

# **Data Structures and Algorithms**

**(CS210A)**

## **Lecture 39**

- **Integer sorting continued**
- **Search data structure for integers : Hashing**

# Types of sorting algorithms

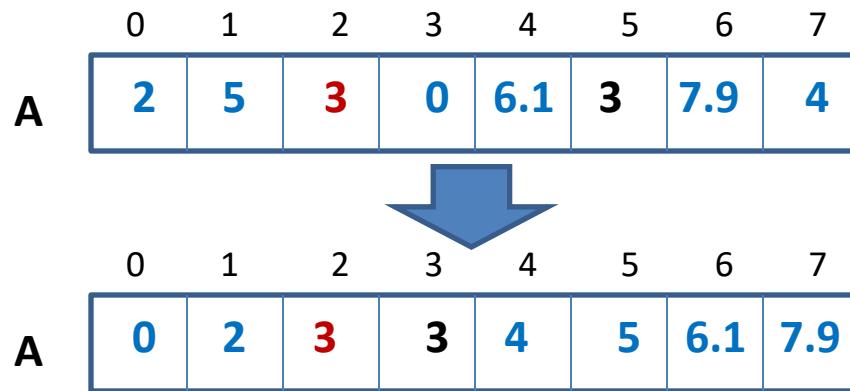
**In Place** Sorting algorithm:

A sorting algorithm which uses

**Example:** Heap sort, Quick sort.

**Stable** Sorting algorithm:

A sorting algorithm which preserves :



**Example:** Merge sort.

# Integer Sorting algorithms

Continued from last class

# Counting sort: algorithm for sorting integers

**Input:** An array  $\mathbf{A}$  storing  $n$  integers in the range  $[0 \dots k - 1]$ .

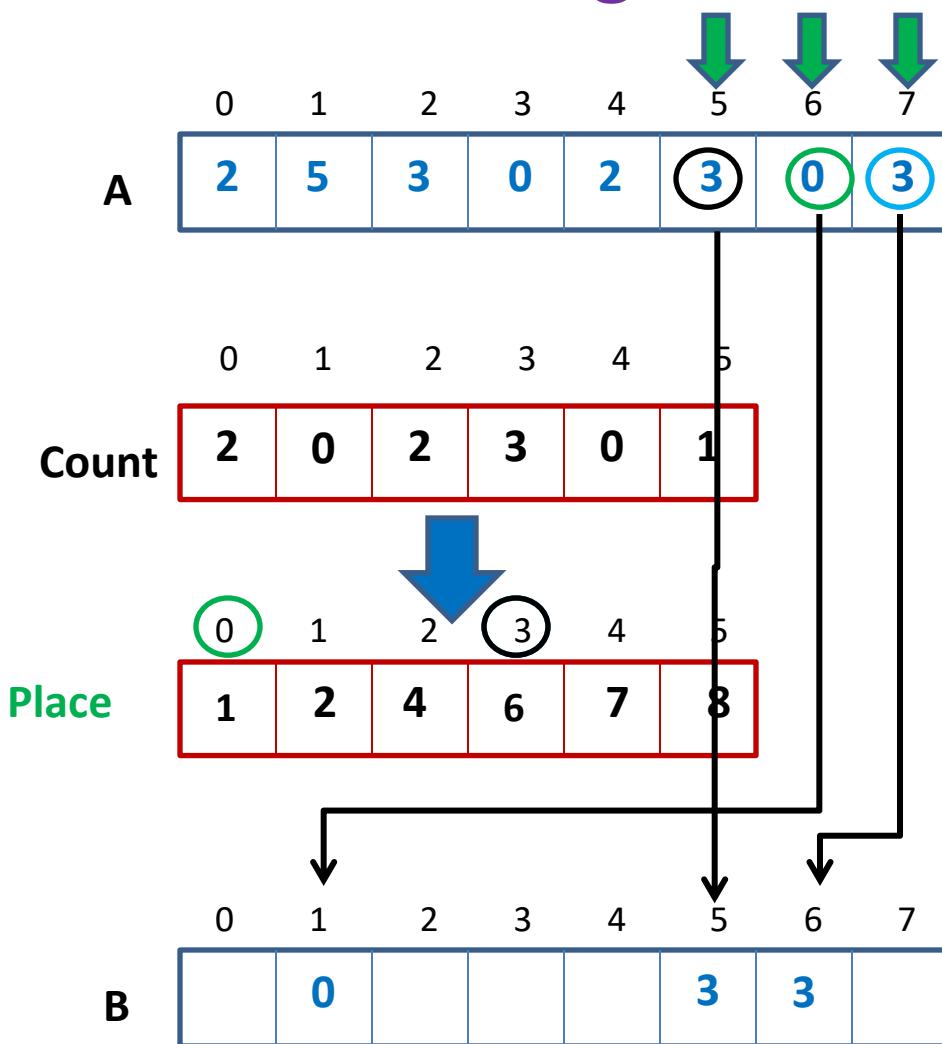
$$k = O(n)$$

**Output:** Sorted array  $\mathbf{A}$ .

**Running time:**  $O(n + k)$  in word RAM model of computation.

**Extra space:**  $O(n + k)$

# Counting sort: a visual description



Why did we scan elements of **A** in reverse order (from index  $n - 1$  to **0**) while placing them in the final sorted array **B**?

Answer:

- To ensure that Counting sort is **stable**.
- The reason why stability is required will become clear soon 😊

# Counting sort: algorithm for sorting integers

Algorithm ( $A[0 \dots n - 1], k$ )

For  $j=0$  to  $k - 1$  do  $\text{Count}[j] \leftarrow 0$ ;

For  $i=0$  to  $n - 1$  do  $\text{Count}[A[i]] \leftarrow \text{Count}[A[i]] + 1$ ;

$\text{Place}[0] \leftarrow \text{Count}[0]$ ;

For  $j=1$  to  $k - 1$  do  $\text{Place}[j] \leftarrow \text{Place}[j - 1] + \text{Count}[j]$  ;

For  $i=n - 1$  to  $0$  do

{      $B[\text{Place}[A[i]] - 1] \leftarrow A[i]$ ;

$\text{Place}[A[i]] \leftarrow \text{Place}[A[i]] - 1$ ;

}

return  $B$ ;

Each arithmetic operations

involves  $O(\log n + \log k)$  bits

# Counting sort: algorithm for sorting integers

## Key points of Counting sort:

- It performs arithmetic operations involving  $O(\log n + \log k)$  bits  
( $O(1)$  time in word RAM).
- It is a **stable** sorting algorithm.

**Theorem:** An array storing  $n$  integers in the range  $[0..k - 1]$   
can be sorted in  $O(n+k)$  time and  
using total  $O(n+k)$  space in word RAM model.

- For  $k \leq n$ ,  
→ For  $k = n^t$ ,  
(too bad for  $t > 1$ . ☹)

## Question:

How to sort  $n$  integers in the range  $[0..n^t]$  in

# Radix Sort

# Digits of an integer

507266

No. of **digits** = 6

value of **digit**  $\in \{0, \dots, 9\}$

1011000101011111

No. of **digits** = 4

value of **digit**  $\in \{0, \dots, 15\}$

It is up to us how we define digit ?

# Radix Sort

**Input:** An array  $\mathbf{A}$  storing  $n$  integers, where

- (i) each integer has exactly  $d$  digits.
- (ii) each digit has value  $< k$
- (iii)  $k < n$ .

**Output:** Sorted array  $\mathbf{A}$ .

**Running time:**

$O(dn)$  in word RAM model of computation.

**Extra space:**

$O(n + k)$

**Important points:**

- makes use of a count sort.
- Heavily relies on the fact that count sort is a stable sort algorithm.

# Demonstration of Radix Sort through example

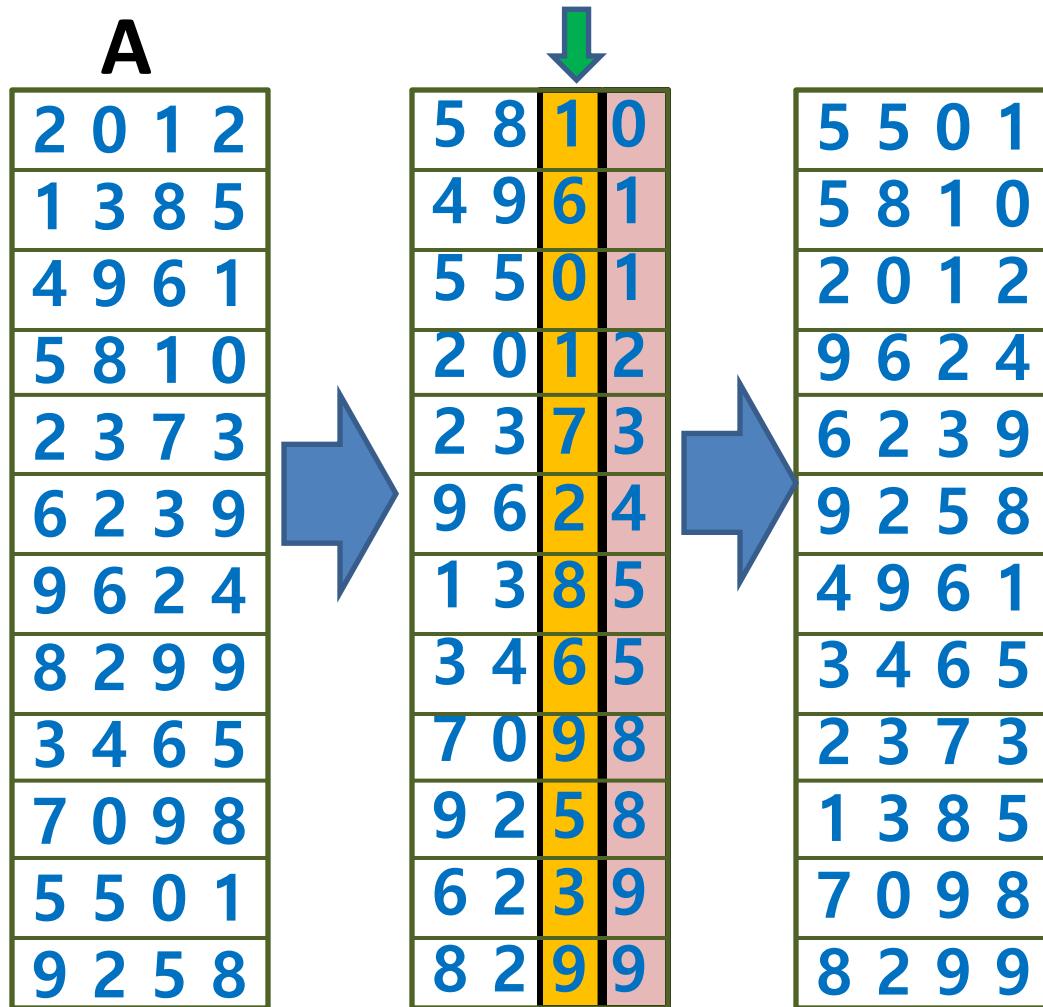
A	↓
2	0
1	3
4	9
5	8
2	3
6	2
9	6
8	2
3	4
7	0
5	5
9	2
2	5
1	8

→

5	8	1	0
4	9	6	1
5	5	0	1
2	0	1	2
2	3	7	3
9	6	2	4
1	3	8	5
3	4	6	5
7	0	9	8
9	2	5	8
6	2	3	9
8	2	9	9

$d = 4$   
 $n = 12$   
 $k = 10$

# Demonstration of Radix Sort through example



# Demonstration of Radix Sort through example

A

2	0	1	2
1	3	8	5
4	9	6	1
5	8	1	0
2	3	7	3
6	2	3	9
9	6	2	4
8	2	9	9
3	4	6	5
7	0	9	8
5	5	0	1
9	2	5	8

5	8	1	0
4	9	6	1
5	5	0	1
2	0	1	2
2	3	7	3
9	6	2	4
6	2	3	9
9	2	5	8
4	9	6	1
3	4	6	5
2	3	7	3
1	3	8	5
8	2	9	9

↓

5	5	0	1
5	8	1	0
2	0	1	2
9	6	2	4
6	2	3	9
9	2	5	8
4	9	6	1
3	4	6	5
2	3	7	3
1	3	8	5
7	0	9	8
8	2	9	9

2	0	1	2
7	0	9	8
6	2	3	9
9	2	5	8
8	2	9	9
2	3	7	3
1	3	8	5
3	4	6	5
5	5	0	1
9	6	2	4
5	8	1	0
4	9	6	1

# Demonstration of Radix Sort through example

A

2 0 1 2	5 8 1 0
1 3 8 5	4 9 6 1
4 9 6 1	5 5 0 1
5 8 1 0	2 0 1 2
2 3 7 3	9 6 2 4
6 2 3 9	6 2 3 9
9 6 2 4	9 2 5 8
8 2 9 9	1 3 8 5
3 4 6 5	4 9 6 1
7 0 9 8	3 4 6 5
9 2 5 8	2 3 7 3
5 5 0 1	1 3 8 5
8 2 9 9	7 0 9 8

5 5 0 1	2 0 1 2
5 8 1 0	9 6 2 4
2 0 1 2	6 2 3 9
9 6 2 4	9 2 5 8
6 2 3 9	4 9 6 1
9 2 5 8	3 4 6 5
4 9 6 1	2 3 7 3
3 4 6 5	1 3 8 5
2 3 7 3	7 0 9 8
1 3 8 5	9 2 5 8
7 0 9 8	8 2 9 9
9 2 5 8	8 2 9 9

↓

2 0 1 2	7 0 9 8	1 3 8 5
6 2 3 9	9 2 5 8	2 0 1 2
9 2 5 8	8 2 9 9	2 3 7 3
2 3 7 3	2 3 7 3	3 4 6 5
1 3 8 5	1 3 8 5	4 9 6 1
7 0 9 8	3 4 6 5	5 5 0 1
9 2 5 8	5 5 0 1	5 8 1 0
8 2 9 9	9 6 2 4	6 2 3 9
8 2 9 9	8 2 9 9	7 0 9 8
9 2 5 8	9 6 2 4	8 2 9 9
9 6 2 4	5 8 1 0	9 2 5 8
4 9 6 1	4 9 6 1	9 6 2 4

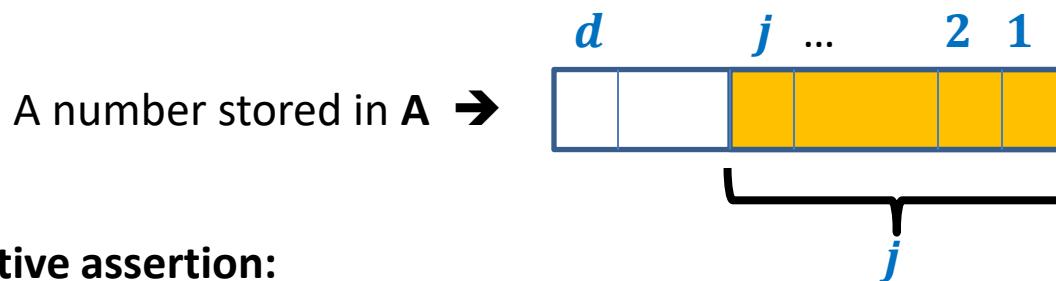
Can you see where we are exploiting the fact that  
Countsort is a **stable** sorting algorithm ?

# Radix Sort

**RadixSort(A[0... $n - 1$ ],  $d$ ,  $k$ )**

```
{  For  $j=1$  to  $d$  do
    Execute CountSort(A,  $k$ ) with  $j$ th digit as the key;
    return A;
}
```

**Correctness:**



**Inductive assertion:**

At the end of  $j$ th iteration, array **A** is sorted according to the last  $j$  digits.

During the induction step, you will have to use the fact that **Countsort** is a **stable** sorting algorithm.

# Radix Sort

**RadixSort(A[0... $n - 1$ ],  $d$ ,  $k$ )**

```
{  For  $j=1$  to  $d$  do
    Execute CountSort(A, $k$ ) with  $j$ th digit as the key;
    return A;
}
```

**Time complexity:**

- A single execution of **CountSort(A, $k$ )** runs in  $O(n + k)$  time and  $O(n + k)$  space.
- For  $k < n$ ,
  - a single execution of **CountSort(A, $k$ )** runs in  $O(n)$  time.
  - Time complexity of radix sort =  $O(dn)$ .
- → Extra space used =  $O(n)$

**Question:** How to use Radix sort to sort  $n$  integers in range  $[0..n^t]$  in  $O(tn)$  time and  $O(n)$  space ?

**Answer:**

The diagram illustrates the bit representation of numbers. On the left, a blue box labeled "1 bit" has a green arrow pointing to a blue box labeled "log  $n$  bits". On the right, a blue box labeled " $t \log n$  bits" has a green arrow pointing to a blue box labeled "t".

$d$	$k$	Time complexity
$t \log n$	2	$O(tn \log n)$ 😕
$t$	$n$	$O(tn)$ 😊



# Power of the word RAM model

- **Very fast** algorithms for **sorting integers**:  
**Example:**  $n$  integers in range  $[0..n^{10}]$  in  $O(n)$  time and  $O(n)$  space ?
- **Lesson:**  
**Do not** always go after **Merge sort** and **Quick sort** when input is integers.
- **Interesting programming exercise** (for summer vacation):  
**Compare** **Quick sort** with **Radix sort** for sorting **long** integers.