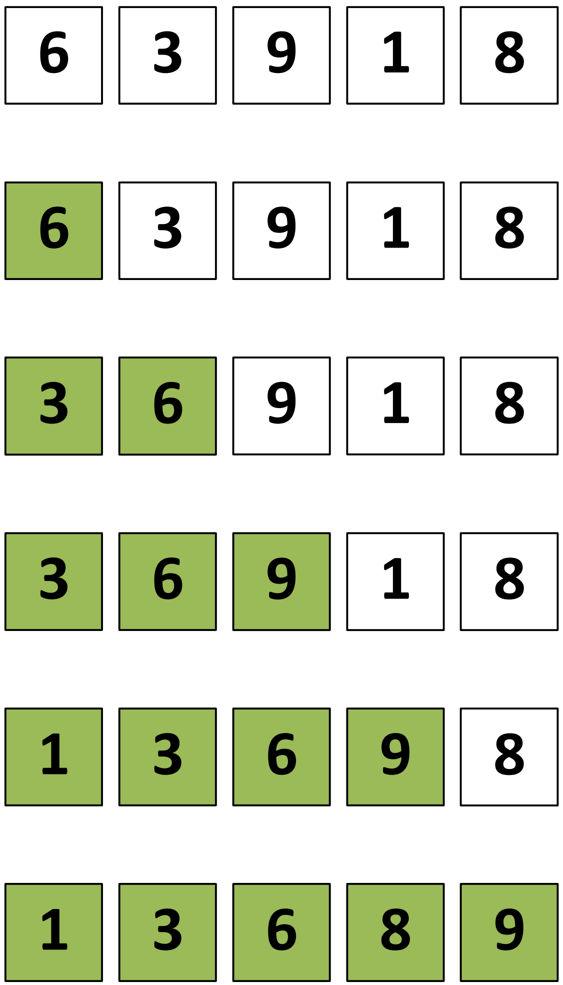
- Best Time Complexity: **Ω** (n^2)
- Average Time Complexity: **Θ** (n^2)
- Worst Time Complexity: **O** (n^2)

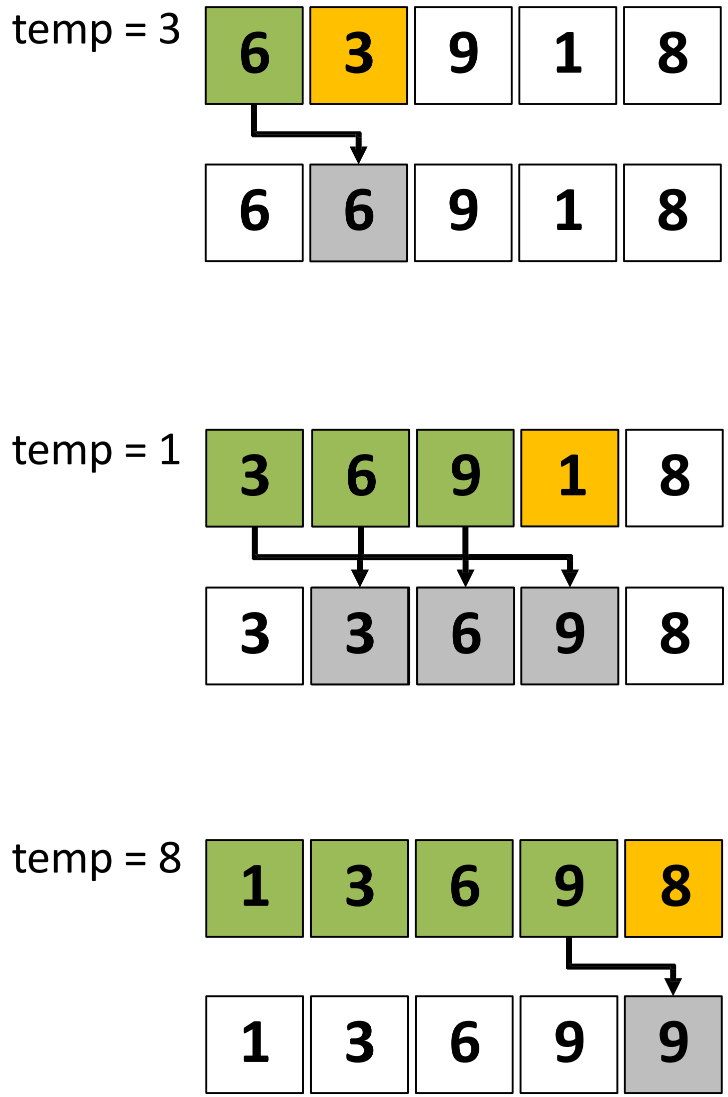
Insertion Sort

Insertion Sort

- An in-place, stable, online comparison-based sorting algorithm.
- Always keeps the lower part of an array in the sorted order.
- A new element will be inserted in the sorted part at an appropriate place.
- The algorithm searches sequentially, move the elements, and inserts the new element in the array.
- Not suitable for large data sets

Example





Algorithm

- **Algorithm insertionSort(a[], n)**
- Input: An array **a** containing **n** elements.
- Output: The elements of **a** get sorted in increasing order.

1. **for** i = 1 to n – 1

2. temp = a[i]

3. j = i

4.while j > 0 and a[j-1] > temp 5.

a[j] = a[j-1]

6. j = j – 1

7. a[j] = temp

Time Complexity – Insertion Sort

- Best Time Complexity: **Ω** (n)
- Average Time Complexity: **Θ** (n^2)
- Worst Time Complexity: **O** (n^2)

Quick Sort

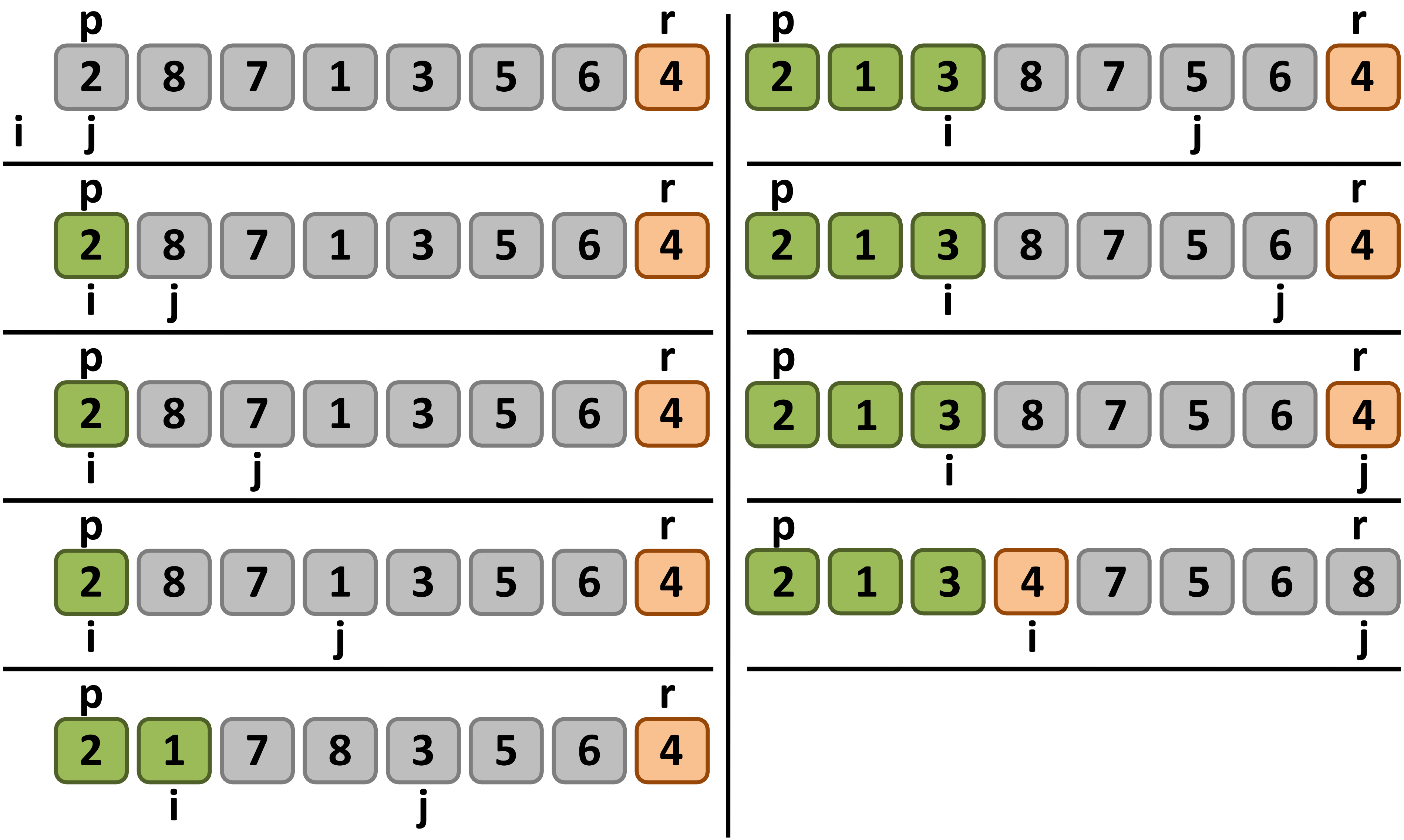
Quick Sort

- Divide and Conquer algorithm. In-place algorithm.
- Picks an element as pivot and partitions the given array around the picked pivot, such that
  - The pivot is placed at its correct position
  - All elements smaller than the pivot are placed before the pivot.
  - All elements greater than the pivot are placed after the pivot.
- Several ways to pick a pivot.
  - The first element.
  - The last element.
  - Any random element.
  - The median.

Algorithm

```
PARTITION(A, p, r)
x = A[r]
i = p – 1
for j = p to r – 1
    if A[j] ≤ x
        i = i + 1
        Exchange A[i] with A[j]
Exchange A[i + 1] with A[r]
return i + 1
```

Partition Procedure



Algorithm

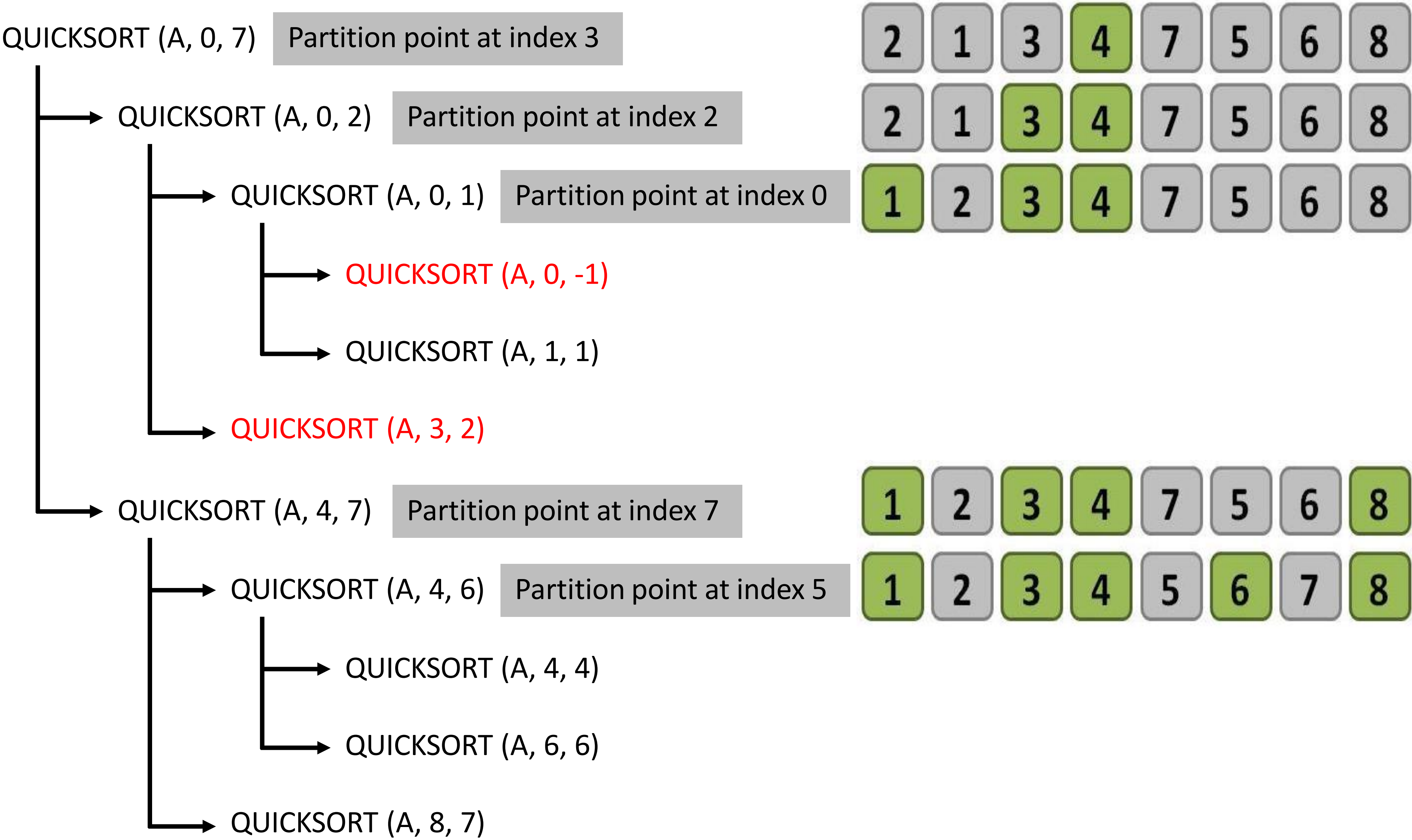
- QUICKSORT(A, p, r)

```
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q – 1)
    QUICKSORT(A, q + 1, r)
```

To sort an array A with n elements, the first call to QUICKSORT is made with p = 0 and r = n – 1.

```
PARTITION(A, p, r)
x = A[r]
i = p – 1
for j = p to r – 1
    if A[j] ≤ x
        i = i + 1
        Exchange A[i] with A[j]
Exchange A[i + 1] with A[r]
return i + 1
```

Call sequence for QUICKSORT



Time Complexity – Quick Sort

- Best Time Complexity: **Ω** $(n\ logn)$
- Average Time Complexity: **Θ** $(n\ logn)$
- Worst Time Complexity: **O** (n^2)

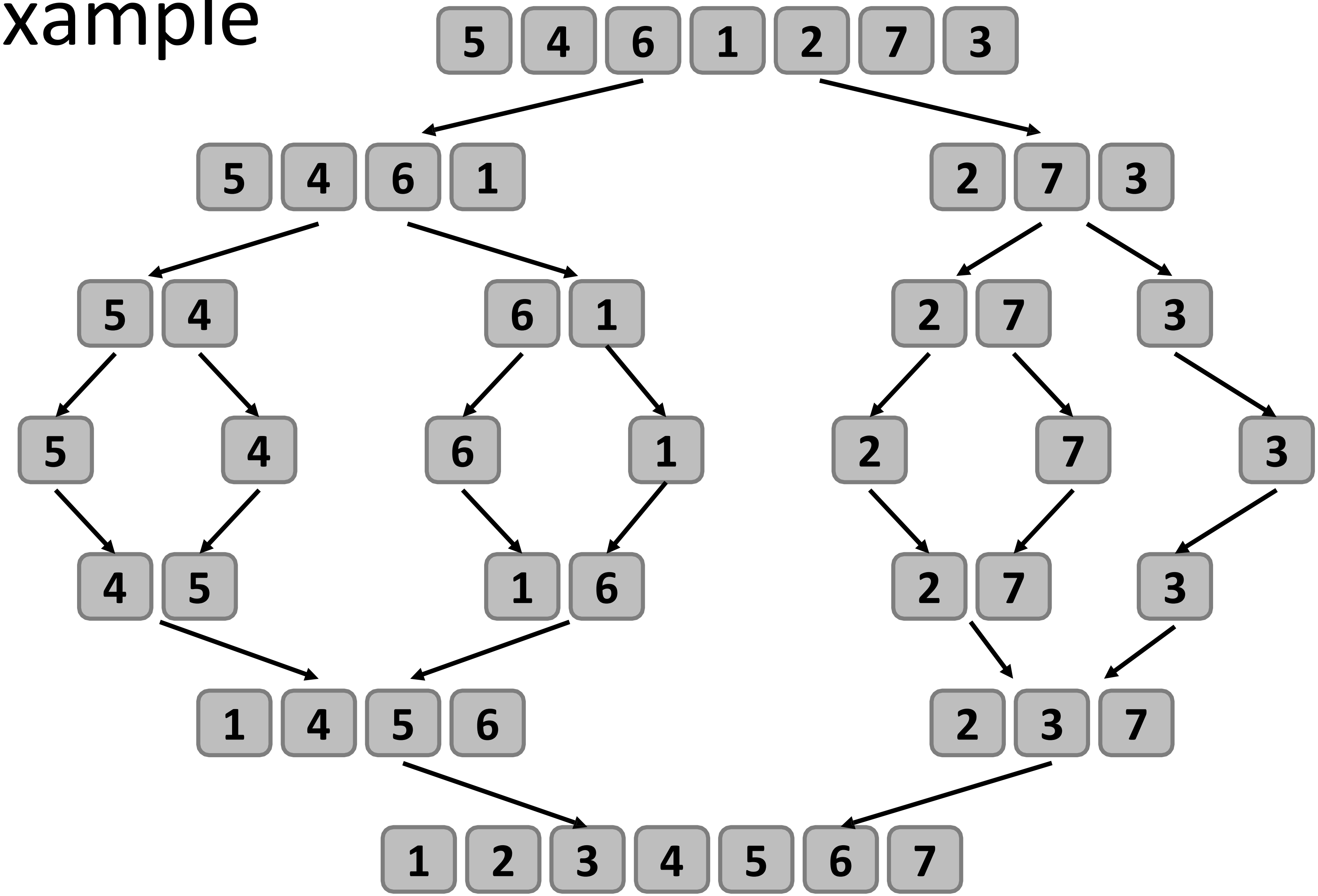
Merge Sort

Merge Sort

- Based on the divide-and-conquer paradigm.
- To sort an array A[p .. r], (initially p = 0 and r = n-1)

1. Divide Step
   - If a given array A has zero or one element, then return as it is already sorted.
   - Otherwise, split A[p…r] into two subarrays A[p…q] and A[q + 1... r], each containing about half of the elements of A[p…r]. That is, q is the halfway point of A[p...r].
2. Conquer Step
   - Recursively sort the two subarrays A[p…q] and A[q + 1…r].
3. Combine Step
   - Combine the elements back in A[p…r] by merging the two sorted subarrays A[p…q] and A[q + 1…r] into a sorted sequence.

Example



Merge Two Sorted Arrays

n1 - #Elements in L
n2 - #Elements in R

L: 1 4 5 6

i i i i i

A: 1 2 3 4 5 6 7

k k k k k k k k

R: 2 3 7

j j j j

```
i = 0, j = 0, and k = p.
while i < n1 and j < n2
        if L[i] ≤ R[j]
                A[k] = L[i]
                i = i + 1
        else
                A[k] = R[j]
                j = j + 1
        k++
while i < n1
        A[k] = L[i]
        i++
        k++
while j < n2
        A[k] = R[j]
        j++
        k++
```

Algorithm

- MERGE-SORT (A, p, r)

1. if p < r
2. q = FLOOR[(p + r)/2]
3. MERGE-SORT(A, p, q)
4. MERGE-SORT(A, q + 1, r)
5. MERGE (A, p, q, r)

- To sort an array A with n elements, the first call to MERGE-SORT is made with p = 0 and r = n – 1.

Contd…

- Algorithm MERGE (A, p, q, r)
- Input: Array A and indices p, q, r such that $p \le q \le r$. Subarrays A[p…q] and A[q + 1…r] are sorted.
- Output: The two subarrays are merged into a single sorted subarray in A[p .. r].

1. $n1 = q - p + 1$
2. $n2 = r - q$
3. Create arrays L[n1] and R[n2]
4. for $i = 0$ to $n1 - 1$
5. $\quad\quad$ $L[i] = A[p + i]$
6. for $j = 0$ to $n2 - 1$
7. $\quad\quad$ $R[j] = A[q + 1 + j]$

Contd…

```
8.  i = 0, j = 0, and k = p.
9.  while i < n1 and j < n2
10.     if L[i] ≤ R[j]
11.         A[k] = L[i]
12.         i = i + 1
13.     else
14.         A[k] = R[j]
15.         j = j + 1
16.     k++
17. while i < n1
18.     A[k] = L[i]
19.     i++
20.     k++
21. while j < n2
22.     A[k] = R[j];
23.     j++;
24.     k++;
```

Call sequence for an array with size 8

- MERGE-SORT (A, 0, 7)
  - MERGE-SORT (A, 0, 3)
    - MERGE-SORT (A, 0, 1)
      - MERGE-SORT (A, 0, 0)
      - MERGE-SORT (A, 1, 1)
      - **MERGE (A, 0, 0, 1)**
    - MERGE-SORT (A, 2, 3)
      - MERGE-SORT (A, 2, 2)
      - MERGE-SORT (A, 3, 3)
      - **MERGE (A, 2, 2, 3)**
    - **MERGE (A, 0, 1, 3)**
  - MERGE-SORT (A, 4, 7)
    - MERGE-SORT (A, 4, 5)
      - MERGE-SORT (A, 4, 4)
      - MERGE-SORT (A, 5, 5)
      - **MERGE (A, 4, 4, 5)**
    - MERGE-SORT (A, 6, 7)
      - MERGE-SORT (A, 6, 6)
      - MERGE-SORT (A, 7, 7)
      - **MERGE (A, 6, 6, 7)**
    - **MERGE (A, 4, 5, 7)**
  - **MERGE (A, 0, 3, 7)**

5	4	6	1	2	7	3	8
5	4	6	1	2	7	3	8
5	4	6	1	2	7	3	8
5	4	6	1	2	7	3	8
5	4	6	1	2	7	3	8
4	5	6	1	2	7	3	8
4	5	6	1	2	7	3	8
4	5	6	1	2	7	3	8
4	5	6	1	2	7	3	8
4	5	1	6	2	7	3	8
1	4	5	6	2	7	3	8
1	4	5	6	2	7	3	8
1	4	5	6	2	7	3	8
1	4	5	6	2	7	3	8
1	4	5	6	2	7	3	8
1	4	5	6	2	7	3	8
1	4	5	6	2	7	3	8
1	4	5	6	2	7	3	8
1	4	5	6	2	7	3	8
1	4	5	6	2	7	3	8
1	4	5	6	2	3	7	8
1	2	3	4	5	6	7	8

Time Complexity – Merge Sort

- Best Time Complexity: **Ω** $(n\ log n)$
- Average Time Complexity: **Θ** $(n\ log n)$
- Worst Time Complexity: **O** $(n\ log n)$