CH08-320201
Algorithms and Data Structures

Lecture 11/12 — 13 Mar 2018

Prof. Dr. Michael Sedlmair

Jacobs University
Spring 2018

This & that

- Don't forget: midterm next week

Last time

- Comparison sorting
  - The lower bound for comparison sorting Ω(n lgn).
- Is it possible to avoid comparisons between elements?
- Yes, if we can make assumptions on the input data.
- E.g., trivial case
  - Input: A[1…n], where A[j] ϵ {1,2,…, n}, and $a_i \neq a_j$ for all $i \neq j$
  - Output: B[1…n]

2.4 Counting Sort

Problem statement

Input: **A[1..n]**, where A[j] $\in \{1,2,\ldots,k\}$.

Output: **B[1..n]**,
which is a sorted version of A [1..n].

Auxiliary storage: **C[1..k]**.

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
```

Example: loop 1

A:					
	4	1	3	4	3

C:				
	0	0	0	0

B:					

```
for i := 1 to k
   do C[i]:=0
```

Example: loop 2

A:

4	1	3	4	3

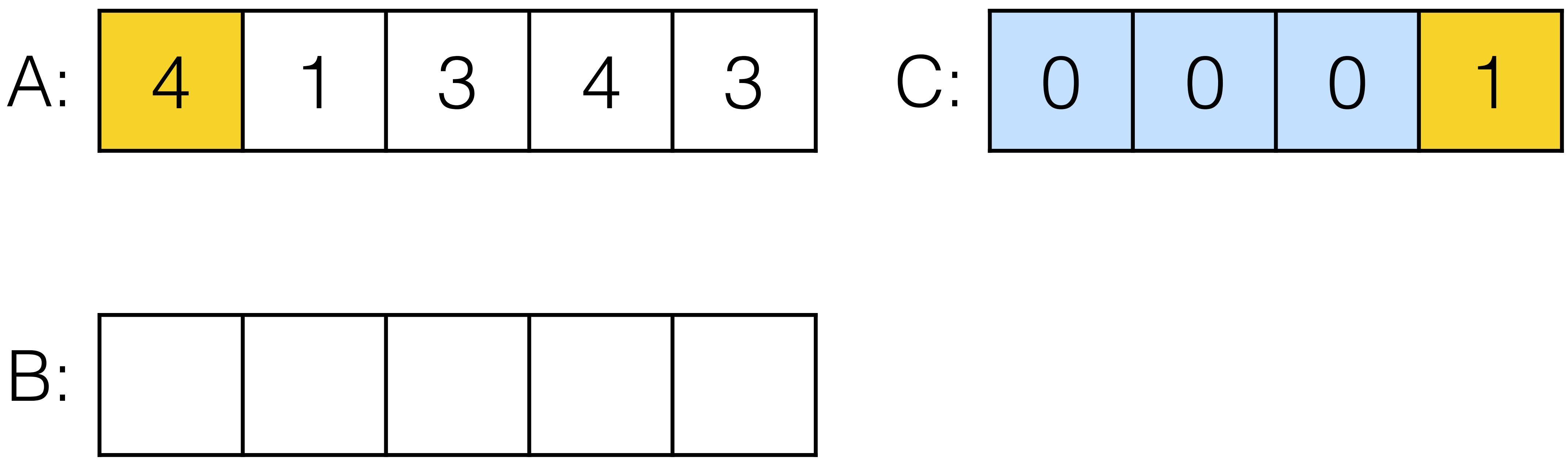
C:

0	0	0	0

B:


```
for j := 1 to n
   do C[A[j]]:=C[A[j]]+1       //C[i]=|{key=i}|
```

Example: loop 2




```
for j := 1 to n
    do C[A[j]]:=C[A[j]]+1        //C[i]=|{key=i}|
```

Example: loop 2

A:

4	1	3	4	3

C:

1	0	0	1

B:


```
for j := 1 to n
   do C[A[j]]:=C[A[j]]+1      //C[i]=|{key=i}|
```

Example: loop 2

etc. etc.

```
for j := 1 to n
   do C[A[j]]:=C[A[j]]+1      //C[i]=|{key=i}|
```

Example: loop 2

A:

4	1	3	4	3

C:

1	0	2	2

B:


```
for j := 1 to n
    do C[A[j]]:=C[A[j]]+1        //C[i]=|{key=i}|
```

Example: loop 3

A:

4	1	3	4	3

C:

1	0	2	2

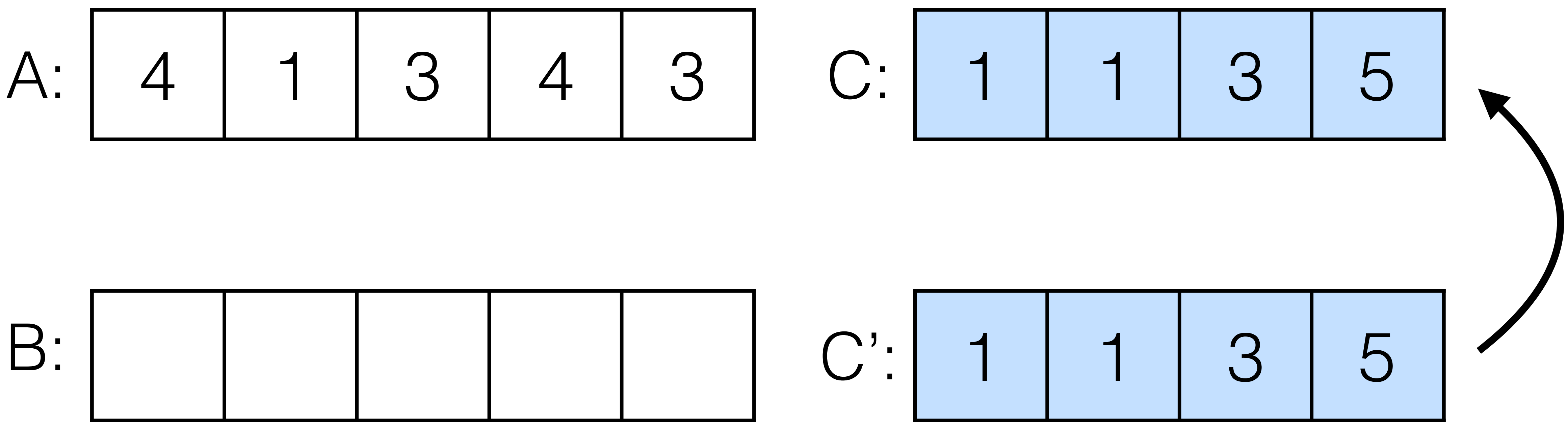
B:

C’:

1	1	3	5

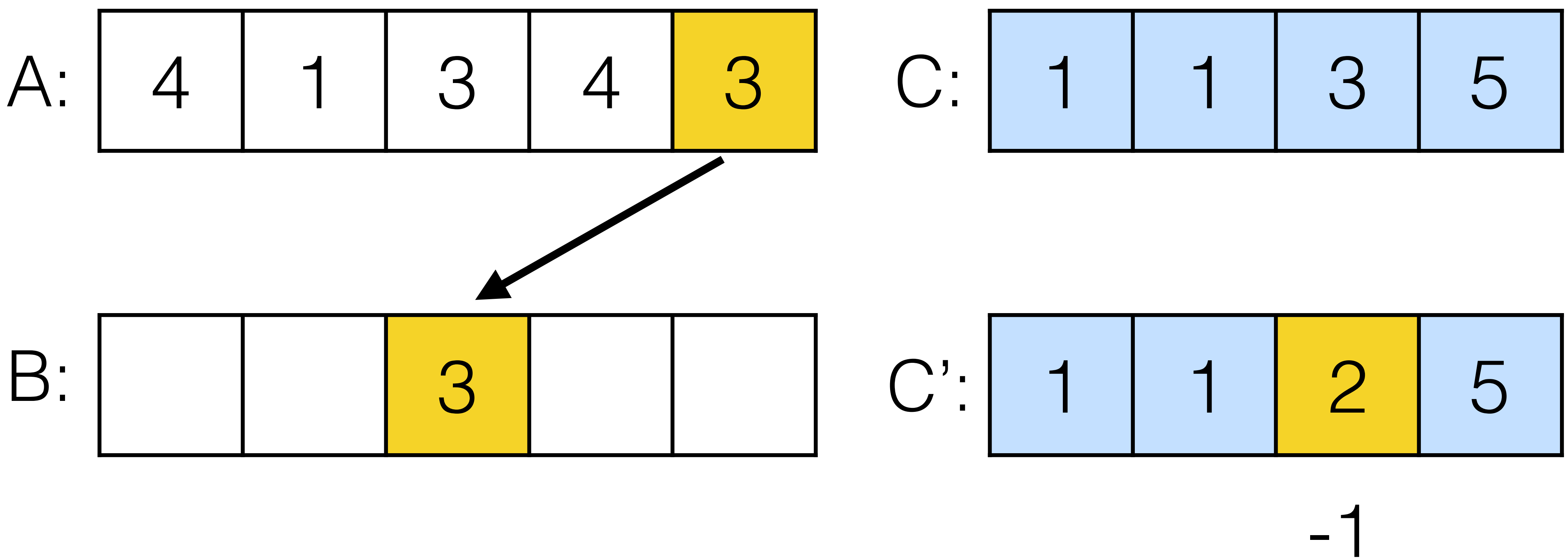
```
for i := 2 to k
   do C[i]:=C[i]+C[i–1]          //C[i]=|{key≤i}|
```

Example: loop 4



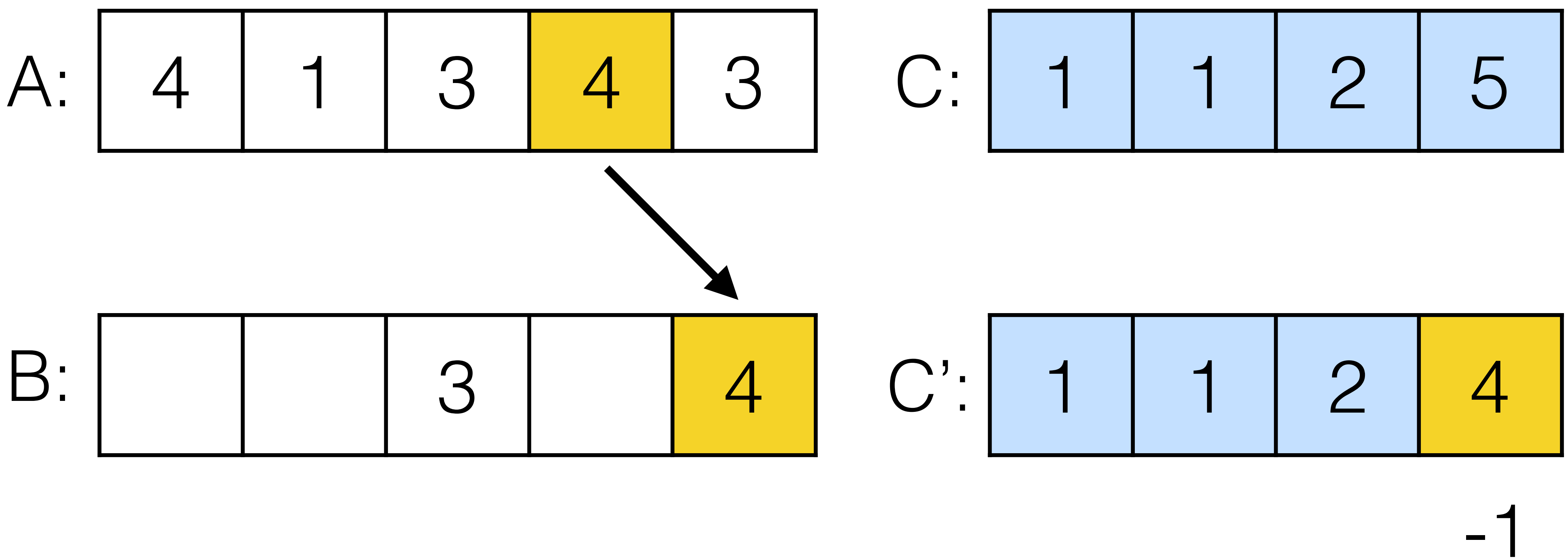

```
for j := n downto 1
   do B[C[A[j]]]←A[j]
      C[A[j]] ←C[A[j]] –1
```

Example: loop 4



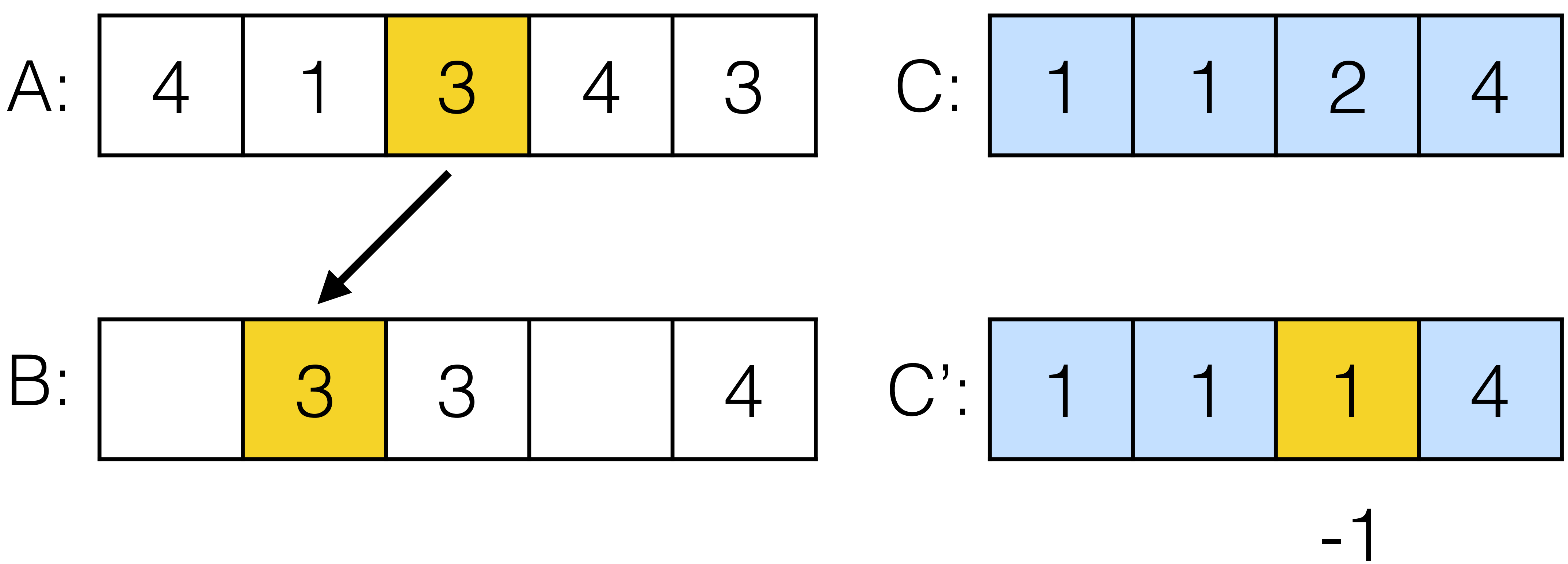

```
for j := n downto 1
    do B[C[A[j]]]←A[j]
       C[A[j]] ←C[A[j]] –1
```

Example: loop 4



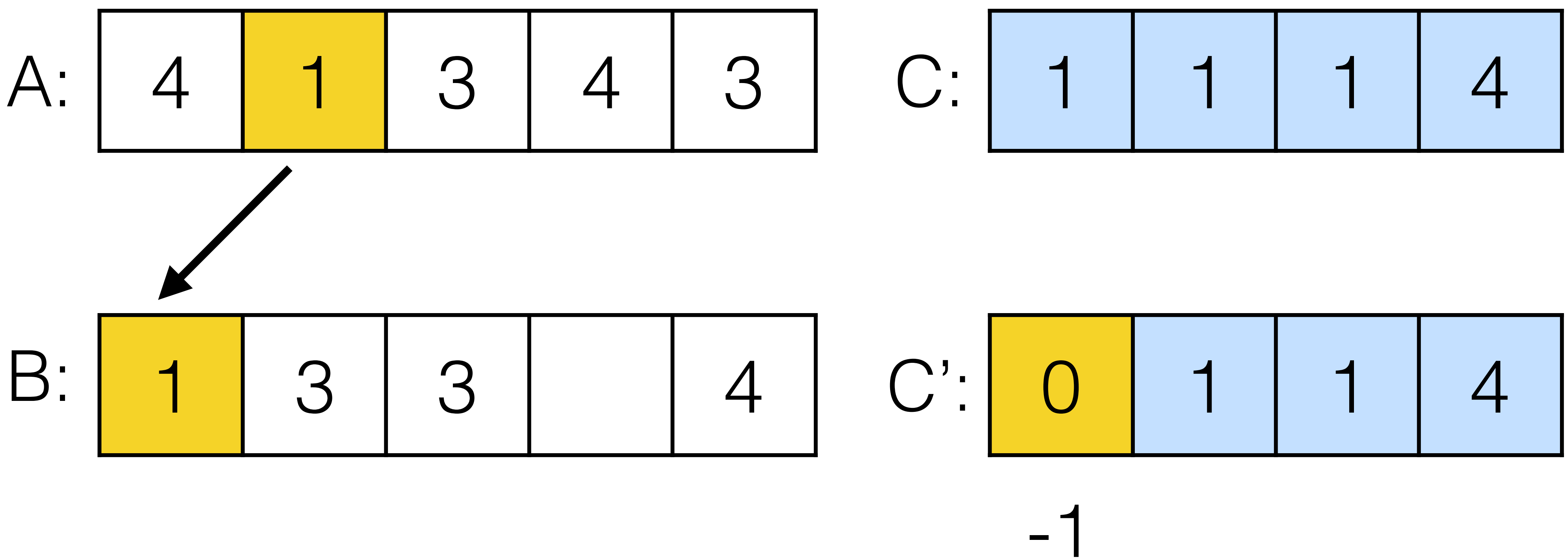

```
for j := n downto 1
    do B[C[A[j]]]←A[j]
       C[A[j]] ←C[A[j]] –1
```

Example: loop 4



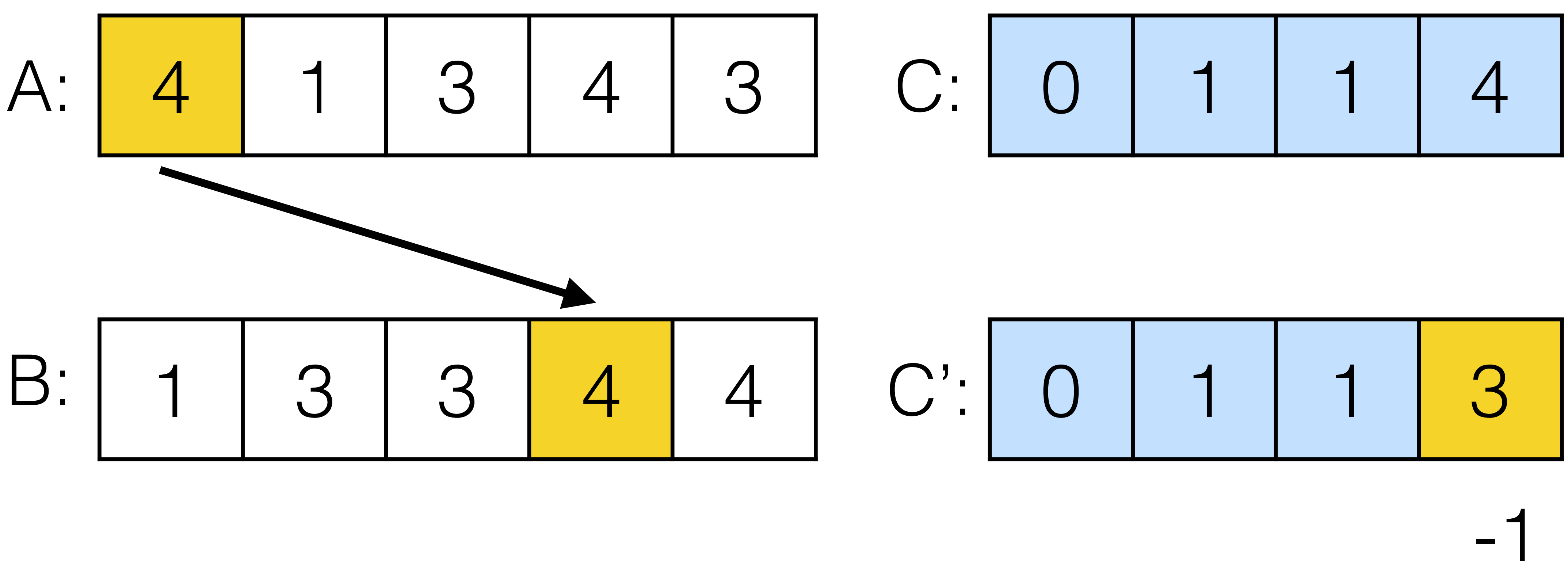

```
for j := n downto 1
   do B[C[A[j]]]←A[j]
      C[A[j]] ←C[A[j]] −1
```

Example: loop 4




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

Example: loop 4




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

Asymptotic Analysis

$\Theta(k)$
```
for i := 1 to k
    do C[i]:=0
```
$\Theta(n)$
```
for j := 1 to n
    do C[A[j]]:=C[A[j]]+1
```
$\Theta(k)$
```
for i := 2 to k
    do C[i]:=C[i]+C[i–1]
```
$\Theta(n)$
```
for j := n downto 1
    do B[C[A[j]]]←A[j]
       C[A[j]] ←C[A[j]] –1
```

$\Theta(n + k)$

Asymptotic Analysis

- If k=O(n), then Counting Sort takes Θ(n) time.
- Remark:
  - Comparison sorting takes Ω(n lg n) time. Counting Sort is not a comparison sort. In fact, not a single comparison between elements occurs!

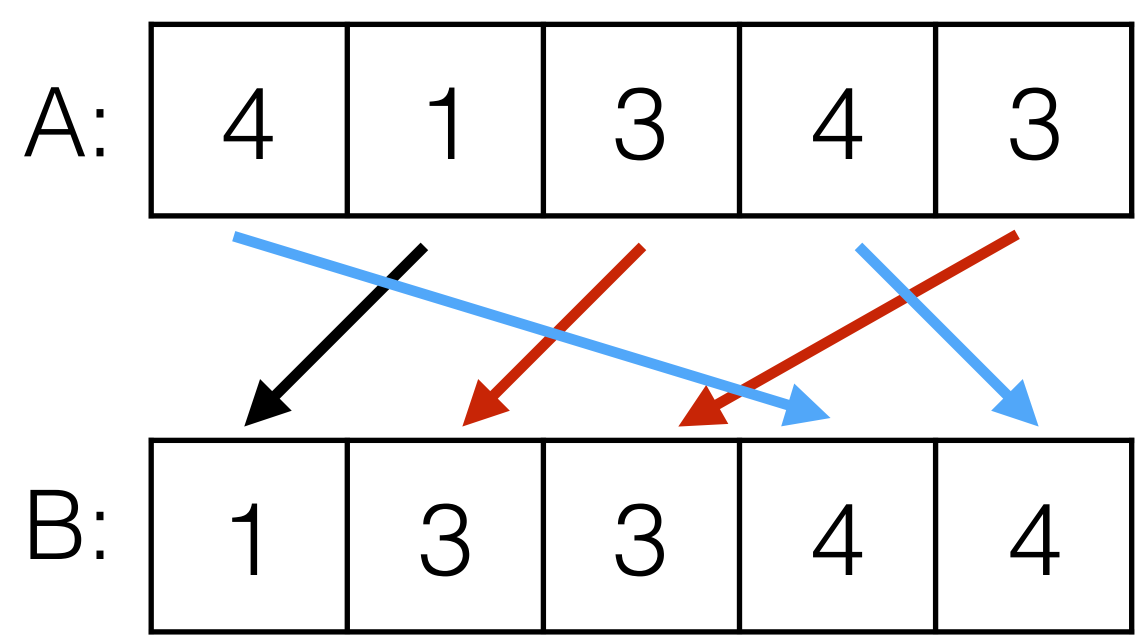
Stable Sorting (hw 4)

Definition:

- Stable sorting: Stable sorting algorithms maintain the relative order of records with equal keys (i.e., values).
- Thus, a sorting algorithm is stable, if whenever there are two records R and S with the same key and with R appearing before S in the original list, R will appear before S in the sorted list.

Is Counting Sort stable?

Yes!



2.5 Radix Sort

Motivation

- Counting Sort is less efficient when processing numbers from a large range, i.e., k is large.
- Can we find an algorithm that efficiently sorts n numbers for large k?

Origin

- The 1880 U.S. Census took almost 10 years to process.
- Herman Hollerith (1860-1929) prototyped a 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 (IBM).

Idea

- Hollerith's idea was to use a digit-by-digit sort.
- He sorted on most significant digit first.
- However, it requires us to keep one sequence for each digit, which then gets sorted recursively.
- It is more efficient to sort on least significant digit first.
- This idea requires a stable sorting algorithm.

Radix Sort

```
RADIX-SORT(A, d)
for i = 1 to d
    use a stable sort to sort array A on digit i
```

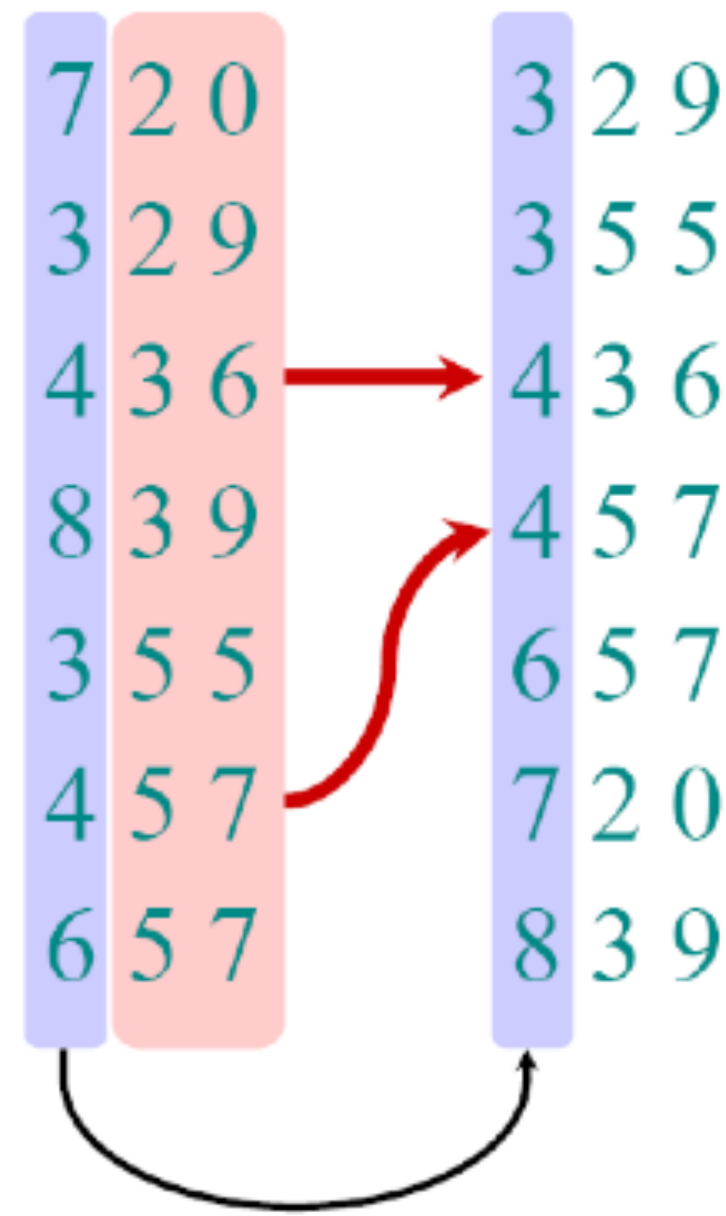
Example

329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

Correctness of radix sort

Induction on digit position:

- Only one digit: trivial.
- 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, i.e., correct order.



Asymptotic Analysis

- Use Counting Sort as stable sorting algorithm.
- Sort n computer words of b bits each.
- Each word can be viewed as having b/r base-2^r digits.
- Example: 32-bit word.
  - $r = 8$:
    $d = b/r = 4$ passes of counting sort on base-2^8 digits;
  - $r = 16$:
    $d = b/r = 2$ passes of counting sort on base-2^{16} digits.
- **How many passes should we make?**

Choosing r

- Counting Sort takes Θ(n+k) time to sort *n* numbers in the range from 0 to *k* – 1.
- If each b-bit word is broken into *r*-bit pieces, each pass of Counting Sort takes $\Theta(n + 2^r)$ time.
- Since there are *b*/*r* passes, we have:

$$T(n,b) = \Theta\left(\frac{b}{r}\left(n + 2^r\right)\right).$$

- Choose *r* to minimize *T*(*n*,*b*).

Choosing r

$$T(n,b) = \Theta\left(\frac{b}{r}\left(n + 2^r\right)\right).$$

- Increasing r means fewer passes, but when $r >> \lg n$ the time grows exponentially.
- We do not want $2^r > n$, but there is no harm asymptotically in choosing r as large as possible subject to this constraint.
- Choosing $r = \lg n$ implies $T(n,b) = \Theta(bn/\lg n)$.
- For numbers in the range from 0 to $n^d - 1$,
  we have $b = dr = d \lg n$,
  i.e., Radix Sort runs in $\Theta(dn)$ time.

Conclusions

- In practice, Radix Sort is fast for large inputs, as well as simple to code and maintain.
- Example (32-bit numbers, i.e., b=32, and n=2000):
  - ***dn***: At most d = 3 passes when sorting 2000 numbers.
  - ***n* lg*n***: Merge Sort and Quicksort do at least ceiling (lg 2000) = 11 passes.

2.6 Bucket Sort

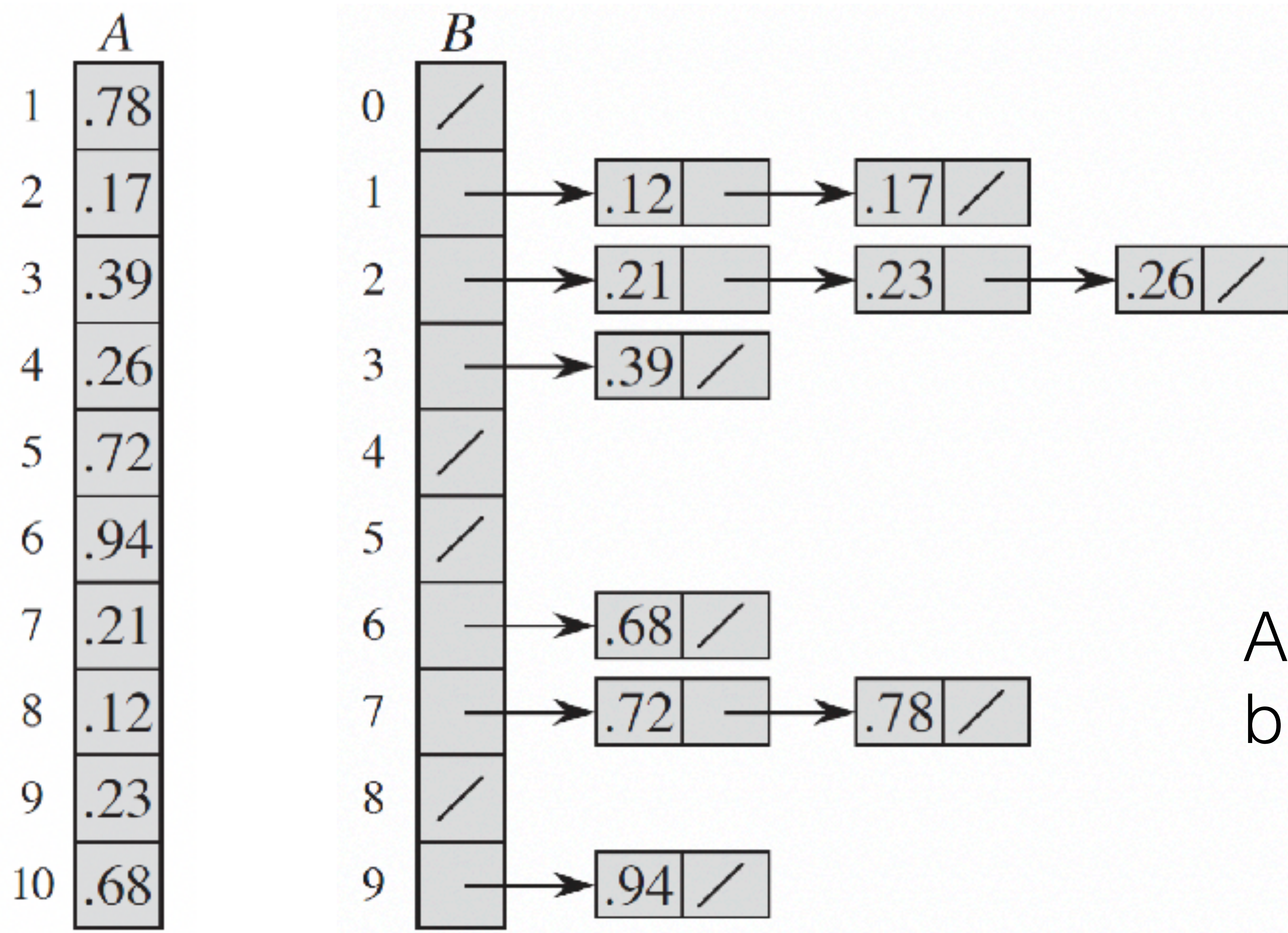
Motivation

- Can we use the idea of Radix Sort to sort any numbers, i.e., without assuming them to be integers?
- In order to do this efficiently, we make a new assumption:
  - The to-be-sorted elements shall distribute uniformly and independently over the interval [0,1).
- Remark:
  - Interval [0,1) is not a real restriction, as we can normalize the elements to this interval in linear time.
  - However, uniform distribution and independence are restrictions and we will see that we need this to assure good expected running time.

Idea

- Assuming that we have to sort *n* numbers, we split the interval [0,1) into *n* subintervals or *buckets*.
- Then, we can distribute the *n* numbers to the *n* buckets.
- Assuming uniform distribution, we can conclude that we have only few numbers falling into each bucket.

Example (n=10)



A[i] is put into bucket $\lfloor nA[i] \rfloor$

Bucket Sort

BUCKET-SORT(A)

let $B[0 \,..\, n-1]$ be a new array
$n = A.length$
for $i = 0$ **to** $n-1$
    make $B[i]$ an empty list
for $i = 1$ **to** n
    insert $A[i]$ into list $B[\lfloor n A[i] \rfloor]$
for $i = 0$ **to** $n-1$
    sort list $B[i]$ with insertion sort
concatenate the lists $B[0], B[1], \ldots, B[n-1]$ together in order

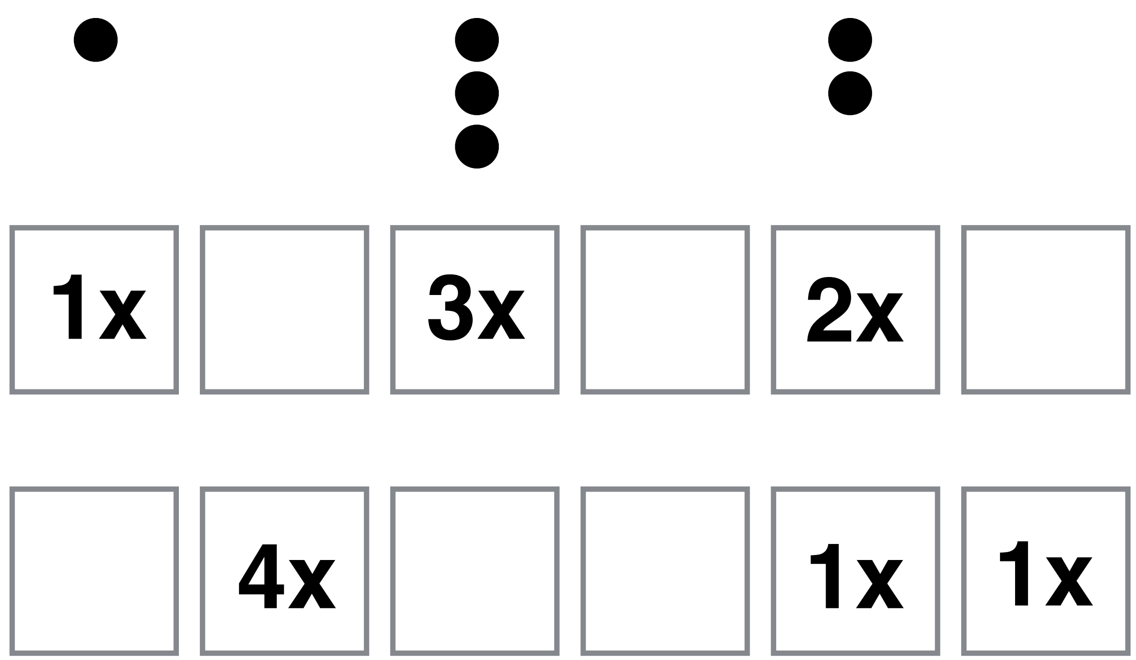
Time complexity

```
BUCKET-SORT(A)
let B[0..n − 1] be a new array
n = A.length
for i = 0 to n − 1
    make B[i] an empty list
for i = 1 to n
    insert A[i] into list B[⌊nA[i]⌋]
for i = 0 to n − 1
    sort list B[i] with insertion sort
concatenate the lists B[0], B[1], ..., B[n − 1] together in order
```

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

where n_i denotes the number of elements in bucket i.

Average case



might fall into boxes like that

or that

...

Expected time complexity

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

What is E [n_i^2]?

Let X_{ij} be the event that $A[j]$ falls into bucket i.
Then,

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

Use assumptions of uniform distribution and independence.

Estimate $E\,[n_i^2]$

$$E[n_i^2] = E\left[\left(\sum_{j=1}^{n} X_{ij}\right)^2\right] = E\left[\sum_{j=1}^{n}\sum_{k=1}^{n} X_{ij}X_{ik}\right]$$

$$= E\left[\sum_{j=1}^{n} X_{ij}^2 + \sum_{j=1}^{n}\sum_{\substack{k=1\\k\neq j}}^{n} X_{ij}X_{ik}\right]$$

$$= \sum_{j=1}^{n} E[X_{ij}^2] + \sum_{j=1}^{n}\sum_{\substack{k=1\\k\neq j}}^{n} E[X_{ij}X_{ik}]$$

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

Estimate $E\left[n_i^2\right]$

$E[X_{ij}]\,E[X_{ik}] = \frac{1}{n} \cdot \frac{1}{n} = \frac{1}{n^2}.$

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

$$
\begin{aligned}
E[n_i^2] &= \sum_{j=1}^{n} E[X_{ij}^2] + \sum_{j=1}^{n} \sum_{\substack{k=1 \\ k \neq j}}^{n} E[X_{ij}]\,E[X_{ik}] \\
&= \sum_{j=1}^{n} \frac{1}{n} + \sum_{j=1}^{n} \sum_{\substack{k=1 \\ k \neq j}}^{n} \frac{1}{n^2} \\
&= \frac{n}{n} + n\,(n-1)\frac{1}{n^2} \\
&= 2 - \frac{1}{n}.
\end{aligned}
$$

Expected time complexity

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

$$\begin{aligned} E[T(n)] &= \Theta(n) + n \cdot O(2 - 1/n) \\ &= \Theta(n) \end{aligned}$$

2.7 Searching

Searching problem

- Given a sorted sequence.
- Find an element in that sequence.

- Example: Find element 9.

Sequence	3	5	7	8	9	12	15

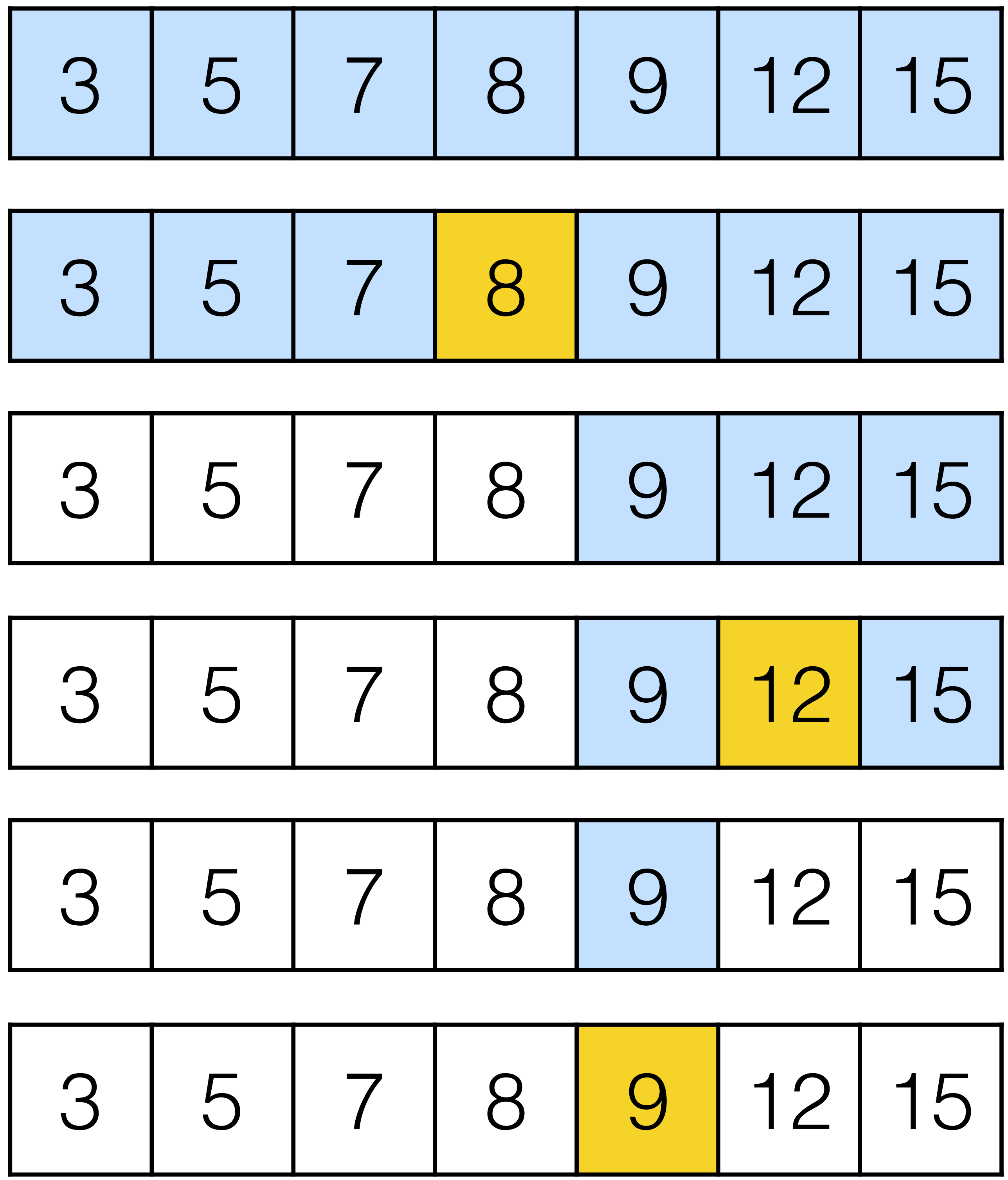
- Brute-force approach (going through the sequence from start until we find the 9) runs in O(*n*).

Binary Search

Idea: Use a divide & conquer strategy.

1. Divide:
   Check middle element.
2. Conquer:
   Recursively search 1 subarray.
3. Combine:
   Nothing to be done.

Example (find 9)



Time Complexity

$T(n) = 1\mathrm{T}(n/2) + \Theta(1)$
$a = 1,\ b = 2$

$\mathrm{n}\ \log_b a = n^{\log_2 1} = 1$
$f(n) = \Theta(1)$

Case 2: $T(n) = \Theta(\lg n)$.

2.8 Summary

Summary

Sorting problem:

- Comparison sorts:
  - InsertionSort: $\Theta(n)$ [best], $\Theta(n^2)$ [average&worst].
  - MergeSort: $\Theta(n\lg n)$.
  - HeapSort: $\Theta(n\lg n)$ / Heap as a data structure
  - Quicksort: $\Theta(n\lg n)$ [best&average], $\Theta(n^2)$ [worst].
  - Decision trees: Worst case does not get better than $\Theta(n\lg n)$.
- Sorting in linear time:
  - Counting Sort: small integers
  - Radix Sort: large integers
  - Bucket Sort: any numbers, but uniform distribution.

Searching Problem:

- Linear Search: $\Theta(1)$ [best], $\Theta(n)$ [average&worst]
- Binary Search: $\Theta(1)$ [best], $\Theta(\lg n)$ [average&worst]