Data Structures and Algorithms

(ESO207)

Lecture 39

- Integer sorting
- Counting Sort and Radix Sort

Integer sorting

Algorithms for Sorting n elements

- **Insertion** sort: O(n^2)
- **Selection** sort: O(n^2)
- **Bubble** sort: O(n^2)
- **Merge** sort: O(n **log** n)
- **Quick** sort: worst case **O**(n^2), average case **O**(n **log** n)
- **Heap** sort: O(n **log** n)

Question: What is common among these algorithms ?

Answer: All of them use only **comparison** operation to perform sorting.

Theorem (we will not prove it in this course):

Every comparison based sorting algorithm

must perform at least n **log** n comparisons in the worst case.

Question: Can we sort in **O**(n) time ?

The answer depends upon

- the ***model of computation***.
- the ***domain*** of input.

Integer sorting

Counting sort: algorithm for sorting integers

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

$k = O(n)$

Output: Sorted array **A**.

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

Extra space: $O(k)$

Motivating example: Indian railways

There are **13 lacs** employees.

Aim : To **sort** them list according to **DOB** (date of birth)

Observation: There are only **14600** different date of births possible.

Counting sort: algorithm for sorting integers

A

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

Count

	0	1	2	3	4	5
Count	2	0	2	3	0	1

Place

	0	1	2	3	4	5
Place	2	2	4	7	7	8

B

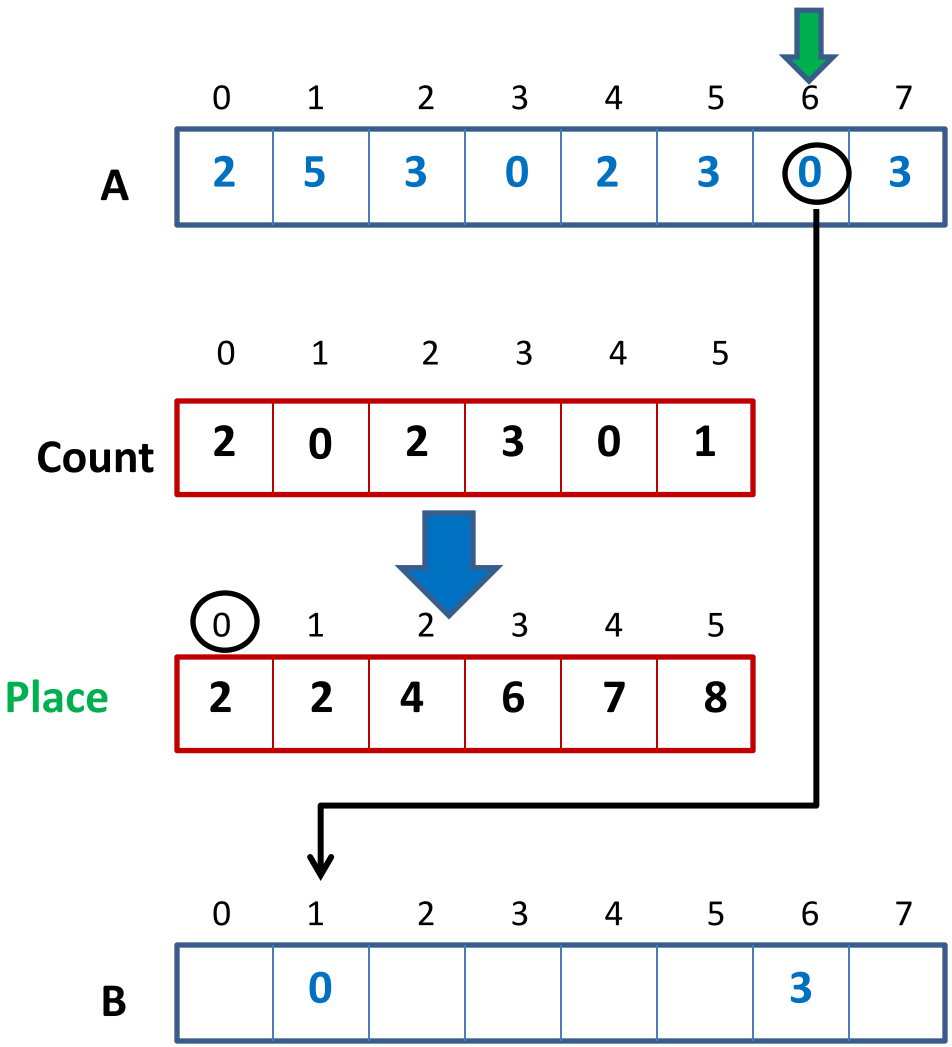
	0	1	2	3	4	5	6	7
B							3	

Final sorted output

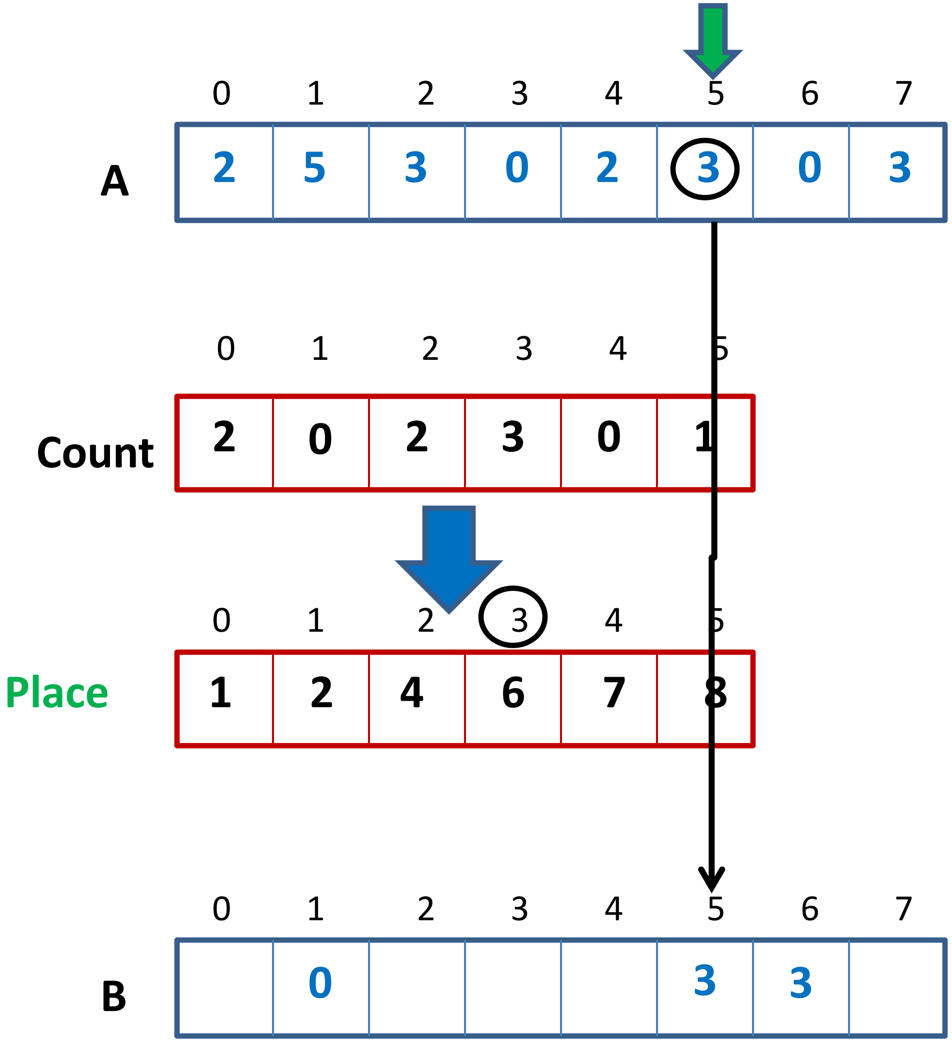
If **A**[i]=j, where should A[i] be placed in **B** ?

Certainly after all those elements in **A** which are **smaller** than j

Counting sort: algorithm for sorting integers



Counting sort: algorithm for sorting integers



Types of sorting algorithms

In Place Sorting algorithm:

A sorting algorithm which uses

Example: Heap sort, Quick sort.

Stable Sorting algorithm:

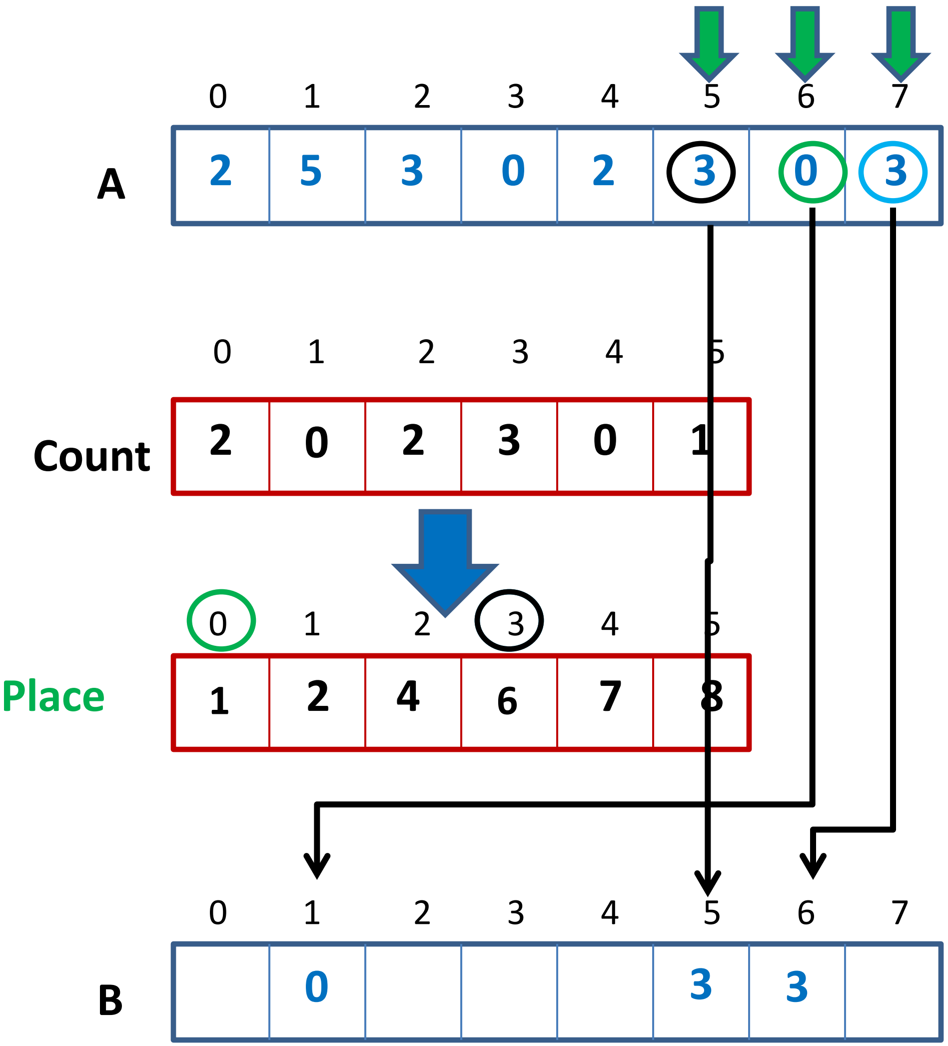
A sorting algorithm which preserves

	0	1	2	3	4	5	6	7
A	**2**	**5**	**3**	**0**	**6.1**	**3**	**7.9**	**4**

	0	1	2	3	4	5	6	7
A	**0**	**2**	**3**	**3**	**4**	**5**	**6.1**	**7.9**

Example: Merge sort.

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[$0 \ldots n-1$], k)

For j=0 to $k-1$ do Count[j]← 0;

For i=0 to $n-1$ do Count[A[i]]← Count[A[i]] +1;

Place[0]← Count[0];

For j=1 to $k-1$ do Place[j]← Place[$j-1$] + Count[j] ;

For i=$n-1$ to 0 do

{ B[Place[A[i]]−1]← A[i];

Place[A[i]] ← 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 ≤ 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** $\in \{0, \ldots, 9\}$

$$\overbrace{1011}\overbrace{0001}\overbrace{0101}\overbrace{1111}$$

No. of **digits** = 4

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

It is up to us how we define digit ?

Radix Sort

Input: An array **A** storing n integers, where

(i) each integer has exactly d **digits**.

(ii) each **digit** has **value** $< k$

(iii) $k < n$.

Output: Sorted array **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	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
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

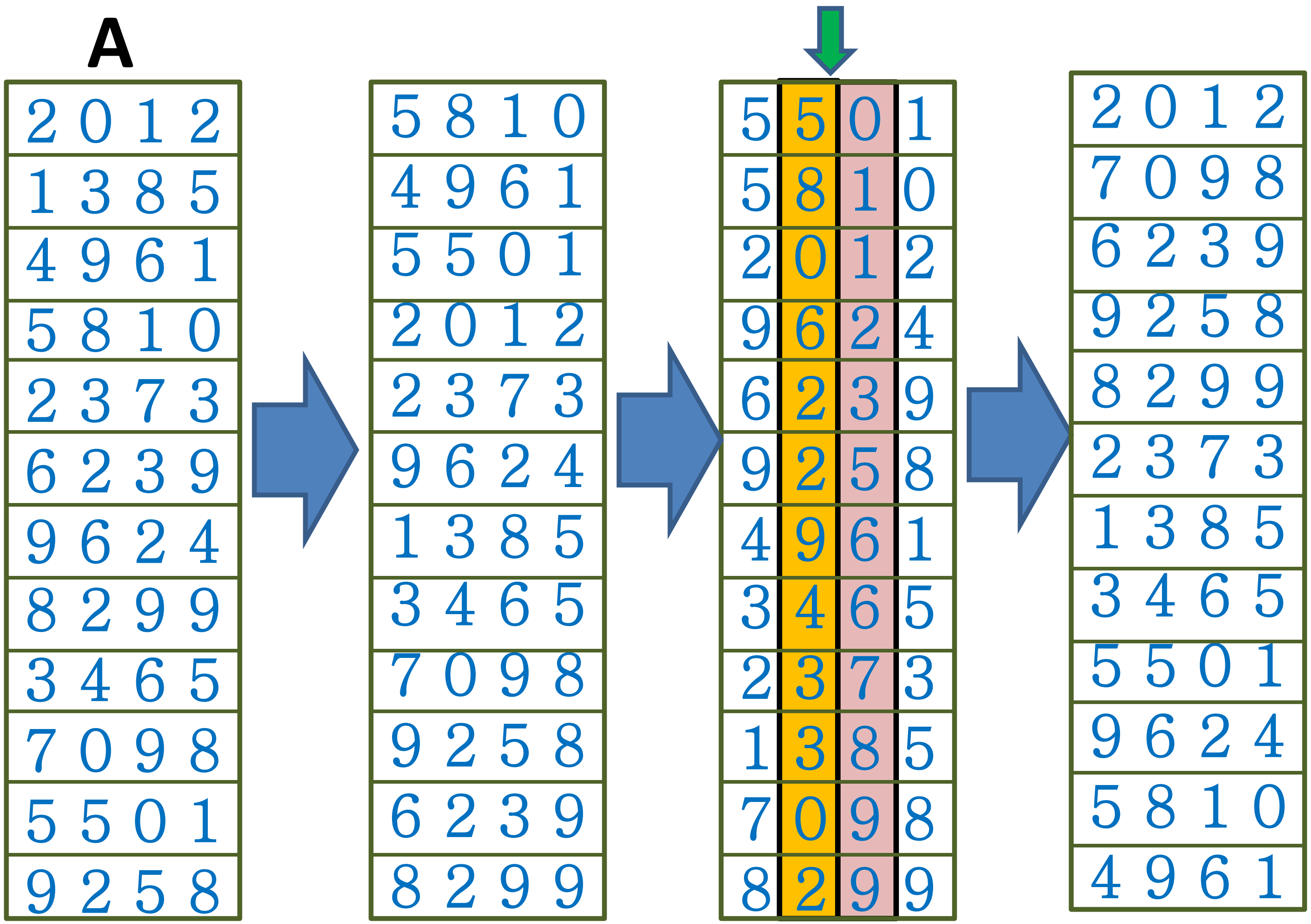
A

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

Demonstration of Radix Sort through example



Demonstration of Radix Sort through example

A				
2012	5810	5501	2012	1385
1385	4961	5810	7098	2012
4961	5501	2012	6239	2373
5810	2012	9624	9258	3465
2373	2373	6239	8299	4961
6239	9624	9258	2373	5501
9624	1385	4961	1385	5810
8299	3465	3465	3465	6239
3465	7098	2373	5501	7098
7098	9258	1385	9624	8299
5501	6239	7098	5810	9258
9258	8299	8299	4961	9624

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

Radix Sort

RadixSort(A[$0 \ldots n-1$], d, k)

{ **For** j=1 **to** d **do**

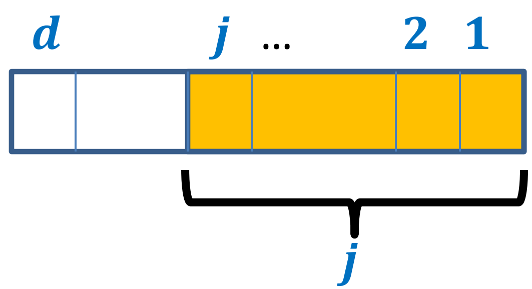
  Execute CountSort(A,k) with j**th digit** as the **key**;

 return **A**;

}

Correctness:

A number stored in **A** ➔



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**[$0 \ldots n-1$], 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:

	d	k	Time complexity
1 bit	$t \log n$	2	**O**($tn \log n$) ☹
$\log n$ bits	t	n	**O**(tn) ☺

What digit to use ?

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:**

  Compare **Quick sort** with **Radix sort** for sorting **long** integers.