# Data Structures & Algorithms (7)

## CS F211 / IS F 211

# Outline of this slide


- Non-comparison-based sorting

# Counting sort

# The simplest setting

- Input: $n$ distinct positive integers $A[1], \ldots, A[n]$ with $A[i] \leq k$ for each $i = 1, 2, \ldots, n$, for some known number $k = O(n)$.

- Output: a reordering of $A$ such that the resulting list is monotonically increasing.

# Solvable in $O(n)$ time?

Note that you are allowed to inspect the value of each $A[i]$.

# An algorithm

allocate an array $C[1 \ldots k]$; — $O(1)$ time

for each $j = 1$ to $k$ — $O(k)$ time
    let $C[j] = 0$;

for each $i = 1$ to $n$ — $O(n)$ time
    increment $C[A[i]]$ by one;

for each $j = 1$ to $k$ — $O(k)$ time
    if $C[j] > 0$
        print $j$;

The running time is $O(n + k) = O(n)$.

# Slightly harder setting

- Input: $n$ ~~distinct~~ positive integers $A[1], \ldots, A[n]$ with $A[i] \leq k$ for each $i = 1, 2, \ldots, n$, for some known number $k = O(n)$.

- Output: a reordering of $A$ such that the resulting list is monotonically increasing.

# An algorithm

allocate an array $C[1 \ldots k]$; — $O(1)$ time

for each $j = 1$ to $k$
    let $C[j] = 0$; — $O(k)$ time

for each $i = 1$ to $n$
    increment $C[A[i]]$ by one; — $O(n)$ time

for each $j = 1$ to $k$
    for each $\ell = 1$ to $C[j]$
        print $j$; — $O(k) + O(n)$ time

The running time is $O(n + k) = O(n)$.

# A general setting

- Input: $n$ ~~distinct~~ positive integers $A[1], \ldots, A[n]$ with $A[i] \leq k$ for each $i = 1, 2, \ldots, n$, for some unknown number $k = O(n)$.

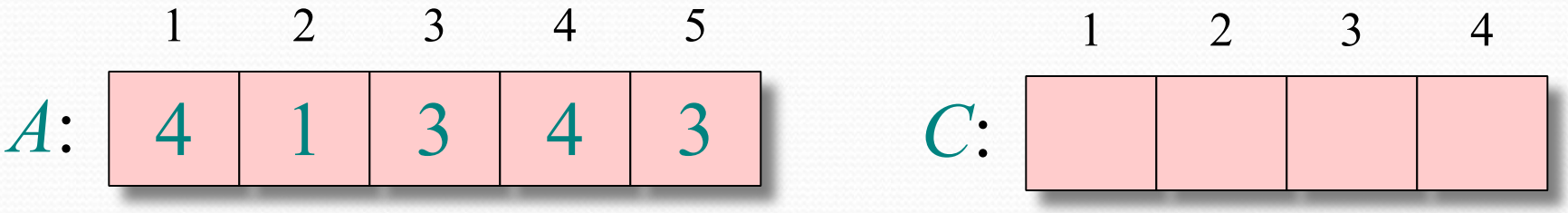
- Output: a reordering of $A$ such that the resulting list is monotonic increasing.

# An algorithm

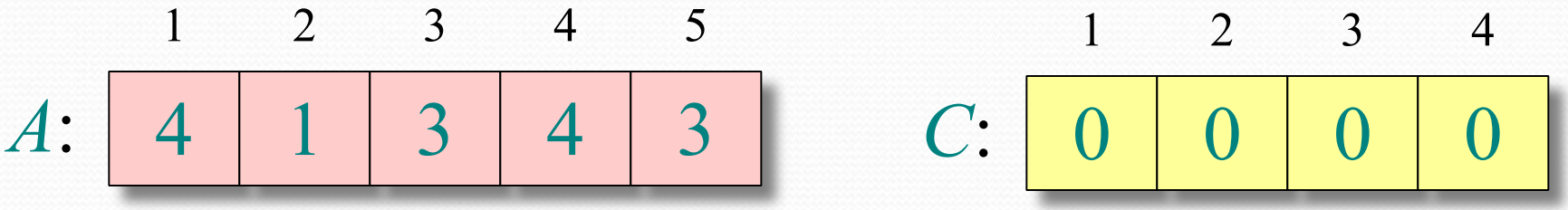
let $k$ be the maximum of $A[1], \ldots, A[n]$; $O(n)$ time
allocate an array $C[1 \ldots k]$;
for each $j = 1$ to $k$
    let $C[j] = 0$;
for each $i = 1$ to $n$
    increment $C[A[i]]$ by one;
for each $j = 1$ to $k$
    for each $\ell = 1$ to $C[j]$
        print $j$;

The running time is $O(n + k) = O(n)$.

# Counting-sort example



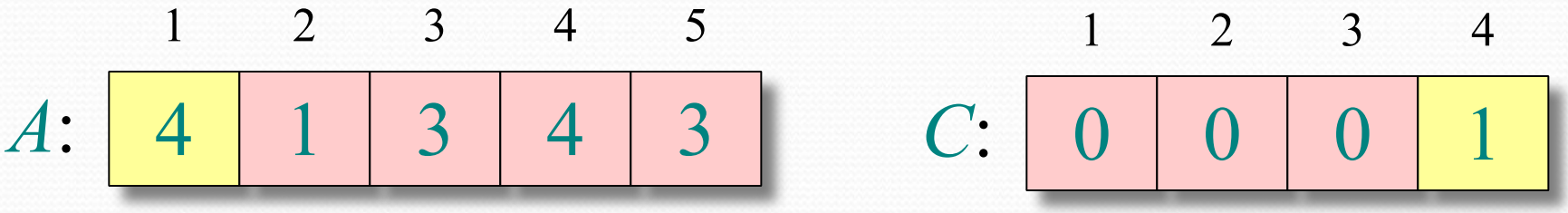
# Loop 1




**for** $j \leftarrow 1$ **to** $k$

**do** $C[j] \leftarrow 0$

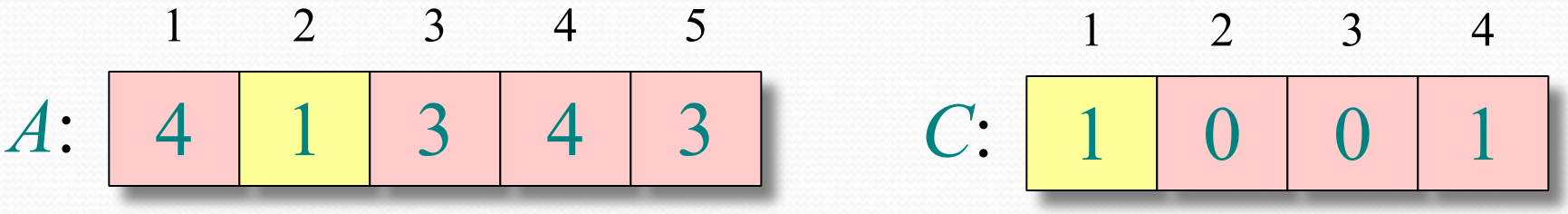
# Loop 2




**for** $i \leftarrow 1$ **to** $n$

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

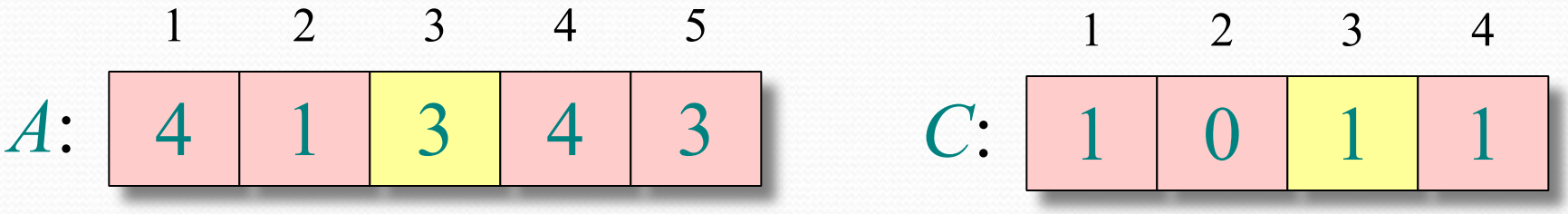
# Loop 2




**for** $i \leftarrow 1$ **to** $n$

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

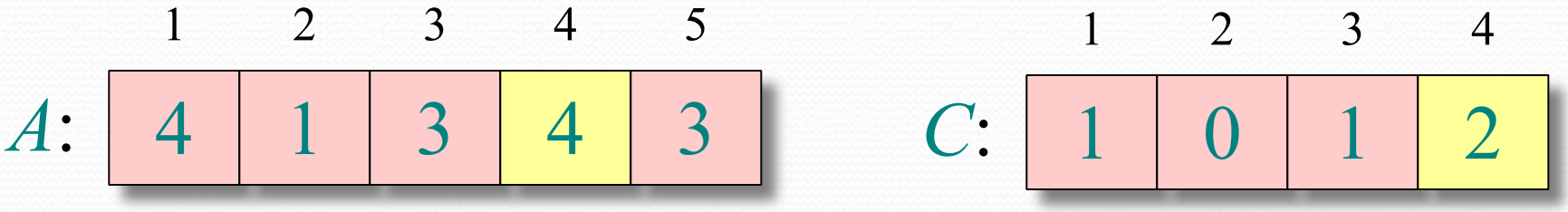
# Loop 2




**for** $i \leftarrow 1$ **to** $n$

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

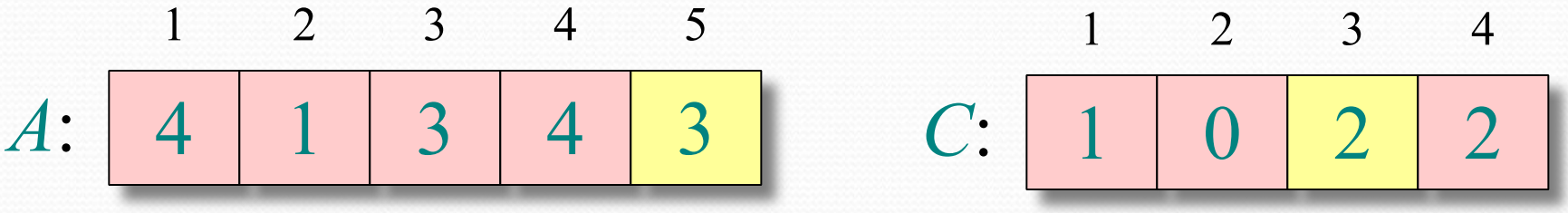
# Loop 2




**for** $i \leftarrow 1$ **to** $n$

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

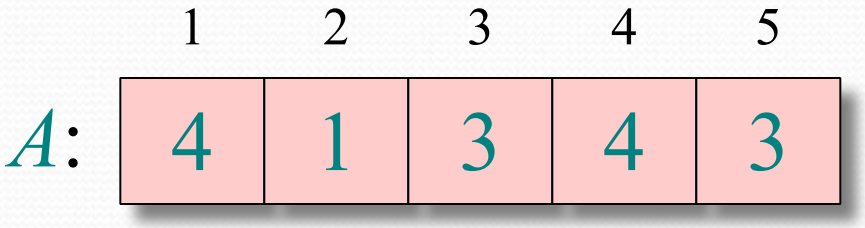
# Loop 2

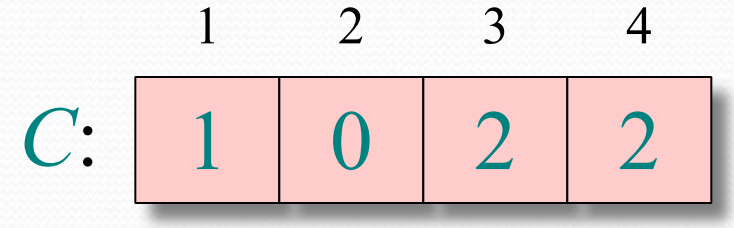



**For** i ← 1 **to** $n$

**do** $C[A[\,\mathrm{i}]] \leftarrow C[A[\,\mathrm{i}]] + 1$ ▷ $C[i] = |\{\mathrm{key} = i\}|$

# Loop 3




```
for each j = 1 to k
    for each ℓ = 1 to C[j]
        print j;
```

1
1
1, 3
1, 3, 3
1, 3, 3, 4
1, 3, 3, 4, 4

# Stable Sort

A sorting algorithm is **stable** if

- Numbers with the same value appear in the output array in the same order as they do in the input array
- Ties between two numbers are broken by the rule that which ever number appears first in the input array appears first in the output array

# Stable Sort


Unstable Sort
Amitab 178
Salman 115
Amir 126
Shahrukh 115
Akshay 126
Abhishek 35
Abhisek 35
Shahrukh 115
Salman 115
Amir 126
Akshay 126
Amitab 178
Abhisek 35
Salman 115
Shahrukh 115
Amir 126
Akshay 126
Amitab 178
Stable Sort

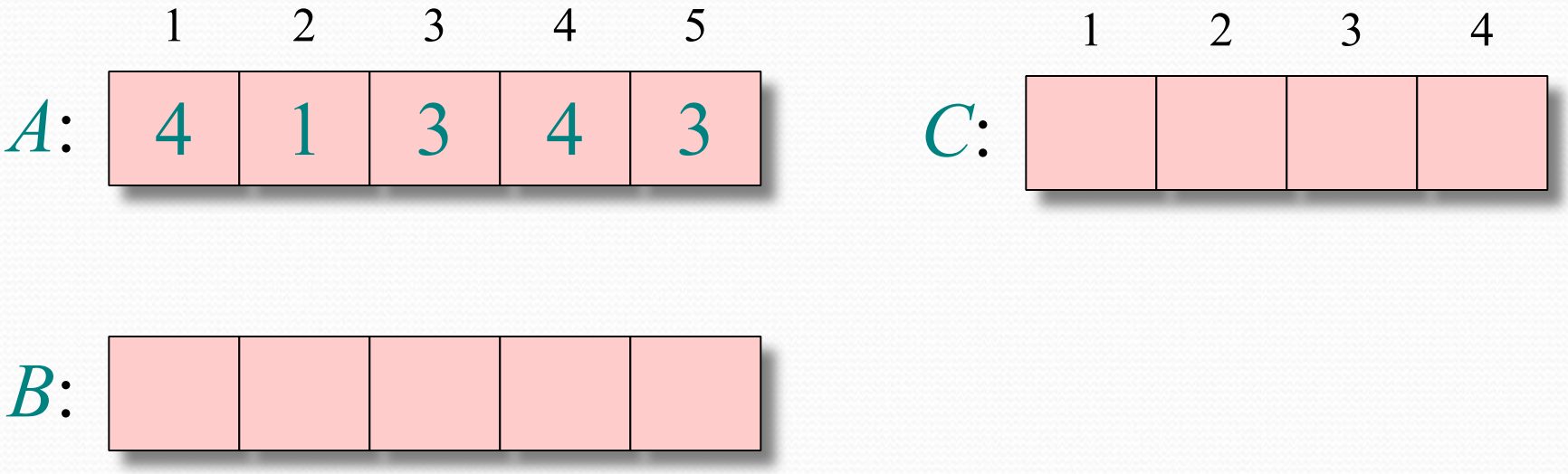
# Counting Sort (stable) Overview

- Assumption: n input elements are integers in the range of 0 to k (integer).
  - $\Theta(n+k)$ ➔ When $k=O(n)$, counting sort: $\Theta(n)$
- Basic Idea
  - For each input element x, determine the number of elements less than or equal to x
  - For each integer i ($0 \le i \le k$), count how many elements whose values are i
    - Then we know how many elements are less than or equal to I
- Algorithm storage
  - A[1..n]: input elements
  - B[1..n]: sorted elements
  - C[1..k]: hold the number of elements less than or equal to i

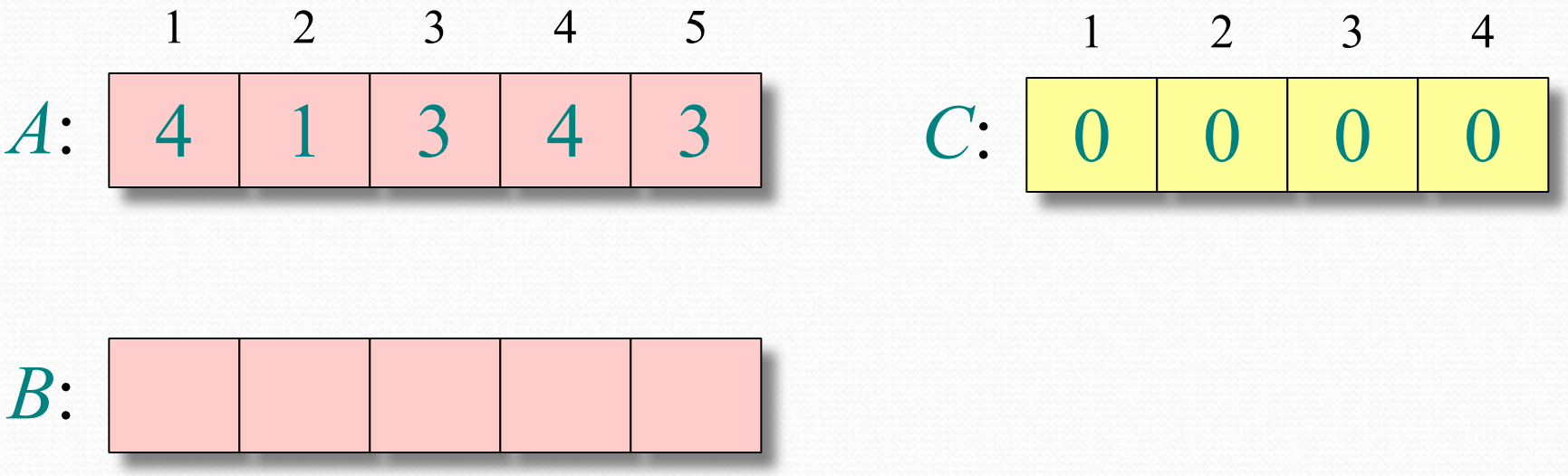
# Counting Sort

```
for i ← 1 to k
    do C[i] ← 0

for j ← 1 to n
    do C[A[j]] ← C[A[j]] + 1        ▷ C[i] = |{key = i}|

for i ← 2 to k
    do C[i] ← C[i] + C[i–1]         ▷ C[i] = |{key ≤ i}|

for j ← n downto 1
    do  B[C[A[j]]] ← A[j]
        C[A[j]] ← C[A[j]] – 1
```

# Counting-sort example



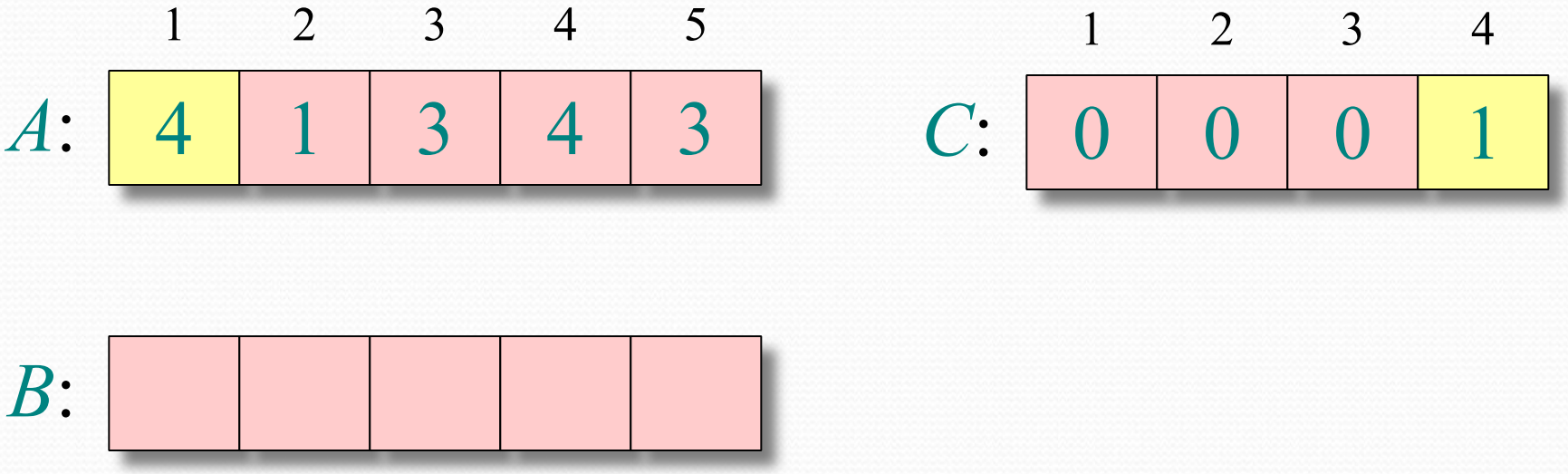
# Loop 1




**for** $i \leftarrow 1$ **to** $k$

**do** $C[i] \leftarrow 0$

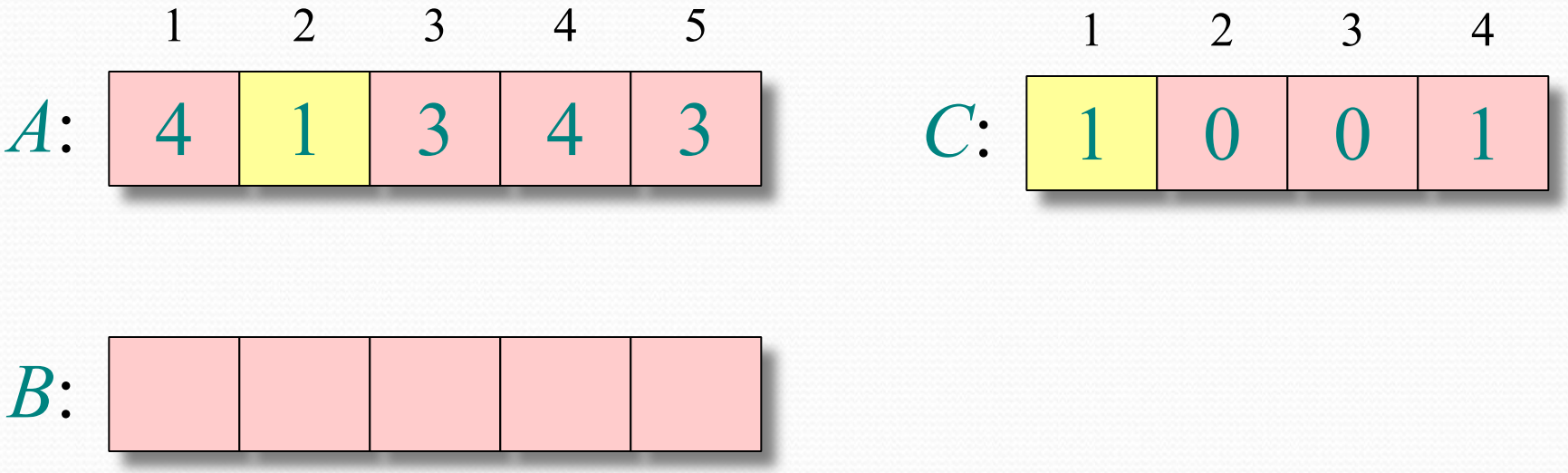
# Loop 2




**for** $j \leftarrow 1$ **to** $n$

**do** $C[A[j]] \leftarrow C[A[j]] + 1$ ▻ $C[i] = |\{\text{key} = i\}|$

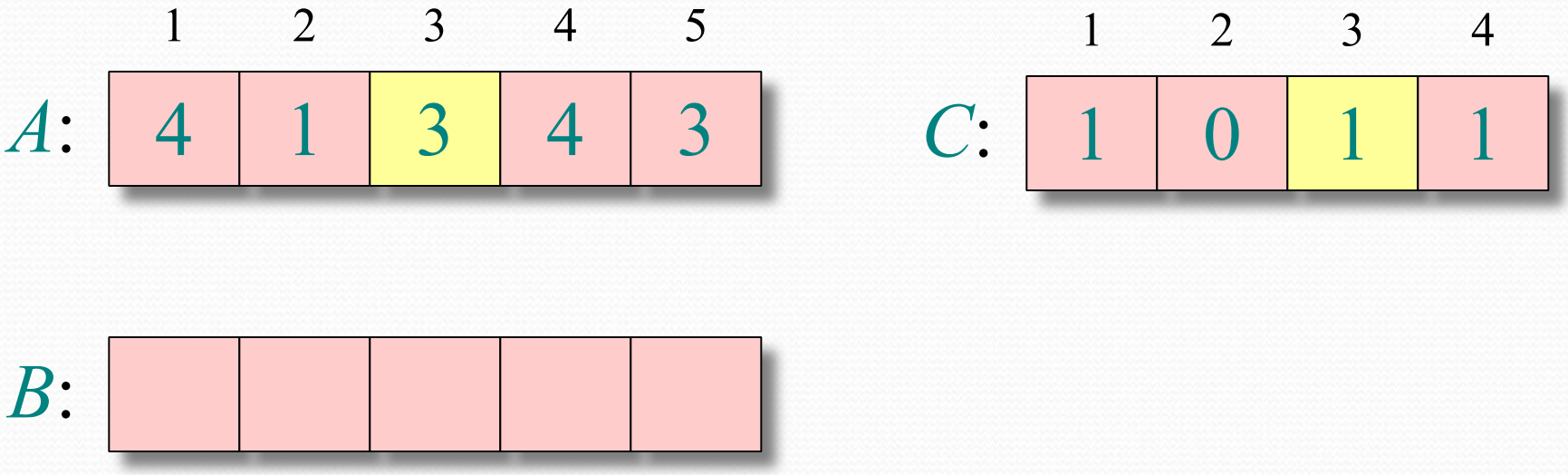
# Loop 2




**for** $j \leftarrow 1$ **to** $n$

**do** $C[A[j]] \leftarrow C[A[j]] + 1$ ▻ $C[i] = |\{\text{key} = i\}|$

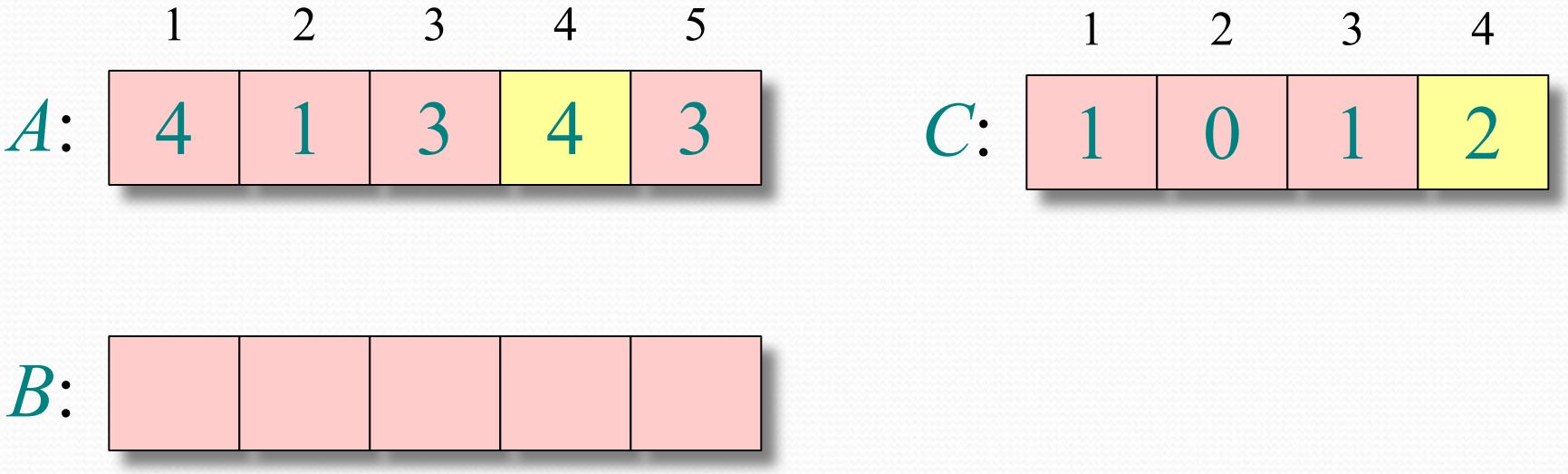
# Loop 2




**for** $j \leftarrow 1$ **to** $n$

**do** $C[A[j]] \leftarrow C[A[j]] + 1$ ▻ $C[i] = |\{\text{key} = i\}|$

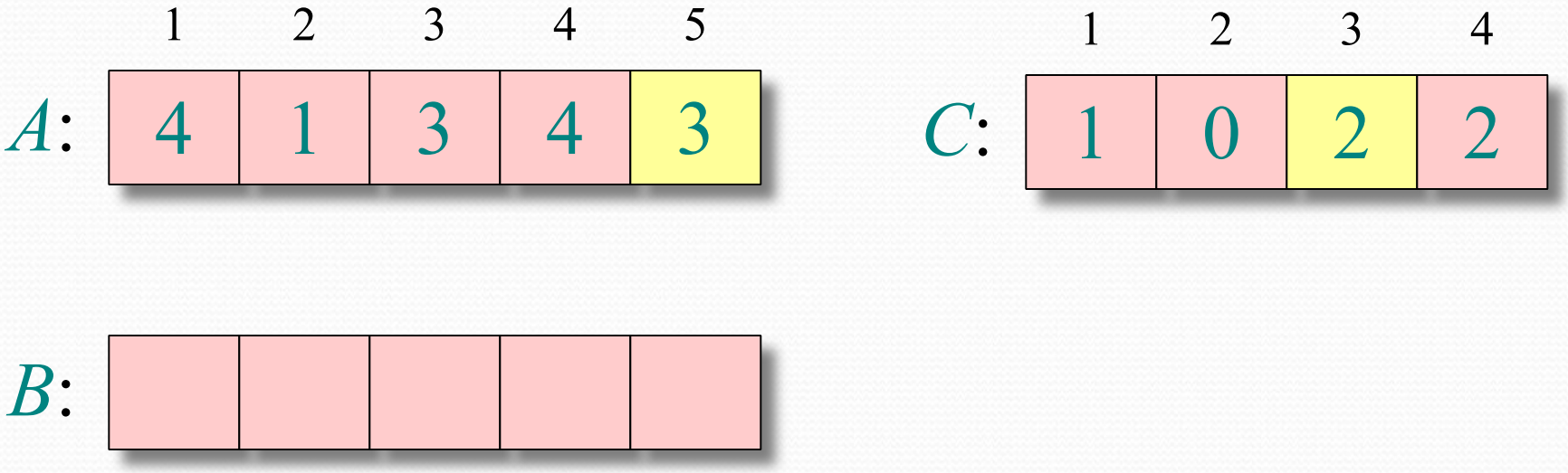
# Loop 2




**for** $j \leftarrow 1$ **to** $n$

**do** $C[A[j]] \leftarrow C[A[j]] + 1$ ▻ $C[i] = |\{\text{key} = i\}|$

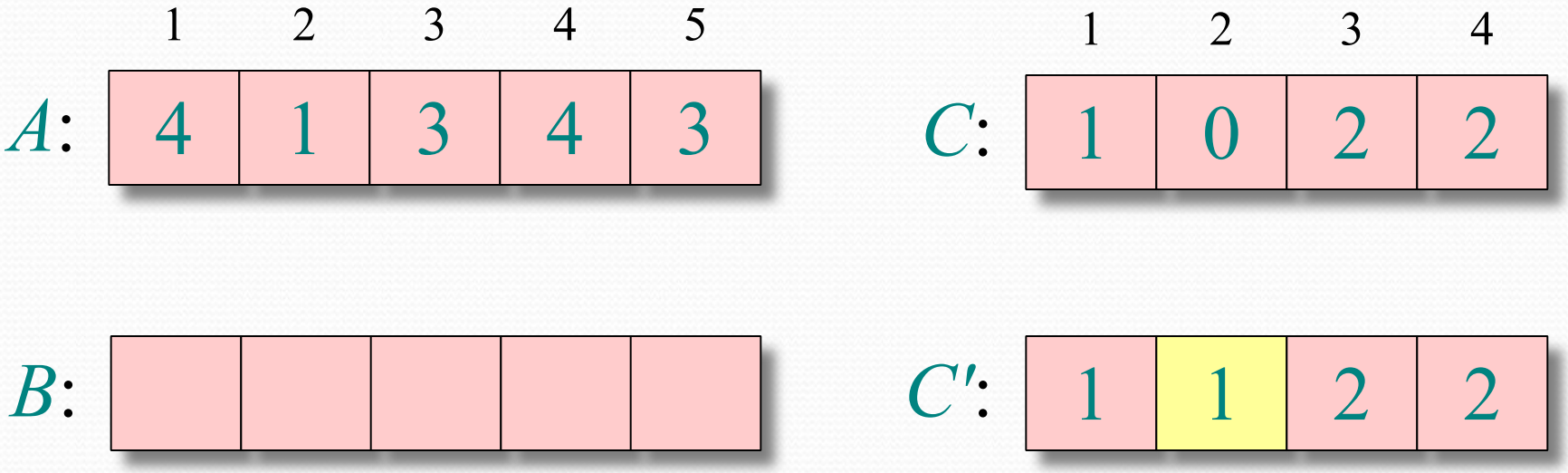
# Loop 2




**for** $j \leftarrow 1$ **to** $n$

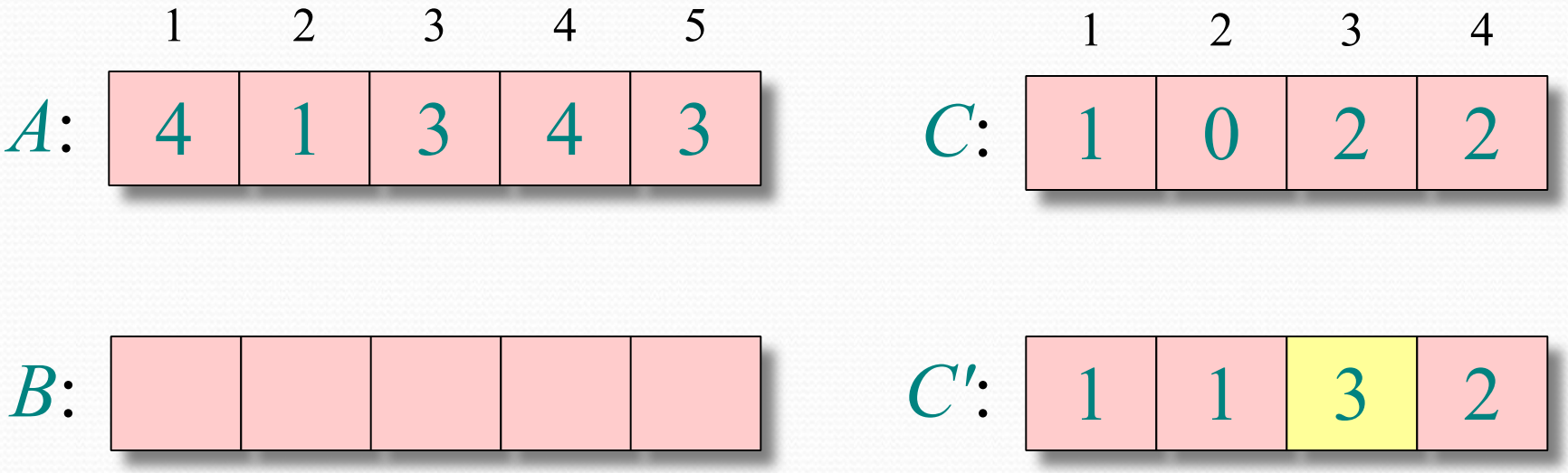
**do** $C[A[j]] \leftarrow C[A[j]] + 1$ ▷ $C[i] = |\{\text{key} = i\}|$

# Loop 3




**for** $i \leftarrow 2$ **to** $k$
  **do** $C[i] \leftarrow C[i] + C[i-1]$ ▷ $C[i] = |\{\text{key} \le i\}|$

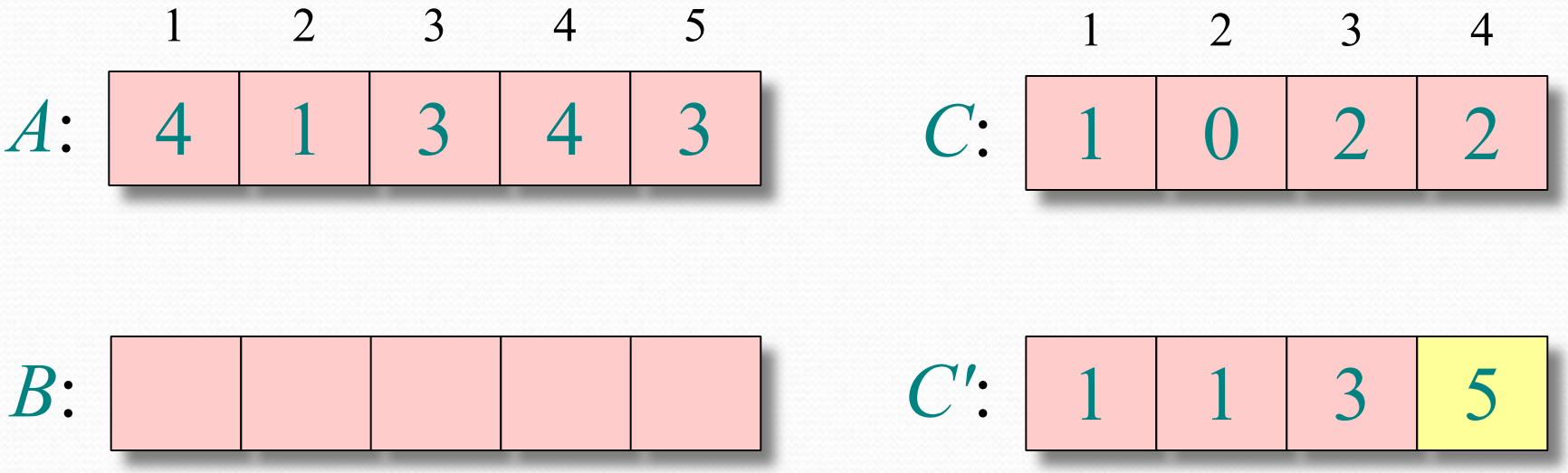
# Loop 3




**for** $i \leftarrow 2$ **to** $k$

**do** $C[i] \leftarrow C[i] + C[i-1]$ ▷ $C[i] = |\{\text{key} \le i\}|$

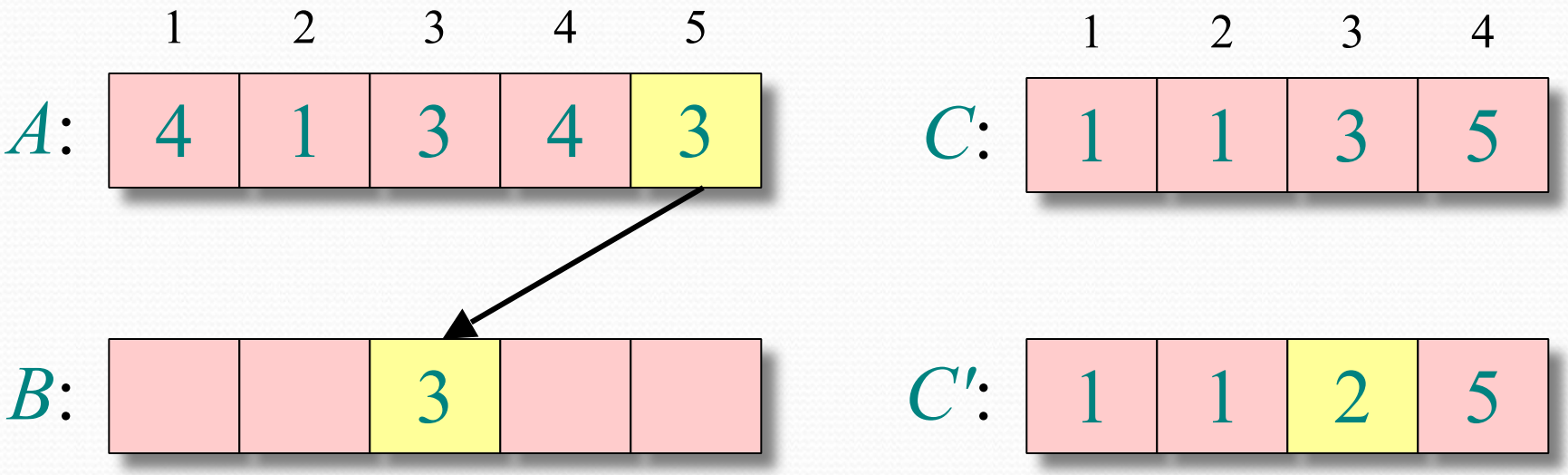
# Loop 3




**for** $i \leftarrow 2$ **to** $k$

**do** $C[i] \leftarrow C[i] + C[i-1]$ ▷ $C[i] = |\{\text{key} \le i\}|$

# Loop 4

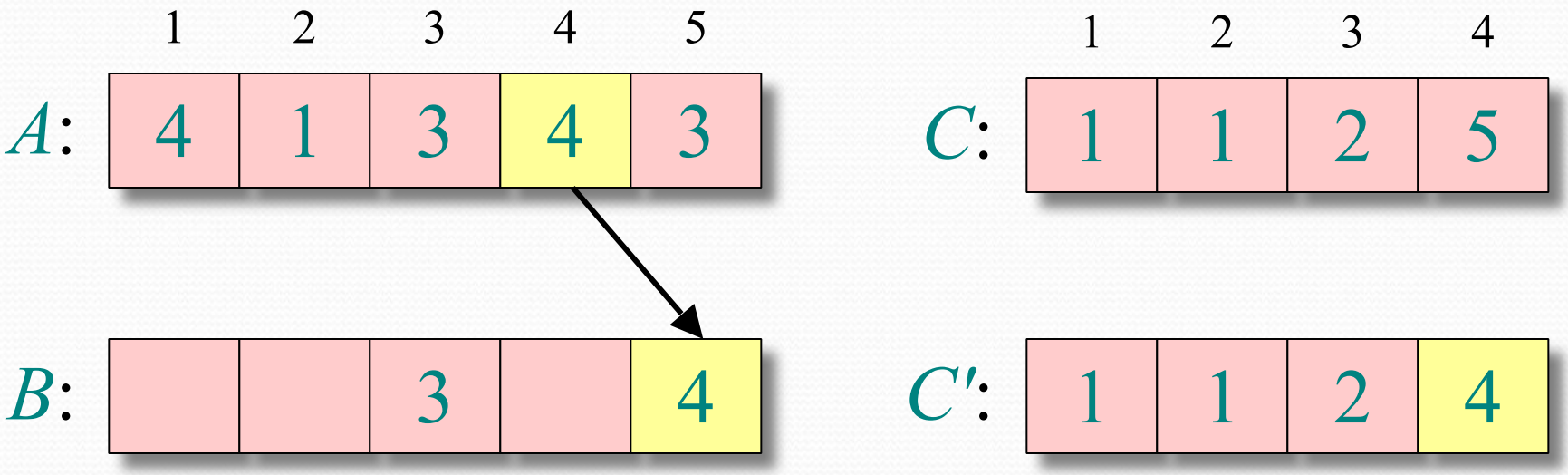



**for** $j \leftarrow n$ **downto** 1

**do** $B[C[A[j]]] \leftarrow A[j]$

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

# Loop 4

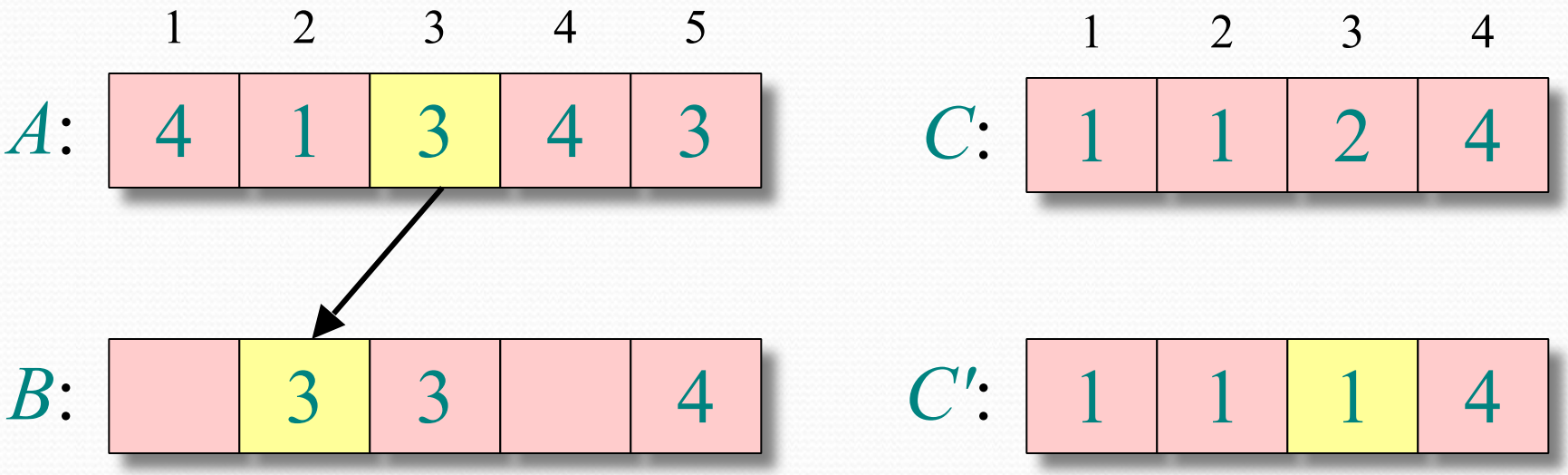



**for** $j \leftarrow n$ **downto** 1

**do** $B[C[A[j]]] \leftarrow A[j]$

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

# Loop 4

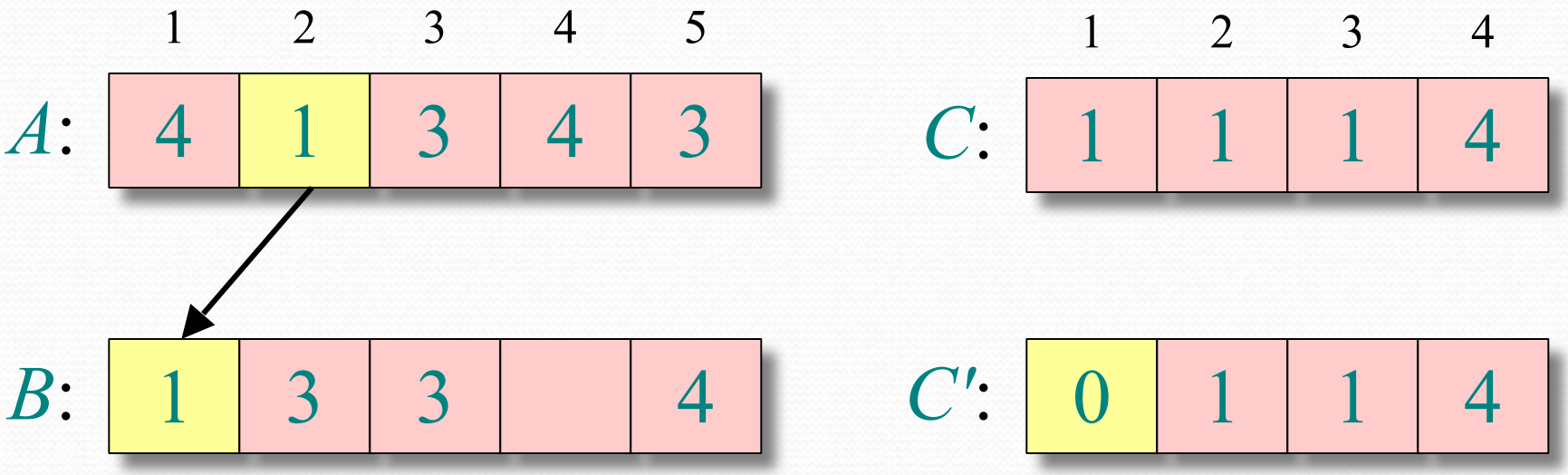



**for** $j \leftarrow n$ **downto** 1

**do** $B[C[A[j]]] \leftarrow A[j]$

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

# Loop 4

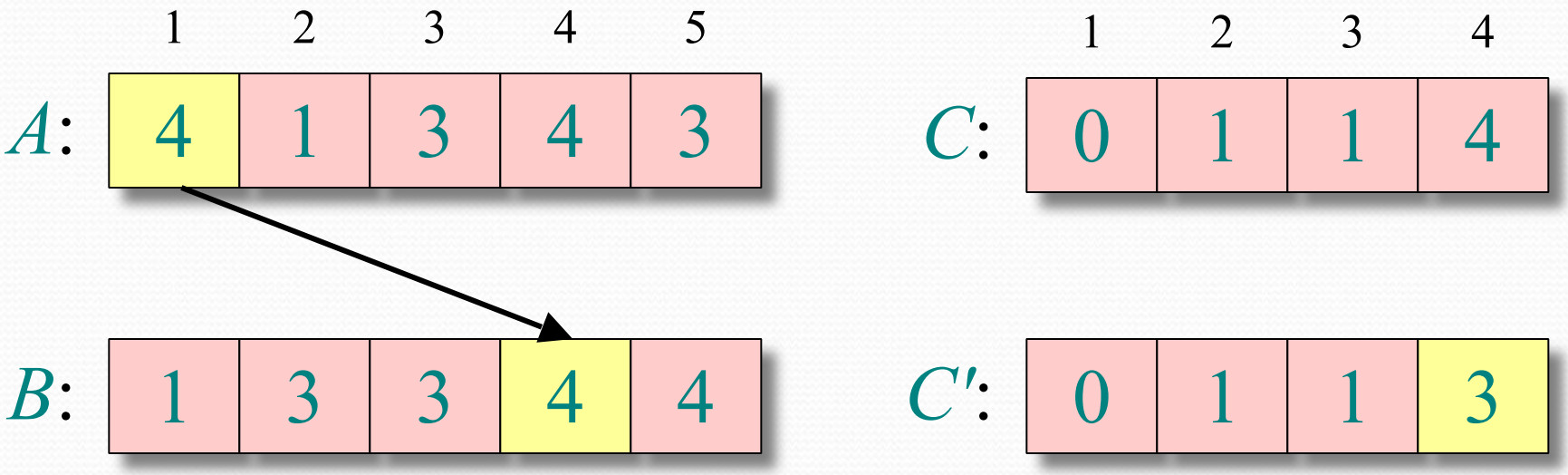



**for** $j \leftarrow n$ **downto** 1

**do** $B[C[A[j]]] \leftarrow A[j]$

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

# Loop 4




**for** $j \leftarrow n$ **downto** 1

**do** $B[C[A[j]]] \leftarrow A[j]$

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

# Analysis

$\Theta(k)$ **for** $i \leftarrow 1$ **to** $k$
**do** $C[i] \leftarrow 0$

$\Theta(n)$ **for** $j \leftarrow 1$ **to** $n$
**do** $C[A[j]] \leftarrow C[A[j]] + 1$

$\Theta(k)$ **for** $i \leftarrow 2$ **to** $k$
**do** $C[i] \leftarrow C[i] + C[i-1]$

$\Theta(n)$ **for** $j \leftarrow n$ **downto** 1
**do** $B[C[A[j]]] \leftarrow \mathrm{A}[j]$
$C[A[j]] \leftarrow C[A[j]] - 1$

$\Theta(n + k)$

# Bucket-Sort

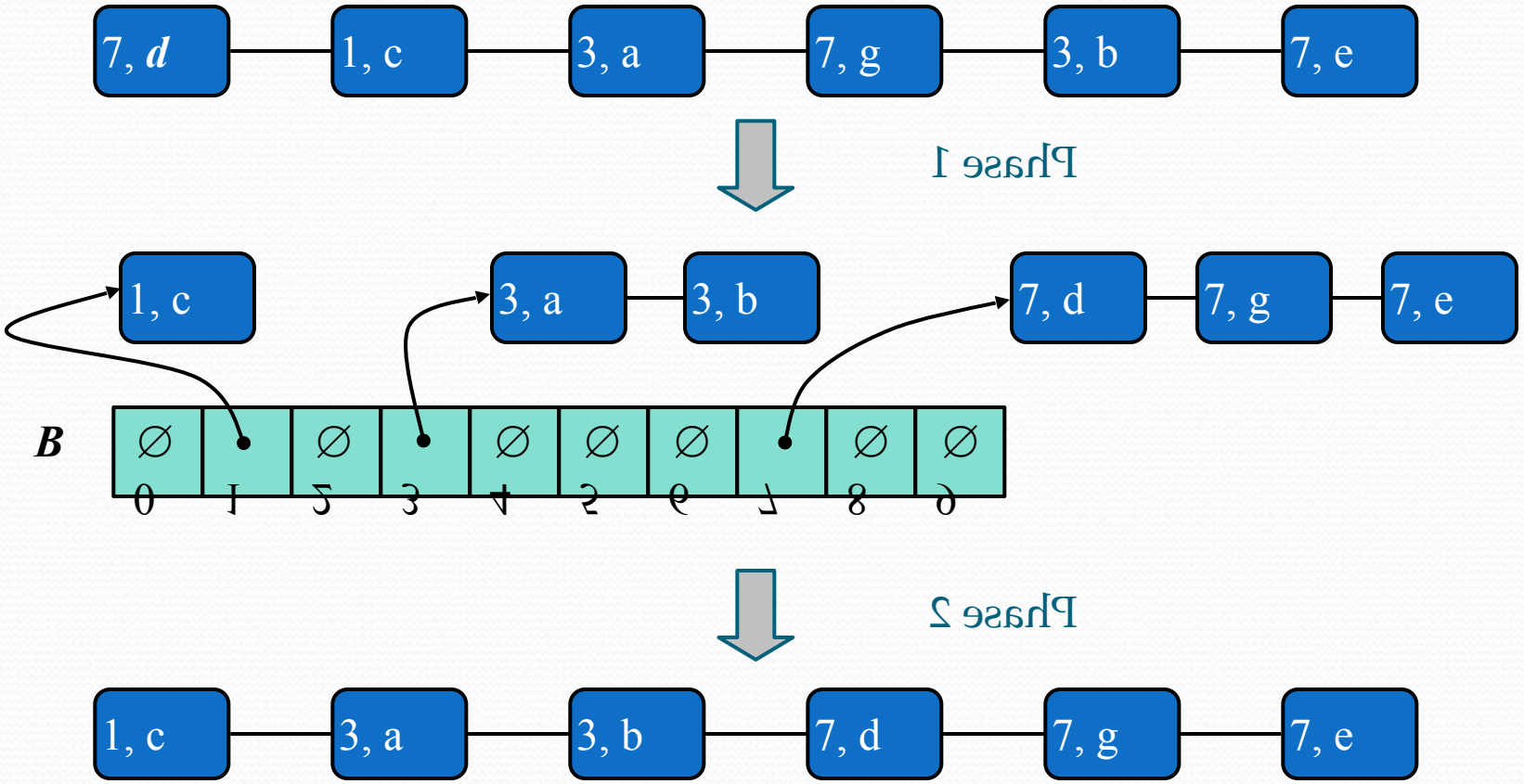
- Let be $S$ be a sequence of $n$ (key, element) entries with keys in the range $[0, N - 1]$
- Bucket-sort uses the keys as indices into an auxiliary array $B$ of sequences (buckets)
  - Phase 1: Empty sequence $S$ by moving each entry $(k, o)$ into its bucket $B[k]$
  - Phase 2: For $i = 0, \ldots, N - 1$, move the entries of bucket $B[i]$ to the end of sequence $S$
- Analysis:
  - Phase 1 takes $O(n)$ time
  - Phase 2 takes $O(n + N)$ time

Bucket-sort takes $O(n + N)$ time

```
Algorithm bucketSort(S, N)
    Input sequence S of (key, element)
        items with keys in the range
        [0, N – 1]
    Output sequence S sorted by
        increasing keys
    B ← array of N empty sequences
    while ¬S.isEmpty()
        f ← S.first()
        (k, o) ← S.remove(f)
        B[k].insertLast((k, o))
    for i ← 0 to N – 1
        while ¬B[i].isEmpty()
            f ← B[i].first()
            (k, o) ← B[i].remove(f)
            S.insertLast((k, o))
```

# Example

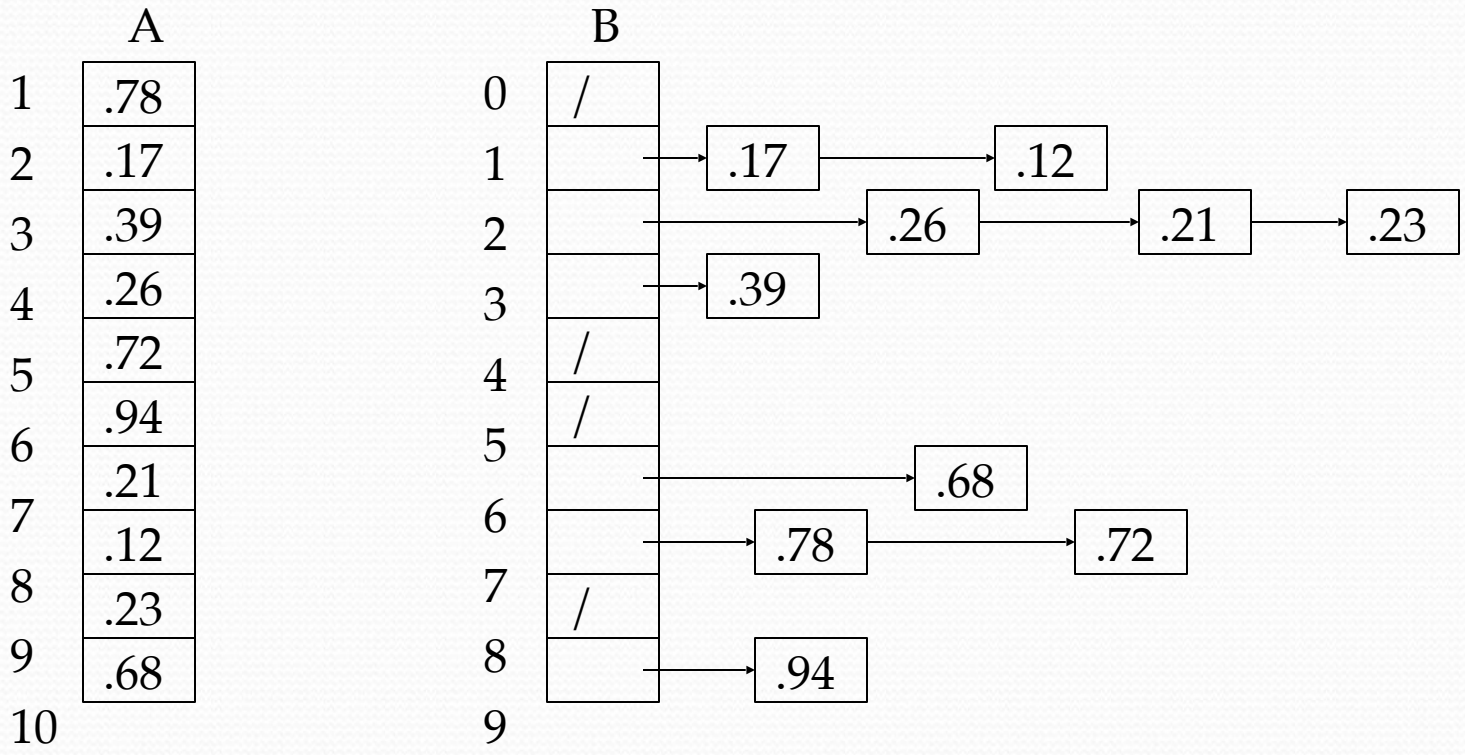
Key range [0, 9]



# Bucket Sort Illustration

- Bucket sort runs in linear time when the input is drawn from a uniform distribution over the interval [0, 1)
- Basic idea
  - Divide [0,1) into n equal-sized subintervals (buckets)
  - Distribute the n numbers into the buckets
  - Sort the numbers in each bucket
  - Go through the buckets in order, list the elements in each

# Bucket Sort Illustration



# Bucket Sort Algorithm

```
BUCKET-SORT(A)
1  n ← length[A]
2  for i ← 1 to n
3        do insert A[i] into list B[⌊nA[i]⌋]
4  for i ← 0 to n − 1
5        do sort list B[i] with insertion sort
6  concatenate the lists B[0], B[1], . . . , B[n − 1] together in order
```

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2) .$$

# Bucket Sort Algorithm – Average Case Analysis

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2) .$$

Taking expectations both sides and using linearity of expressions

$$\begin{aligned} \mathrm{E}[T(n)] &= \mathrm{E}\left[\Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)\right] \\ &= \Theta(n) + \sum_{i=0}^{n-1} \mathrm{E}[O(n_i^2)] \quad \text{(by linearity of expectation)} \\ &= \Theta(n) + \sum_{i=0}^{n-1} O\left(\mathrm{E}[n_i^2]\right) \end{aligned}$$

Define an indicator random variable, $X_{i,j}$ as follows:

$X_{i,j}$ = 1 if A[j] falls in bucket i

= 0 otherwise

$$n_i = \sum_{j=1}^{n} X_{ij} .$$

# Bucket Sort Algorithm – Average Case Analysis

To compute $E[n_i^2]$, we expand the square and regroup terms:

$$
\begin{aligned}
E[n_i^2] &= E\left[\left(\sum_{j=1}^{n} X_{ij}\right)^2\right] \\
&= E\left[\sum_{j=1}^{n}\sum_{k=1}^{n} X_{ij}X_{ik}\right] \\
&= E\left[\sum_{j=1}^{n} X_{ij}^2 + \sum_{1\le j\le n}\ \sum_{\substack{1\le k\le n \\ k\ne j}} X_{ij}X_{ik}\right] \\
&= \sum_{j=1}^{n} E[X_{ij}^2] + \sum_{1\le j\le n}\ \sum_{\substack{1\le k\le n \\ k\ne j}} E[X_{ij}X_{ik}] ,
\end{aligned}
$$

# Bucket Sort Algorithm – Average Case Analysis

$$\begin{aligned} \mathrm{E}[X_{ij}^2] &= 1 \cdot \frac{1}{n} + 0 \cdot \left(1 - \frac{1}{n}\right) \\ &= \frac{1}{n}. \end{aligned}$$

When $k \neq j$, the variables $X_{ij}$ and $X_{ik}$ are independent, and hence

$$\begin{aligned} \mathrm{E}[X_{ij} X_{ik}] &= \mathrm{E}[X_{ij}]\,\mathrm{E}[X_{ik}] \\ &= \frac{1}{n} \cdot \frac{1}{n} \\ &= \frac{1}{n^2}. \end{aligned}$$

# Bucket Sort Algorithm – Average Case Analysis

$$\mathrm{E}[n_j^2] = \sum_{j=1}^{n} \mathrm{E}[X_{ij}^2] + \sum_{1 \le j \le n} \sum_{\substack{1 \le k \le n \\ k \ne j}} \mathrm{E}[X_{ij} X_{ik}] ,$$

$$\mathrm{E}[n_i^2] = \sum_{j=1}^{n} \frac{1}{n} + \sum_{1 \le j \le n} \sum_{\substack{1 \le k \le n \\ k \ne j}} \frac{1}{n^2}$$

$$= n \cdot \frac{1}{n} + n(n-1) \cdot \frac{1}{n^2}$$

$$= 1 + \frac{n-1}{n}$$

$$= 2 - \frac{1}{n} ,$$

$$\mathrm{E}[T(n)] = \mathrm{E}\left[\Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)\right]$$

$$= \Theta(n) + \sum_{i=0}^{n-1} O\left(\mathrm{E}[n_i^2]\right)$$

$= \Theta(n) + O(n)$ (since n (2 – 1/n) = 2n – 1 = O(n)

# Herman Hollerith (1860-1929)

The 1880 U.S. Census took almost 10 years to process.

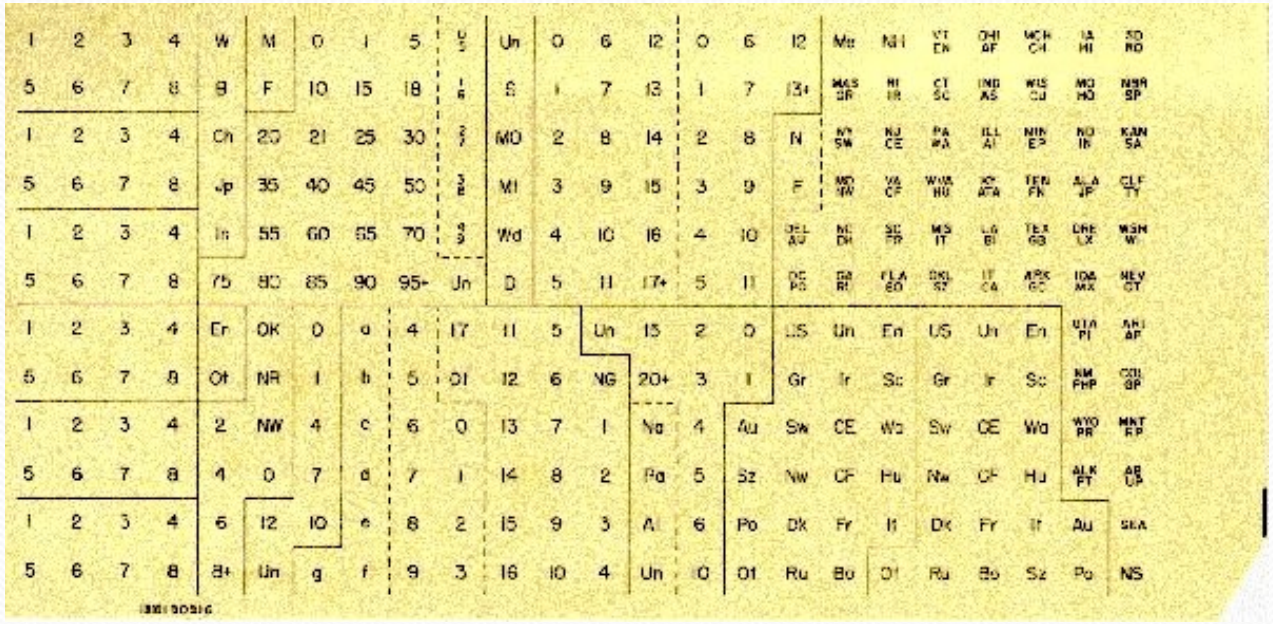
While a lecturer at MIT, Hollerith prototyped punched-card technology.

His machines, including a "card sorter," allowed the 1890 census total to be reported in 6 weeks.

He founded the Tabulating Machine Company in 1911, which merged with other companies in 1924 to form International Business Machines.

# Punched cards

- Punched card = data record.
- Hole = value.
- Algorithm = machine + human operator.



Replica of punch card from the 1900 U.S. census. [Howells 2000]

# Hollerith's tabulating system

- Pantograph card punch
- Hand-press reader
- Dial counters
- Sorting box

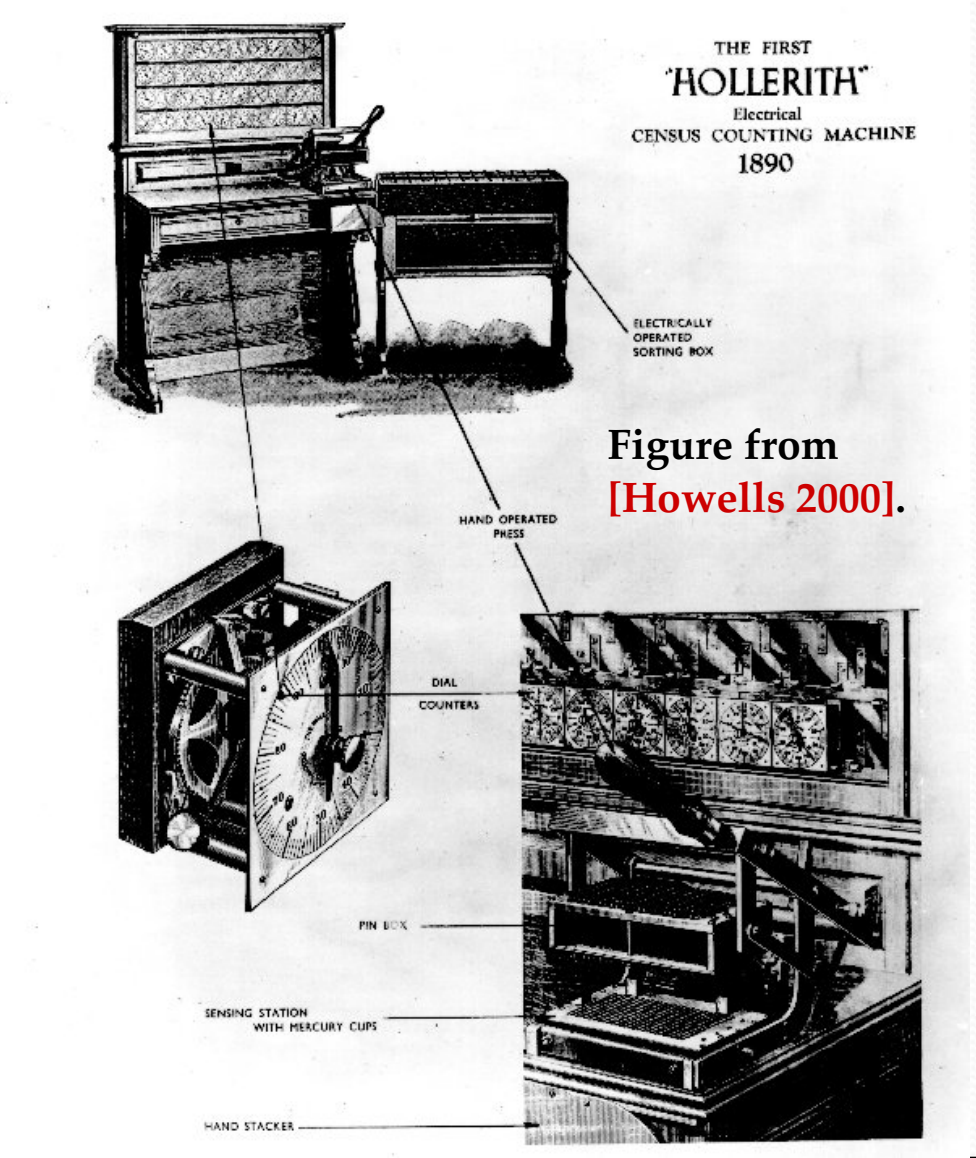



**Figure from [Howells 2000].**

# Origin of radix sort

Hollerith's original 1889 patent alludes to a most-significant-digit-first radix sort:

> *"The most complicated combinations can readily be counted with comparatively few counters or relays by first assorting the cards according to the first items entering into the combinations, then reassorting each group according to the second item entering into the combination, and so on, and finally counting on a few counters the last item of the combination for each group of cards."*

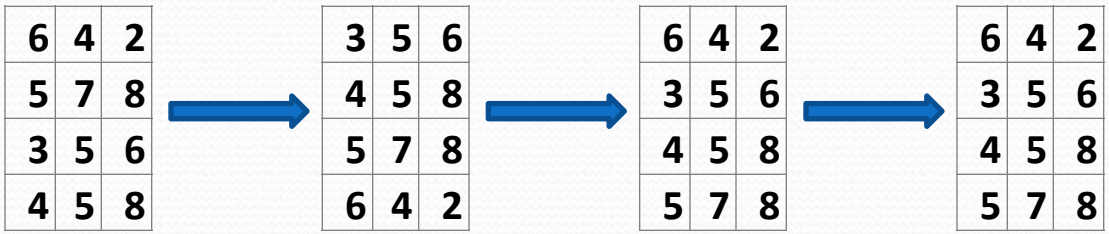
Least-significant-digit-first radix sort seems to be a folk invention originated by machine operators.

Most-significant-digit-first radix sort will not work.

Example: 642

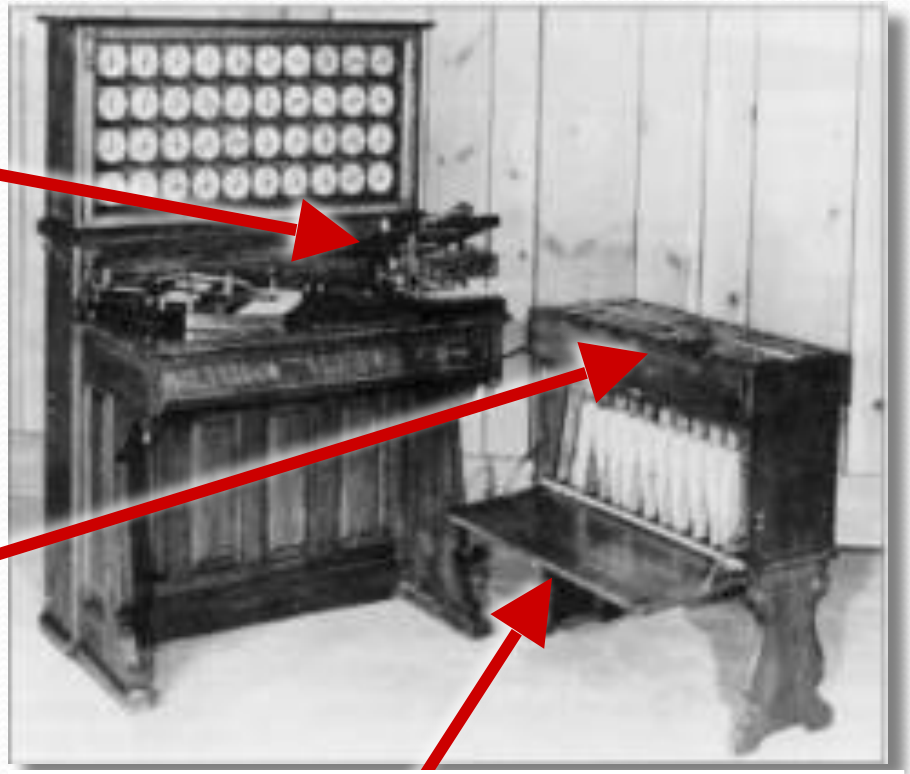
# Origin of radix sort

Most-significant-digit-first radix sort will not work.
Example: 642, 578, 356, 458



# Operation of the sorter

- An operator inserts a card into the press.
- Pins on the press reach through the punched holes to make electrical contact with mercury-filled cups beneath the card.
- Whenever a particular digit value is punched, the lid of the corresponding sorting bin lifts.



Hollerith Tabulator, Pantograph, Press, and Sorter

- The operator deposits the card into the bin and closes the lid.
- When all cards have been processed, the front panel is opened, and the cards are collected in order, yielding one pass of a stable sort.

# Radix Sort

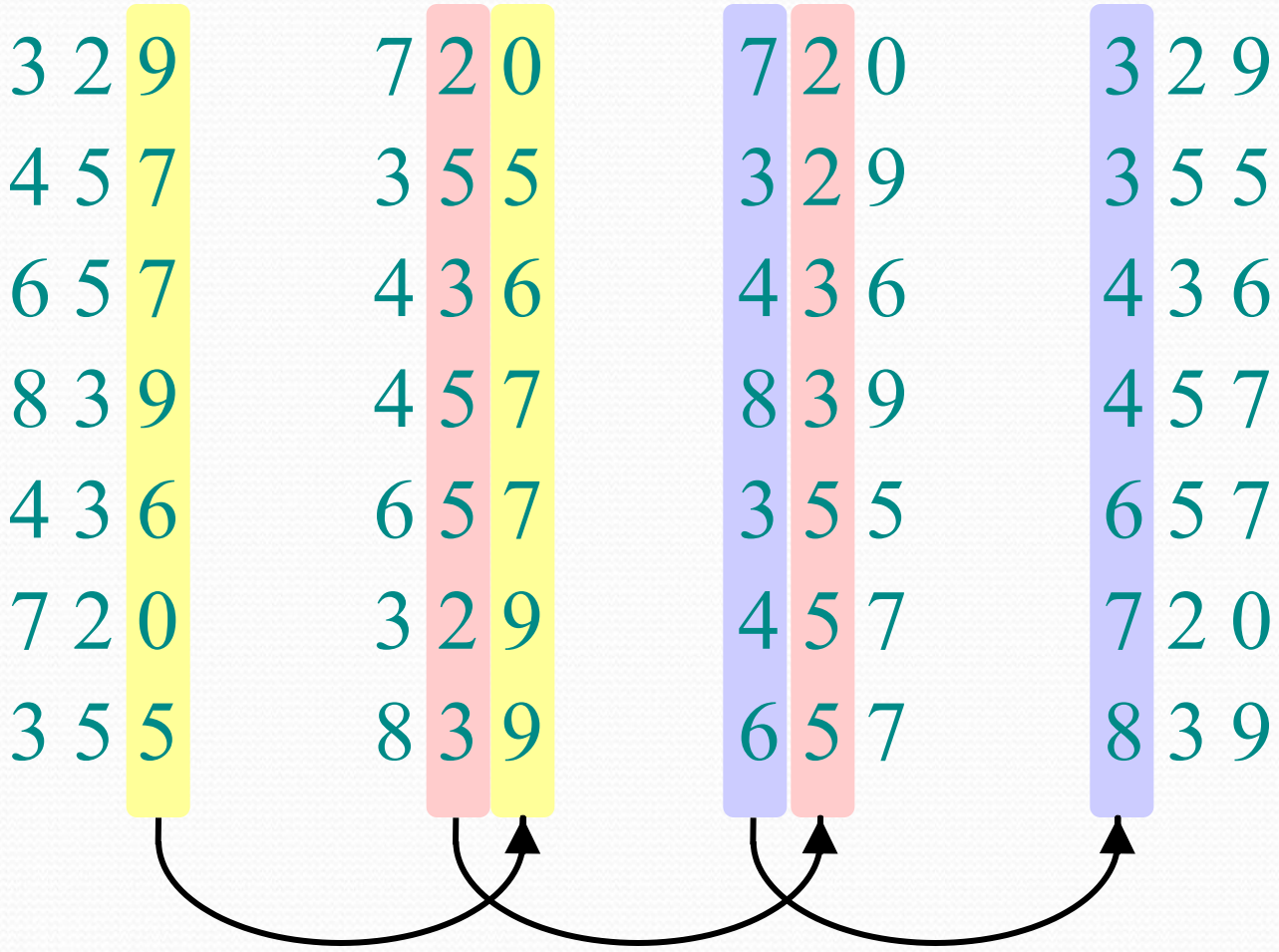
- Intuitively, you might sort on the most significant digit, then the second most significant, etc.

- Problem: lots of intermediate piles of cards to keep track of

- Key idea: sort the *least* significant digit first

```
RadixSort(A, d)
    for i=1 to d
        StableSort(A) on digit i
```

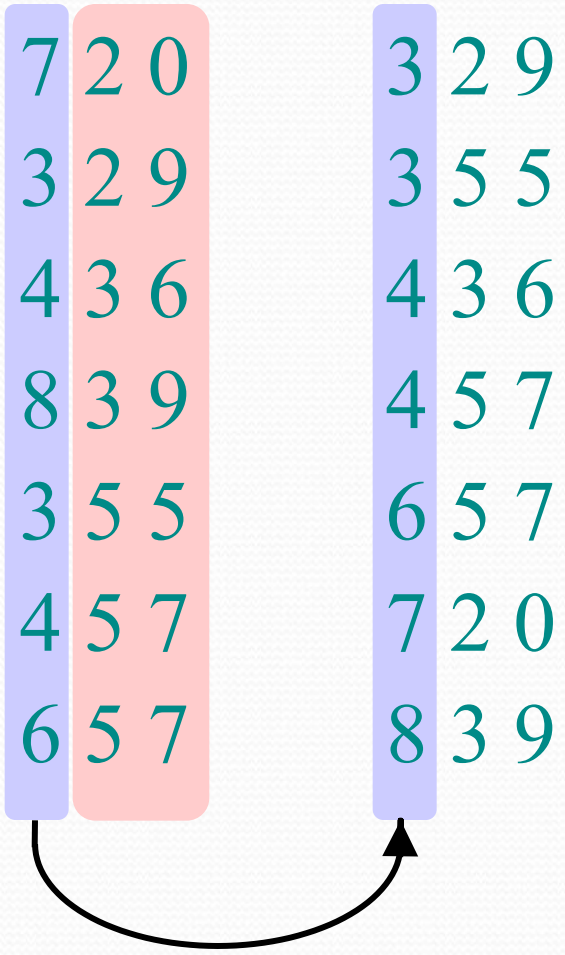
# Operation of radix sort



# Correctness of radix sort

*Induction on digit position*

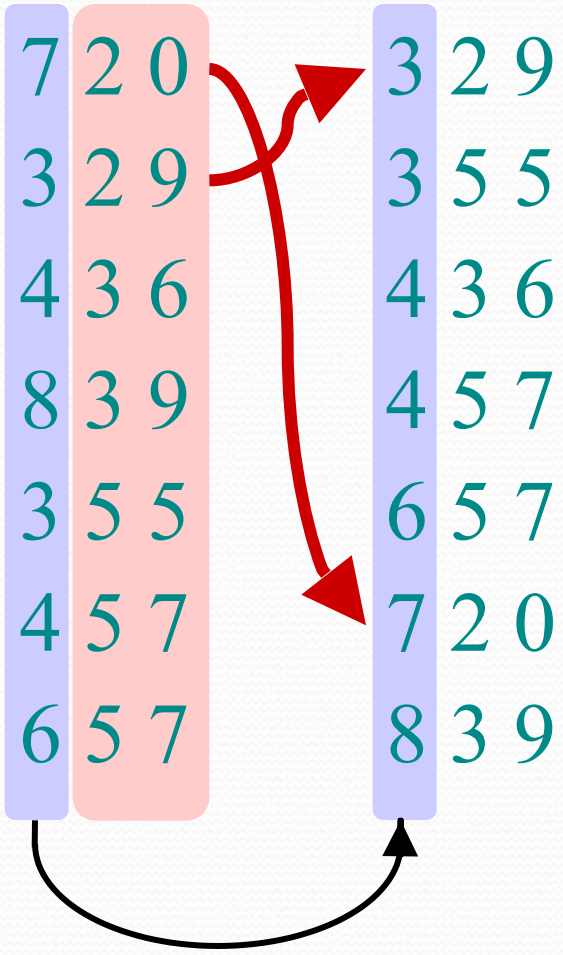
- Assume that the numbers are sorted by their low-order $t - 1$ digits.
- Sort on digit $t$



# Correctness of radix sort

*Induction on digit position*

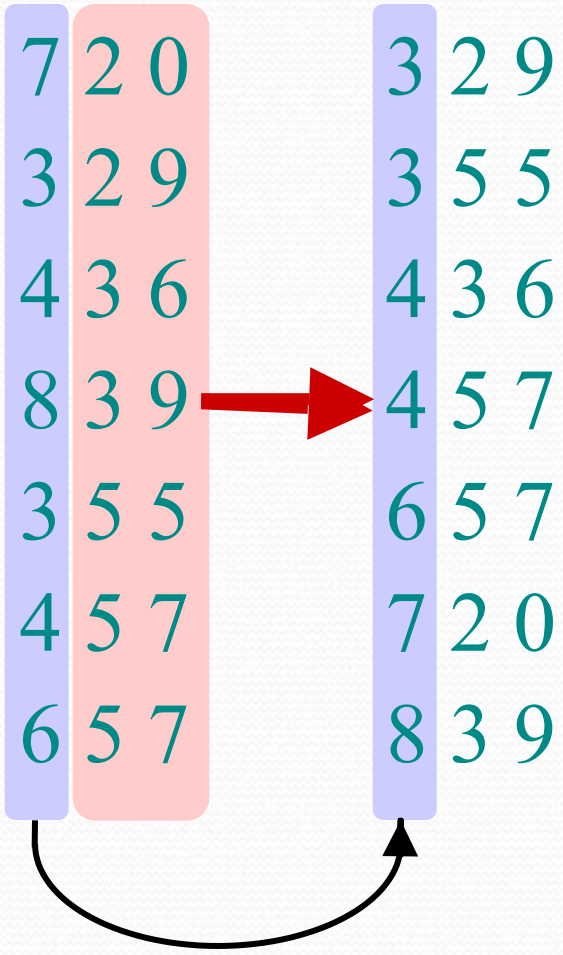
- Assume that the numbers are sorted by their low-order $t - 1$ digits.
- Sort on digit $t$
  - Two numbers that differ in digit $t$ are correctly sorted.



# Correctness of radix sort

*Induction on digit position*

- Assume that the numbers are sorted by their low-order $t - 1$ digits.
- Sort on digit $t$
  - 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.



# Radix Sort

- *What sort will we use to sort on digits?*
- Bucket sort is a good choice:
  - Sort $n$ numbers on digits that range from 1..$N$
  - Time: O($n$ + $N$)
- Each pass over $n$ numbers with $d$ digits takes time O($n$+$k$), so total time O($dn$+$dk$)
  - When $d$ is constant and $k$=O($n$), takes O($n$) time

# Summary: Radix Sort

- Radix sort:
  - Assumption: input has $d$ digits ranging from 0 to $k$
  - Basic idea:
    - Sort elements by digit starting with *least* significant
    - Use a stable sort (like bucket sort) for each stage
  - Each pass over $n$ numbers with *1* digit takes time $O(n+k)$, so total time $O(dn+dk)$
    - When $d$ is constant and $k=O(n)$, takes $O(n)$ time
  - Fast, Stable, Simple

*Thank You!!*