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

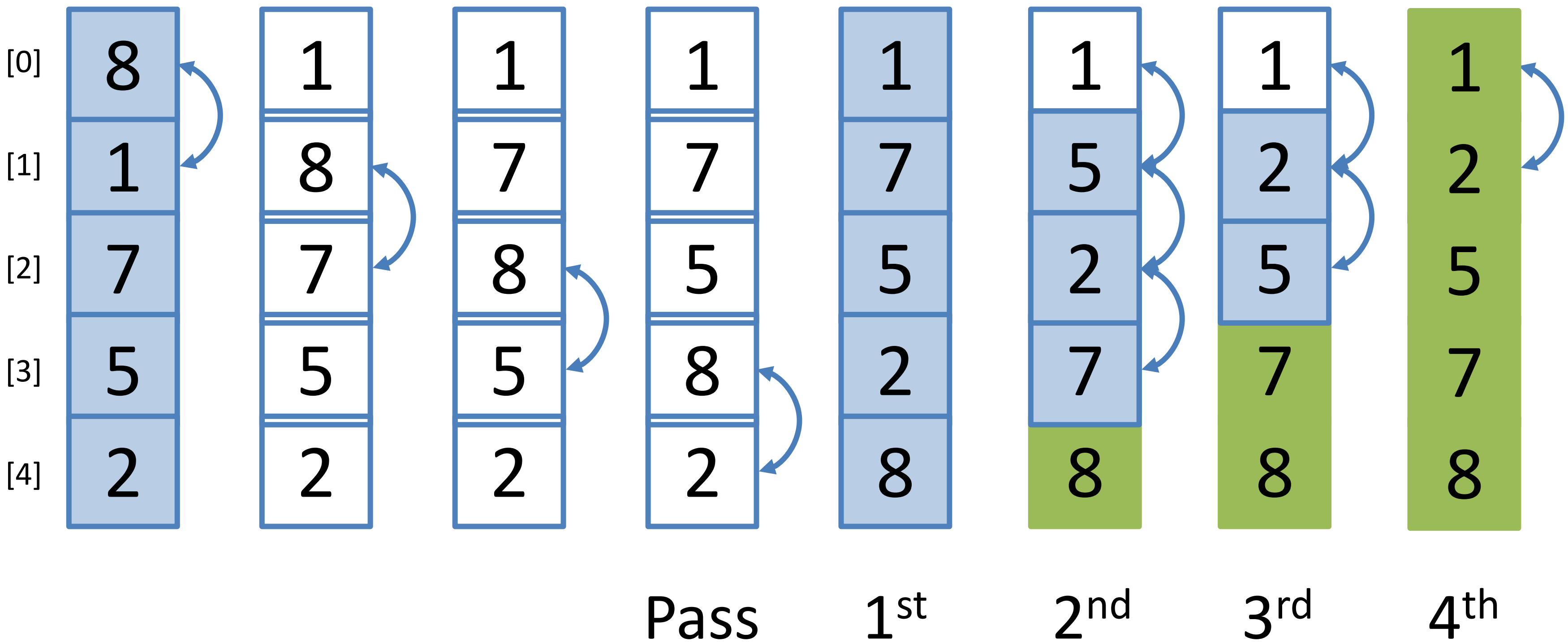
- In place.
- Stable.
- Online.

10	30	10	50	70	40	10	20

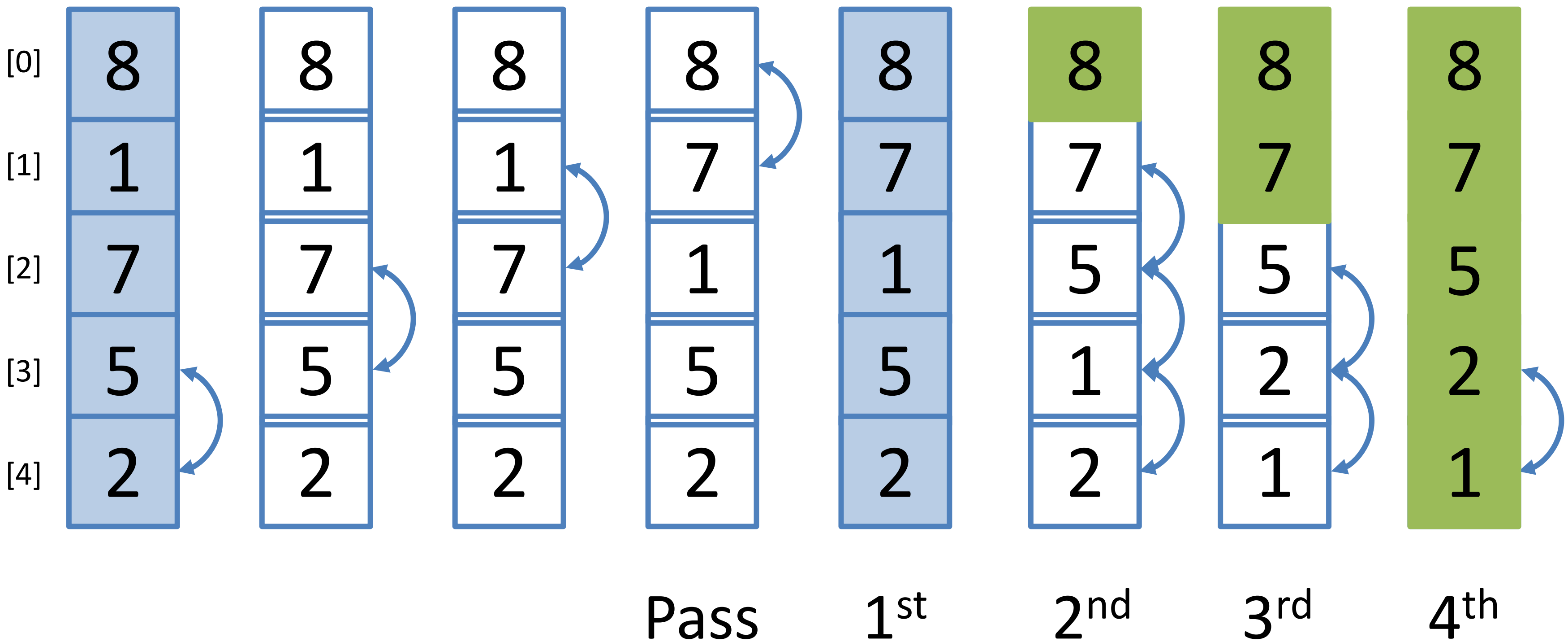
10	10	10	20	30	40	50	70

Bubble Sort

Bubble Sort – Ascending



Bubble Sort – Descending



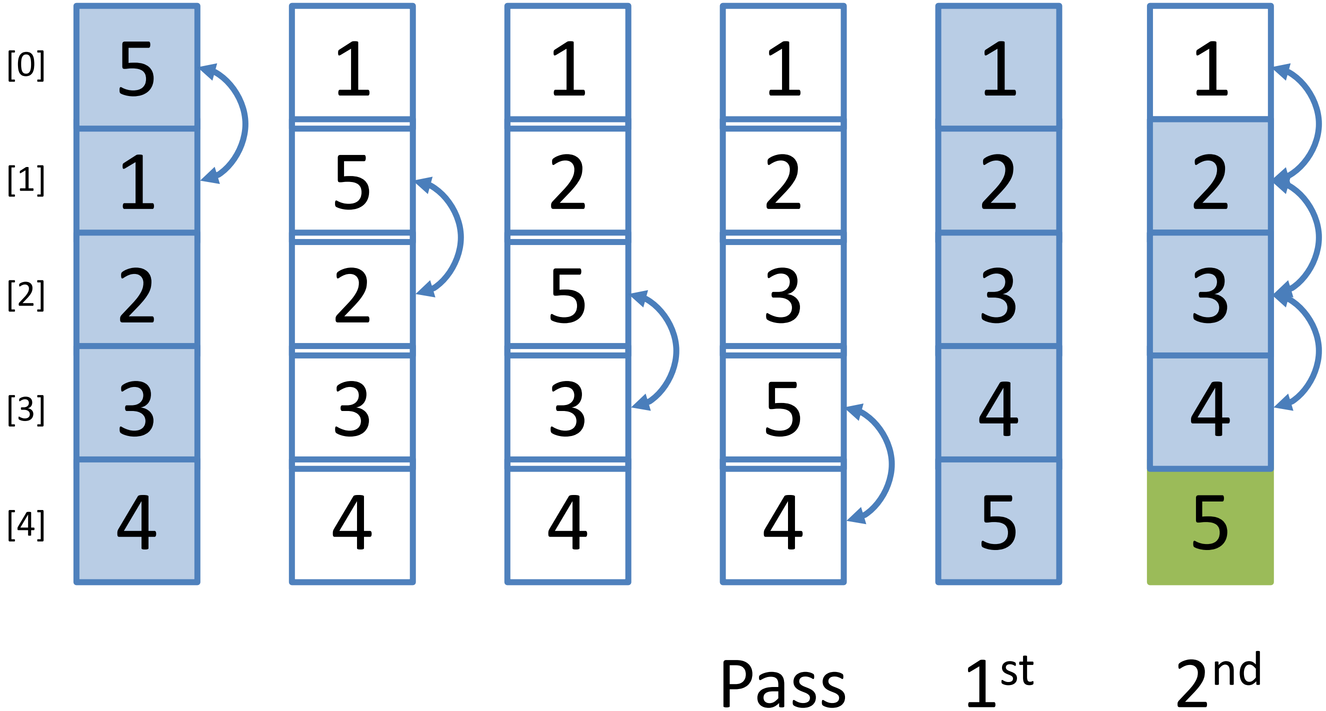
Algorithm – Bubble Sort

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]

In all the cases, complexity is of the order of 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;
```

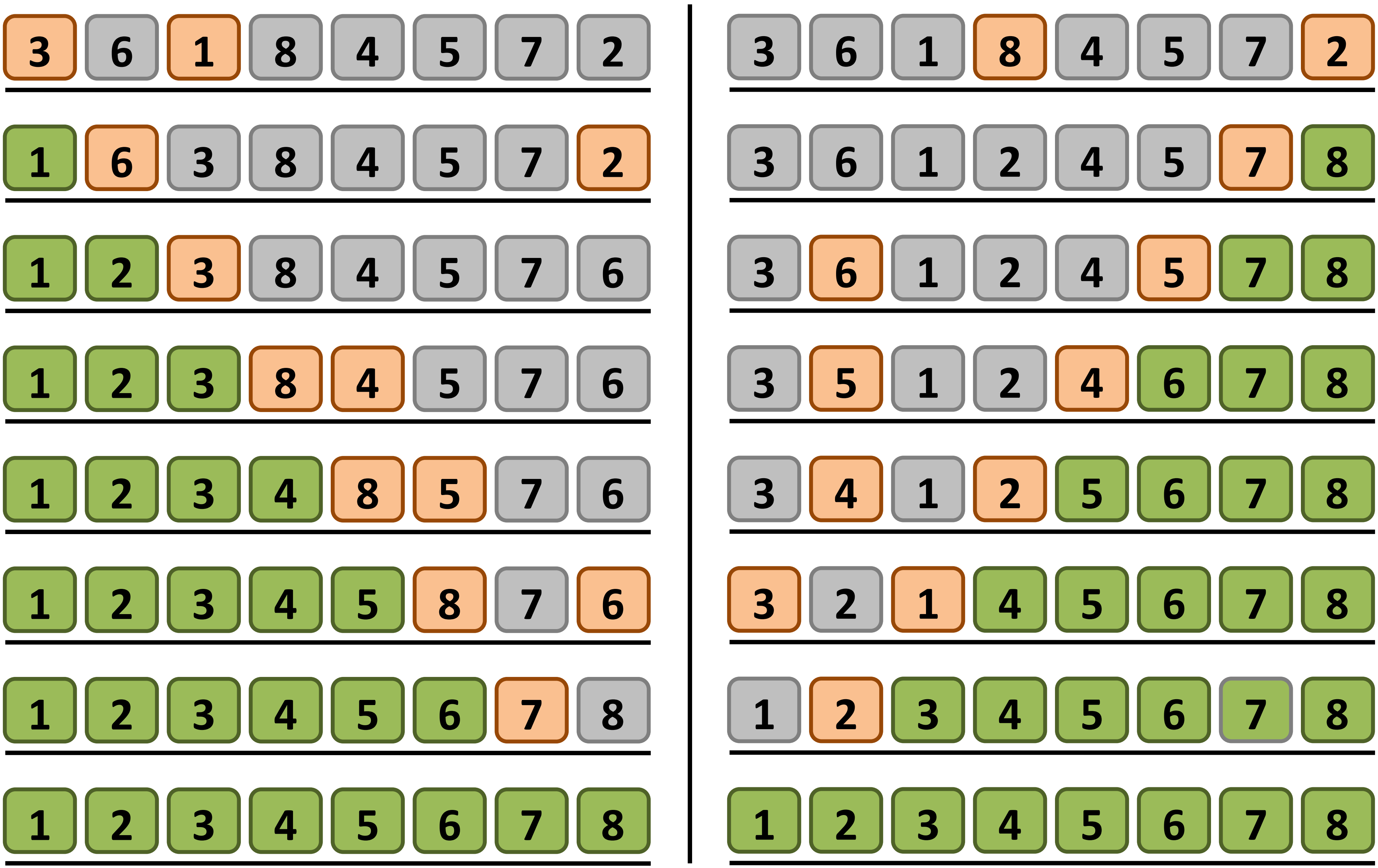
The best case complexity reduces to the order of n, but the worst and average is still n^2. So, overall the complexity is of the order of n^2 again.

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.
- Complexity is $O(n^2)$, where n is the number of elements.

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.

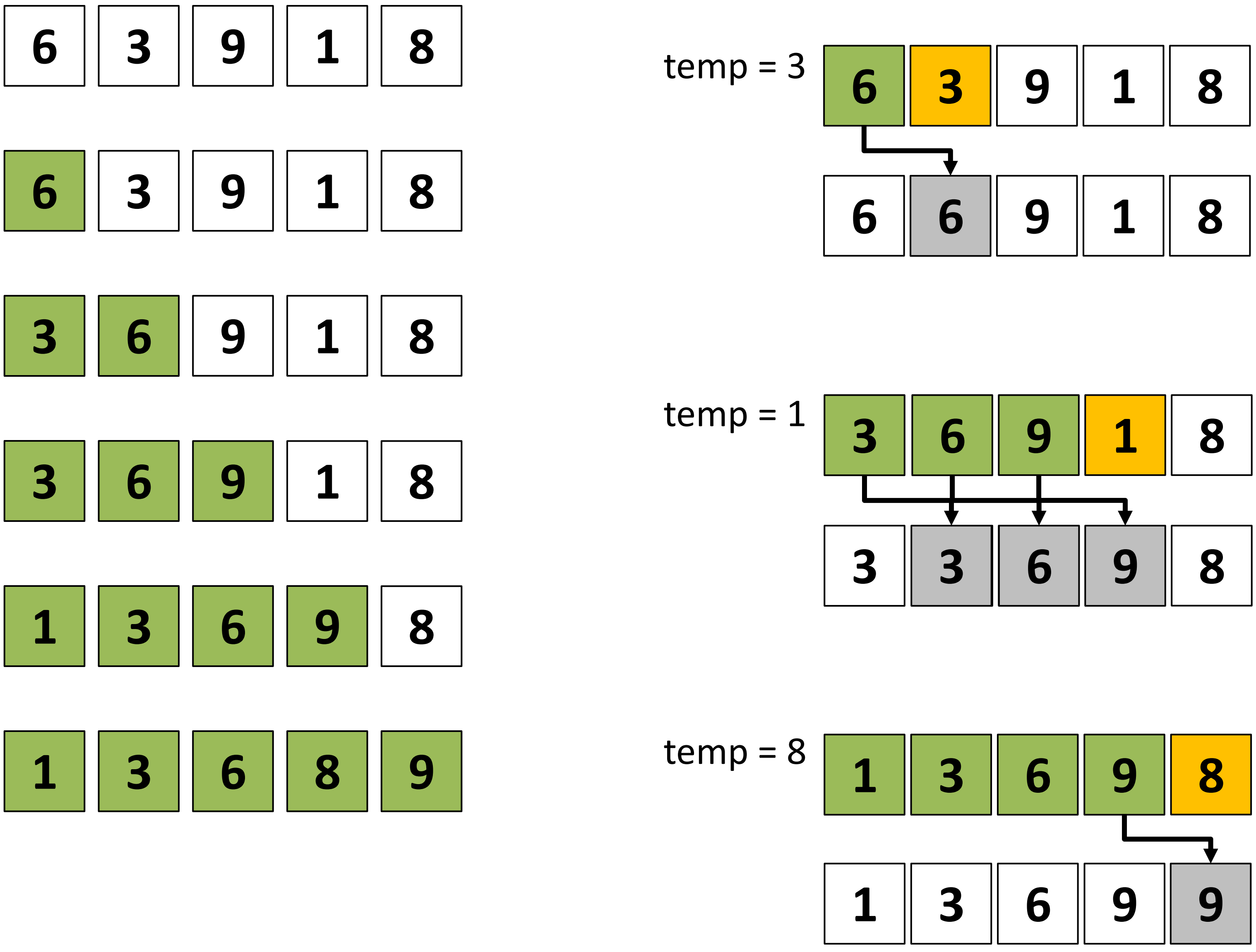
```
1.  for i = 0 to n – 2
2.          min = i
3.          for j = i+1 to n – 1
4.                  if a[j] < a[min]
5.                          min = j
6.          if min != i
7.                  Exchange a[min] with a[i]
```

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
- Complexity is $\boldsymbol{O}(\boldsymbol{n}^2)$, where n is the number of elements.
- Best case complexity is $\boldsymbol{O}(\boldsymbol{n})$.

Example



6 3 9 1 8
6 3 9 1 8
3 6 9 1 8
3 6 9 1 8
1 3 6 9 8
1 3 6 8 9
temp = 3
6 3 9 1 8
6 6 9 1 8
temp = 1
3 6 9 1 8
3 3 6 9 8
temp = 8
1 3 6 9 8
1 3 6 9 9

Algorithm

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

```
for i = 1 to n – 1
        temp = a[i]
        j = i
        while j > 0 and a[j-1] > temp
                a[j] = a[j-1]
                j = j – 1
        a[j] = temp
```

Shell Sort

Shell Sort

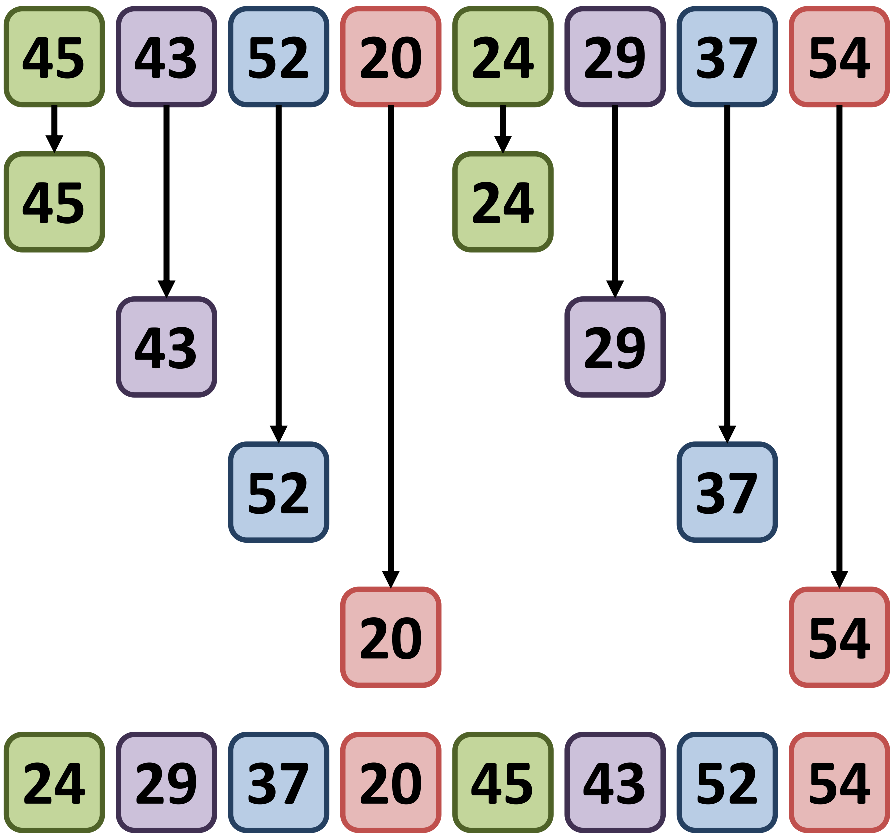
- Improved or generalized insertion sort.
- An in–place comparison sort.
- Also known as diminishing increment sort.
- Breaks the original list into a number of smaller sublists, each of which is sorted using an insertion sort.
- Instead of breaking the list into sublists of contiguous items, the algorithm uses a unique way to choose the sublists.
  - An increment (say h), sometimes called the gap or the interval.
  - A sublist contains all the elements that are i elements apart.
- Complexity lies in between $O(n)$ and $O(n^2)$. Still an open problem.

Contd…

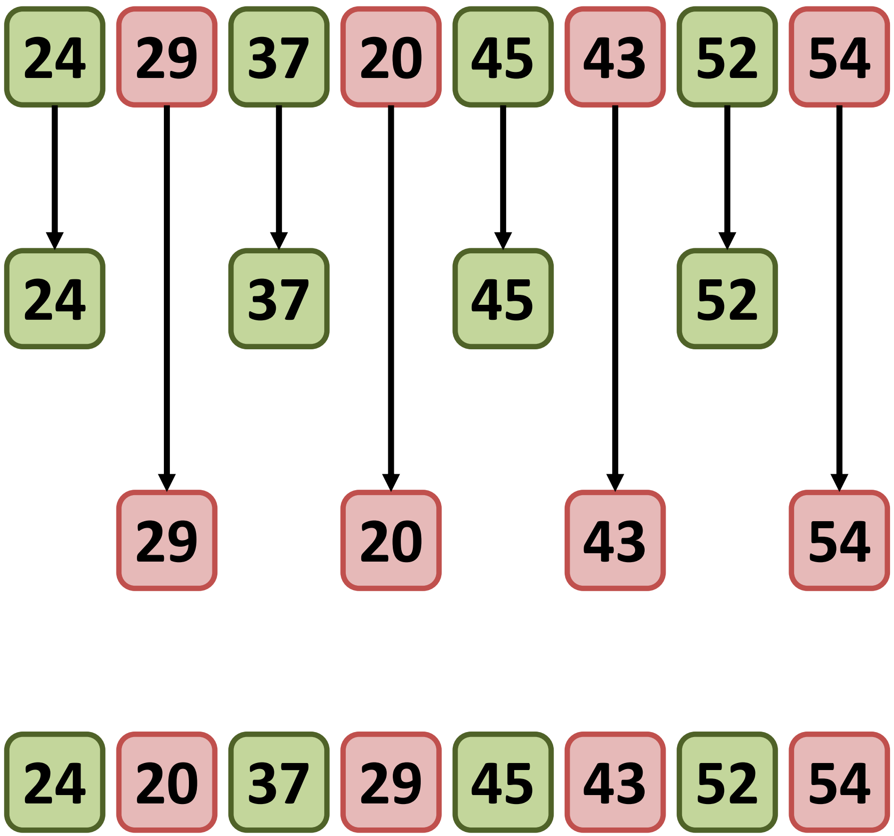
- Increment Sequences:
  - Shell's original sequence: N/2 , N/4 , …, 1
  - Hibbard's increments: 1, 3, 7, …, $2^k - 1$
  - Knuth's increments: 1, 4, 13, …, (3k + 1)
  - Sedgewick's increments: 1, 5, 19, 41, 109, ….
    - Merging of $(9 \times 4^i) - (9 \times 2^i) + 1$ and $4^i - (3 \times 2^i) - 1$
- Start with higher intervals and then reduce the interval after each pass as per the chosen sequence.

Example

- Using original sequence
  - h = N/2 = 8/2 = 4.



  - h = h/2 = 4/2 = 2.



Contd…

– h = h/2 = 2/2 = 1.

24	20	37	29	45	43	52	54
24	20	37	29	45	43	52	54
20	24	37	29	45	43	52	54
20	24	37	29	45	43	52	54
20	24	37	29	45	43	52	54
20	24	29	37	45	43	52	54
20	24	29	37	45	43	52	54
20	24	29	37	45	43	52	54
20	24	29	37	43	45	52	54
20	24	29	37	43	45	52	54
20	24	29	37	43	45	52	54

Algorithm

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

```
for (gap = n/2; gap > 0; gap /=2)
{ for (i = gap; i < n; i++)
    { temp = a[i]
    for j = i; j >= gap && a[j - gap] > temp; j -= gap)
            a[j] = a[j-gap]
    a[j]= temp;       }
}
```

Quick Sort

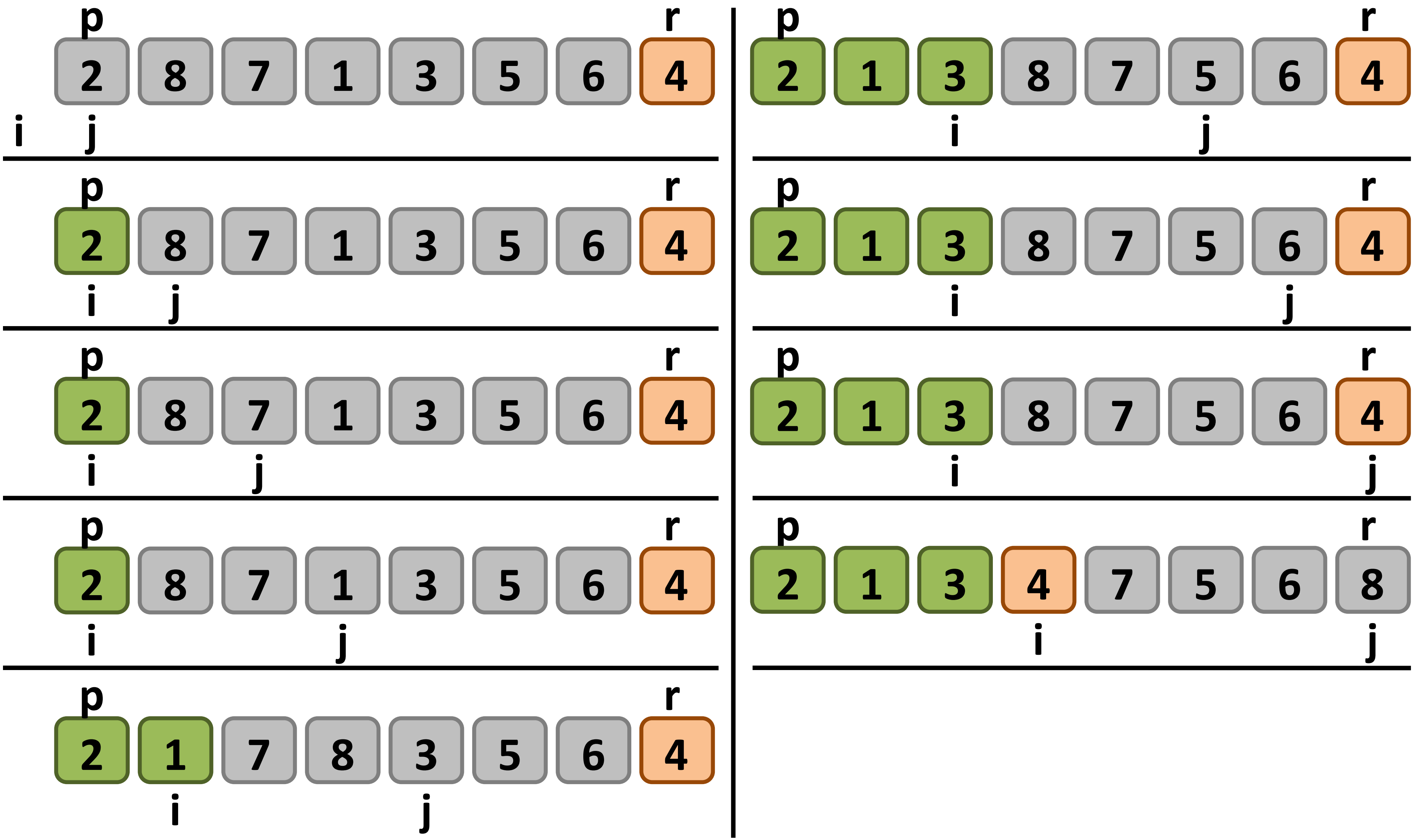
Quick Sort

- 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

2	8	7	1	3	5	6	4

- QUICKSORT (A, 0, 7) — Partition point at index 3 — 2 1 3 4 7 5 6 8
  - QUICKSORT (A, 0, 2) — Partition point at index 2 — 2 1 3 4 7 5 6 8
    - QUICKSORT (A, 0, 1) — Partition point at index 0 — 1 2 3 4 7 5 6 8
      - QUICKSORT (A, 0, -1)
      - QUICKSORT (A, 1, 1)
    - QUICKSORT (A, 3, 2)
  - QUICKSORT (A, 4, 7) — Partition point at index 7 — 1 2 3 4 7 5 6 8
    - QUICKSORT (A, 4, 6) — Partition point at index 5 — 1 2 3 4 5 6 7 8
      - QUICKSORT (A, 4, 4)
      - QUICKSORT (A, 6, 6)
    - QUICKSORT (A, 8, 7)

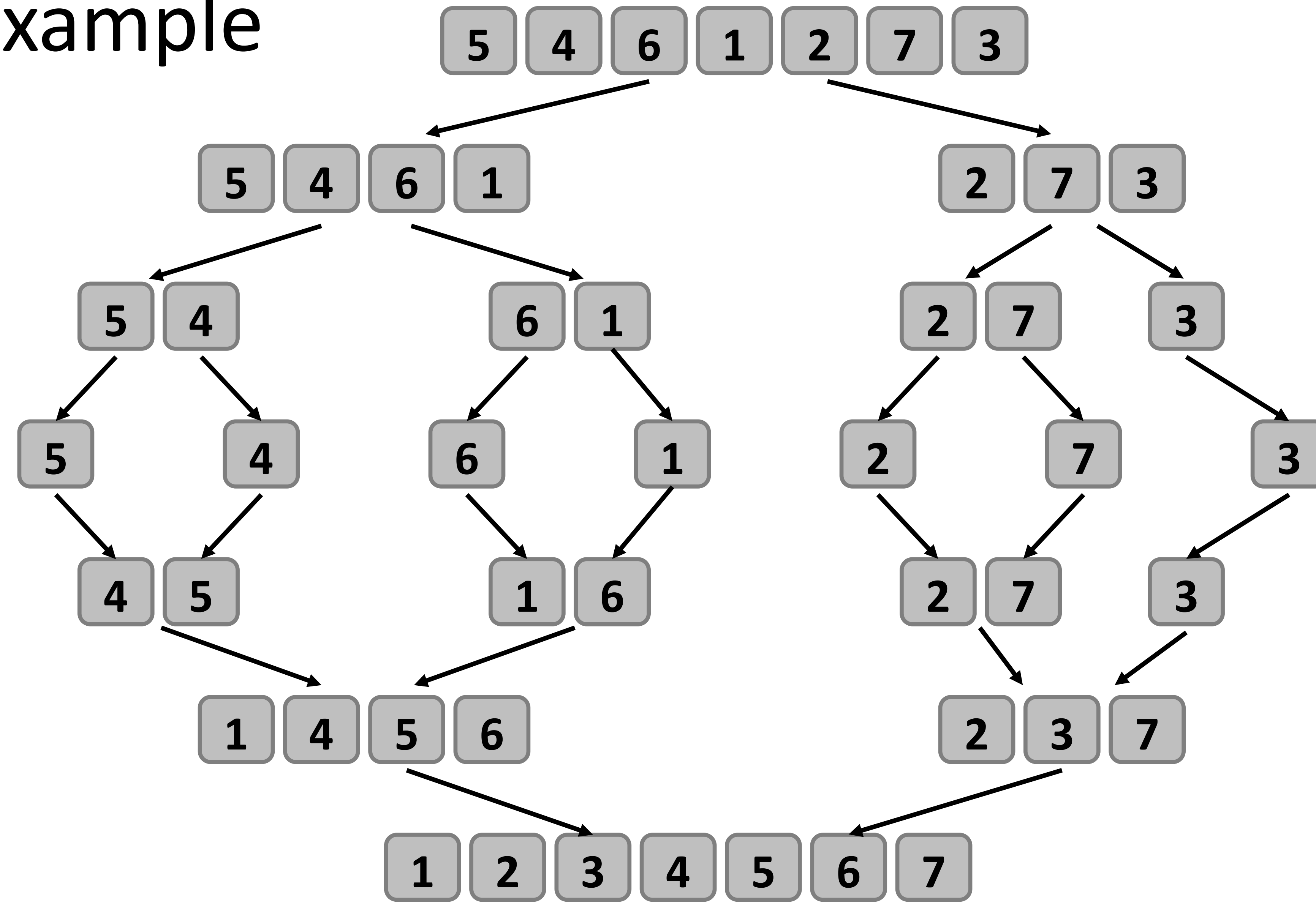
Merge Sort

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)

```
if p < r
    q = FLOOR[(p + r)/2]
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    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. L[i] = A[p + i]
6. for j = 0 to n2 – 1
7. R[j] = A[q + 1 + j]

Contd…

```
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++;
```

Call sequence for an array with size 8

Call								
MERGE-SORT (A, 0, 7)	5	4	6	1	2	7	3	8
→ MERGE-SORT (A, 0, 3)	5	4	6	1	2	7	3	8
→ → MERGE-SORT (A, 0, 1)	5	4	6	1	2	7	3	8
→ → → MERGE-SORT (A, 0, 0)	5	4	6	1	2	7	3	8
→ → → MERGE-SORT (A, 1, 1)	5	4	6	1	2	7	3	8
→ → → **MERGE (A, 0, 0, 1)**	4	5	6	1	2	7	3	8
→ → MERGE-SORT (A, 2, 3)	4	5	6	1	2	7	3	8
→ → → MERGE-SORT (A, 2, 2)	4	5	6	1	2	7	3	8
→ → → MERGE-SORT (A, 3, 3)	4	5	6	1	2	7	3	8
→ → → **MERGE (A, 2, 2, 3)**	4	5	1	6	2	7	3	8
→ → **MERGE (A, 0, 1, 3)**	1	4	5	6	2	7	3	8
→ MERGE-SORT (A, 4, 7)	1	4	5	6	2	7	3	8
→ → MERGE-SORT (A, 4, 5)	1	4	5	6	2	7	3	8
→ → → MERGE-SORT (A, 4, 4)	1	4	5	6	2	7	3	8
→ → → MERGE-SORT (A, 5, 5)	1	4	5	6	2	7	3	8
→ → → **MERGE (A, 4, 4, 5)**	1	4	5	6	2	7	3	8
→ → MERGE-SORT (A, 6, 7)	1	4	5	6	2	7	3	8
→ → → MERGE-SORT (A, 6, 6)	1	4	5	6	2	7	3	8
→ → → MERGE-SORT (A, 7, 7)	1	4	5	6	2	7	3	8
→ → → **MERGE (A, 6, 6, 7)**	1	4	5	6	2	7	3	8
→ → **MERGE (A, 4, 5, 7)**	1	4	5	6	2	3	7	8
→ **MERGE (A, 0, 3, 7)**	1	2	3	4	5	6	7	8

Counting Sort

Counting Sort

- Assumes that the input consists of integers in a small range 1 to k, for some integer k.
- Runs in $O(n + k)$ time.
  - $k = O(n)$, the sort runs in $\theta(n)$ time.
- For each element x, the algorithm
  - First determines the number of elements less than x.
  - Then directly place the element into its correct position.

Example

A[]

0	1	2	3	4	5	6	7
3	**6**	**4**	**1**	**3**	**4**	**1**	**4**

k = 6
n = 8

- Compute frequency of k elements, i.e. array C.

C[]

0	1	2	3	4	5	6
0	**2**	**0**	**2**	**3**	**0**	**1**

- Update C to store cumulative frequency.

C[]

0	1	2	3	4	5	6
0	**2**	**2**	**4**	**7**	**7**	**8**

A[]									C[]								B[]								
0	1	2	3	4	5	6	7		0	1	2	3	4	5	6		0	1	2	3	4	5	6	7	8
3	**6**	**4**	**1**	**3**	**4**	**1**	**4**		**0**	**2**	**2**	**4**	**7**	**7**	**8**									**4**	
0	1	2	3	4	5	6	7		0	1	2	3	4	5	6		0	1	2	3	4	5	6	7	8
3	**6**	**4**	**1**	**3**	**4**	**1**	**4**		**0**	**2**	**2**	**4**	**6**	**7**	**8**				**1**					**4**	
0	1	2	3	4	5	6	7		0	1	2	3	4	5	6		0	1	2	3	4	5	6	7	8
3	**6**	**4**	**1**	**3**	**4**	**1**	**4**		**0**	**1**	**2**	**4**	**6**	**7**	**8**				**1**				**4**	**4**	
0	1	2	3	4	5	6	7		0	1	2	3	4	5	6		0	1	2	3	4	5	6	7	8
3	**6**	**4**	**1**	**3**	**4**	**1**	**4**		**0**	**1**	**2**	**4**	**5**	**7**	**8**				**1**		**3**		**4**	**4**	
0	1	2	3	4	5	6	7		0	1	2	3	4	5	6		0	1	2	3	4	5	6	7	8
3	**6**	**4**	**1**	**3**	**4**	**1**	**4**		**0**	**1**	**2**	**3**	**5**	**7**	**8**			**1**	**1**		**3**		**4**	**4**	
0	1	2	3	4	5	6	7		0	1	2	3	4	5	6		0	1	2	3	4	5	6	7	8
3	**6**	**4**	**1**	**3**	**4**	**1**	**4**		**0**	**0**	**2**	**3**	**5**	**7**	**8**			**1**	**1**		**3**	**4**	**4**	**4**	
0	1	2	3	4	5	6	7		0	1	2	3	4	5	6		0	1	2	3	4	5	6	7	8
3	**6**	**4**	**1**	**3**	**4**	**1**	**4**		**0**	**0**	**2**	**3**	**4**	**7**	**8**			**1**	**1**		**3**	**4**	**4**	**4**	**6**
0	1	2	3	4	5	6	7		0	1	2	3	4	5	6		0	1	2	3	4	5	6	7	8
3	**6**	**4**	**1**	**3**	**4**	**1**	**4**		**0**	**0**	**2**	**2**	**4**	**7**	**7**			**1**	**1**	**3**	**3**	**4**	**4**	**4**	**6**

Algorithm

- Algorithm countingSort(A,n,k)
- Input: Array A, its size n, and the maximum integer k in the list.
- Output: The elements of A get sorted in increasing order.

```
for i = 0 to k
        C[i] = 0
for i = 0 to n – 1
        C[A[i]] = C[A[i]] + 1
for i = 1 to k
        C[i] = C[i] + C[i-1]
for i = n – 1 to 0
        B[C[A[i]]] = A[i]
        C[A[i]] = C[A[i]] - 1
for i = 0 to n – 1
        A[i] = B[i+1]
```

Radix Sort

- Similar to alphabetizing a large list of names.
  - List of names is first sorted according to the first letter of each names, that is, the names are arranged in 26 classes.
  - Then sort on the next most significant letter, and so on.
- Radix sort do counter-intuitively by sorting on the least significant digits first.
  - First pass sorts entire list on the least significant digit.
  - Second pass sorts entire list again on the second least-significant digits and so on.

Example: Least Significant Digit

INPUT	1st pass	2nd pass	3rd pass
329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

Algorithm

- Assumption: Each element in the n-element array A has d digits, where digit 1 is the least-significant digit and d is the most-significant digit.

- radixSort(A, d)

```
1. for i = 1 to d
2.     use a stable sort to sort A on digit i
       // counting sort will do the job
```