

Lower Bounds for Sorting, Linear Time Sorting, Counting Sort

(Class 15)

From Book's Chapter 8

In-place, Stable Sorting

- In merge sort we needed a temporary array when we perform merging the two subarrays.
- In Quick sort, the algorithm updates the original array without the need of temporary array.
- Another fact Stable sort or unstable sort.

- For example, if the array has duplicate numbers, then which position they will appear.
- An *in-place* sorting algorithm is one that uses no additional array for storage.
- A sorting algorithm is stable if duplicate elements remain in the same relative position after sorting.

9 3 3' 5 6 5' 2 1 3''

unsorted

1 2 3 3' 3'' 5 5' 6 9

stable sort

1 2 3' 3 3'' 5' 5 6 9

unstable

- Bubble sort, insertion sort and selection sort are in-place sorting algorithms.
- Bubble sort and insertion sort can be implemented as stable algorithms, but selection sort cannot (without significant modifications).
- Merge sort is a stable algorithm but not an in-place algorithm.
- It requires extra array storage.

- Quicksort is not stable but is an in-place algorithm.
- Heapsort is an in-place algorithm but is not stable.
- Stable sorting is important because if our key has satellite data (the whole record) has to be moved after sorting.
- It can also leads to disturb the data because if two keys are same then after sorting which key points to which record?
- It can affect the performance.

Lower Bounds for Sorting

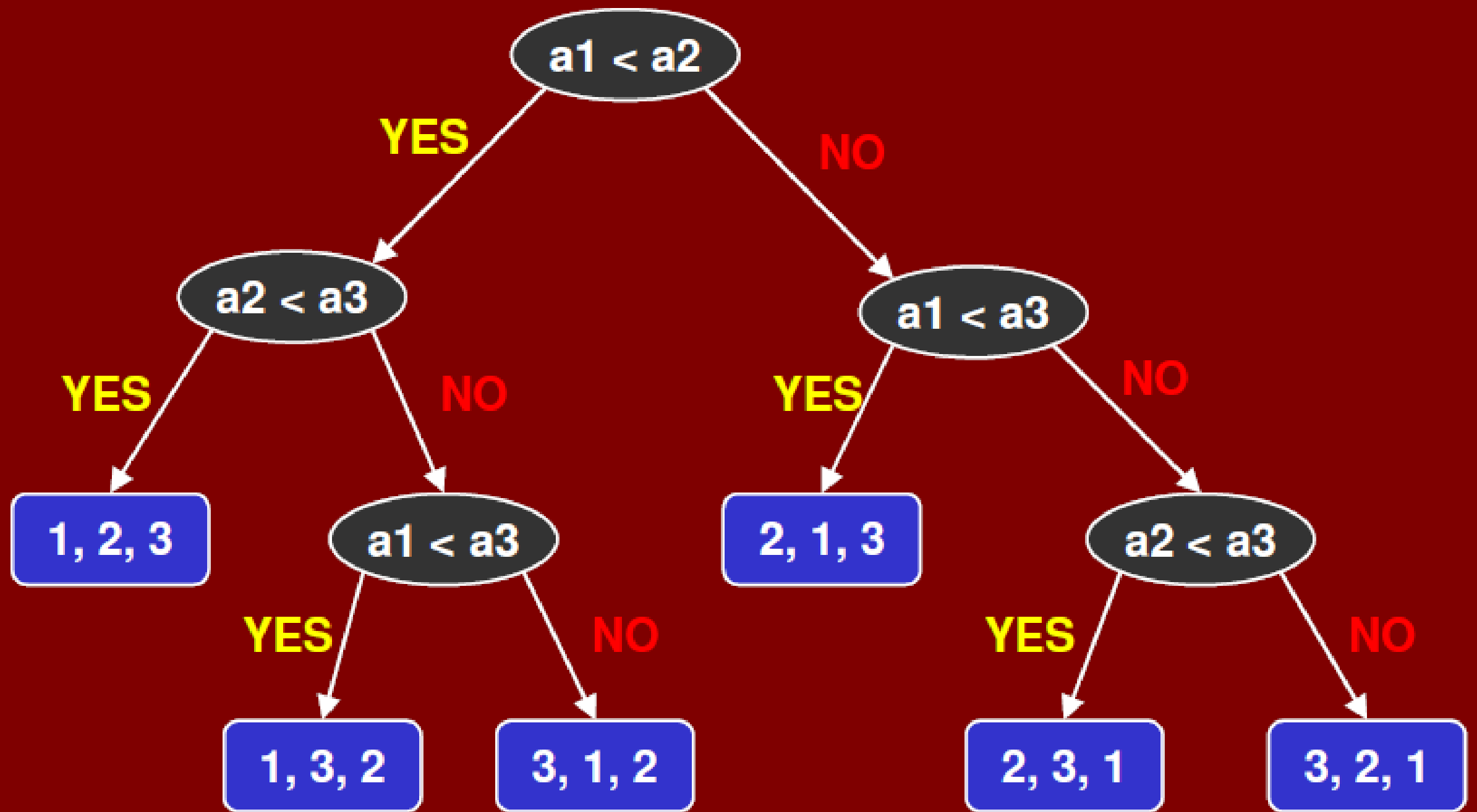
- The best we have seen so far is $O(n \log n)$ algorithms for sorting.
- Is it possible to do better than $O(n \log n)$?
- If a sorting algorithm is solely based on comparison of keys in the array, then it is impossible to sort more efficiently than $\Omega(n \log n)$ time.
- All algorithms we have seen so far are comparison-based sorting algorithms.

- Consider sorting three distinct numbers a_1, a_2, a_3 .
- There are $3! = 6$ possible combinations:

$(a_1, a_2, a_3), (a_1, a_3, a_2), (a_3, a_2, a_1)$

$(a_3, a_1, a_2), (a_2, a_1, a_3), (a_2, a_3, a_1)$

- One of these permutations leads to the numbers in sorted order.
- To make these combinations we have to perform the comparisons.
- The comparison-based algorithm defines a decision tree.
- Here is the tree for the three numbers.



- For n elements, there will be $n!$ possible permutations.
- The height of the tree is exactly equal to $T(n)$, the running time of the algorithm.
- The height is $T(n)$ because any path from the root to a leaf corresponds to a sequence of comparisons made by the algorithm.
- Any binary tree of height $T(n)$ has at most $2^{T(n)}$ leaves. ($n = 2^h$)

- Thus, a comparison-based sorting algorithm can distinguish between at most $2^{T(n)}$ different final outcomes.
- So, we have:

$$2^{T(n)} = n!$$

- And therefore (taking log on both sides):

$$T(n) = \log n!$$

- We can use *Stirling's approximation* for $n!$

$$n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$$

- Where e is the base of natural log.

- Therefore

$$\begin{aligned} T(n) &= \log(\sqrt{2\pi n} \left(\frac{n}{e}\right)^n) \\ &= \log(\sqrt{2\pi n}) + \log\left(\frac{n}{e}\right)^n \\ &= \log(\sqrt{2\pi n}) + n \log n + n \log e \end{aligned}$$

- The dominating term is $n \log n$.

$$\mathbf{T(n) \in \Omega(n \log n)}$$

- We thus have the following theorem.
- **Theorem 1:** Any comparison-based sorting algorithm has worst-case running time $(n \log n)$.

Linear Time Sorting

- The lower bound implies that if we hope to sort numbers faster than $O(n \log n)$, we cannot do it by making comparisons alone.
- Is it possible to sort without making comparisons?
- The answer is yes, but only under very restrictive circumstances.

- Many applications involve sorting small integers (e.g., sorting characters, exam scores, etc.).
- We present three algorithms based on the theme of speeding up sorting in special cases, by not making comparisons.

Counting Sort

- We will consider three algorithms that are faster and work by not making comparisons:
 - Counting Sort
 - Bucket or Bin Sort
 - Radix Sort

- Counting sort assumes that the numbers to be sorted are in the range 1 to k where k is small.
- The basic idea is to determine the rank of each number in final sorted array.
- Recall that the rank of an item is the number of elements that are less than or equal to it.
- Once we know the ranks, we simply place all the numbers to their final rank position in an output array.

- The question is how to find the rank of an element without comparing it to the other elements of the array?.
- The algorithm uses three arrays.:
 - $A[1 \dots n]$: holds the initial input.
 - $B[1 \dots n]$: holds the sorted output.
 - $C[1 \dots k]$: is an array of integers.
- $C[x]$ is the rank of x in A , where $x \in [1..k]$.

- The algorithm is remarkably simple, but deceptively clever.
- The algorithm operates by first constructing C .
- This is done in two steps.
- First, we set $C[x]$ to be the number of elements of $A[j]$ that are equal to x .

- We can do this initializing C to zero, and then for each j , from 1 to n , we increment $C[A[j]]$ by 1.
- Thus, if $A[j] = 5$, then the 5th element of C is incremented, indicating that we have seen one more 5.
- To determine the number of elements that are less than or equal to x , we replace $C[x]$ with the sum of elements in the sub array $R[1:x]$.
- This is done by just keeping a running total of the elements of C .

- $C[x]$ now contains the rank of x .
- This means that if $x = A[j]$ then the final position of $A[j]$ should be at position $C[x]$ in the final sorted array.
- Thus, we set $B[C[x]] = A[j]$.
- Notice we need to be careful if there are duplicates, since we do not want them to overwrite the same location of B .
- To do this, we decrement $C[i]$ after copying.

Counting Sort Algorithm

COUNTING-SORT (array A, array B, int k)

1 for i \leftarrow 1 to k

2 C[i] \leftarrow 0

k times

3 for j \leftarrow 1 to length[A]

4 C[A[j]] \leftarrow C[A[j]] + 1

n times

// C[i] now contains the number of elements = i

5 for i \leftarrow 2 to k

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

k-1 times

// C[i] now contains the number of elements \leq i

7 for j \leftarrow length[A] downto 1

8 B[C[A[j]]] \leftarrow A[j]

9 C[A[j]] \leftarrow C[A[j]] - 1

n times

- There are four (unnested) loops, executed k times, n times, $k - 1$ times, and n times, respectively.
- So, the total running time is:

$$T(n) = n + k + k - 1 + n$$

$$T(n) = 2n + 2k - 1$$

- As $k \leq n$:

$$\mathbf{T(n) = O(n)}$$

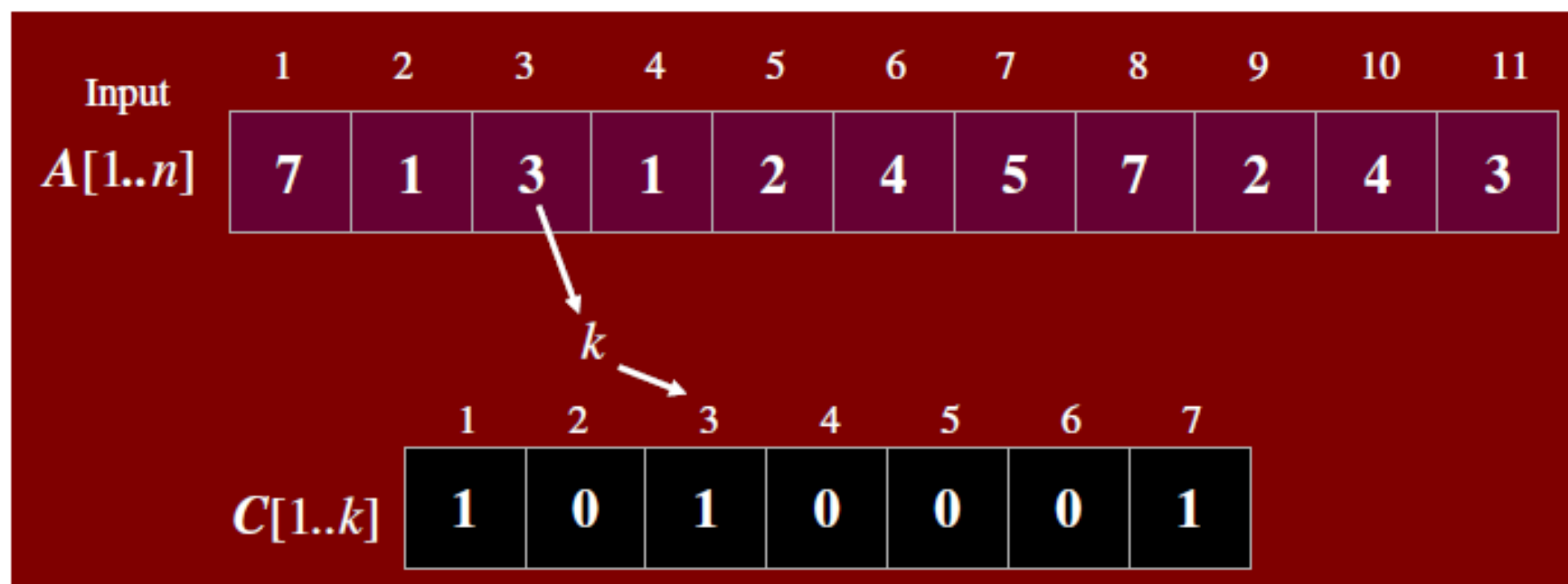
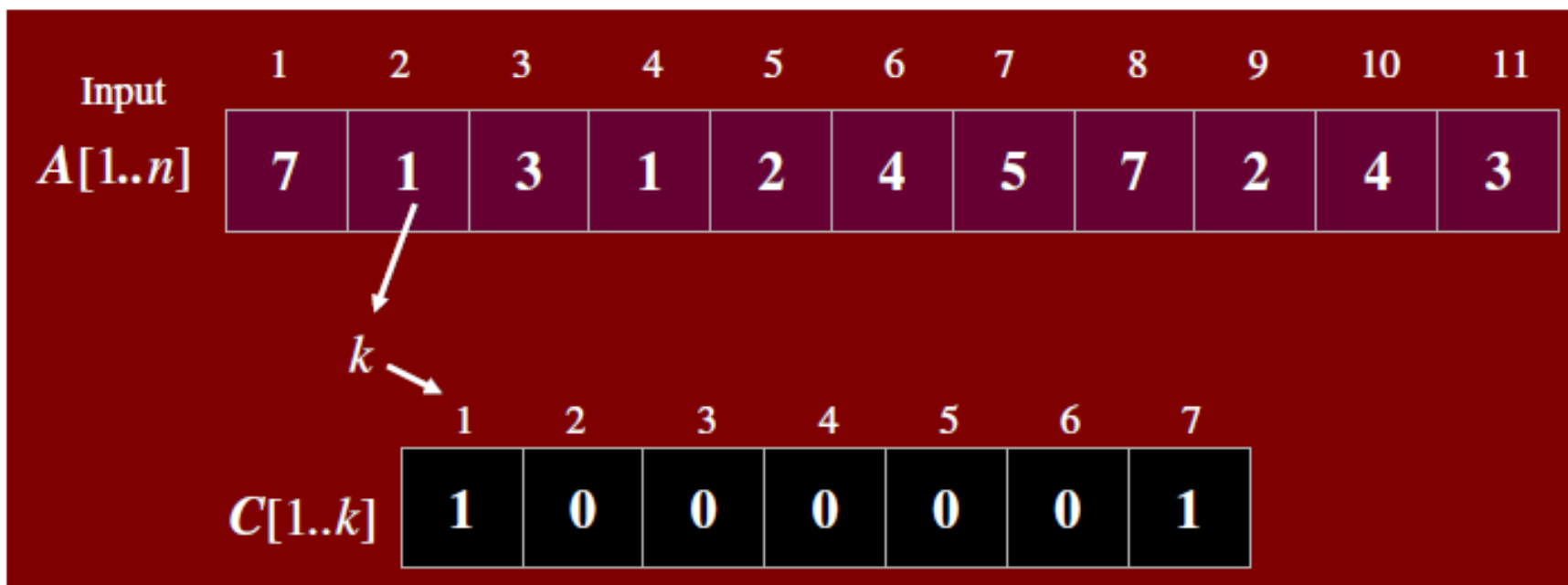
Input	1	2	3	4	5	6	7	8	9	10	11
$A[1..n]$	7	1	3	1	2	4	5	7	2	4	3

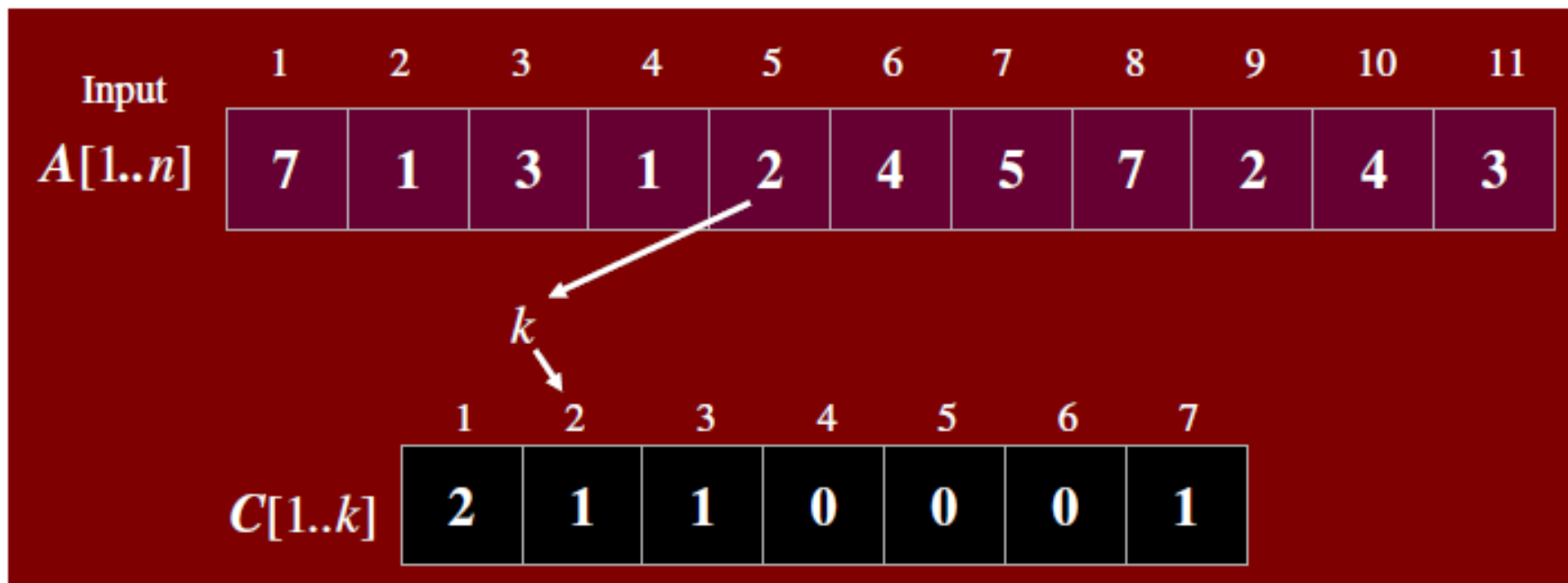
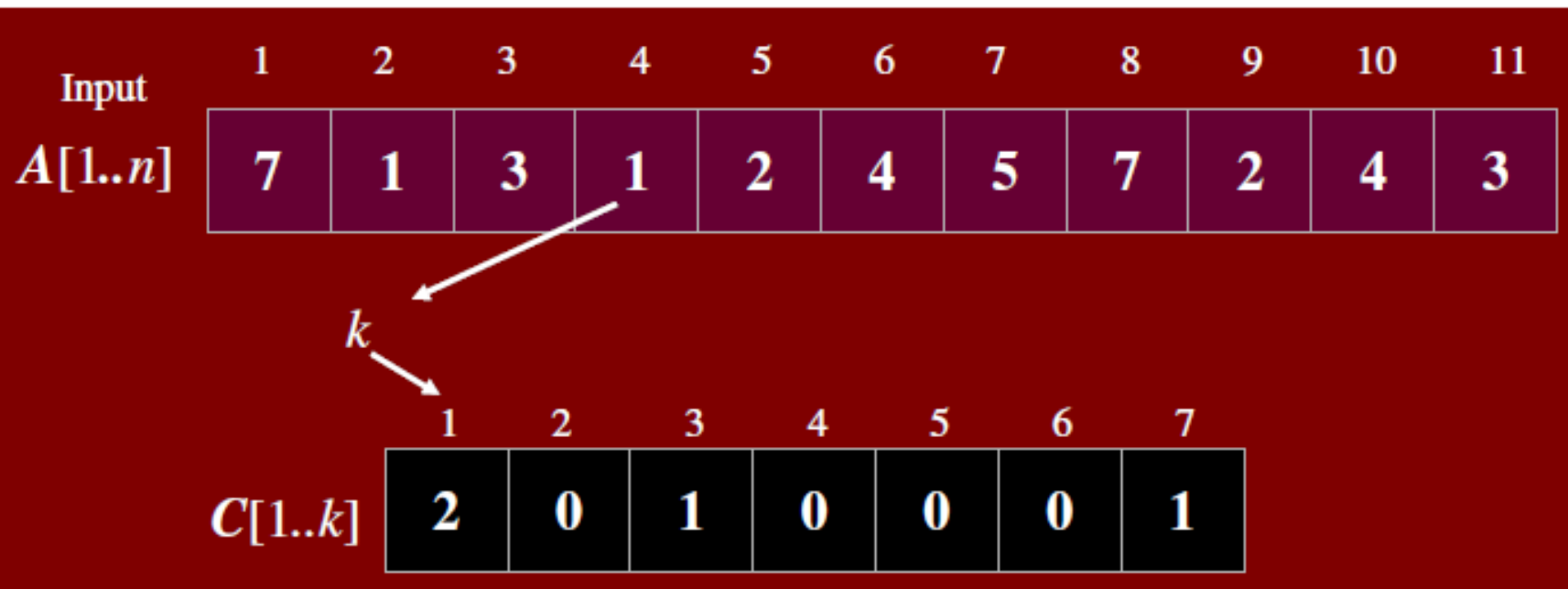
$k = 7$

	1	2	3	4	5	6	7
$C[1..k]$	0	0	0	0	0	0	0

Input	1	2	3	4	5	6	7	8	9	10	11
$A[1..n]$	7	1	3	1	2	4	5	7	2	4	3

	1	2	3	4	5	6	7
$C[1..k]$	0	0	0	0	0	0	1





Input	1	2	3	4	5	6	7	8	9	10	11
$A[1..n]$	7	1	3	1	2	4	5	7	2	4	3
<i>finally</i>											
	1	2	3	4	5	6	7				
$C[1..k]$	2	2	2	2	1	0	2				

Input	1	2	3	4	5	6	7	8	9	10	11
$A[1..n]$	7	1	3	1	2	4	5	7	2	4	3

	1	2	3	4	5	6	7
$C[1..k]$	2	2	2	2	1	0	2

for $i = 2$ to 7
do $C[i] = C[i] + C[i-1]$

	1	2	3	4	5	6	7
C	2	4	6	8	9	9	11

↓
6 elements ≤ 3

Input											
$A[1..n]$	7	1	3	1	2	4	5	7	2	4	3
	1	2	3	4	5	6	7	8	9	10	11

Output											
$B[1..n]$						3					

$$B[6] = B[C[3]] = B[C[A[11]]] = A[11] = 3$$

	1	2	3	4	5	6	7
C	2	4	6	8	9	9	11

$$C[A[11]] = C[A[11]] - 1$$

C	2	4	5	8	9	9	11
-----	---	---	---	---	---	---	----

Input											
$A[1..n]$	7	1	3	1	2	4	5	7	2	4	3
	1	2	3	4	5	6	7	8	9	10	11

Output											
$B[1..n]$						3		4			

$$B[8] = B[C[4]] = B[C[A[10]]] = A[10] = 4$$

	1	2	3	4	5	6	7
C	2	4	5	8	9	9	11

$$C[A[10]] = C[A[10]] - 1$$

C	2	4	5	7	9	9	11
-----	---	---	---	---	---	---	----

Input											
$A[1..n]$	7	1	3	1	2	4	5	7	2	4	3
	1	2	3	4	5	6	7	8	9	10	11

Output											
$B[1..n]$				2		3		4			

$$B[4] = B[C[2]] = B[C[A[9]]] = A[9] = 2$$

	1	2	3	4	5	6	7
C	2	4	5	7	9	9	11

$$C[A[9]] = C[A[9]] - 1$$

C	2	3	5	7	9	9	11
-----	---	---	---	---	---	---	----

Input											
$A[1..n]$	7	1	3	1	2	4	5	7	2	4	3
	1	2	3	4	5	6	7	8	9	10	11
Output											
$B[1..n]$				2		3		4	5		7

$$B[9] = B[C[5]] = B[C[A[7]]] = A[7] = 5$$

	1	2	3	4	5	6	7
C	2	3	5	7	9	9	10

$$C[A[5]] = C[A[5]] - 1$$

C	2	3	5	7	8	9	10
-----	---	---	---	---	---	---	----

Input											
$A[1..n]$	7	1	3	1	2	4	5	7	2	4	3
	1	2	3	4	5	6	7	8	9	10	11
Output											
$B[1..n]$				2		3	4	4	5		7

$$B[7] = B[C[4]] = B[C[A[6]]] = A[6] = 4$$

	1	2	3	4	5	6	7
C	2	3	5	7	8	9	10

$$C[A[6]] = C[A[6]] - 1$$

C	2	3	5	6	8	9	10
-----	---	---	---	---	---	---	----

Input											
$A[1..n]$	7	1	3	1	2	4	5	7	2	4	3
	1	2	3	4	5	6	7	8	9	10	11
Output											
$B[1..n]$			2	2		3	4	4	5		7

$$B[3] = B[C[2]] = B[C[A[5]]] = A[5] = 2$$

	1	2	3	4	5	6	7
C	2	3	5	7	8	9	10

$$C[A[5]] = C[A[5]] - 1$$

C	2	2	5	6	8	9	10
-----	---	---	---	---	---	---	----

Input											
$A[1..n]$	7	1	3	1	2	4	5	7	2	4	3
	1	2	3	4	5	6	7	8	9	10	11
Output											
$B[1..n]$		1	2	2		3	4	4	5		7

$$B[2] = B[C[1]] = B[C[A[4]]] = A[4] = 1$$

	1	2	3	4	5	6	7
C	2	2	5	7	8	9	10

$$C[A[4]] = C[A[4]] - 1$$

C	1	2	5	6	8	9	10
-----	---	---	---	---	---	---	----

Input											
$A[1..n]$	7	1	3	1	2	4	5	7	2	4	3
	1	2	3	4	5	6	7	8	9	10	11
Output											
$B[1..n]$		1	2	2		3	4	4	5		7

$$B[2] = B[C[1]] = B[C[A[4]]] = A[4] = 1$$

	1	2	3	4	5	6	7
C	2	2	5	7	8	9	10

$$C[A[4]] = C[A[4]] - 1$$

C	1	2	5	6	8	9	10
-----	---	---	---	---	---	---	----

Input											
$A[1..n]$	7	1	3	1	2	4	5	7	2	4	3
	1	2	3	4	5	6	7	8	9	10	11

Output											
$B[1..n]$	1	1	2	2	3	3	4	4	5		7

$$B[1] = B[C[1]] = B[C[A[2]]] = A[2] = 1$$

	1	2	3	4	5	6	7
C	1	2	4	7	8	9	10

$$C[A[3]] = C[A[3]] - 1$$

C	0	2	4	6	8	9	10
-----	---	---	---	---	---	---	----

Input											
$A[1..n]$	7	1	3	1	2	4	5	7	2	4	3
	1	2	3	4	5	6	7	8	9	10	11
Output											
$B[1..n]$	1	1	2	2	3	3	4	4	5	7	7

$$B[10] = B[C[7]] = B[C[A[1]]] = A[1] = 7$$

	1	2	3	4	5	6	7
C	0	2	4	7	8	9	10

C

0	2	4	6	8	9	9
---	---	---	---	---	---	---

$$C[A[1]] = C[A[1]] - 1$$

- Counting sort is not an in-place sorting algorithm but it is stable.
- Stability is important because data are often carried with the keys being sorted. radix sort (which uses counting sort as a subroutine) relies on it to work correctly.
- Stability achieved by running the last loop down from n to 1 and not the other way around.

- The numbers 1, 2, 3, 4, and 7, each appear twice. The two 4's have been given the superscript "*".
- Numbers are placed in the output B array starting from the right.
- The two 4's maintain their relative position in the B array.
- If the sorting algorithm had caused 4** to end up on the left of 4*, the algorithm would be termed unstable.

Input											
$A[1..n]$	7	1	3	1	2	4*	5	7	2	4**	3
	1	2	3	4	5	6	7	8	9	10	11
Output											
$B[1..n]$						3		4**			

$$B[8] = B[C[4]] = B[C[A[10]]] = A[10] = 4$$

	1	2	3	4	5	6	7
C	2	4	5	8	9	9	11

$$C[A[10]] = C[A[10]] - 1$$

C	2	4	5	7	9	9	11
-----	---	---	---	---	---	---	----

Input											
$A[1..n]$	7	1	3	1	2	4*	5	7	2	4**	3
	1	2	3	4	5	6	7	8	9	10	11
Output											
$B[1..n]$				2		3	4*	4**	5		7

$$B[7] = B[C[4]] = B[C[A[6]]] = A[6] = 4$$

	1	2	3	4	5	6	7
C	2	3	5	7	8	9	10

$$C[A[6]] = C[A[6]] - 1$$

C	2	3	5	6	8	9	10
-----	---	---	---	---	---	---	----

Input											
$A[1..n]$	7'	1^	3#	1^^	2+	4*	5	7''	2++	4**	3##
	1	2	3	4	5	6	7	8	9	10	11

Output											
$B[1..n]$	1^	1^^	2+	2++	3#	3##	4*	4**	5	7'	7''