

Data structures and algorithms

Spring 2025

Sorting Algorithms II

Lecturer: Do Thuy Duong

Contents

- Quick sort
- Heap sort
- Linear time sorting algorithm
 - Counting sort (Bucket sort)
 - Radix sort

Quicksort [1]

Quick sort is a kind of **divide and conquer** algorithm

1. Select a value as a pivot
2. Divide array of data into 2 parts: left < pivot value, right > pivot value
3. Repeat (1) and (2) for the left part and right part


Quicksort [2]

For example, we simply use **the first element** (ie. 25) as pivot for partitioning:

$a[0..N-1]$

Eg. 25 10 57 48 37 12 92 86 33

=> 12 10 **25** 48 37 57 92 86 33


pivot

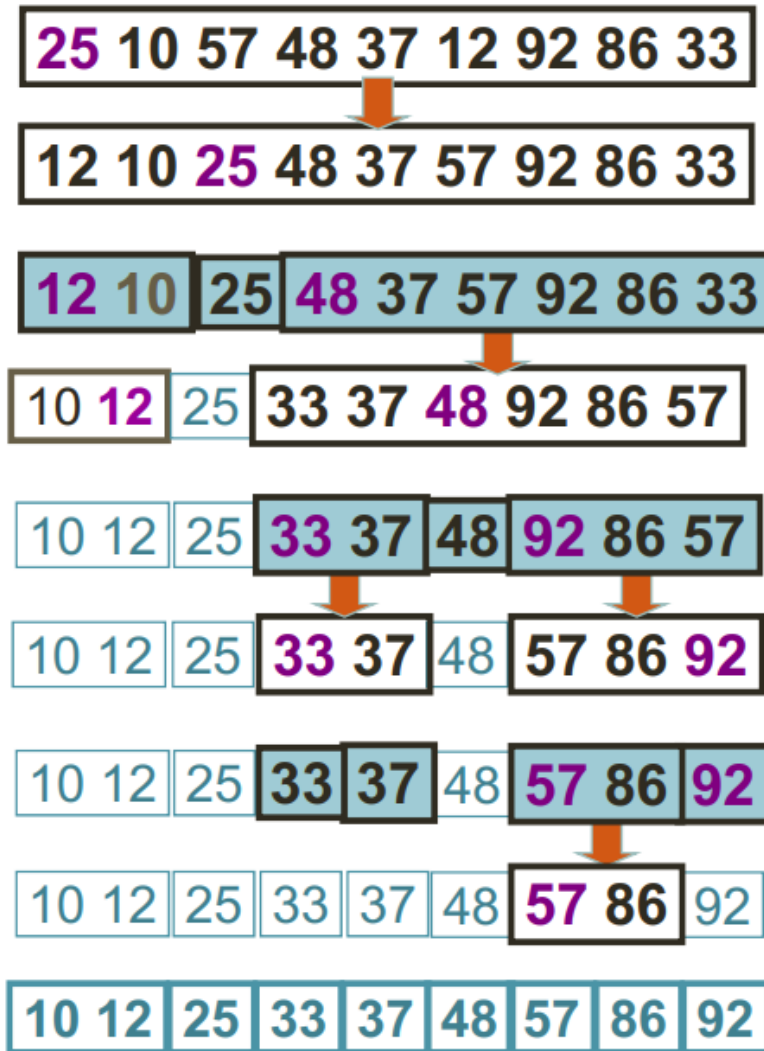
After the partitioning step, we received two sub-lists, the left contains all elements $<$ pivot, the right contains all elements $>$ pivot.

After this step pivot is in its correct position.

We repeat sorting for the left and the right sub-list.

Quicksort[3]

Original



Sorted

Quicksort [4]

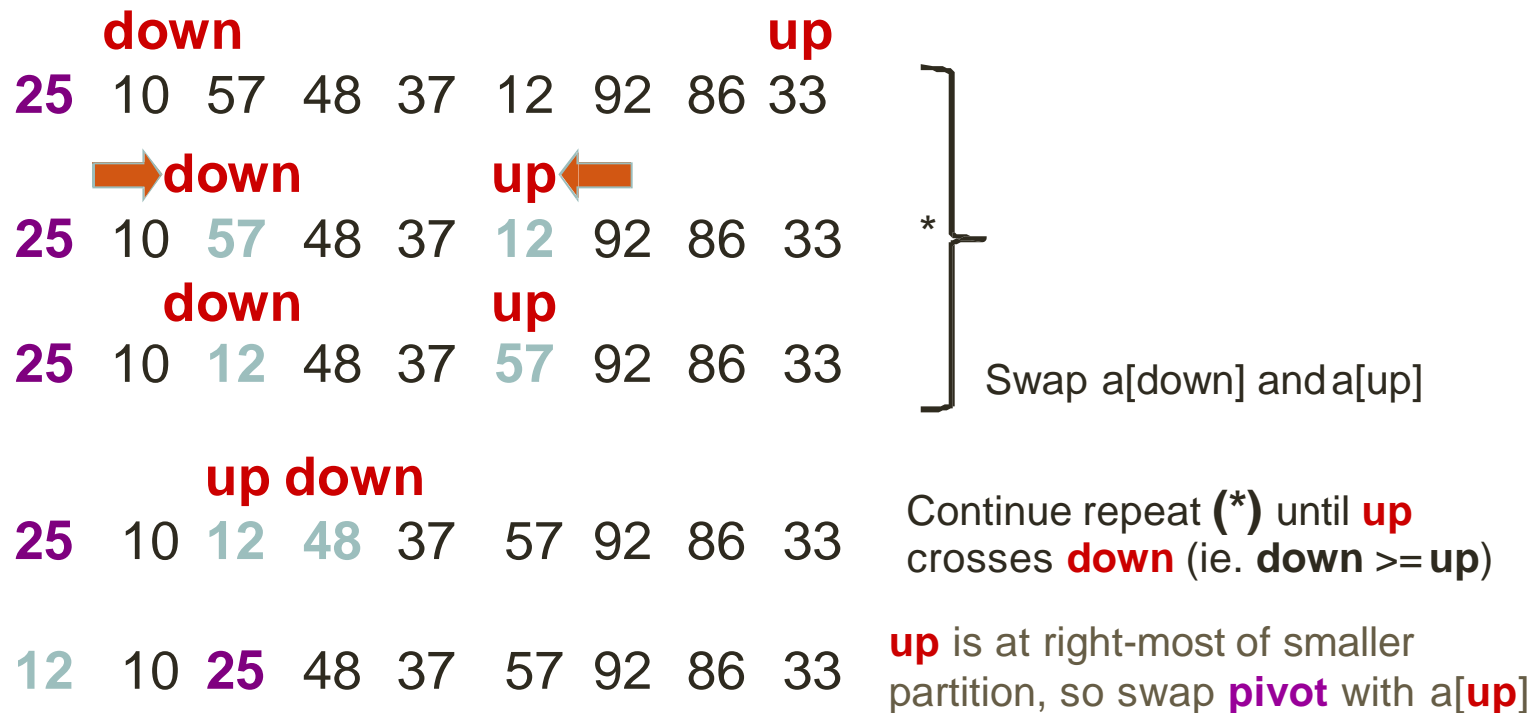
Algorithm to partition the array:

1. Having 2 indices of sorting range: 1st and the last called **down** index and **up** index
2. Move **down** towards **up** until $a[\text{down}] > \text{pivot}$
3. Move **up** towards **down** until $a[\text{up}] \leq \text{pivot}$
4. Swap $a[\text{up}]$ and $a[\text{down}]$ and repeat from step 2 until $\text{up} \leq \text{down}$
5. Swap **pivot** and $a[\text{up}]$, then return **up**

Quicksort[5]

Original: 25 10 57 48 37 12 92 86 33

Partitioning: Select pivot=25. Use 2 indices:



Quicksort[6]

Code

Algorithm QuickSort(A , $left$, $right$):

Input: An array A storing integer elements.

Output: A is sorted in ascending order from left to right

if $left < right$ **then**

$pos \leftarrow \text{Partition}(A, left, right)$

 QuickSort(A , $left$, $pos-1$)

 QuickSort(A , $pos+1$, $right$)

Quicksort[7]

Code

Algorithm Partition(*A*, *left*, *right*):

Input: Sub-array *A* from *left* to *right*
Choose *a*[*left*] as the pivot

Output: Rearranged sub-array so that the left part is the elements < pivot, the right part is the elements > pivot. New position of pivot is returned.

down \leftarrow *left* *up* \leftarrow *right* *pivot* \leftarrow *a*[*left*]

while *down* < *up*

while *a*[*down*] ≤ *pivot*

down++

while *a*[*up*] > *pivot*

up--

tmp \leftarrow *a*[*down*] *a*[*down*] \leftarrow *a*[*up*] *a*[*up*] \leftarrow *tmp*

endwhile

a[*left*] \leftarrow *a*[*up*] *a*[*up*] \leftarrow *pivot*

return *up*

Quicksort[8]

Analysis of Quicksort:

- See textbook section 7.7.5
- Suppose that the partitioning divides the original problem A into two sub-problems A_1 and A_2 with the size of A_1 is i and the side of A_2 is $N-1-i$.

$$T(N) = T(N-1-i) + T(i) + cN$$

- We consider the following cases

Quicksort[9]

- **The worst-case:**

The partitioning produces one sub-problem with $n-1$ elements and one with 0 element ($i=0$).

$$T(N) = T(N-1) + cN$$

- The time complexity is $O(T(n)) = O(n^2)$

- **The best-case:**

Every time the partitioning produces two sub-problems that have equal size, $i=n/2$.

$$T(N) = 2T(N/2) + cN$$

- The time complexity is $O(T(n)) = O(n \log n)$

Quicksort[10]

- **The average-case:**

Assume the size i of A_1 appears with the probability $1/N$. Then we have:

$$T(i) = T(N - 1 - i) = \frac{1}{N} \sum_{j=0}^{N-1} T(j)$$

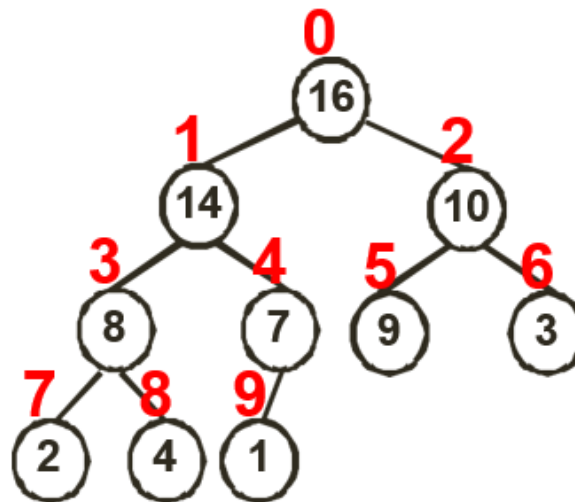
$$T(N) = \frac{1}{N} \sum_{j=0}^{N-1} T(j) + cN$$

- The time complexity is **$O(T(n)) = O(n \log n)$**

Heap sort[1]

A **(binary) heap** is a **complete Binary Tree** used to store data efficiently to get the max or min element based on its type. A Binary Heap is **either Min Heap or Max Heap**.

- A **complete binary tree**: all levels of the tree, except possibly the last one are fully filled; if the last level of the tree is not complete, the nodes of that level are filled from left to right.
- In a **Min/Max Binary Heap**, any given node is always smaller/greater than its child nodes and the root node is the smallest/largest among all other nodes.

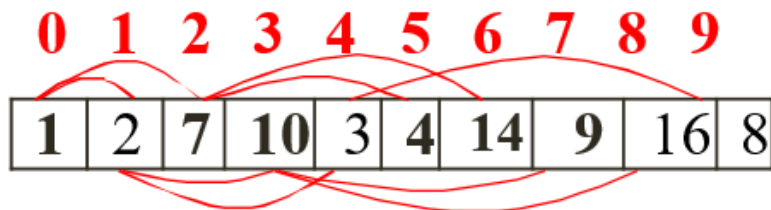


Heap sort

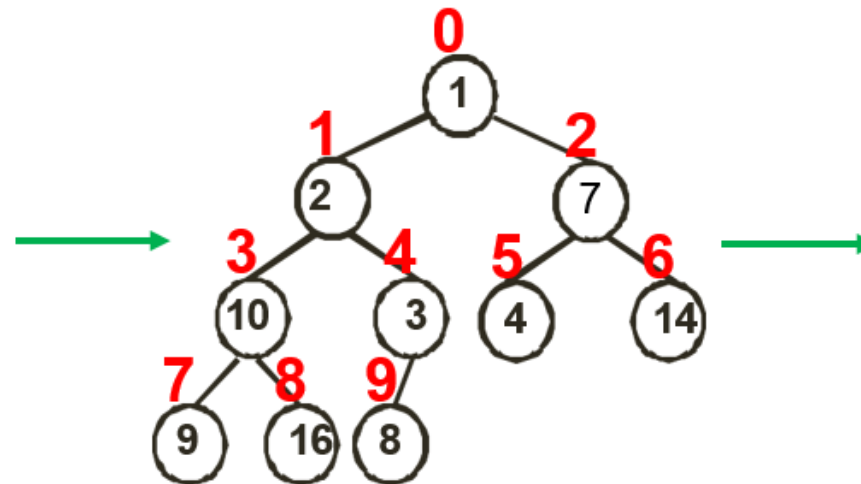
The goal: is to sort the given array **A**.



Idea:



Array



A complete binary tree

**A max
heap?**

$$\text{Parent}(i) = \lfloor (i-1)/2 \rfloor$$

$$\text{Left}(i) = 2i+1$$

$$\text{Right}(i) = 2i+2$$

Heap sort[2]

Heap sort algorithm:

Step 1: Build max/min heap: convert the array to a max heap (or min heap) (heap has N nodes)

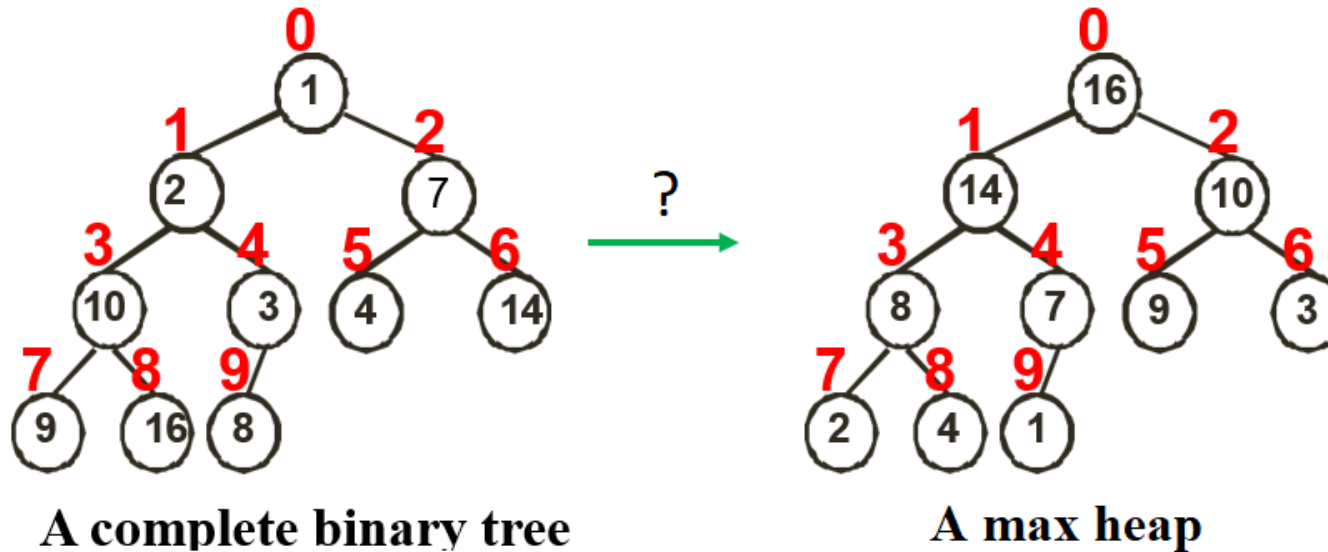
Step 2: Sorting:

- While the heap has more than 1 node
 - **Step 2.1: Swap** the root and the last leaf (the first and the last element of the array). **Reduce the size** of the heap by one
 - **Step 2.2: Re-arrange** remaining elements to **build a max heap**. (build a heap with N-1 nodes)

Heap sort[3]

Step 1. Build a max heap from the array

- Convert the array of unsorted elements into a max heap



- In a max-heap, each sub-tree must be a max-heap too.
- Except leaf nodes, all other subtrees may not fulfill the requirement for a max-heap → **trickle-down all non-leaf nodes one by one, starting from the lowest ones to the top. ie. 3, 10, 7, 2, 1**

Heap sort[4]

Step 1. Build a max heap from the array

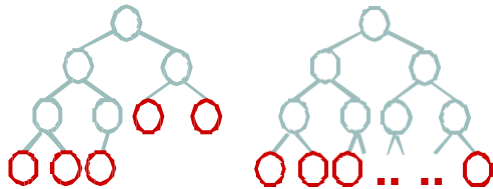
How to trickle down

- The left and right sub-trees of the root are max- heaps. But the root may be **smaller** than its children.
- Step 1: We determine the largest between: root, left child and right child of root.
- If root is the largest, then its position is correct. Otherwise, swap root and the largest. Repeat Step 1 again.

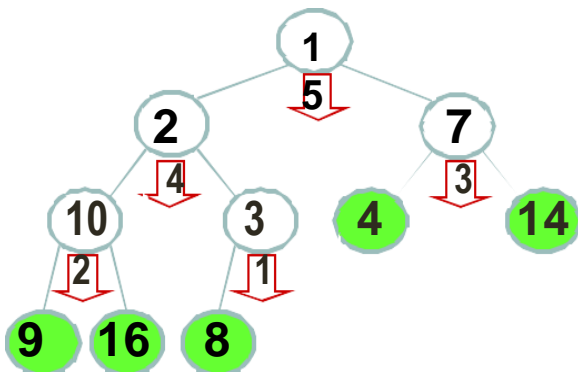
Heap sort[5]

Step 1. Build a max heap from the array

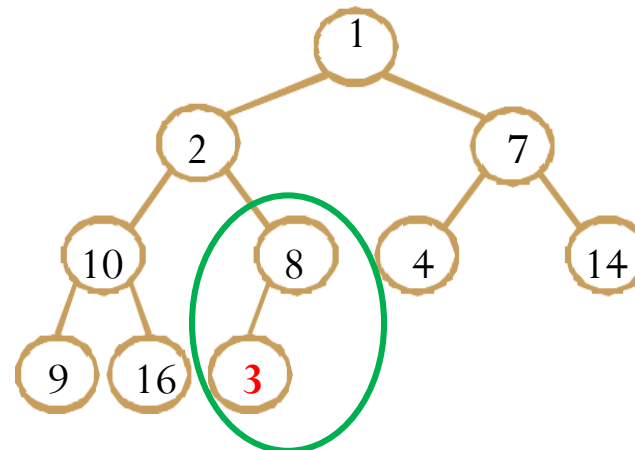
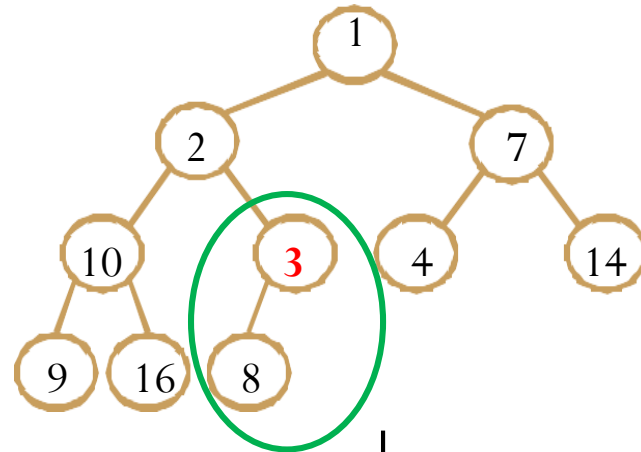
- In any complete binary tree, the last $\lceil N/2 \rceil$ elements are leaf nodes. The first $\lfloor N/2 \rfloor$ are not.



- Trickle down the first $\lfloor N/2 \rfloor$ elements. Starting from the lowest ones to the top.



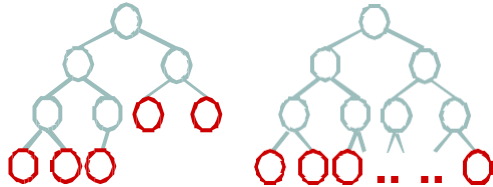
1. Trickle down 3



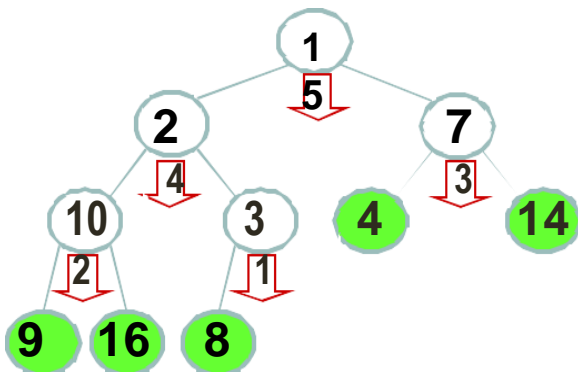
Heap sort[6]

Step 1. Build a max heap from the array

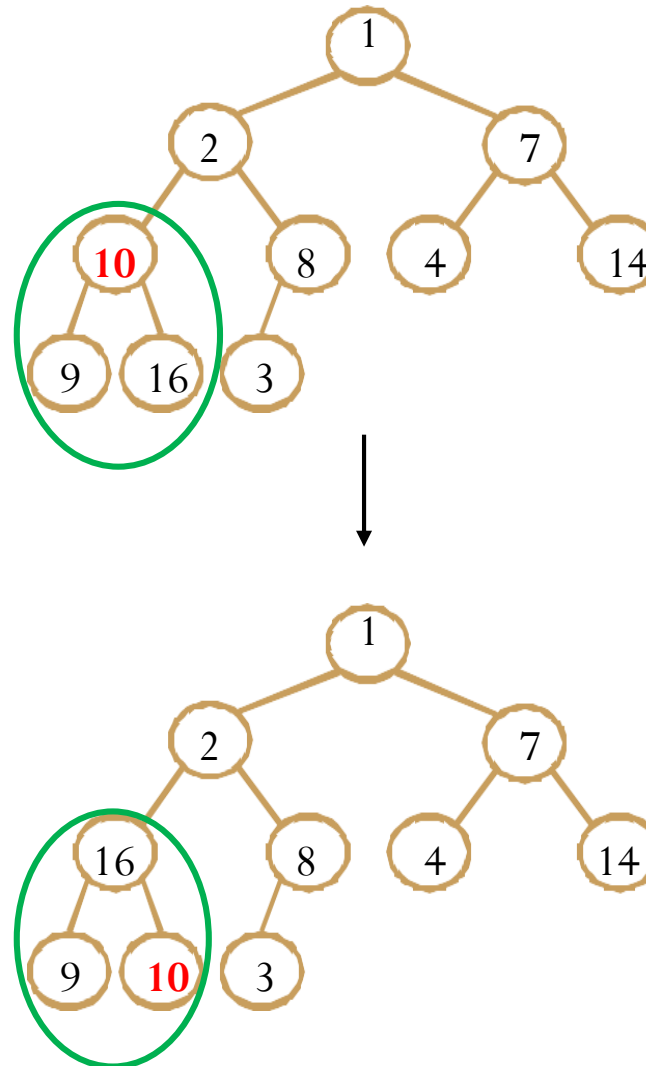
- In any complete binary tree, the last $\lceil N/2 \rceil$ elements are leaf nodes. The first $\lfloor N/2 \rfloor$ are not.



- Trickle down the first $\lfloor N/2 \rfloor$ elements. Starting from the lowest ones to the top.



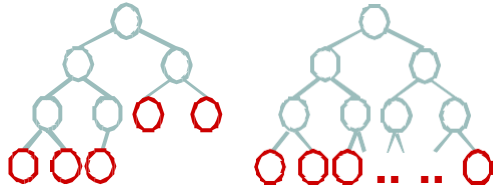
2. Trickle down 10



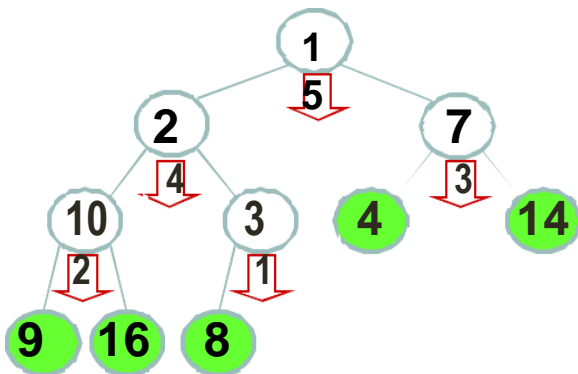
Heap sort[7]

Step 1. Build a max heap from the array

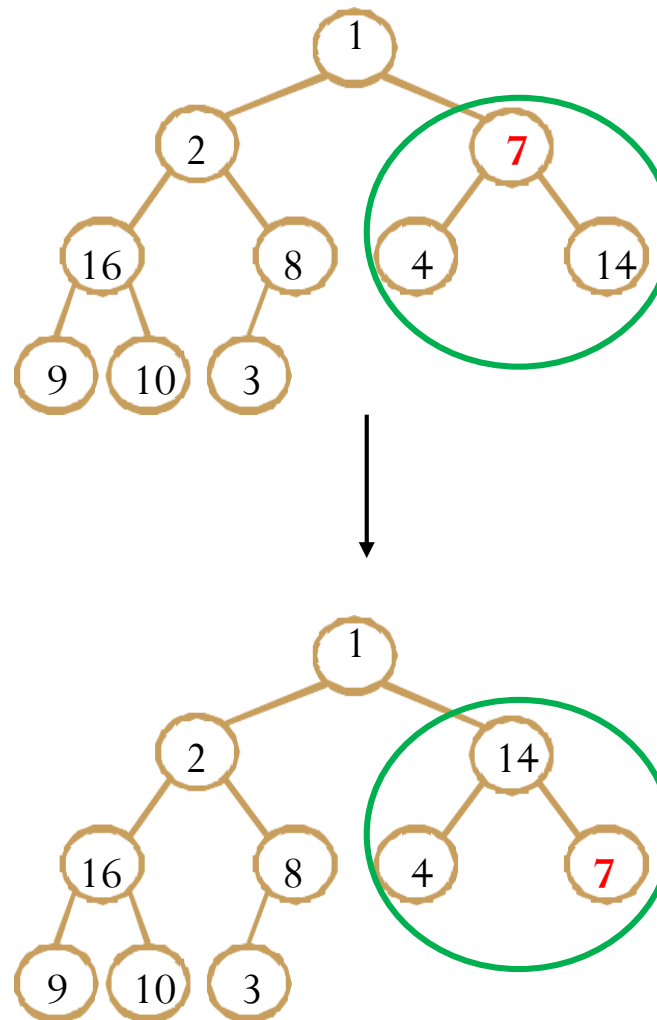
- In any complete binary tree, the last $\lceil N/2 \rceil$ elements are leaf nodes. The first $\lfloor N/2 \rfloor$ are not.



- Trickle down the first $\lfloor N/2 \rfloor$ elements. Starting from the lowest ones to the top.



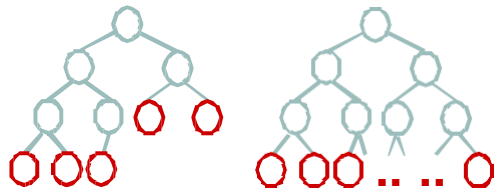
3. Trickle down 7



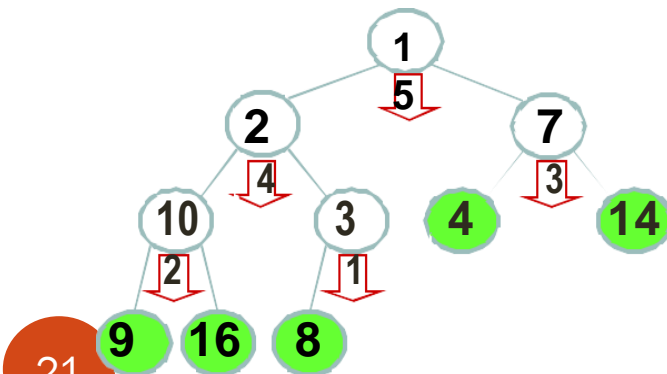
Heap sort[8]

Step 1. Build a max heap from the array

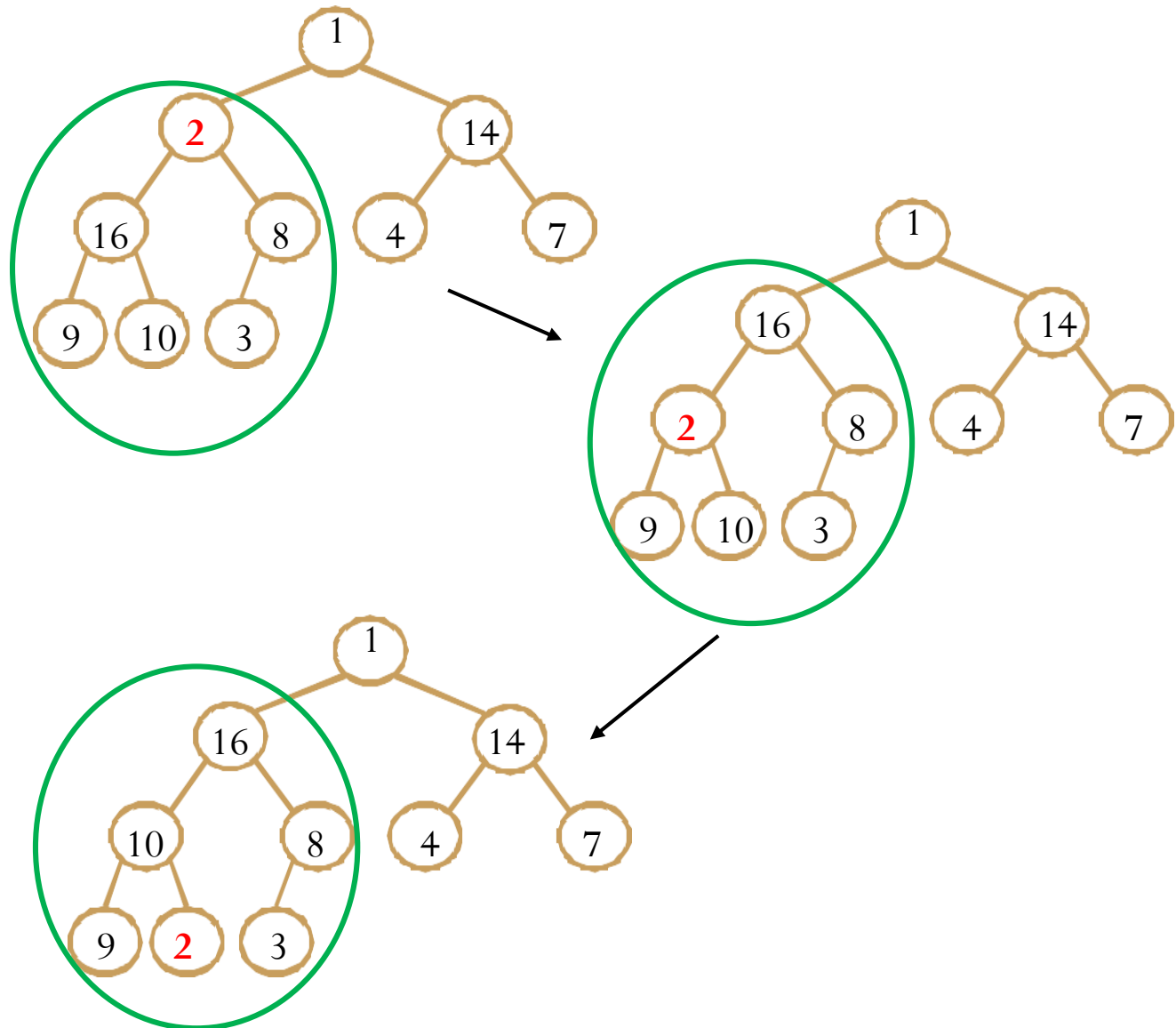
- In any complete binary tree, the last $\lceil N/2 \rceil$ elements are leaf nodes. The first $\lfloor N/2 \rfloor$ are not.



- Trickle down the first $\lfloor N/2 \rfloor$ elements. Starting from the lowest ones to the top.



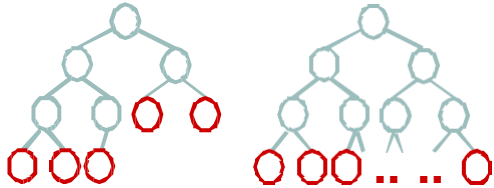
4. Trickle down 2



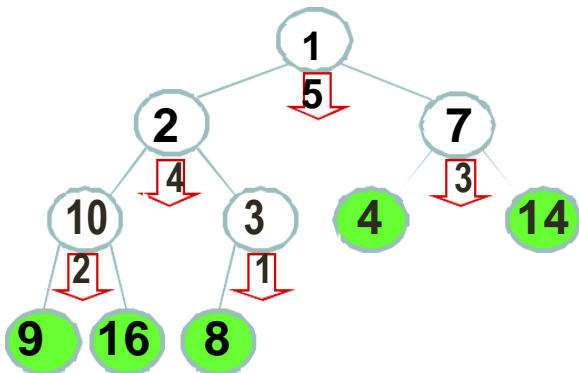
Heap sort[9]

Step 1. Build a max heap from the array

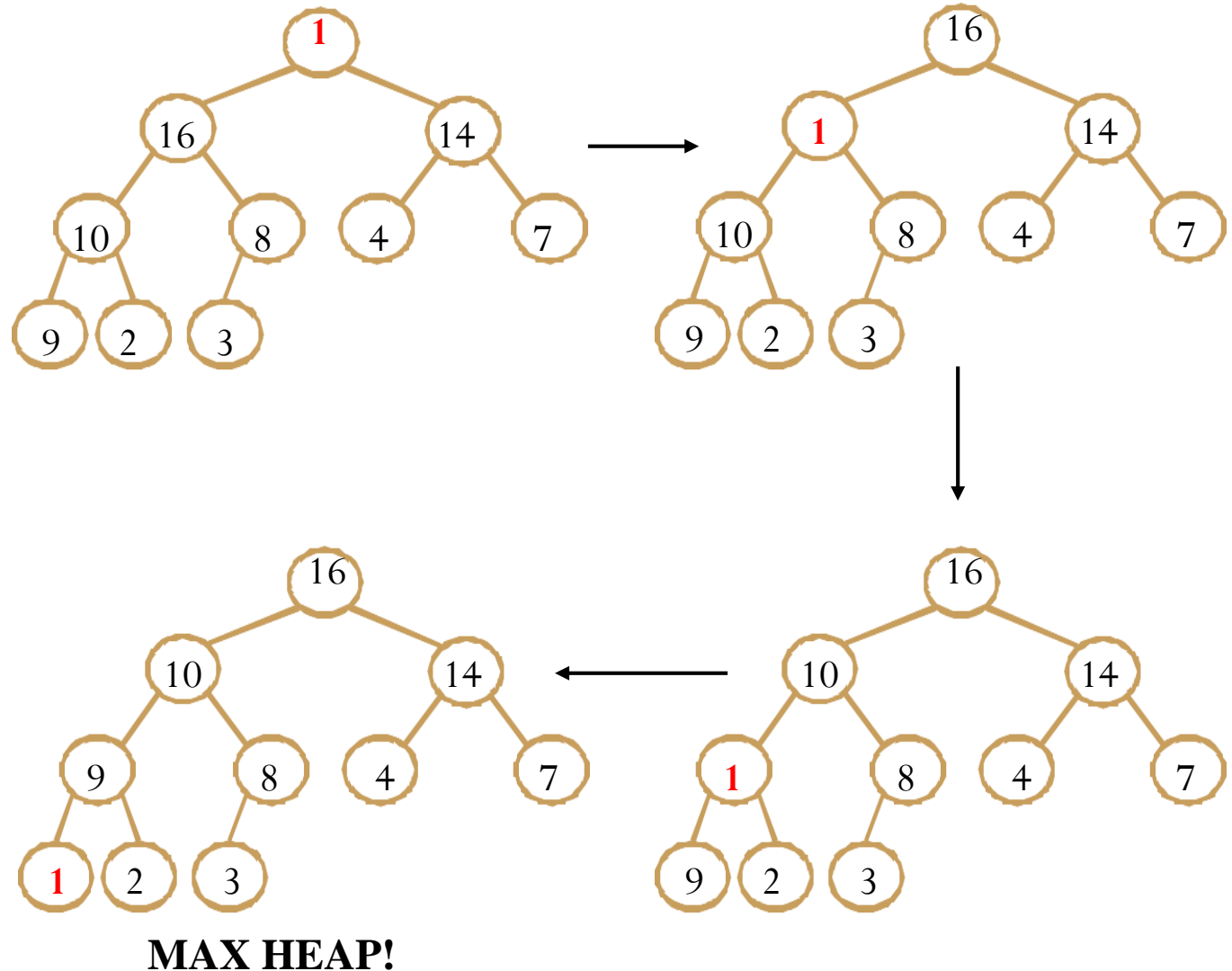
- In any complete binary tree, the last $\lceil N/2 \rceil$ elements are leaf nodes. The first $\lfloor N/2 \rfloor$ are not.



- Trickle down the first $\lfloor N/2 \rfloor$ elements. Starting from the lowest ones to the top.



5. Trickle down 1



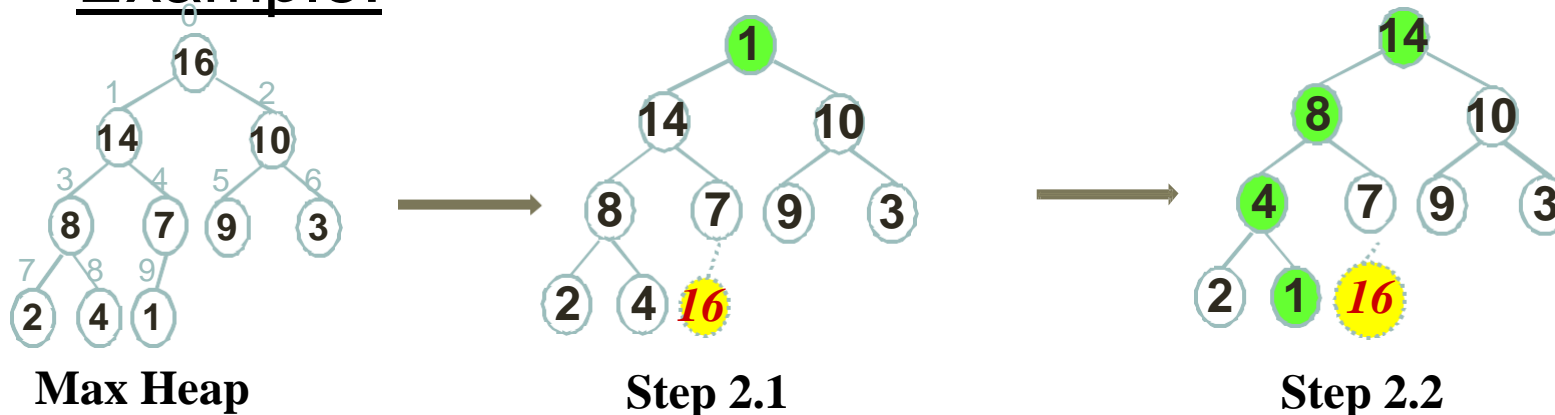
Heap sort[10]

Step 2. Sorting

Step 2.1. Swap the root and the last leaf (the first and the last element of the array). Reduce the size of the heap by one

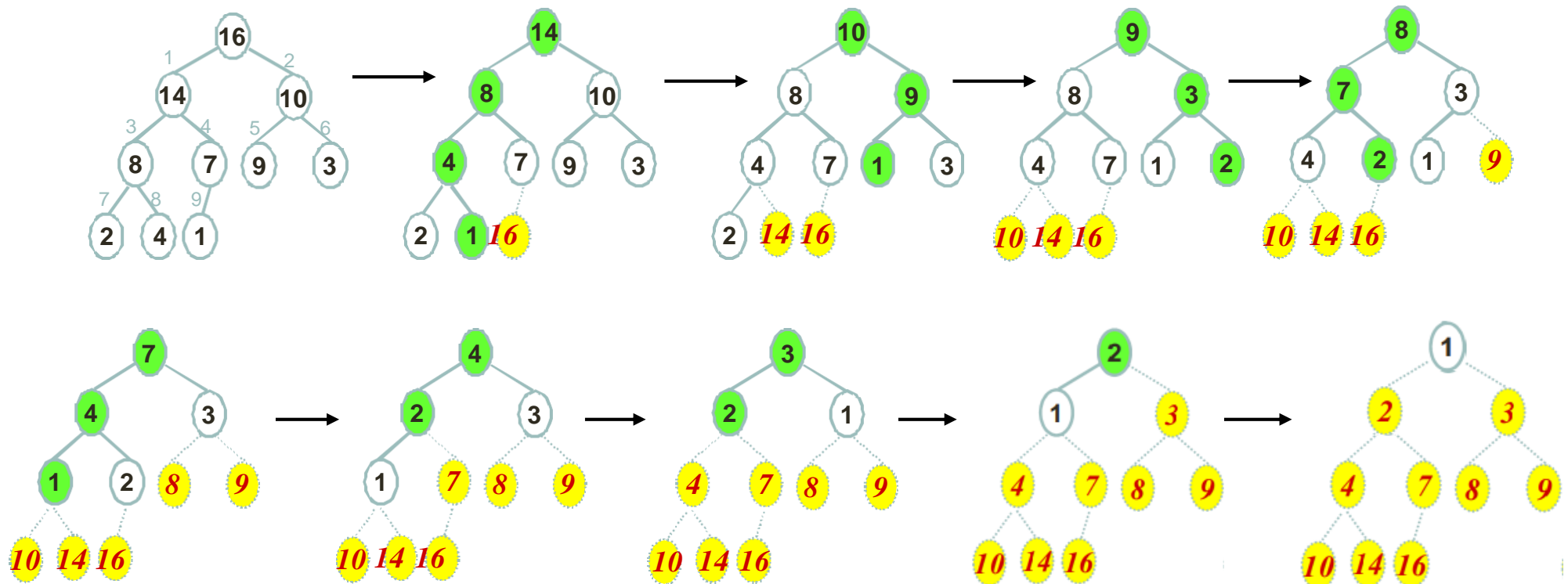
Step 2.2. Trickle down the new root (node 1) to a correct position (in order to build max heap again).

Example:



Heap sort [11]

- Repeat step 2.1 & 2.2 until the heap has only one node



Heap sort[12]

Heap sort algorithm

Code

Algorithm Heapsort (A) :

Input: Array unsorted A

Output: A is sorted

$n \leftarrow a.length$

BuildHeap (a, n)

For $i \leftarrow n-1$ **downto** 1 **do**

$tmp \leftarrow a[0] \quad a[0] \leftarrow a[i] \quad a[i] \leftarrow tmp$

$n \leftarrow n-1$

TrickleDown (A, n, 0)

Endfor

Heap sort[13]

Build heap algorithm

Code

Algorithm BuildHeap(A, n)

Input: Array A of the size n , starting from 1

Output: A becomes max heap.

For $i \leftarrow n/2 - 1$ **down to** 0

do

TrickleDown(A, n, i)

Endfor

Heap sort[14]

Trickle down algorithm

Code

Algorithm TrickleDown(A, n, i):

Input: The binary tree A of the size n starting from 1, i is the root of the tree

Output: Tree A becomes max heap.

$l \leftarrow \text{LeftChild}(i)$ $r \leftarrow \text{RightChild}(i)$ $\text{maxpos} \leftarrow i$

If ($l \leq n$) and ($a[l] > a[\text{maxpos}]$) **then**
 $\text{maxpos} \leftarrow l$

If ($r \leq n$) and ($a[r] > a[\text{maxpos}]$) **then**
 $\text{maxpos} \leftarrow r$

If ($\text{maxpos} \neq i$) **then**
 $\text{tmp} \leftarrow a[i]$ $a[i] \leftarrow a[\text{maxpos}]$ $a[\text{maxpos}] \leftarrow \text{tmp}$
 TrickleDown(A, n, maxpos)
Endif

Heap sort[15]

Time complexity analysis of heap sort

- Reference book **chapter 6** Heap sort
- Running time = Time to build max heap + $(n-1) \times$
Time to trickle down
- Time to build max heap is **$O(n)$**
- Time to trickle down is **$O(\log n)$** .
- The time complexity of heap sort is:
 $O(T(n)) = O(n) + (n-1)O(\log n) = O(n \log n)$

Lineartimesortingalgorithm[1]

- All sorting algorithm introduced so far shared the same property:
 - The sorted order is determined **based only on the comparisons between input elements**.
 - They are called **comparison sort**.
 - The lower bound time complexity for any comparison sort is **$\Omega(n \log n)$** for the worst case (see **chapter 8.1** reference book). It means the fastest comparison sort algorithm has the time complexity **$O(n \log n)$** .

Linear time sorting algorithm [2]

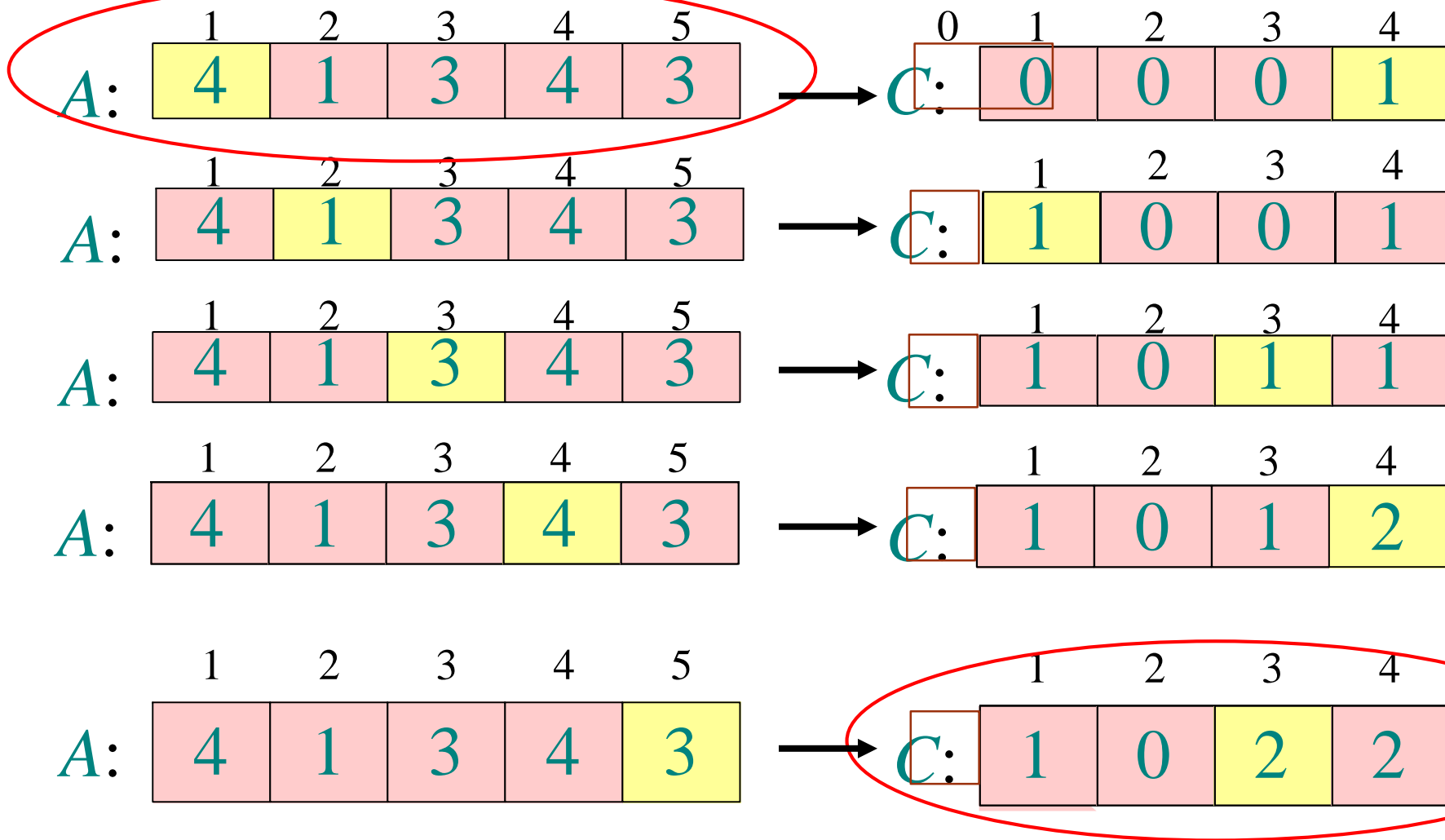
- There are other sorting algorithms that **used operations other than comparisons** to determine the sorted order
 - Counting sort
 - Radix sort
- They can sort n elements in the time **$O(n)$**
 - They are called **linear time sorting algorithm**

Counting sort[1]

- We want to sort n integers, each integer is in the range $[0..k]$.
 - We determine, for each input element x , the number of elements less than x .
 - This information can be used to place directly into its correct position. For example, if there 17 elements less than x , then x belongs in output position 18.
- For example, we want to sort array $A[1..5]$, where $A[j] \in \{1, 2, 3, 4\}$. We use array $B[1..5]$ to store the sorted elements, array $C[1..4]$ to store counting information

Counting sort[2]

- How many times does $A[i]$ appear in A



Counting sort[3]

- How many times does $A[i]$ appear in A

Code

```
For  $i \leftarrow 1$  to  $n$  do
```

```
     $C[A[i]] \leftarrow C[A[i]] + 1$ 
```

```
Endfor
```

Counting sort[4]

- $C[x]$ is the number of elements equal to x
- Now we count the number of elements $\leq x$ ($C[x]$ is the number of elements $\leq x$)

A:

1	2	3	4	5
4	1	3	4	3

C:

1	2	3	4
1	0	2	2

C:

1	2	3	4
1	1	2	2

C:

1	2	3	4
1	1	3	2

C:

1	2	3	4
1	1	3	5

Counting sort[5]

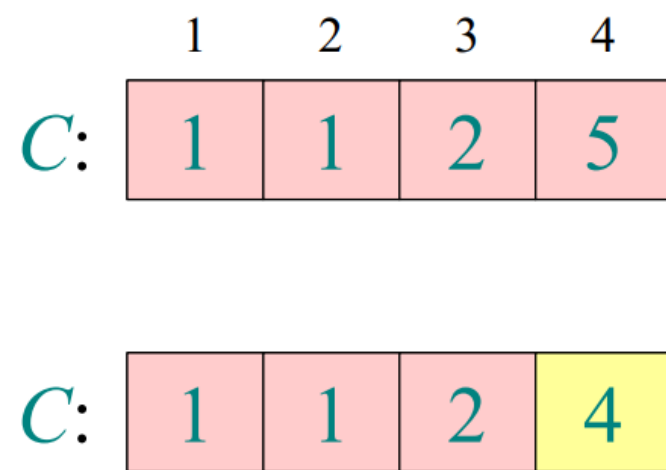
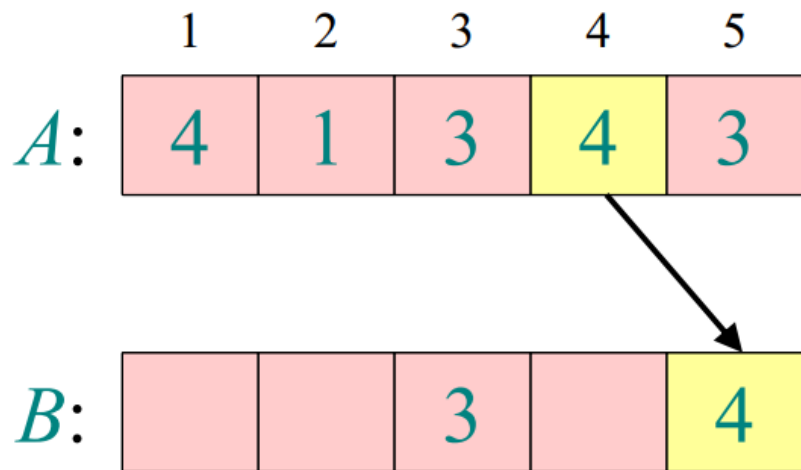
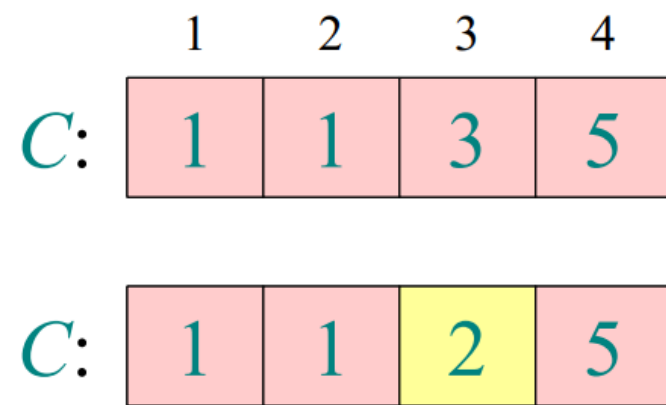
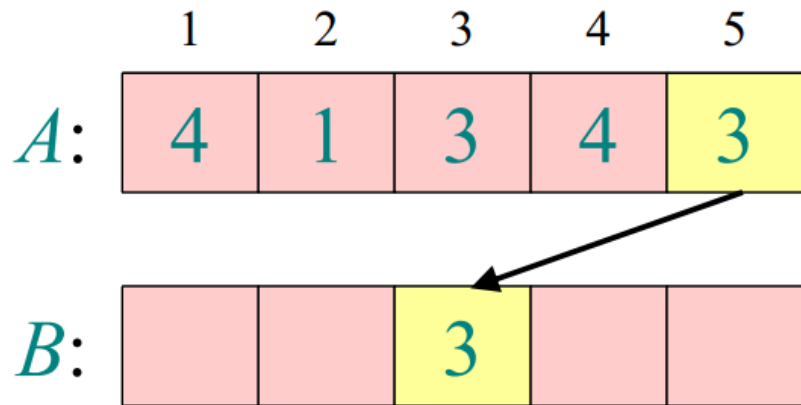
- Count the elements $\leq A[i]$ in A

Code

```
For  $i \leftarrow 2$  to  $k$  do  
     $C[i] \leftarrow C[i] + C[i-1]$   
Endfor
```

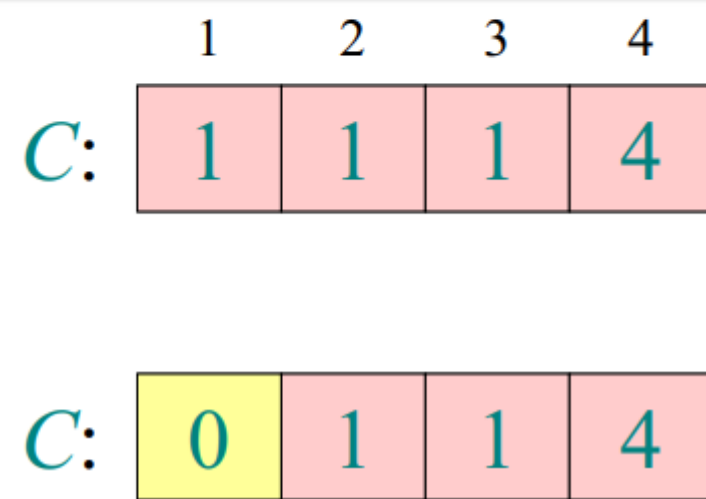
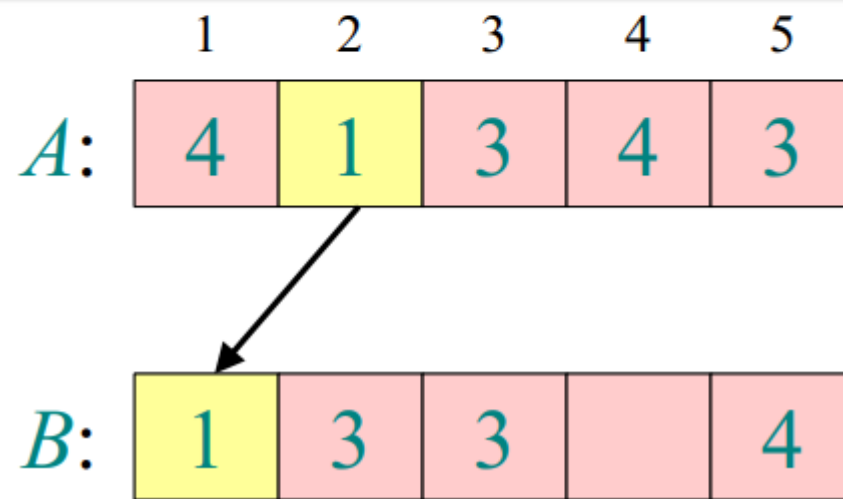
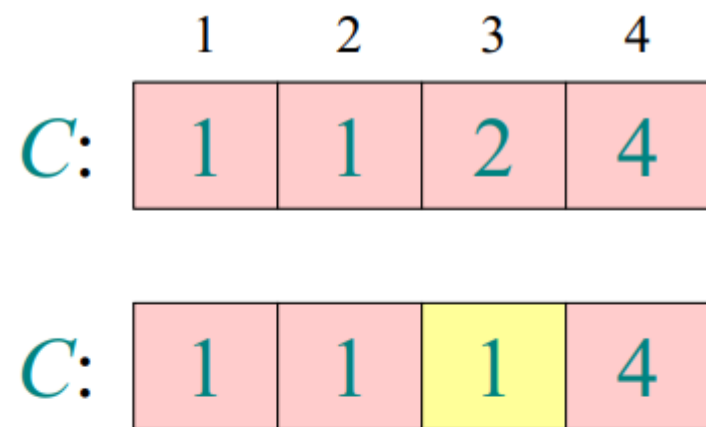
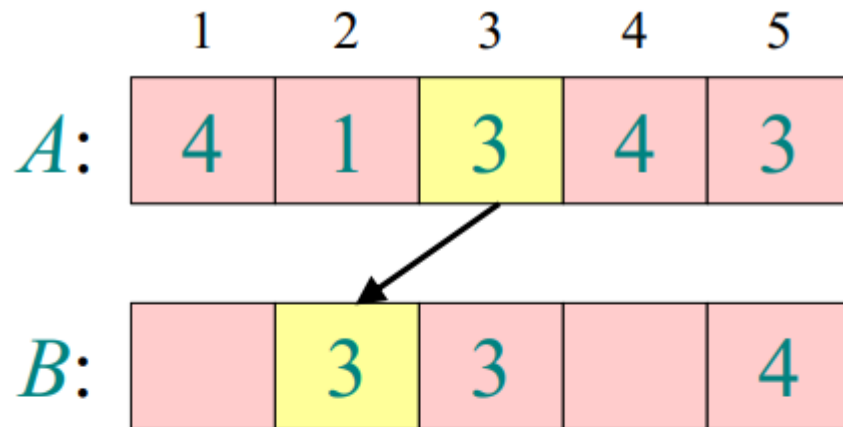
Counting sort[6]

- How to arrange $A[i]$ into B



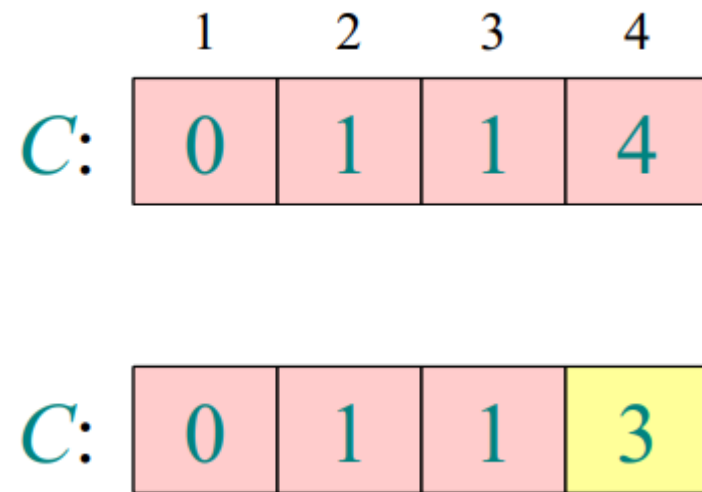
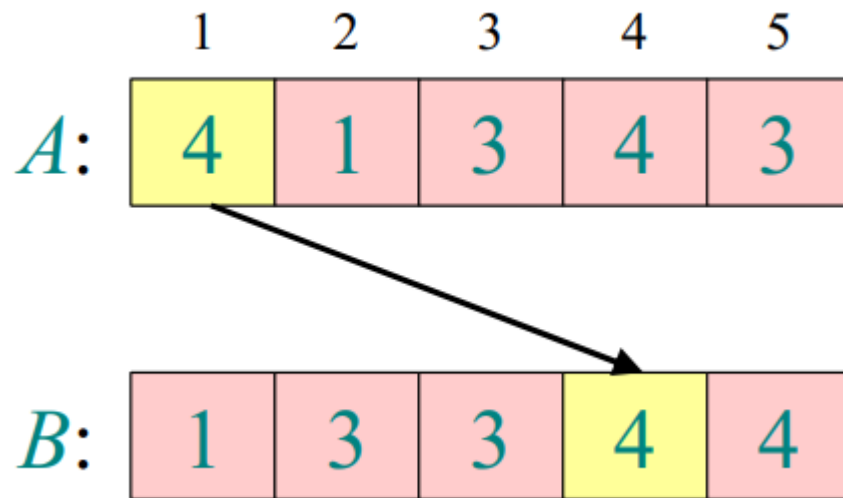
Counting sort[7]

- How to arrange $A[i]$ into B



Counting sort[8]

- How to arrange $A[i]$ into B



- Now B is storing sorted elements

Counting sort[9]

- How to arrange $A[i]$ into B

Code

```
For  $i \leftarrow n$  downto 1 do
```

```
     $B[C[A[i]]] \leftarrow A[i]$ 
```

```
     $C[A[i]] \leftarrow C[A[i]] - 1$ 
```

```
Endfor
```

Counting sort [10]

Code

Algorithm CountingSort(A,n,k):

Input: Array unsorted A of the size n, a[i]
in the range [1..k]

Output: A is sorted and stored in B

For $i \leftarrow 1$ **to** k **do**

$c[i] \leftarrow 0$

For $i \leftarrow 1$ **to** n **do**

$c[a[i]] \leftarrow c[a[i]] + 1$

For $i \leftarrow 2$ **to** k **do**

$c[i] \leftarrow c[i] + c[i-1]$

For $i \leftarrow n$ **downto** 1 **do**

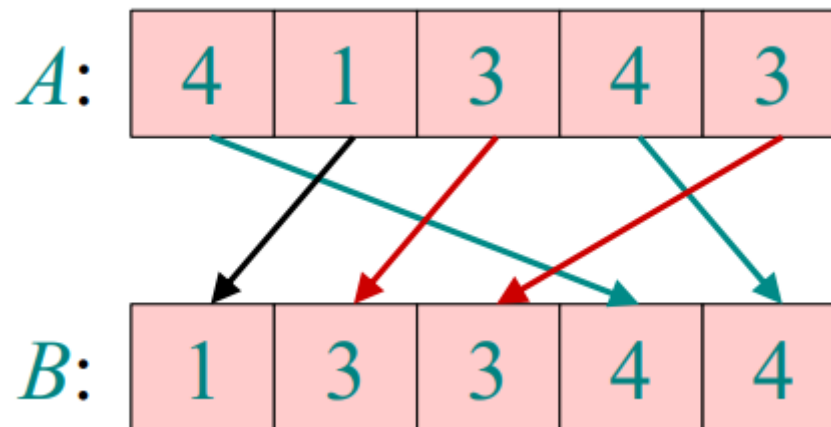
$b[c[a[i]]] \leftarrow a[i]$

$c[a[i]] \leftarrow c[a[i]] - 1$

Endfor

Counting sort[11]

- Time complexity of counting sort algorithm is $O(n+k)$
 - Doesn't compare the elements of array A
- Counting sort is a **stable** sort: it preserves the input order among equal elements.

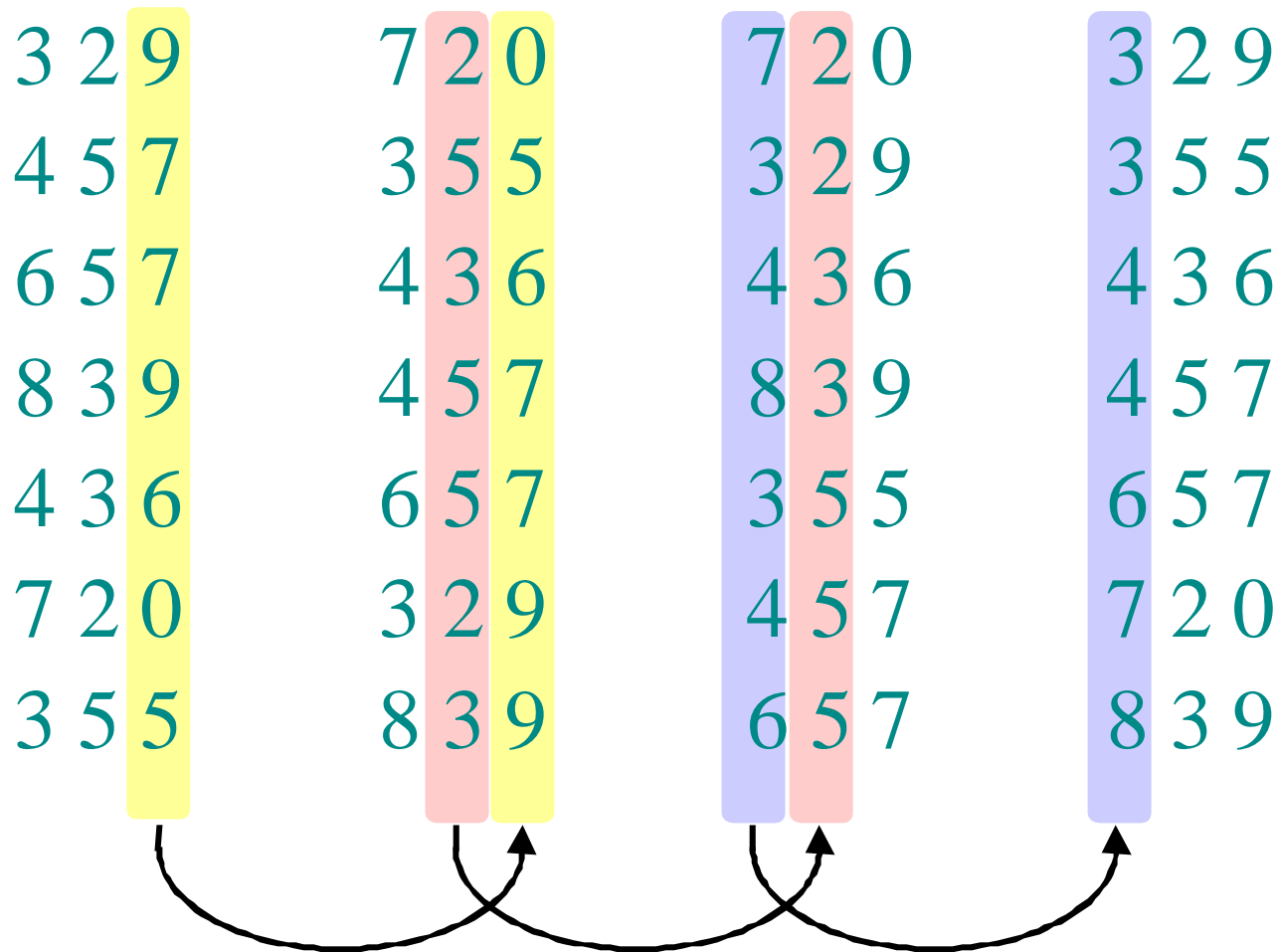


Radixsort [1]

- Counting sort is inefficient when **k is big**:
 - $k=1000$, 10000 , or 100000 ...
- We want to sort an array of n elements, each element is an integer which has **maximum k digits**
 - Sort the elements **by its digits**, one by one **from right to left**.
 - **Use a stable sorting algorithm** to sort elements by digits.
- The algorithm is called **radix sort** or column sort

Radixsort [2]

- Radix sort example: $n=7$; $k=3$;



Radixsort [3]

- Radix sort algorithm

Code

Algorithm RadixSort(A, n, k):

Input: Array unsorted A of the size n ,
 $a[i]$ has maximum k digits. Digit 1 is the
right most and digit k is the left most.

Output: A is sorted

For $i \leftarrow 1$ **to** k **do**

Column_sort(A, n, i)

//use a stable sort to sort A on digit i

Endfor

Radixsort [4]

- **Time complexity of radix sort algorithm**
- **N** elements, each element has **k** digits, each digit is in the range **[0..9]**. Using counting sort as the stable sorting algorithm:
 $O(n) = k \cdot O(n+10) = O(k(n+10))$
- **N** elements, each element has **p** digits in base **b**, each digit is in the range **[0..b-1]**. Using counting sort as the stable sorting algorithm:
 $O(n) = p \cdot O(n+b) = O(p(n+b))$

Radixsort [5]

- **Does radix sort algorithm work on negative number or non-integer number?**
 - The answer is yes, but we need some modification and a proper representation of the elements

Best sorting algorithm?

- **Linear time sorting algorithm**
 - Fastest, but need special input data
- **The fastest comparison sorting algorithms has $O(N \log N)$**
 - For large N (say >1000), Quicksort is faster, on most machines, by a factor of 1.5 or 2. However, it requires a bit of extra memory and is a moderately complicated program.
 - Heap sort is a true “sort in place”, and is more compact to program.
 - For $N < 50$, an $O(N^2)$ sorting algorithm, is concise and fast enough.
 - For $50 < N < 1000$, Shell sort is usually the method of choice (see section 7.4 text book). It is $O(N^{3/2})$ as in the worst case, but is usually faster.

Tutorial & next topic

- **Preparing for the tutorial:**
 - Practice with examples and exercises in Tutorial 5
- **Preparing for next topic:**
 - Read textbook chapter 3 List.