CP312
Algorithm Design and Analysis I

LECTURE 9: SORTING IN LINEAR TIME

Comparison-based Sorting

- All the sorting algorithms we have seen so far are **comparison sorts**. That is, they only use comparisons to determine relative order of elements.

- Is there a comparison sort that can do better than $O(n \lg n)$?
  - Answer: NO!
  - Any comparison-based sorting algorithm must do at least $n \lg n$ amount of work in the worst-case

Sorting in Linear Time

- Is there any way to sort without doing comparisons?
- And if there is, how fast can they get?

Counting Sort

- **Input**: $A[1, \ldots, n]$ where $A[j] \in \{1, \ldots, k\}$
- **Output**: $A'[1, \ldots, n]$ which is A sorted

- Takes time $\Theta(n + k)$
- So if $k = O(n)$ then it would take time $\Theta(n)$

Counting Sort

Initialize $C[1, \ldots, k] \leftarrow [0, \ldots, 0]$

for $i = 1$ **to** n

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

for $i = 2$ **to** k

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

for $i = n$ **to** 1

$$A'\Big[C\big[A[i]\big]\Big] \leftarrow A[i]$$

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

Counting Sort: Example

	1	2	3	4	5
A	4	1	3	4	3

	1	2	3	4
C	0	0	0	0

Initialize $C[1, \dots, k] \leftarrow [0, \dots, 0]$

$k = 4$

Counting Sort: Example

	1	2	3	4	5
A	4	1	3	4	3

	1	2	3	4
C	0	0	0	0

for $i = 1$ **to** n

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

Counting Sort: Example

	1	2	3	4	5
A	4	1	3	4	3

	1	2	3	4
C	0	0	0	1

for $i = 1$ **to** n

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

Counting Sort: Example

	1	2	3	4	5
A	4	1	3	4	3

	1	2	3	4
C	1	0	0	1

for $i = 1$ **to** n

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

Counting Sort: Example

	1	2	3	4	5
A	4	1	3	4	3

	1	2	3	4
C	1	0	1	1

for $i = 1$ **to** n

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

Counting Sort: Example

	1	2	3	4	5
A	4	1	3	4	3

	1	2	3	4
C	1	0	1	2

for $i = 1$ **to** n

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

Counting Sort: Example

	1	2	3	4	5
A	4	1	3	4	3

	1	2	3	4
C	1	0	2	2

for $i = 1$ **to** n

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

Counting Sort: Example

	1	2	3	4	5
A	4	1	3	4	3

	1	2	3	4
C	1	0	2	2

for $i = 2$ **to** k

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

Counting Sort: Example

	1	2	3	4	5
A	4	1	3	4	3

	1	2	3	4
C	1	1	2	2

for $i = 2$ **to** k

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

Counting Sort: Example

	1	2	3	4	5
A	4	1	3	4	3

	1	2	3	4
C	1	1	3	2

for $i = 2$ **to** k

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

Counting Sort: Example

	1	2	3	4	5
A	4	1	3	4	3

	1	2	3	4
C	1	1	3	5

for $i = 2$ **to** k

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

Counting Sort: Example

	1	2	3	4	5
A	4	1	3	4	3

	1	2	3	4
C	1	1	3	5

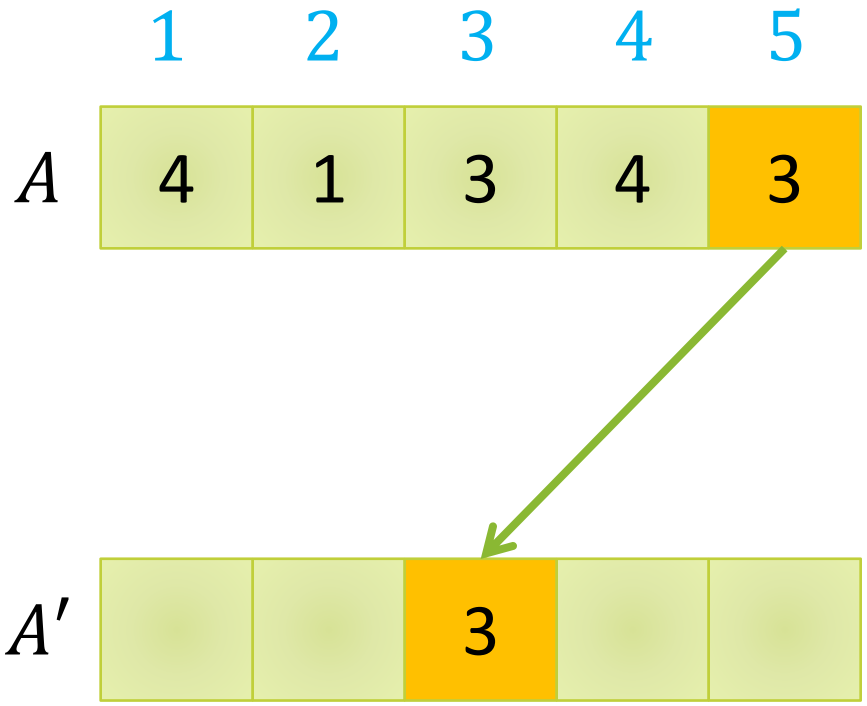
A'					

for $i = n$ **to** 1

$$A'\left[C\left[A[i]\right]\right] \leftarrow A[i]$$

$$C\left[A[i]\right] \leftarrow C\left[A[i]\right] - 1$$

Counting Sort: Example



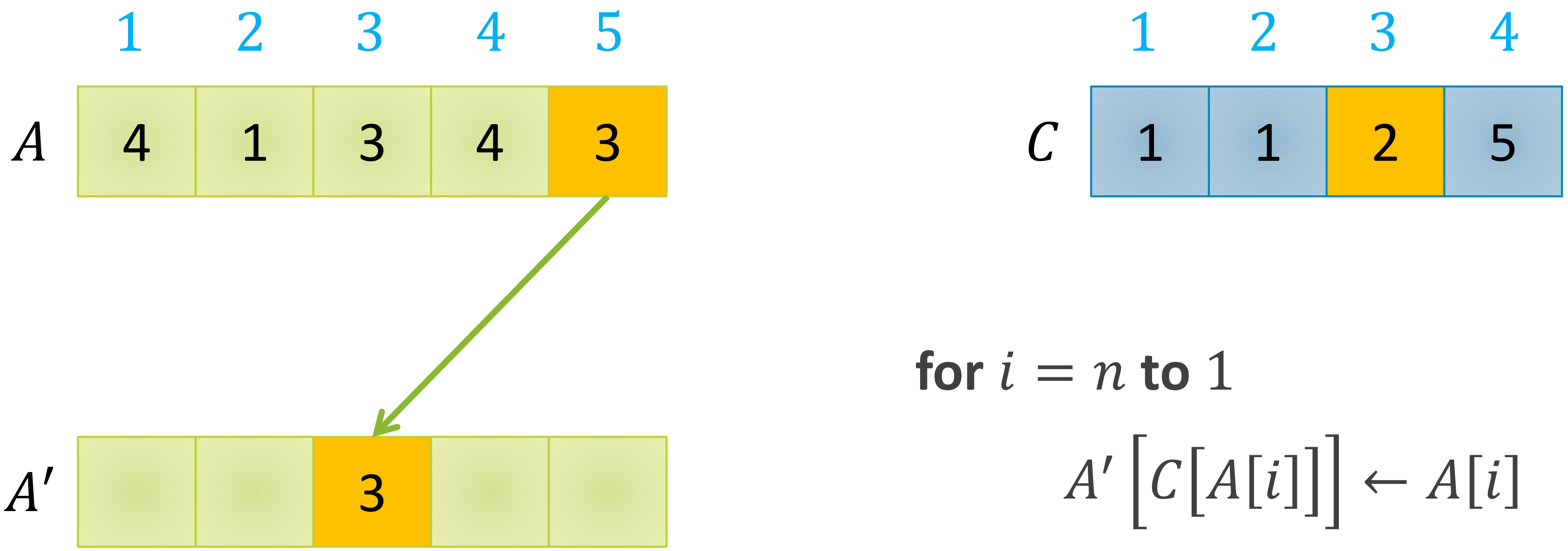
	1	2	3	4
C	1	1	3	5

for $i = n$ **to** 1

$$A'\left[C[A[i]]\right] \leftarrow A[i]$$

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

Counting Sort: Example



for $i = n$ **to** 1

$$A'\left[C\left[A[i]\right]\right] \leftarrow A[i]$$

$$C\left[A[i]\right] \leftarrow C\left[A[i]\right] - 1$$

Counting Sort: Example

	1	2	3	4	5
A	4	1	3	4	3

	1	2	3	4
C	1	1	2	5

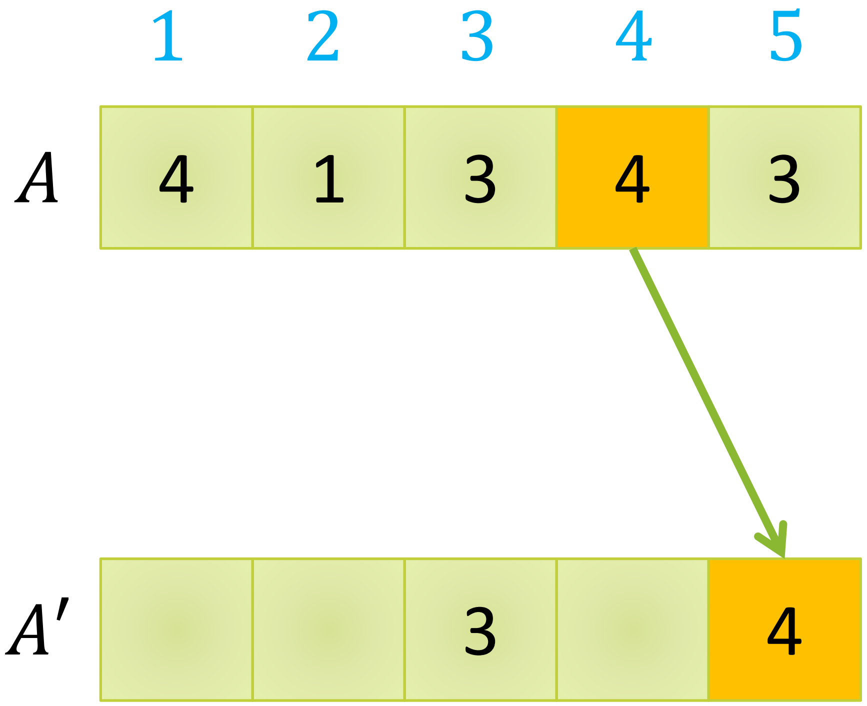
A'			3		4

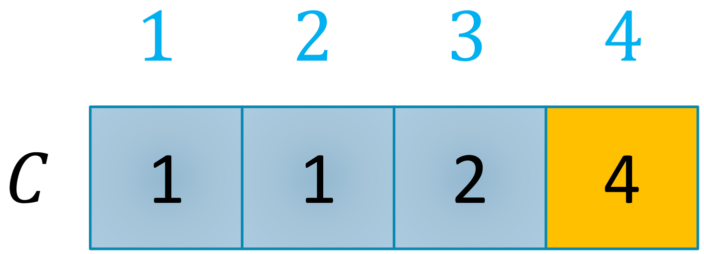
for $i = n$ **to** 1

$$A'\left[C\left[A[i]\right]\right] \leftarrow A[i]$$

$$C\left[A[i]\right] \leftarrow C\left[A[i]\right] - 1$$

Counting Sort: Example



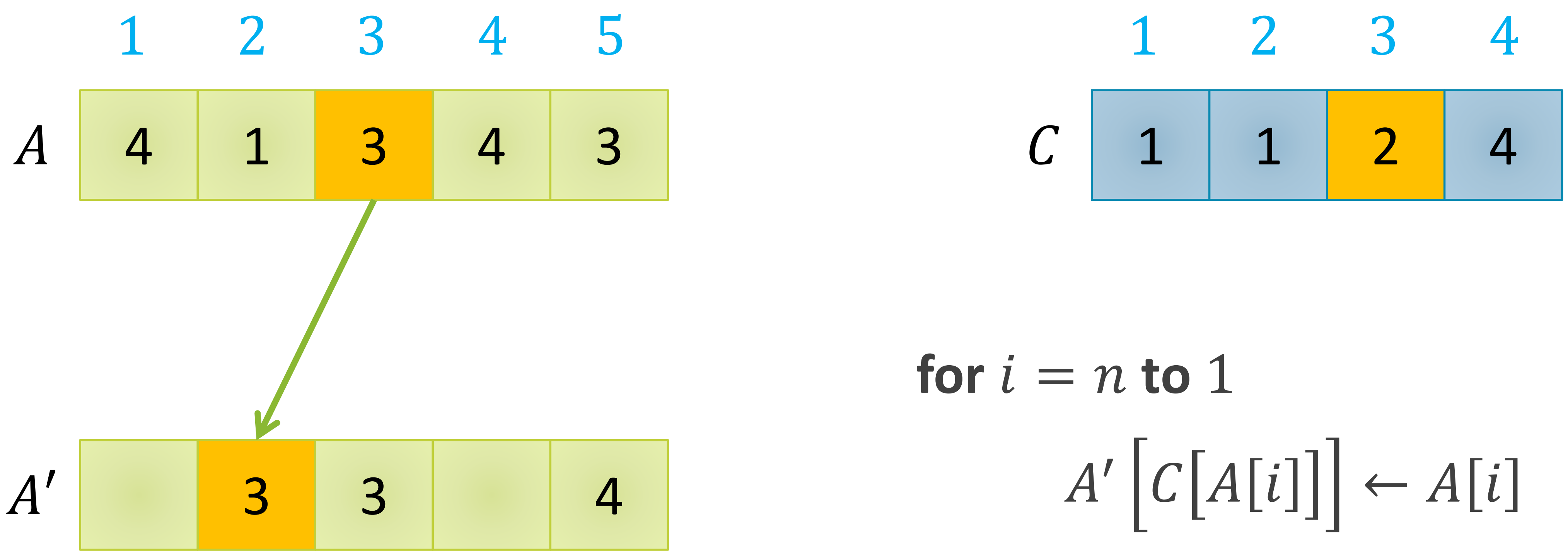


for $i = n$ **to** 1

$$A'\left[C\left[A[i]\right]\right] \leftarrow A[i]$$

$$C\left[A[i]\right] \leftarrow C\left[A[i]\right] - 1$$

Counting Sort: Example

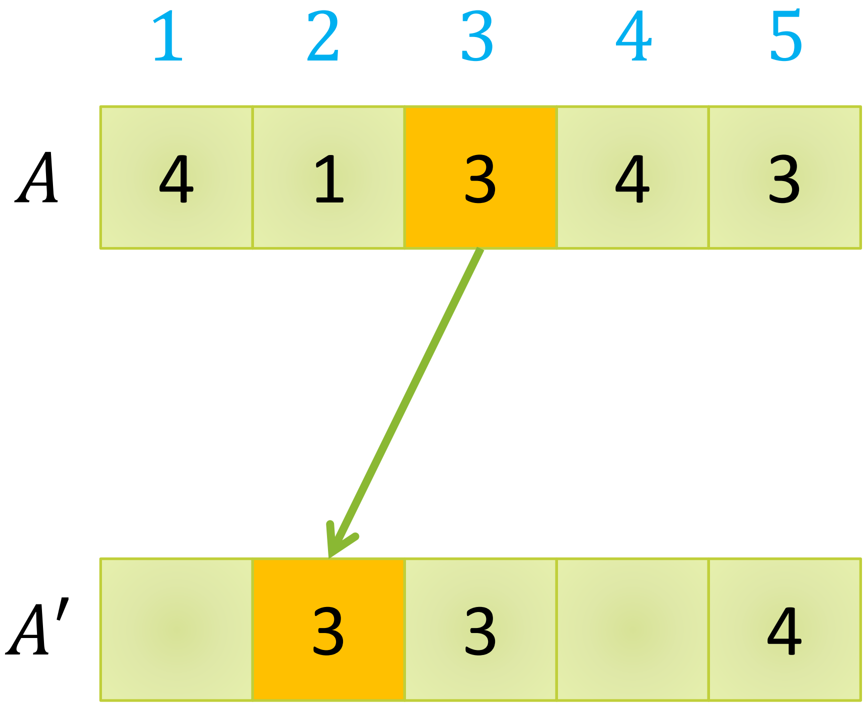


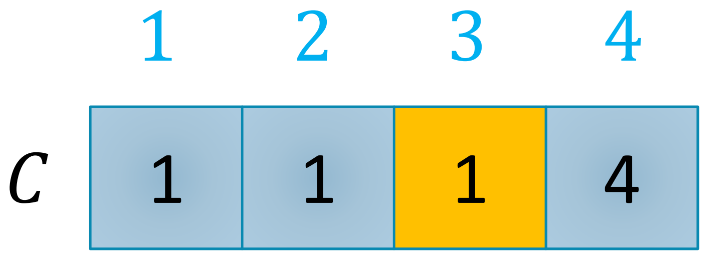
for $i = n$ **to** 1

$$A'\left[C\left[A[i]\right]\right] \leftarrow A[i]$$

$$C\left[A[i]\right] \leftarrow C\left[A[i]\right] - 1$$

Counting Sort: Example





for $i = n$ **to** 1

$$A'\left[C\left[A[i]\right]\right] \leftarrow A[i]$$

$$C\left[A[i]\right] \leftarrow C\left[A[i]\right] - 1$$

Counting Sort: Example

	1	2	3	4	5
A	4	1	3	4	3

	1	2	3	4
C	1	1	1	4

A'	1	3	3		4

for $i = n$ **to** 1

$$A'\left[C\left[A[i]\right]\right] \leftarrow A[i]$$

$$C\left[A[i]\right] \leftarrow C\left[A[i]\right] - 1$$

Counting Sort: Example

	1	2	3	4	5
A	4	1	3	4	3

	1	2	3	4
C	0	1	1	4

A'	1	3	3	4	4

for $i = n$ **to** 1

$$A'\left[C\left[A[i]\right]\right] \leftarrow A[i]$$

$$C\left[A[i]\right] \leftarrow C\left[A[i]\right] - 1$$

Counting Sort: Example

	1	2	3	4	5
A	4	1	3	4	3

	1	2	3	4
C	0	1	1	3

A'	1	3	3	4	4

for $i = n$ **to** 1

$$A'\left[C\left[A[i]\right]\right] \leftarrow A[i]$$

$$C\left[A[i]\right] \leftarrow C\left[A[i]\right] - 1$$

Counting Sort: Example

	1	2	3	4	5
A	4	1	3	4	3

	1	2	3	4
C	0	1	1	3

A'	1	3	3	4	4

for $i = n$ **to** 1

$$A'\left[C[A[i]]\right] \leftarrow A[i]$$

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

Counting Sort: Running-Time Analysis

Initialize $C[1, \dots, k] \leftarrow [0, \dots, 0]$

$\Theta(k)$

for $i = 1$ **to** n

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

$\Theta(n)$

for $i = 2$ **to** k

$\quad C[i] \leftarrow C[i] + C[i-1]$

$\Theta(k)$

for $i = n$ **to** 1

$\quad A'[C[A[i]]] \leftarrow A[i]$

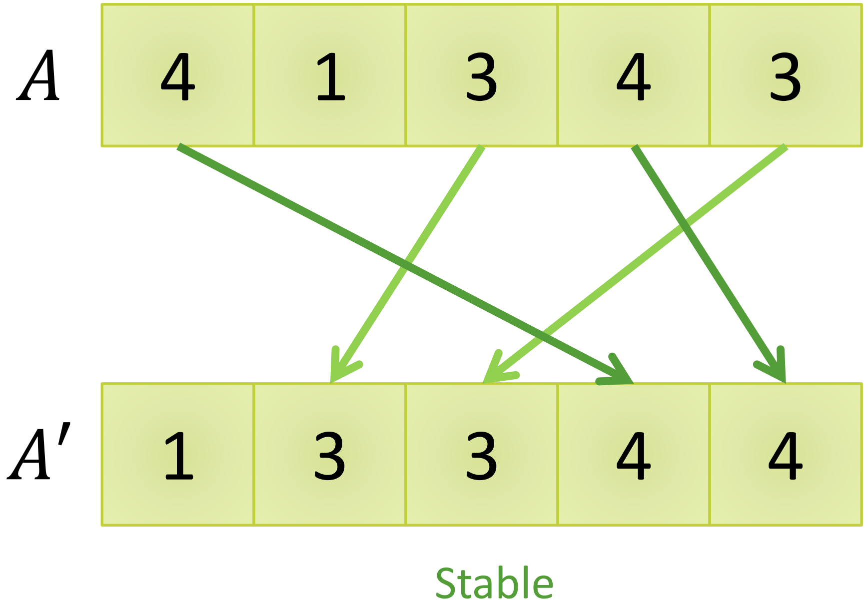
$\quad C[A[i]] \leftarrow C[A[i]] - 1$

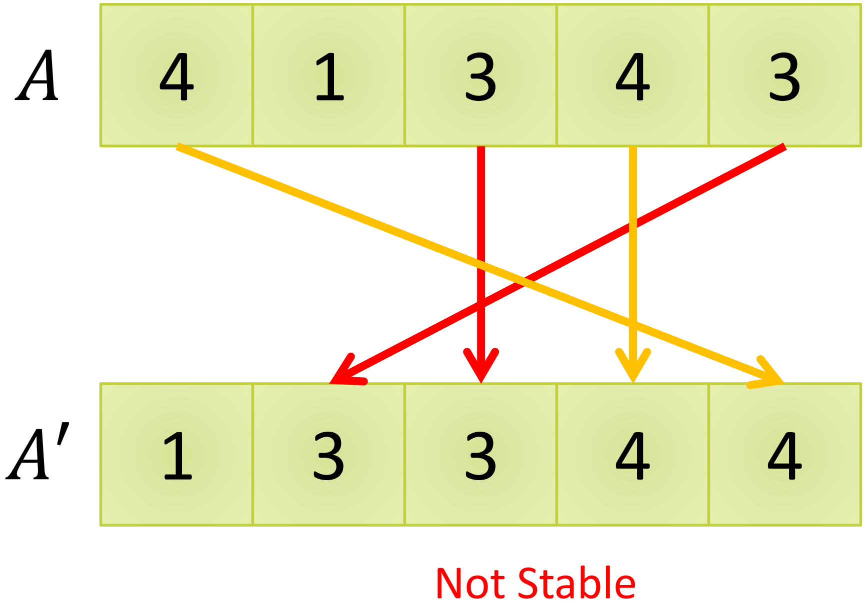
$\Theta(n)$

$$T(n) = \Theta(n + k)$$

Stability in Sorting

- Counting sort is a **stable** sort: it preserves the input order among **equal** elements.





An Issue with Counting Sort

- Suppose we want to sort the following array using counting sort.

329	457	657	839	436	720	355

- We need to create a HUGE auxiliary storage array to count them since the range k is large.

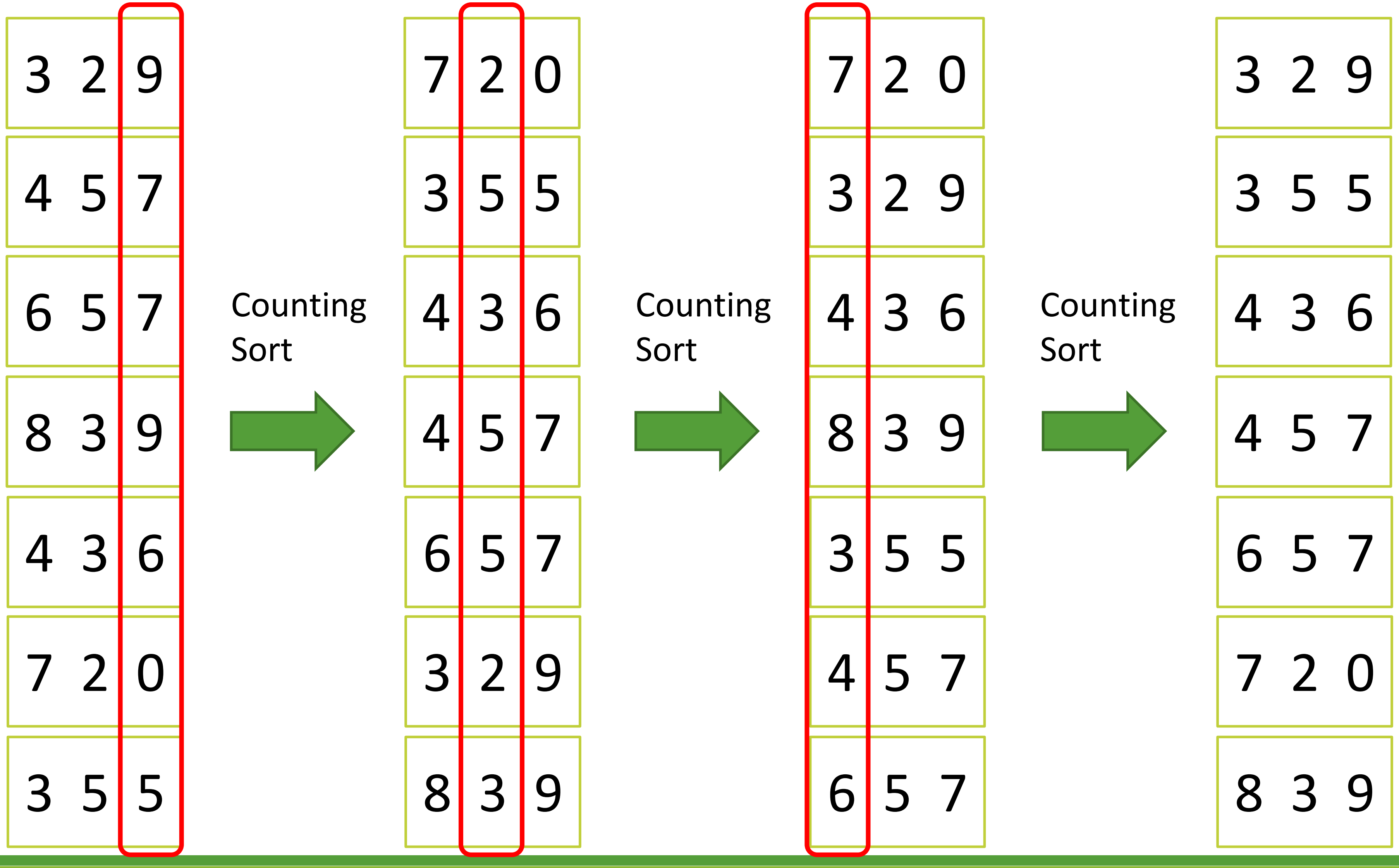
329	330	331	332		838	839
1	0	0	0	....	0	1

Radix Sort

- Digit-by-digit sorting on **least significant digit first**.
- Requires an auxiliary stable sort

329	457	657	839	436	720	355

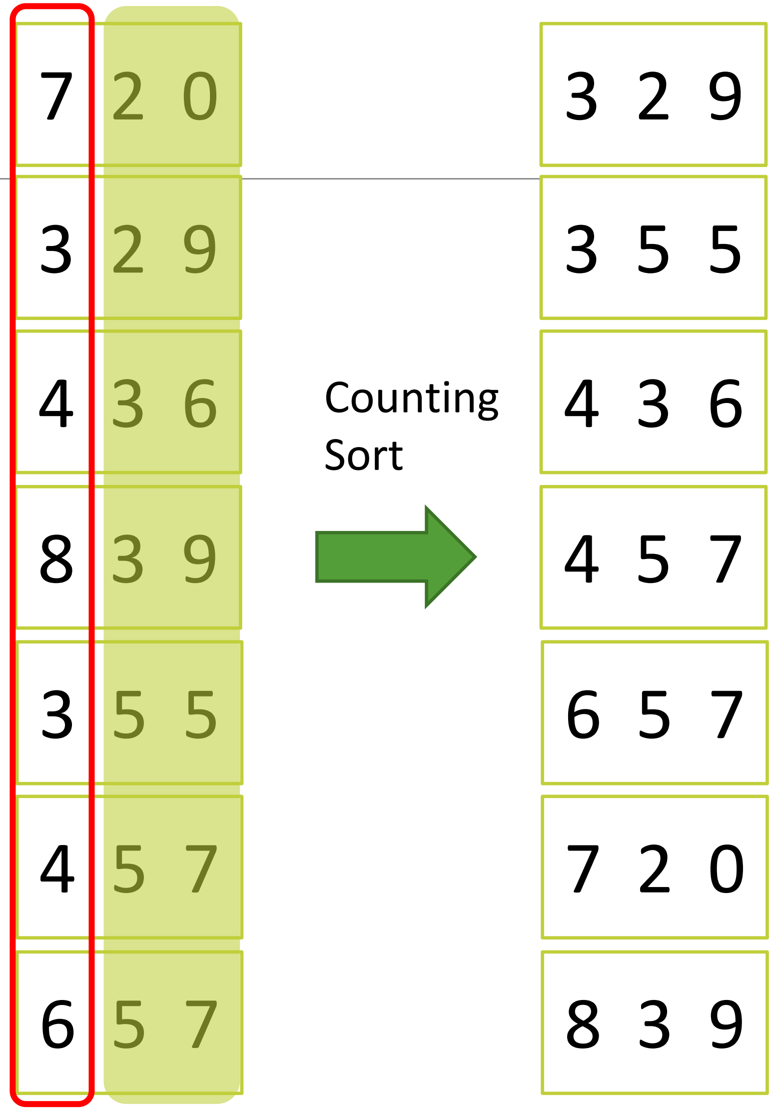
Radix Sort



3 2 9
4 5 7
6 5 7
8 3 9
4 3 6
7 2 0
3 5 5
Counting Sort
7 2 0
3 5 5
4 3 6
4 5 7
6 5 7
3 2 9
8 3 9
Counting Sort
7 2 0
3 2 9
4 3 6
8 3 9
3 5 5
4 5 7
6 5 7
Counting Sort
3 2 9
3 5 5
4 3 6
4 5 7
6 5 7
7 2 0
8 3 9

Correctness of Radix Sort

- Induction on digit position:
- Assume that the numbers are sorted by their low-order $k-1$ digits
- Sort on digit k:
  - Two numbers that differ in digit t are correctly sorted.
  - Two numbers equal in digit t are put in the same order as the input $\Rightarrow$ correct order.



Analysis of Radix Sort

- Complexity:
  - Each pass in the for loop takes O(n+k).
  - We have d passes => O(dn+dk).
  - When k = O(n) => O(dn+dn) = O(2dn) = O(n) (Assuming that d is a constant).

Radix Sort

- In practice, radix sort is fast for large inputs, as well as simple to code and maintain.
- **Example:** 4-digit number
  - At most 3 passes when sorting ≥ 2000 numbers
  - Merge sort and quicksort do at least $\lg 2000 \approx 11$ passes
- Downside: Unlike quicksort, radix sort displays little locality of reference, and thus a well-tuned quicksort fares better on modern processor, which feature steep memory hierarchies.

Summary of Sorting Algorithms

	Algorithm	Worst-Case Running Time	Average-Case Running Time	In-Place
Comparison-based Sorts	Insertion Sort	$\Theta(n^2)$	$\Theta(n^2)$	Y
	Merge sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$	N
	Quicksort	$\Theta(n^2)$	$\Theta(n \lg n)$	Y
	Randomized Quicksort	Expected: $\Theta(n \lg n)$	-	Y
	BST-Sort	$\Theta(n^2)$	$\Theta(n \lg n)$	Y
	Heapsort	$O(n \lg n)$	$O(n \lg n)$	Y
Distribution-based Sorts	Counting Sort	$\Theta(k + n)$	$\Theta(k + n)$	N
	Radix Sort	$\Theta(d(k + n))$	$\Theta(d(k + n))$	N
	Bucket Sort	$\Theta(n^2)$	$\Theta(n)$	N

Other Sorting Algorithms

- Bubble Sort
- Selection Sort
- Shell Sort
- Bitonic Sort
- Timsort