Sorting

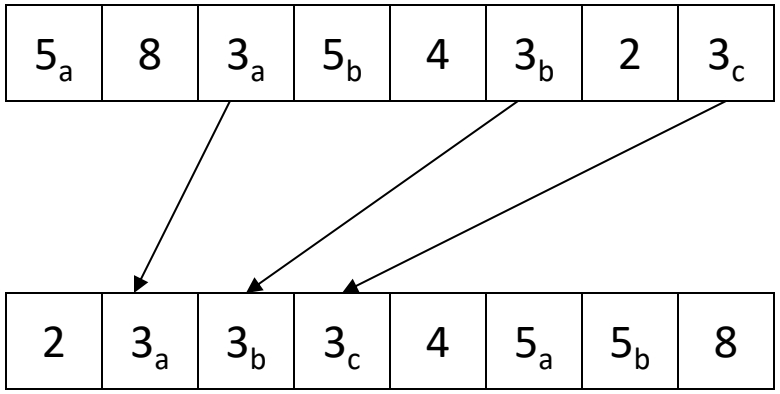
CLRS Chapters 6-8

- Input:
  - An array A of data records
  - A key in each data record (e.g., integers)
  - A comparison function which imposes a consistent ordering on the keys
- Output:
  - Reorganize the elements of *A* such that

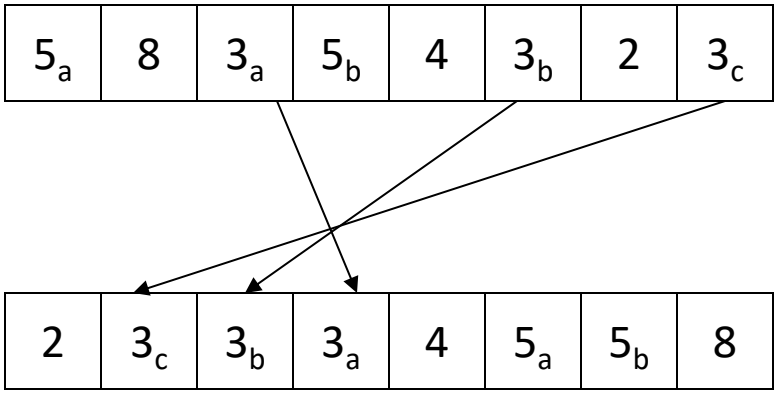
    for any *i* and *j*, if $i < j$ then $A[i] \leq A[j]$
- Performance:
  - time complexity
  - space - do we need additional space? how much? in-place
  - stability – respect original order in case of ties

Reichman University

Example



Stable Sort



Unstable Sort

Naïve algorithms

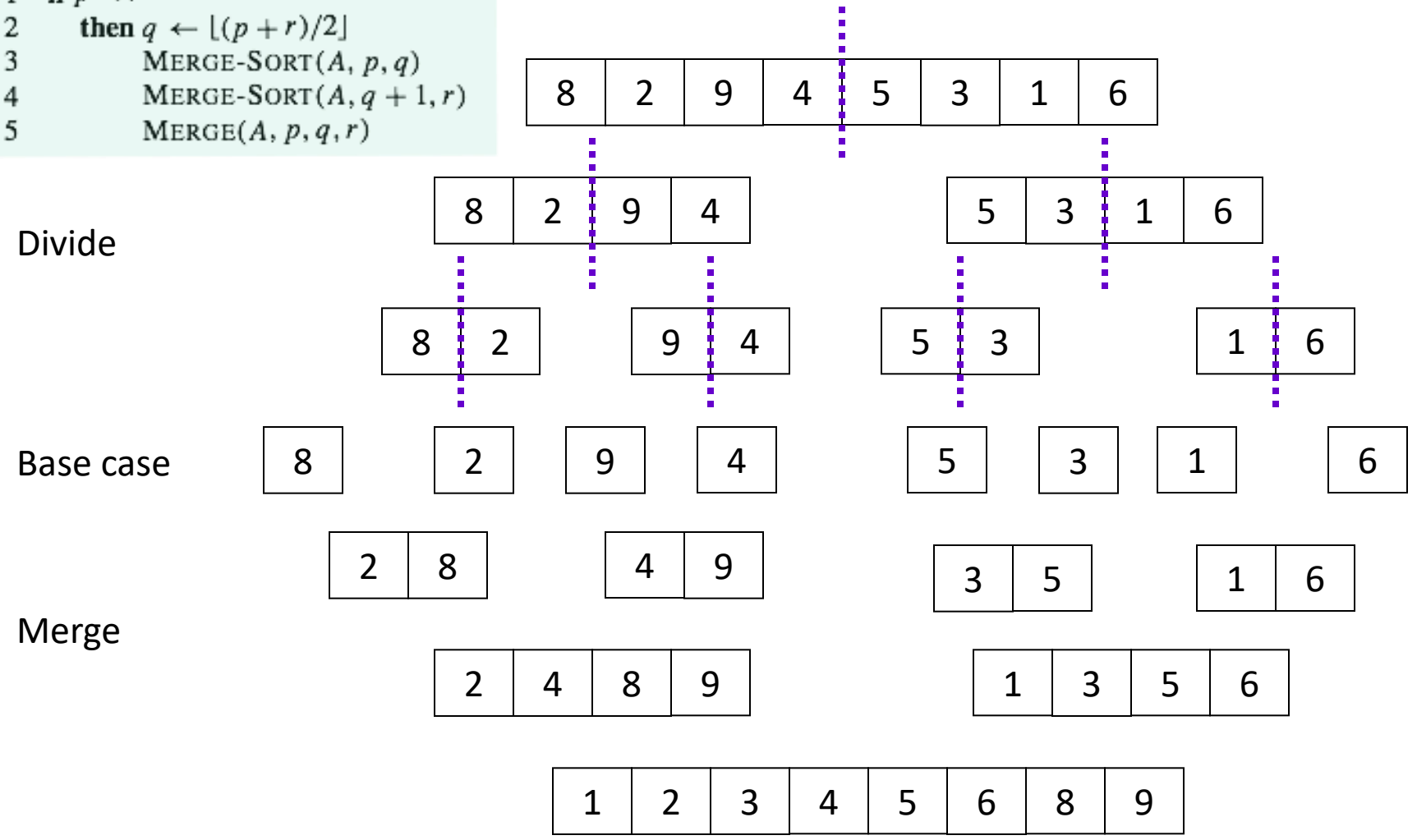
Reichman University

- naïve algorithms (min-sort, insertion sort, bubble sort etc.)
  - find minimum key. place it in first position. repeat on remaining array
  - time $\Theta(n^2)$
  - in place (uses $O(1)$ additional space)
  - stable

- heap-sort
  - make-heap from the input array
  - delete-min one element at a time
  - time $\Theta(n \log n)$
  - in place (uses $O(1)$ additional space)
  - not stable

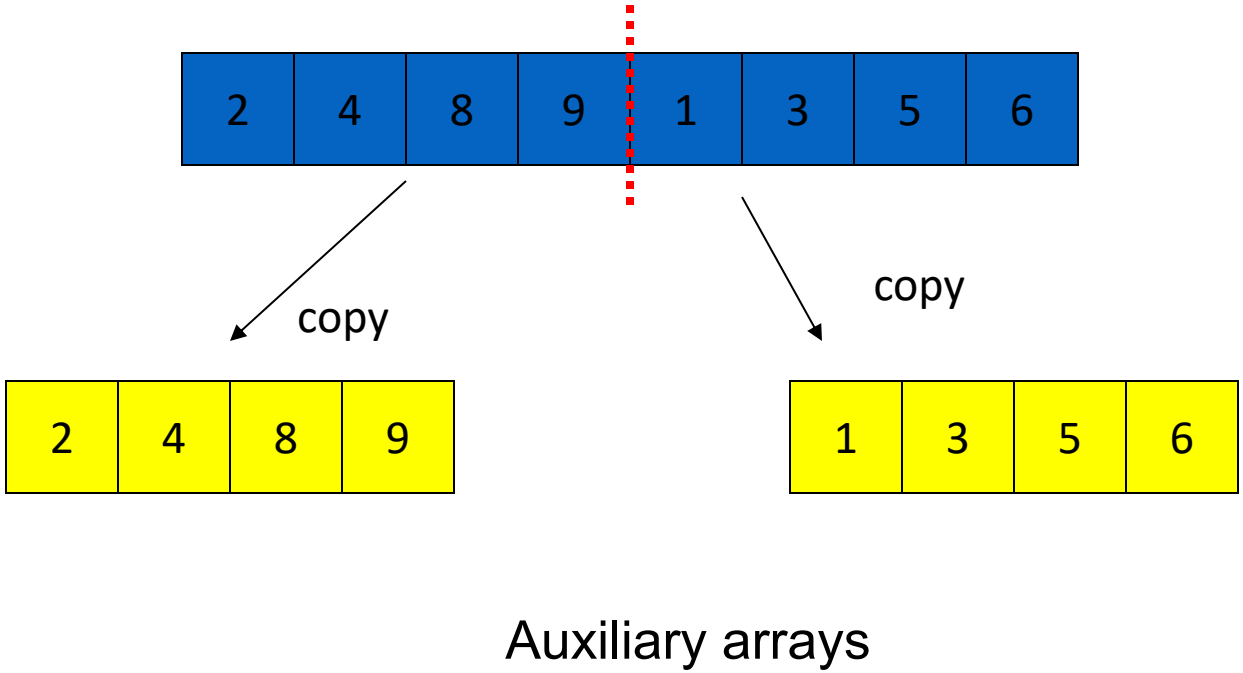
Merge Sort (divide and conquer)

```
MERGE-SORT(A, p, r)
1  if p < r
2     then q ← ⌊(p + r)/2⌋
3          MERGE-SORT(A, p, q)
4          MERGE-SORT(A, q + 1, r)
5          MERGE(A, p, q, r)
```



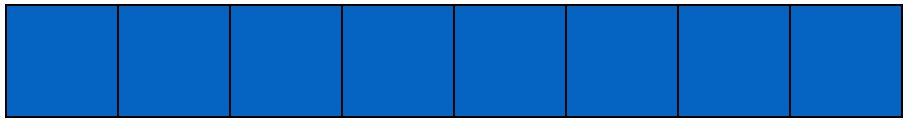
Merging two sorted subarrays

- The merging requires auxiliary arrays.



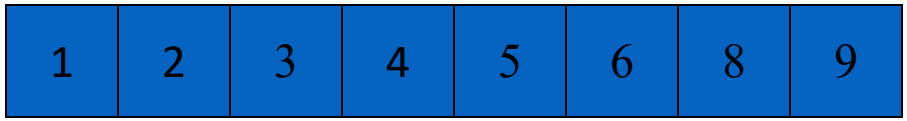
Merging two sorted subarrays

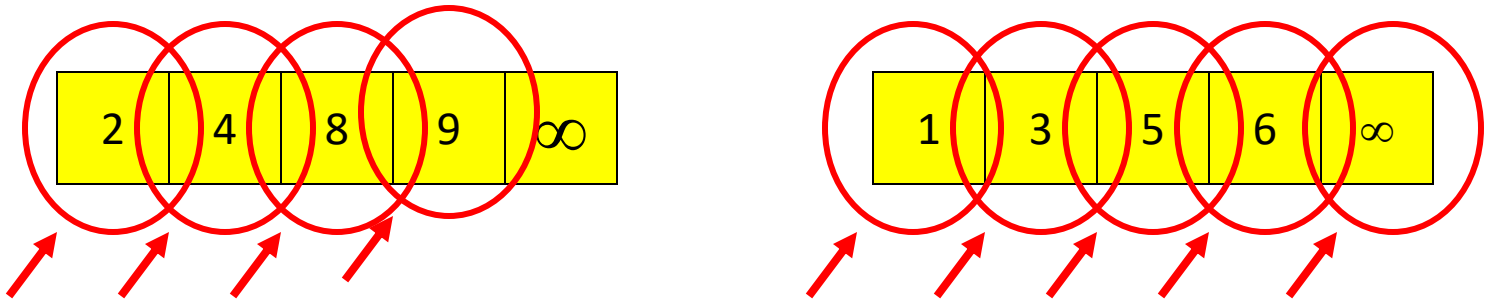
- The merging requires auxiliary arrays.



Auxiliary arrays

Merging two sorted subarrays





Auxiliary arrays

Merge Sort - analysis

```
MERGE-SORT(A, p, r)
1  if p < r
2     then q ← ⌊(p + r)/2⌋
3          MERGE-SORT(A, p, q)
4          MERGE-SORT(A, q + 1, r)
5          MERGE(A, p, q, r)
```

```
MERGE(A, p, q, r)
 1  n1 ← q − p + 1
 2  n2 ← r − q
 3  create arrays L[1 .. n1 + 1] and R[1 .. n2 + 1]
 4  for i ← 1 to n1
 5       do L[i] ← A[p + i − 1]
 6  for j ← 1 to n2
 7       do R[j] ← A[q + j]
 8  L[n1 + 1] ← ∞
 9  R[n2 + 1] ← ∞
10  i ← 1
11  j ← 1
12  for k ← p to r
13       do if L[i] ≤ R[j]
14            then A[k] ← L[i]
15                 i ← i + 1
16            else A[k] ← R[j]
17                 j ← j + 1
```

- the time to merge two sorted subarrays with total n elements is $O(n)$
- let $T(n)$ be the worst case time of Merge-Sort on an array of size n
- $T(n) \leq T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + O(n)$
  - O(log n) levels of recursion
  - calls in a given level are element disjoint so total time per level is $O(n)$
- total time is $O(n \log n)$
- can prove more formally by induction

- not in place (need to copy)
- stable

Merge Sort (non-recursive view)

- lower copy / recursion overhead

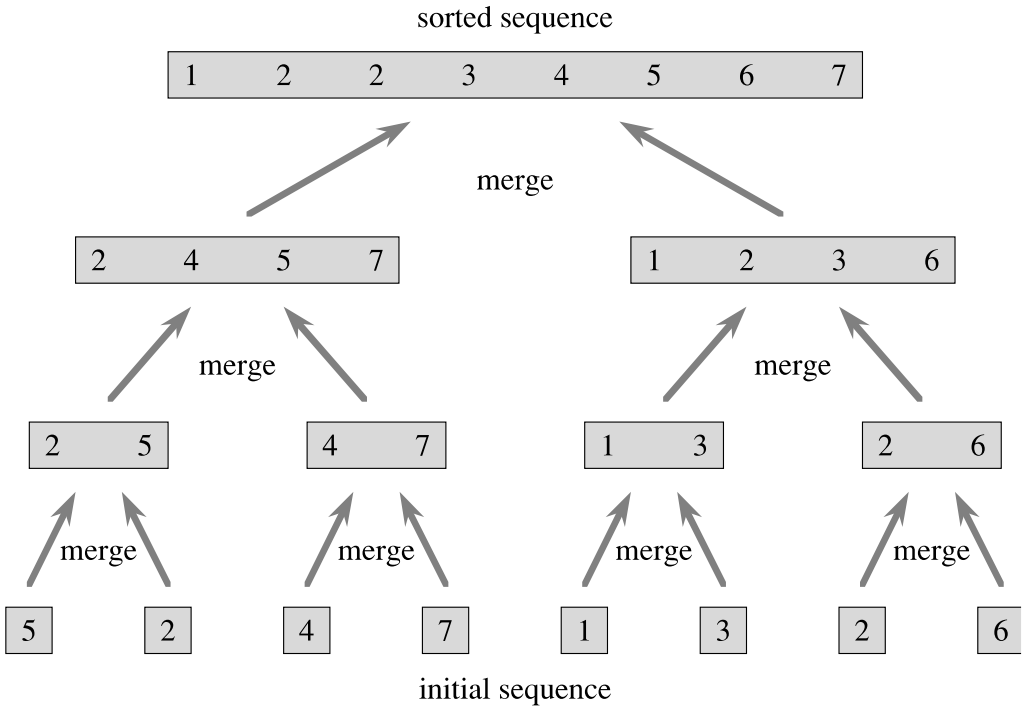
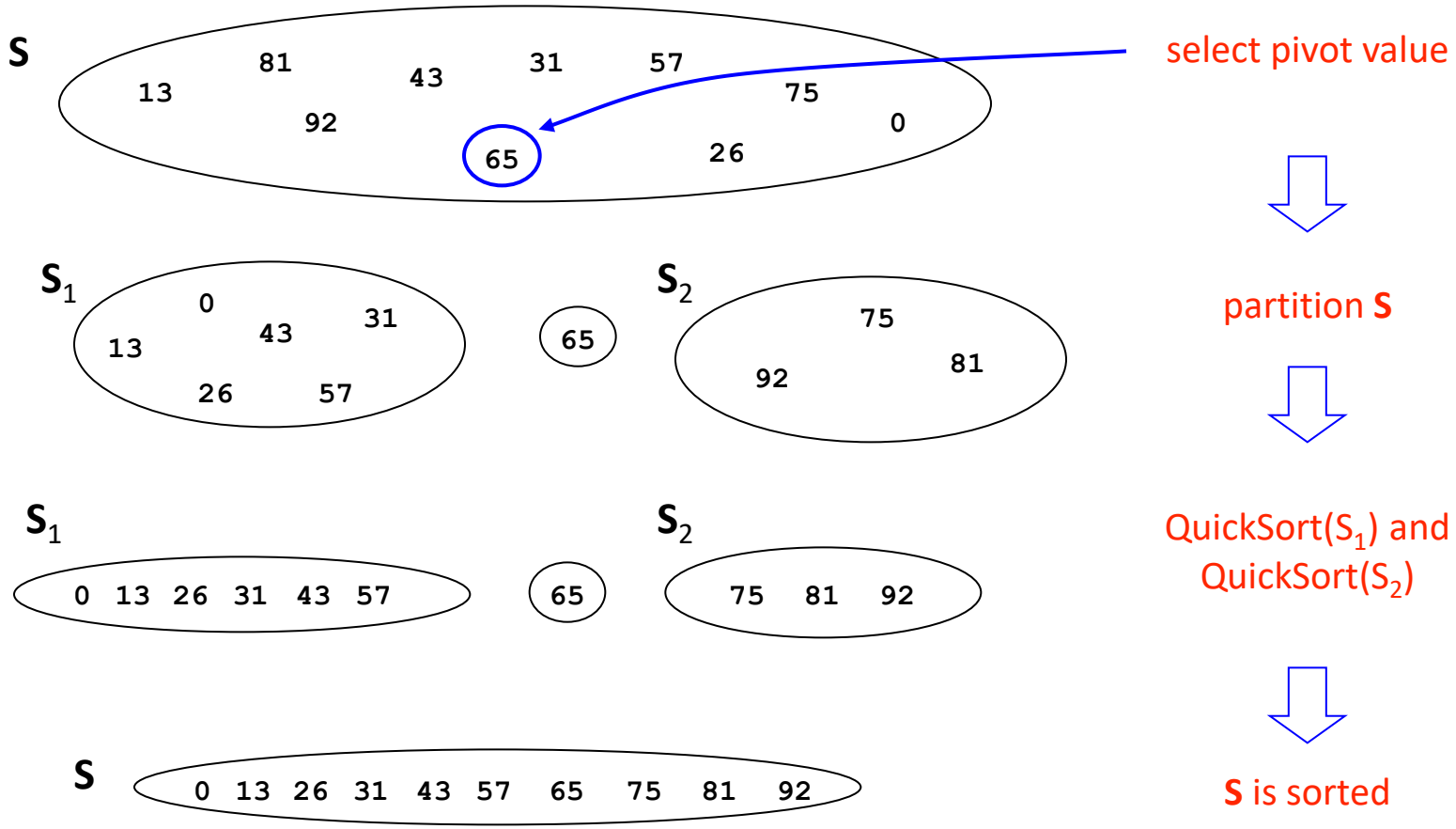


Figure 2.4 The operation of merge sort on the array $A = \langle 5, 2, 4, 7, 1, 3, 2, 6 \rangle$. T

Quicksort

- also divide and conquer
- but can be done in-place



Quicksort

Reichman University

- Choose pivot element q (many possibilities how this is done)
- move all elements smaller than q left of q
- move all elements greater than q right of q
- recurse on left and right parts

```
Quicksort(A, p, r)
if p+2 < r
    then q ←Partition(A, p, r)
         Quicksort(A, p, q-1)
         Quicksort(A, q+1,r)
else SimpleSort(A,p,r) %less than 3 elements
```

Partition

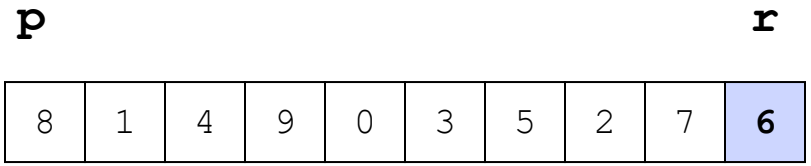
- Set pointers i and j to start and end of array
- Increment i until you hit element A[i] ≥ pivot
- Decrement j until you hit element A[j] < pivot
- Swap A[i] and A[j]
- Repeat until i and j cross

8	1	4	9	0	3	5	2	7	6

Partition

- Set pointers i and j to start and end of array
- Increment i until you hit element A[i] ≥ pivot
- Decrement j until you hit element A[j] < pivot
- Swap A[i] and A[j]
- Repeat until i and j cross

Pivot is x=6

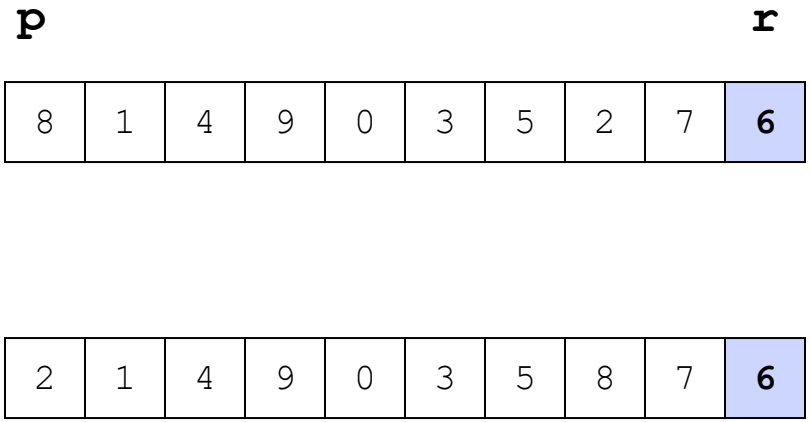



```
Partition(A,p,r)
x=A[r]; j=r; i=p-1;
While true
     repeat j ←j - 1
          until A[j]<x
     repeat i ← i +1
          until A[i]≥x
     if i<j
        then A[i]↔A[j] % swap
        else A[j+1] ↔ A[r]
           return j+1
```

Partition

- Set pointers i and j to start and end of array
- Increment i until you hit element A[i] ≥ pivot
- Decrement j until you hit element A[j] < pivot
- Swap A[i] and A[j]
- Repeat until i and j cross

Pivot is x=6



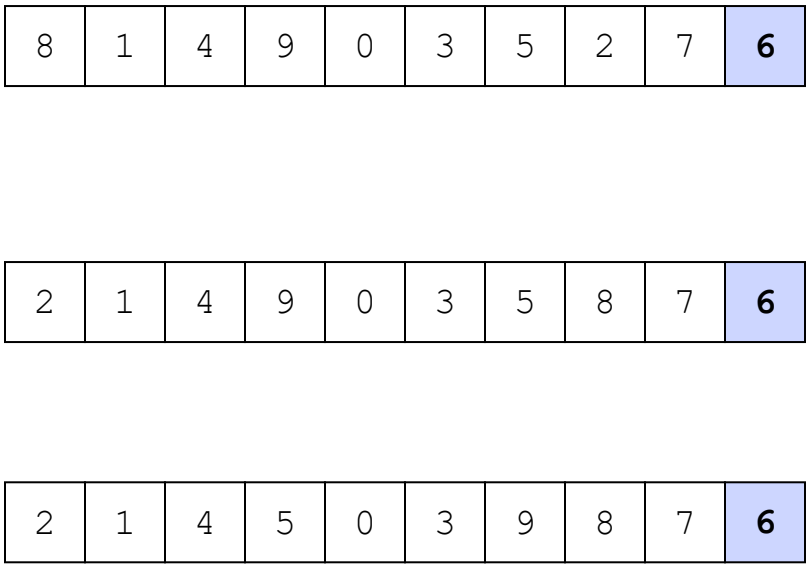

```
Partition(A,p,r)
x=A[r]; j=r; i=p-1;
While true
     repeat j ←j - 1
          until A[j]<x
     repeat  i ← i +1
          until A[i]≥x
     if i<j
        then A[i]↔A[j] % swap
        else A[j+1] ↔ A[r]
           return j+1
```

Partition

- Set pointers i and j to start and end of array
- Increment i until you hit element A[i] ≥ pivot
- Decrement j until you hit element A[j] < pivot
- Swap A[i] and A[j]
- Repeat until i and j cross

```
Partition(A,p,r)
x=A[r]; j=r; i=p-1;
While true
      repeat j ←j - 1
            until A[j]<x
      repeat i ← i +1
            until A[i]≥x
      if i<j
         then A[i]↔A[j] % swap
         else A[j+1] ↔ A[r]
            return j+1
```

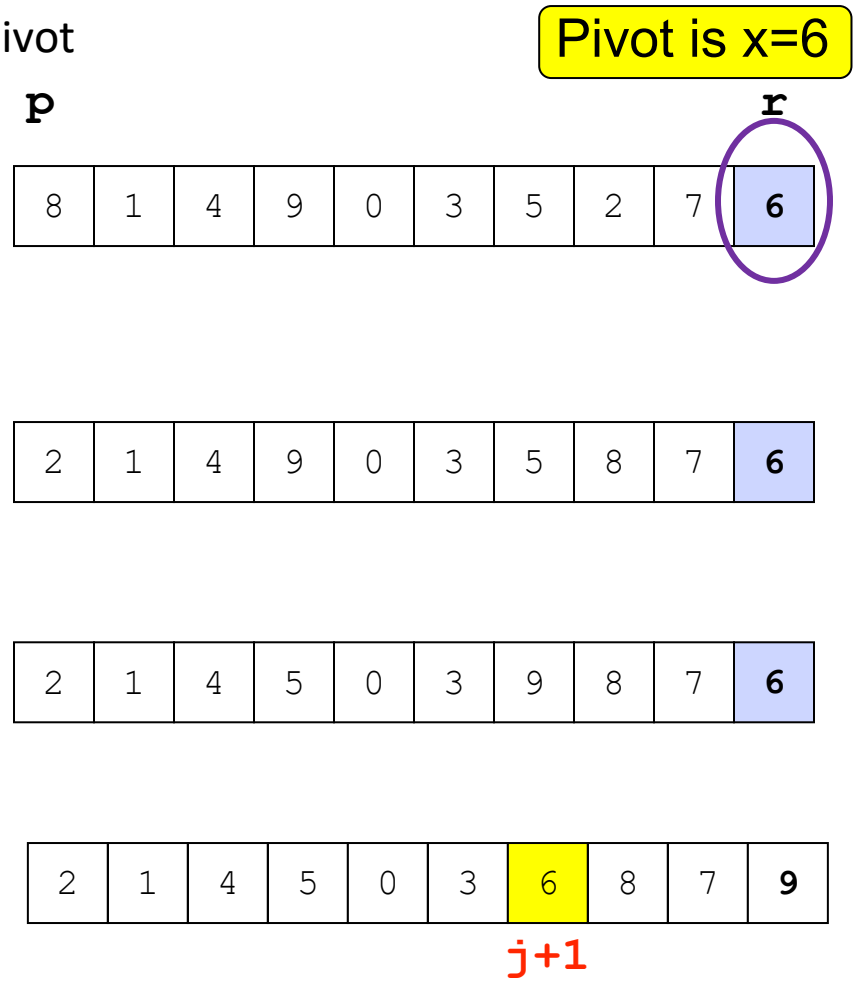
Pivot is x=6



Partition

- Set pointers i and j to start and end of array
- Increment i until you hit element A[i] ≥ pivot
- Decrement j until you hit element A[j] < pivot
- Swap A[i] and A[j]
- Repeat until i and j cross

```
Partition(A,p,r)
x=A[r]; j=r; i=p-1;
While true
      repeat j ←j - 1
            until A[j]<x
      repeat  i ← i +1
            until A[i]≥x
      if i<j
         then A[i]↔A[j] % swap
         else A[j+1] ↔ A[r]
            return j+1
```



Choice of pivot

Reichman University

- what is the ideal pivot?
- the median: $T(n) \leq 2T\left(\frac{n}{2}\right) + O(n) = O(n \log n)$
- not clear how to efficiently find the median (possible in $O(n)$ time)
- use first/last/middle/random element
- worst case: $T(n) = T(n-1) + O(n) = O(n^2)$

- How about $T(n) \leq T\left(\frac{n}{10}\right) + T\left(\frac{9n}{10}\right) + O(n)$?

- We will analyze the expected running time when pivot chosen uniformly at random
- The running time is dominated by the number of comparisons made by the algorithms

Randomized algorithms

Reichman University

- new assumption – algorithm has access to randomness
- each time we run the algorithm the behavior might be different

- in the case of quicksort – the algorithm always returns a sorted array, but what elements are compared, and how many comparisons are made depends on the randomness

- so the running time of the algorithm is a random variable
- the sample space is the set of possible random choices
- Let T(A,r) denote the worst case running time of the algorithm on an input array A when the randomness is r
- We are interested in $\max_{|A|=n} \{E_r[T(A,r)]\}$ – worst case expected running time
- the expectation is over the random choices made by the algorithm (not over the possible inputs!)

Expected # of comparisons

- Let $z_1, z_2, \ldots, z_n$ be the elements in sorted order
- any two elements are compared by the algorithm at most once (when one of them is the pivot)
- define r.v. X_{ij} to be 1 iff z_i is compared to z_j
- define r.v. X to be total # of comparisons $X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr[z_i \text{ is compared with } z_j]$$

- key observation: elements z_i and z_j are compared iff one of them is chosen as the pivot before any of the elements $z_{i+1}, z_{i+2}, \ldots, z_{j-1}$.
- this happens with probability $\frac{2}{j-i+1}$

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n-1} \sum_{k=2}^{n-i+1} \frac{2}{k} < \sum_{i=1}^{n-1} \sum_{k=2}^{n} \frac{2}{k} = 2 \sum_{i=1}^{n-1} \sum_{k=2}^{n} \frac{1}{k} <$$

$$2 \sum_{i=1}^{n-1} \int_{1}^{n} \frac{dx}{x} \leq 2n \ln n = \Theta(n \log n)$$

Quicksort

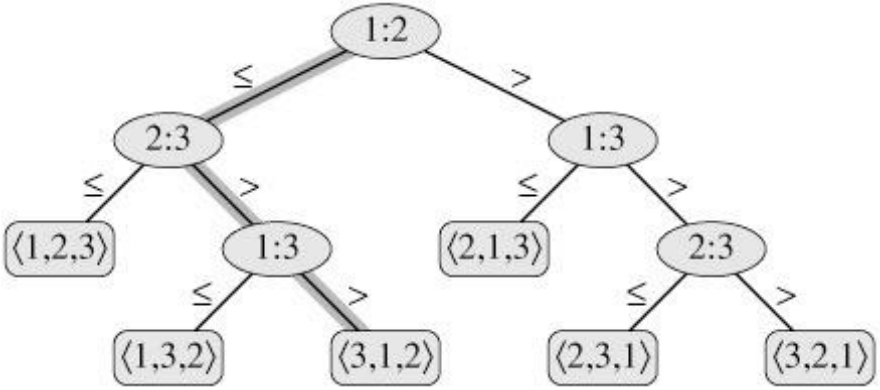
Reichman University

- worst case time $O(n^2)$
- expected time $O(n \log n)$
- in-place, but needs additional space (for maintaining the recursion)
- not stable

$\Omega(n \log n)$ lower bound for sorting

Reichman University

- all our sorting algorithms (so far) use comparisons
- sorting the input = finding the permutation that sorts
- algorithms differ by choice of which elements to compare
- each comparison has two outcomes (suppose no duplicates)
- an algorithm cannot terminate before the comparisons it made uniquely determine the permutation that sorts the input
- describe the possible executions of an algorithm on all different inputs by a binary tree
  - each internal node represents a single comparison
  - each leaf represents a permutation
  - each root-to-leaf path represents the execution of the algorithm on a specific input

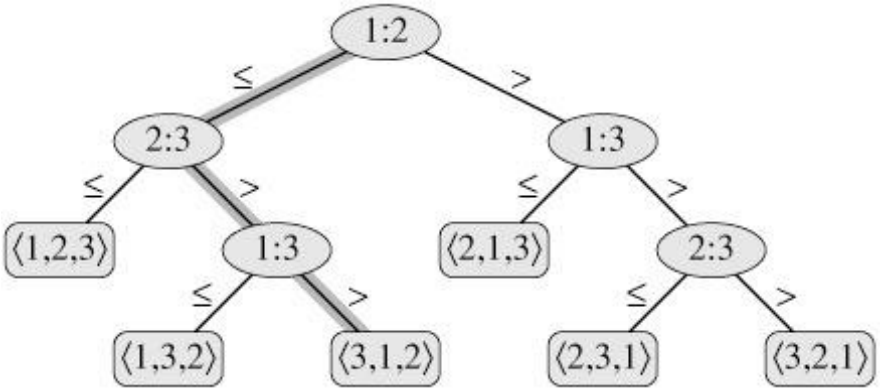


$\Omega(n \log n)$ lower bound for sorting

Reichman University

- a correct algorithm must be able to correctly identify all permutations
- there are $n!$ different permutations on n elements
- so corresponding binary tree has at least $n!$ leaves
- any binary tree with x leaves has depth at least $\log_2 x$
- so every comparison based algorithm must have an execution that makes $\log(n!) = \log(\prod_{k=1}^{n} k) = \sum_{k=1}^{n} \log k \geq \sum_{k=\frac{n}{2}}^{n} \log \frac{n}{2} = \frac{n}{2}(\log n - 1) = \Omega(n \log n)$ comparisons

- in fact $\Omega(n \log n)$ holds not just for worst case, but also for average case



Sorting in $O(n)$ time

Reichman University

- not comparison based
- counting sort
- bucket sort
- radix sort

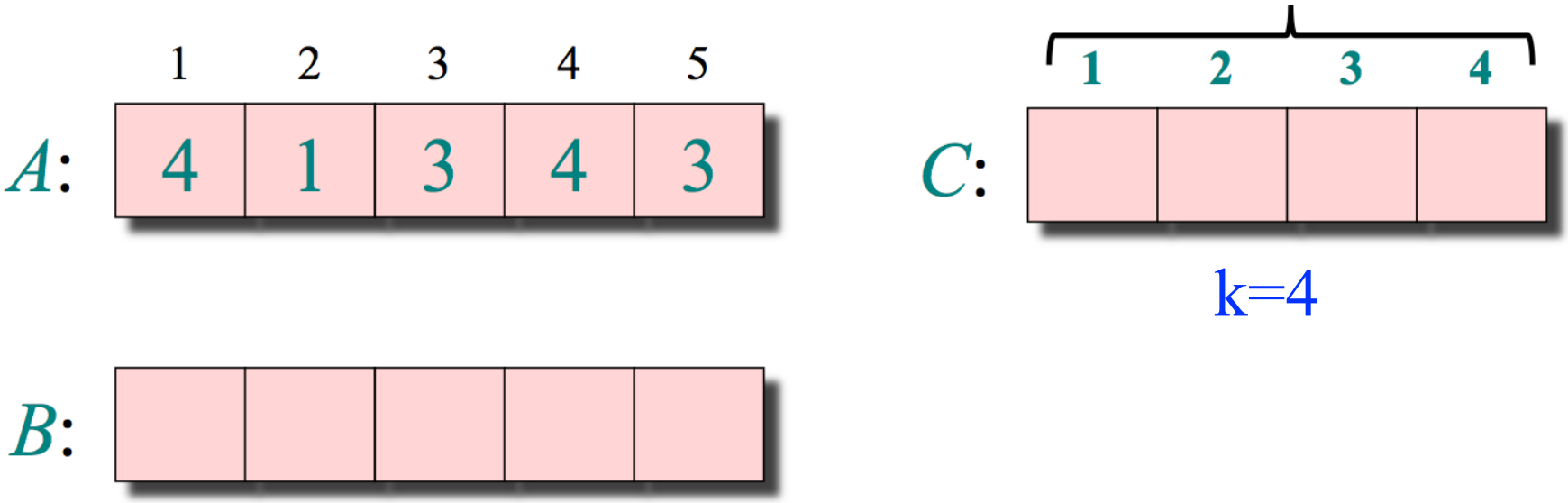
Counting sort

Reichman University

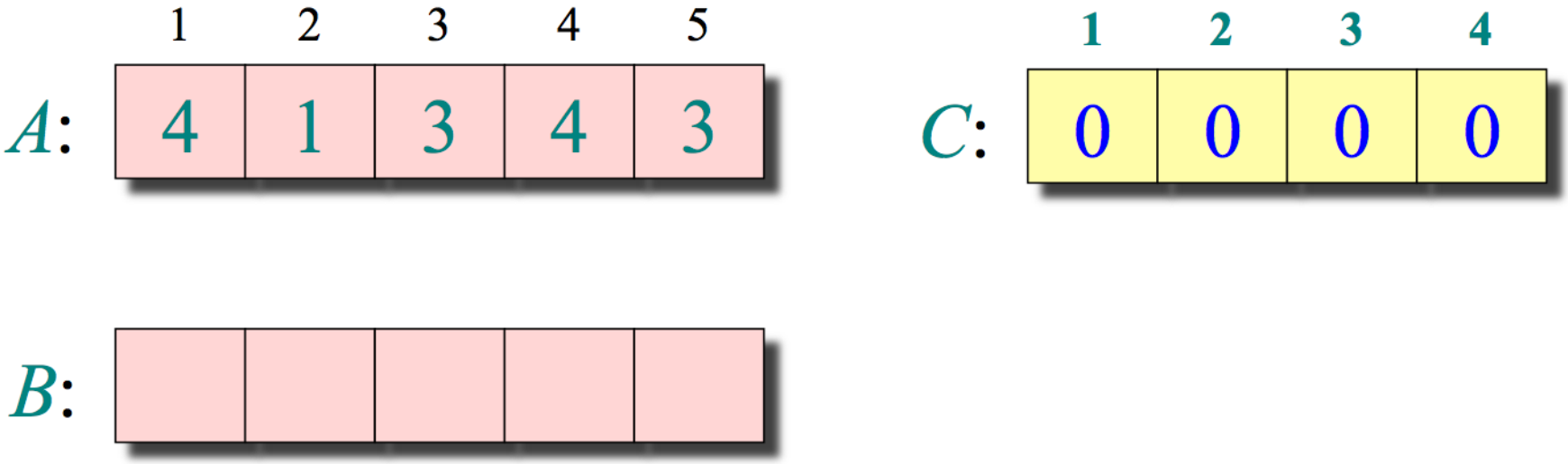
- not comparison based
- input array A[1…n] of integers in $\{1,2,\ldots,k\}$
- output array B[1…n] a sorted permutation of A

- auxiliary array C[1…k]

Counting sort



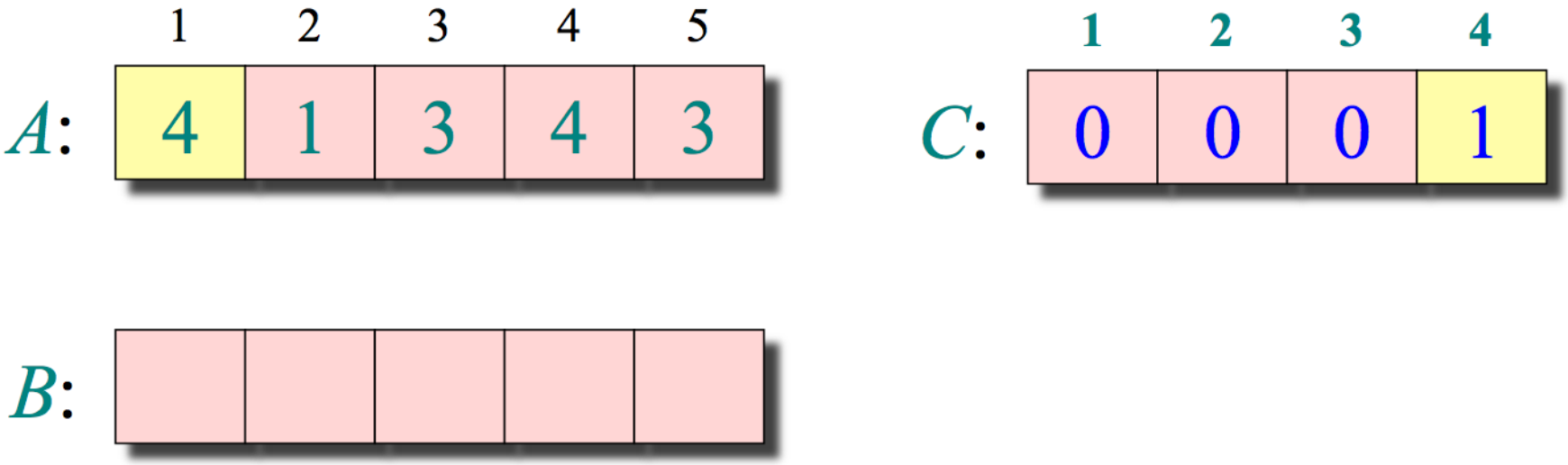
initialization



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

do $C[i] \leftarrow 0$

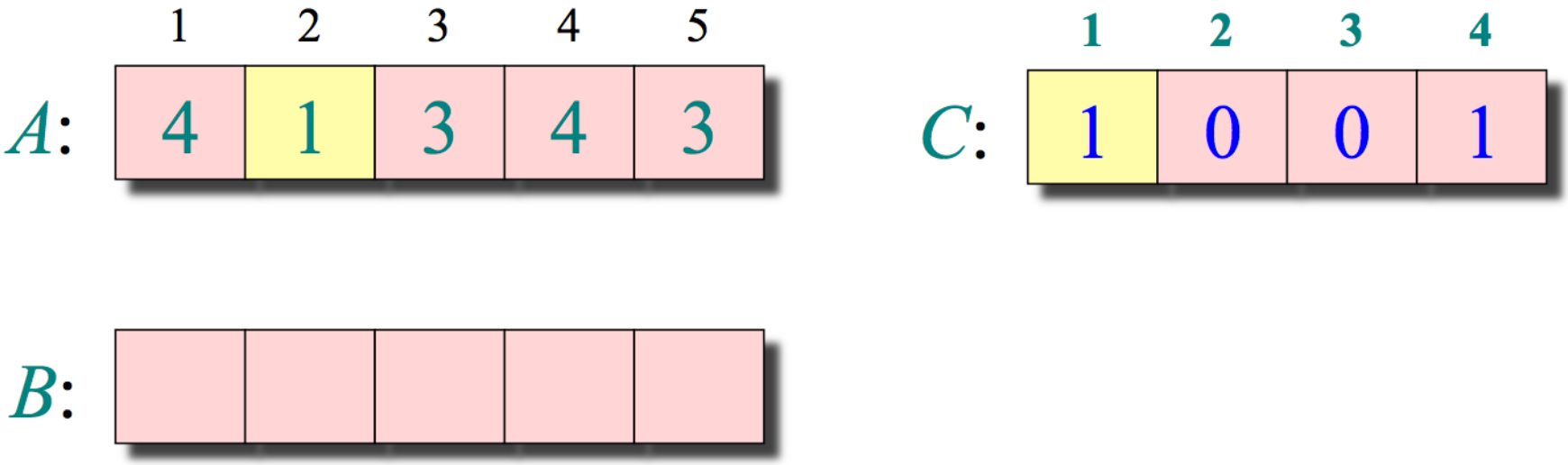
Loop 1: count



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

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

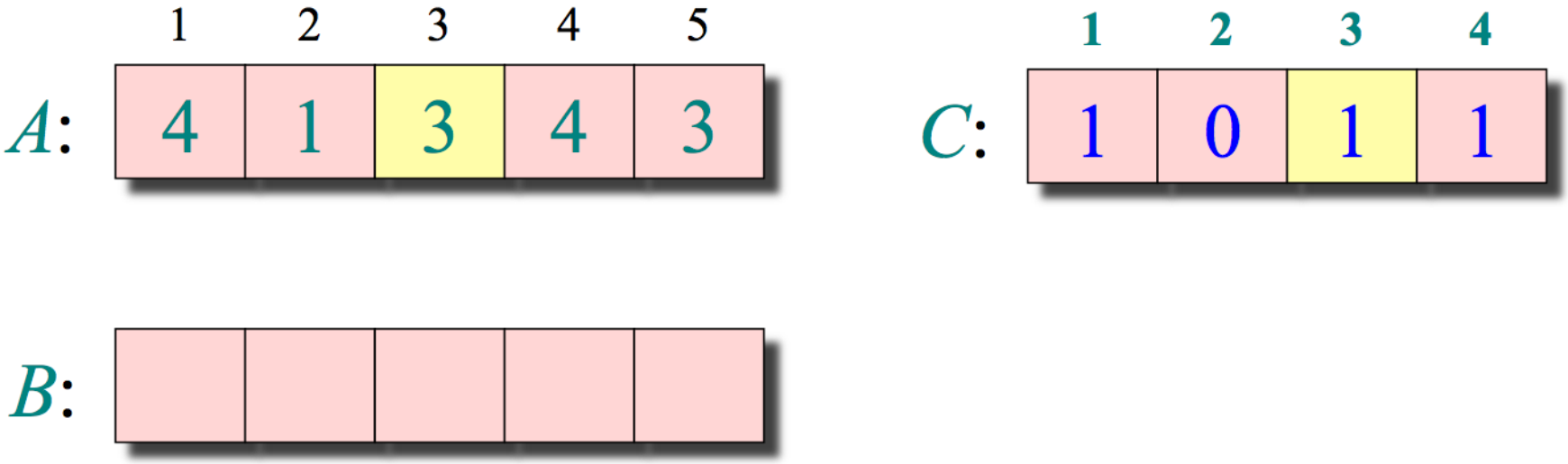
Loop 1: count



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

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

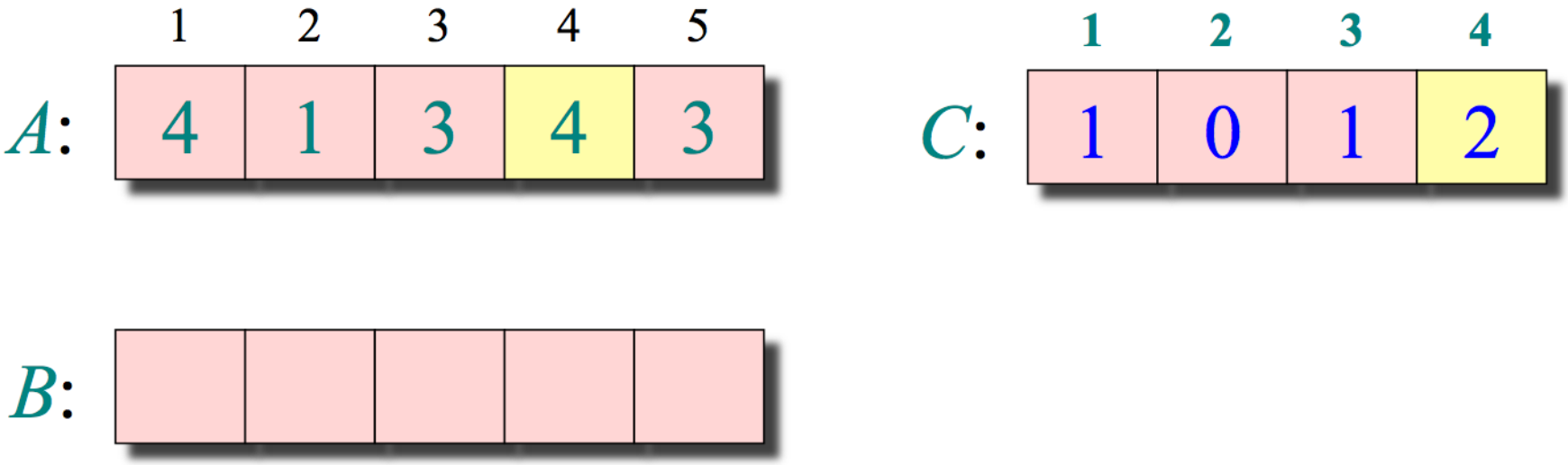
Loop 1: count



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

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

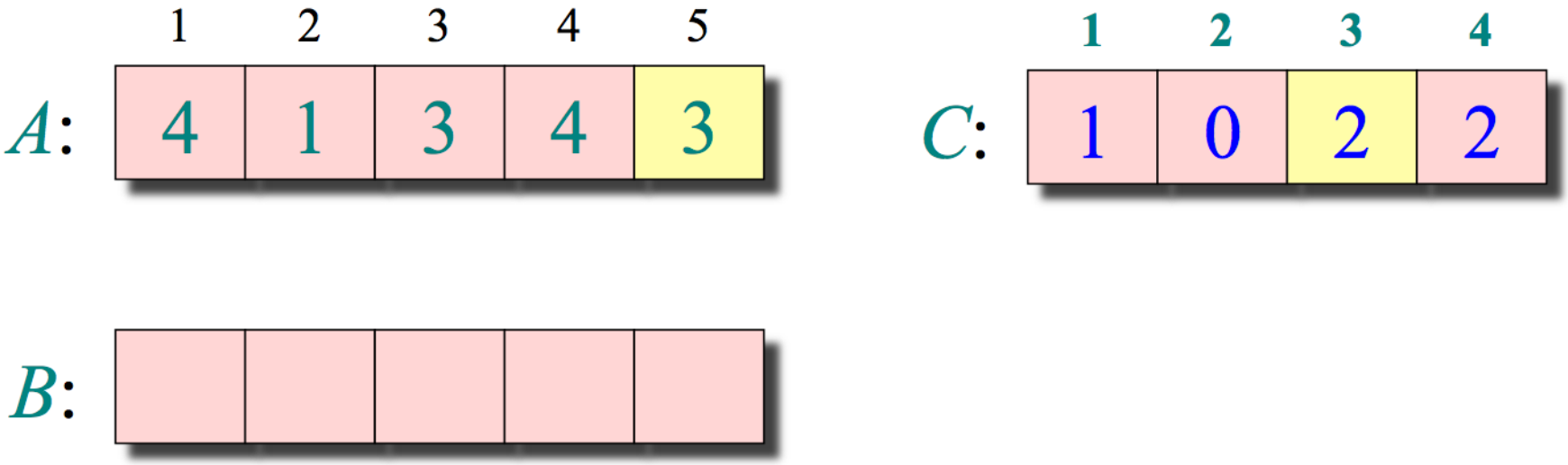
Loop 1: count



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

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

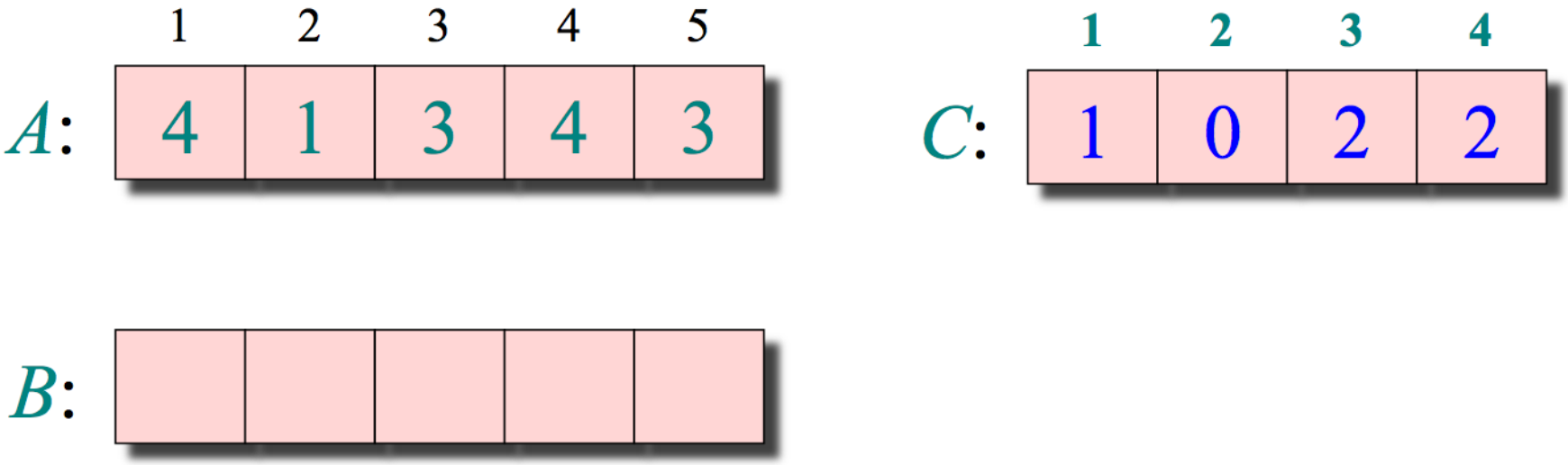
Loop 1: count



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

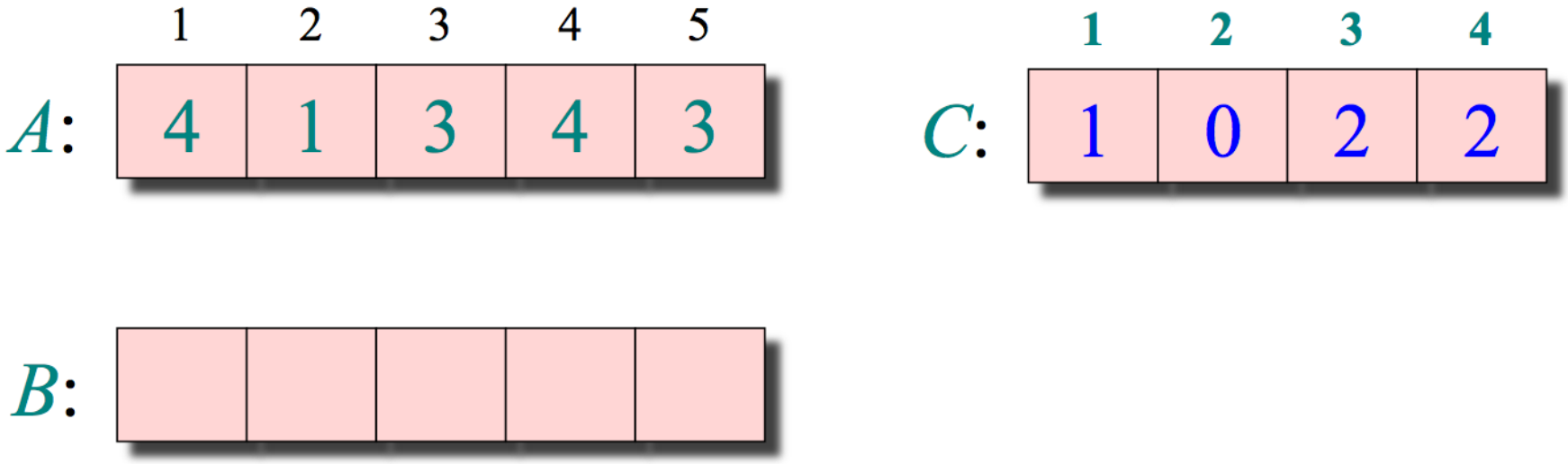
do $C[A[j]] \leftarrow C[A[j]] + 1$ ▷ $C[i] = |\{\text{key} = i\}|$

Loop 1: count

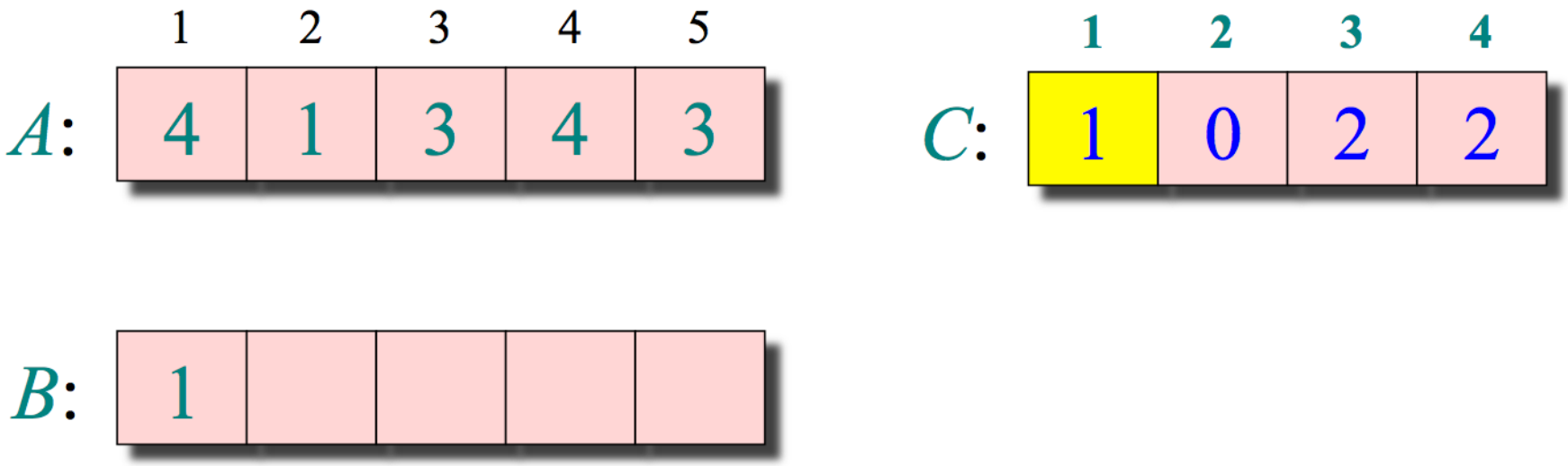


for $j \leftarrow 1$ **to** n

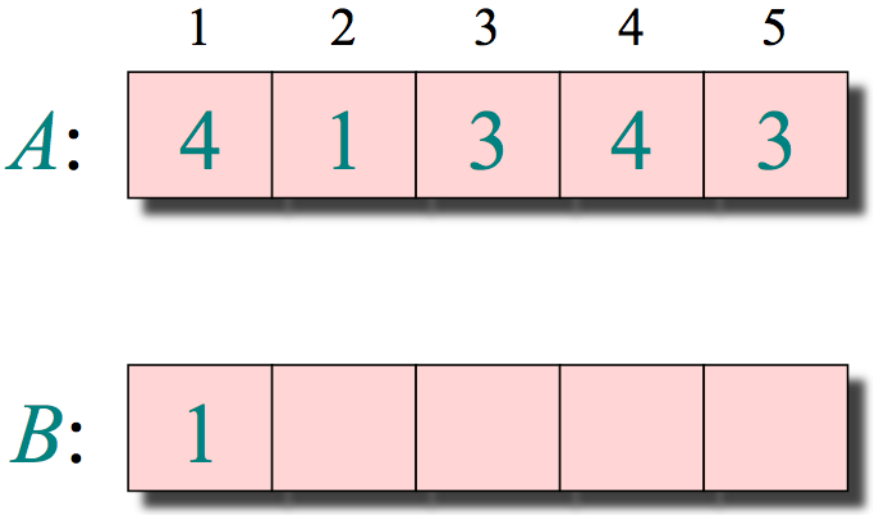
do $C[A[j]] \leftarrow C[A[j]] + 1$ ▷ $C[i] = |\{\text{key} = i\}|$

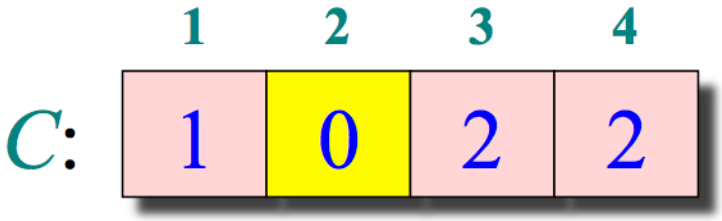

1 2 3 4 5
A:
4 1 3 4 3
1 2 3 4
C:
1 0 2 2
B:

Actually, we're done…

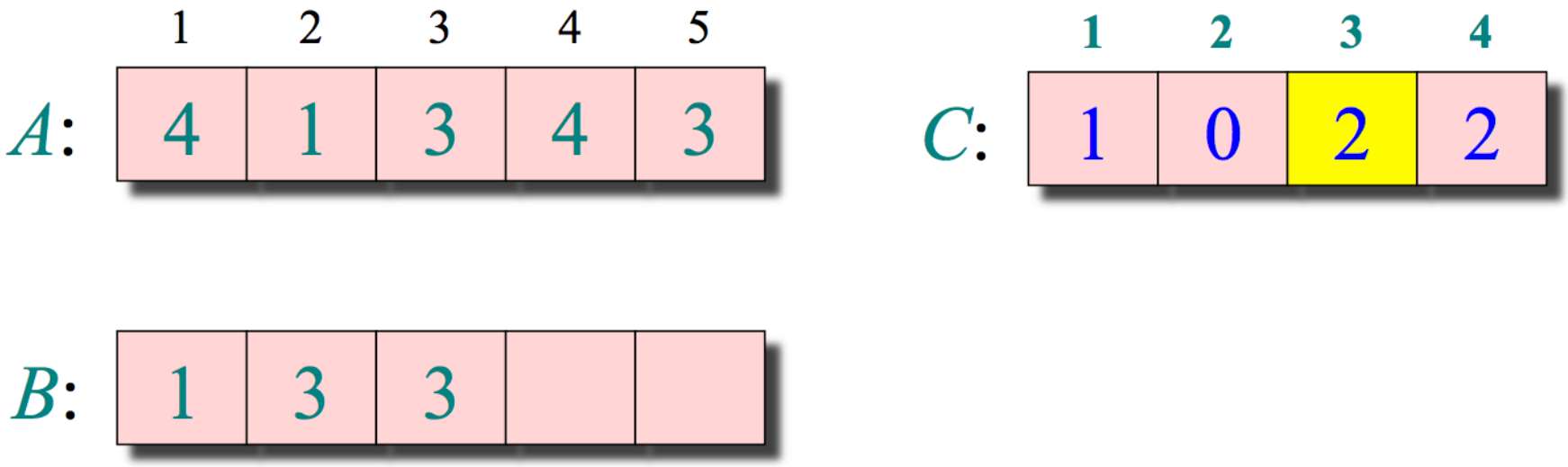


Actually, we're done…

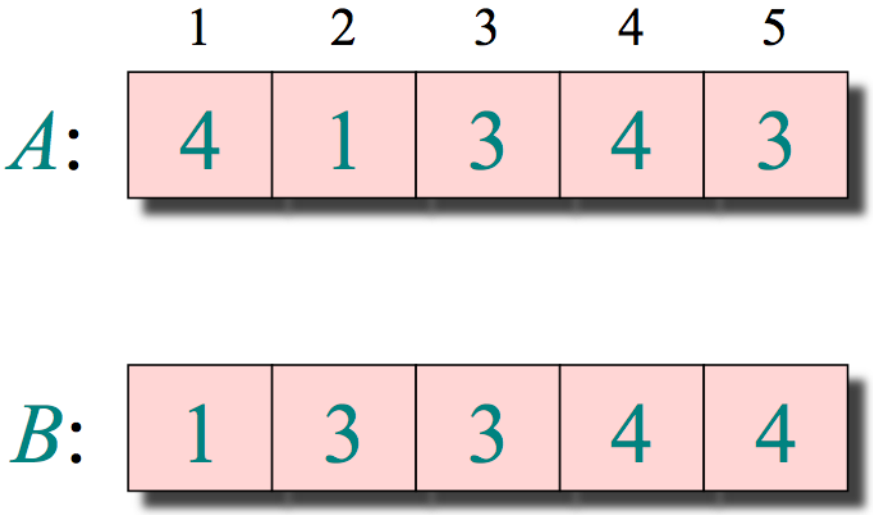


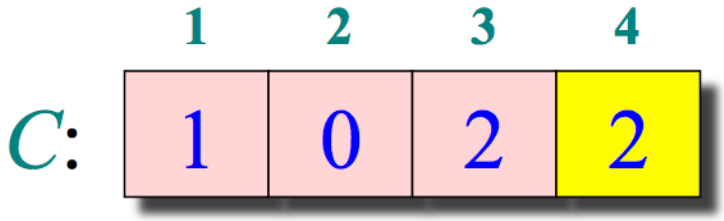


Actually, we're done…



Actually, we're done…

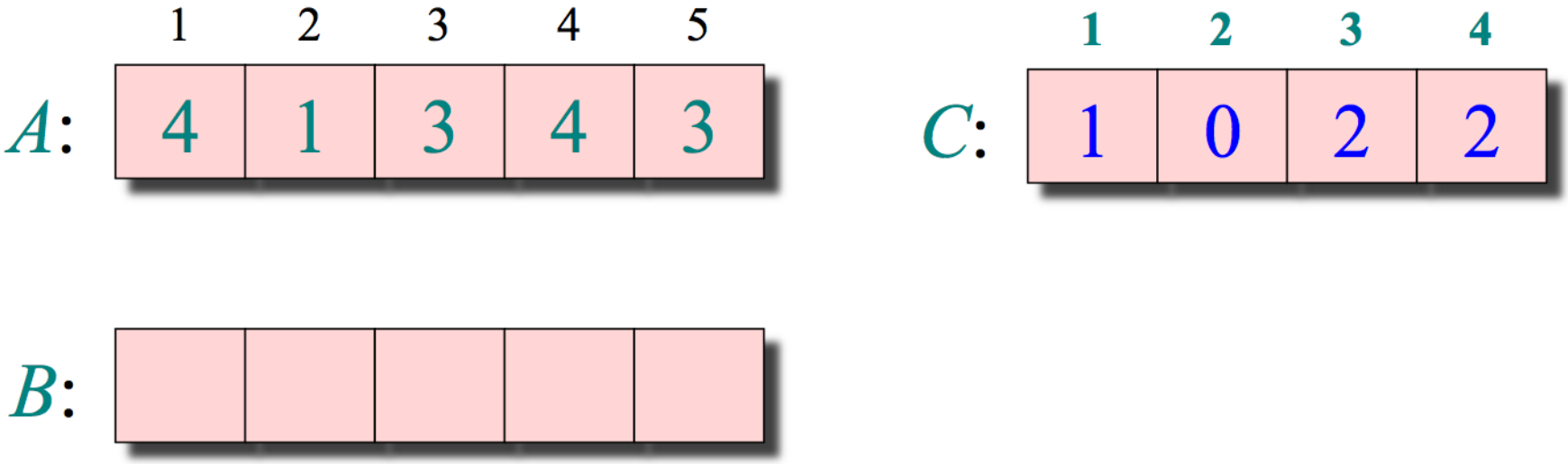




B is sorted!

What about the permutation π transforming A to B?

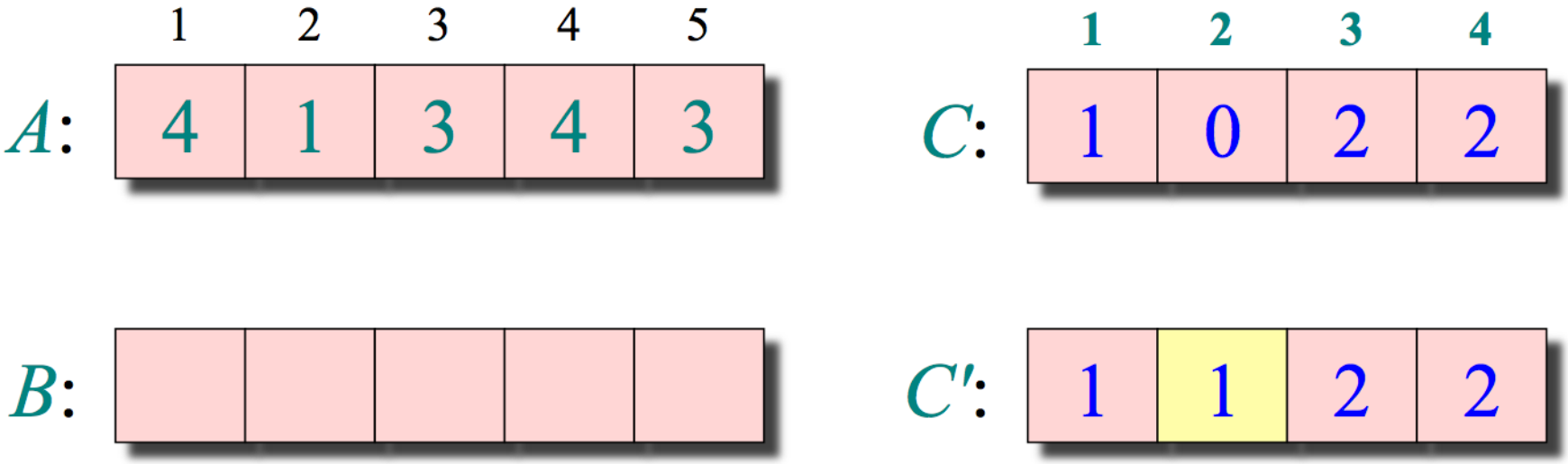
Loop 2: cumulative count



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

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

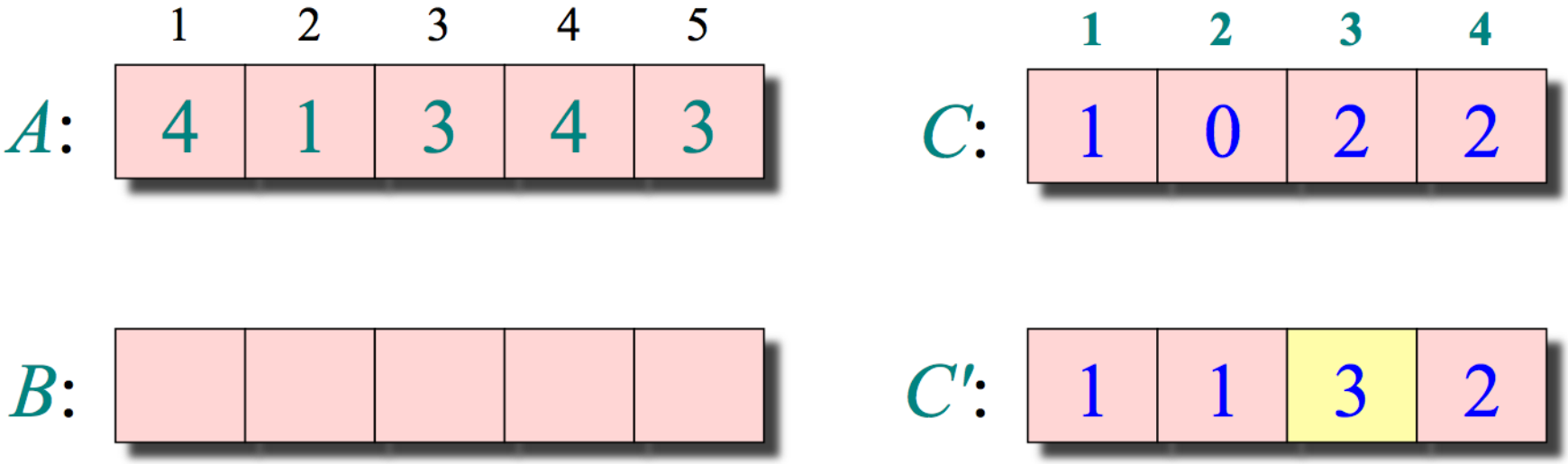
Loop 2: cumulative count



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

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

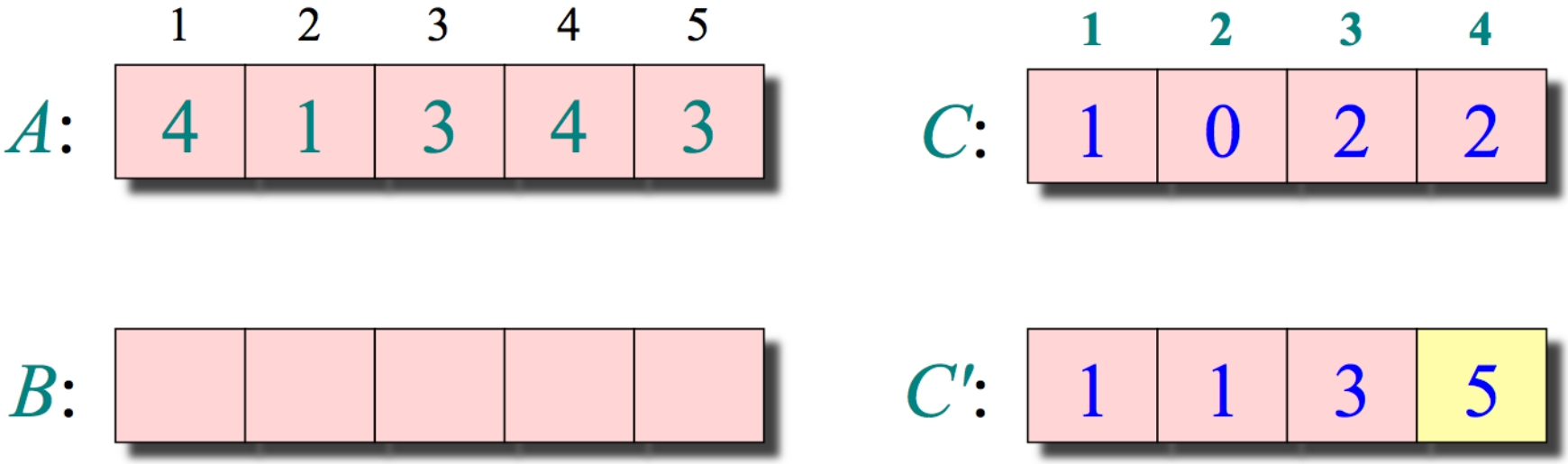
Loop 2: cumulative count



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

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

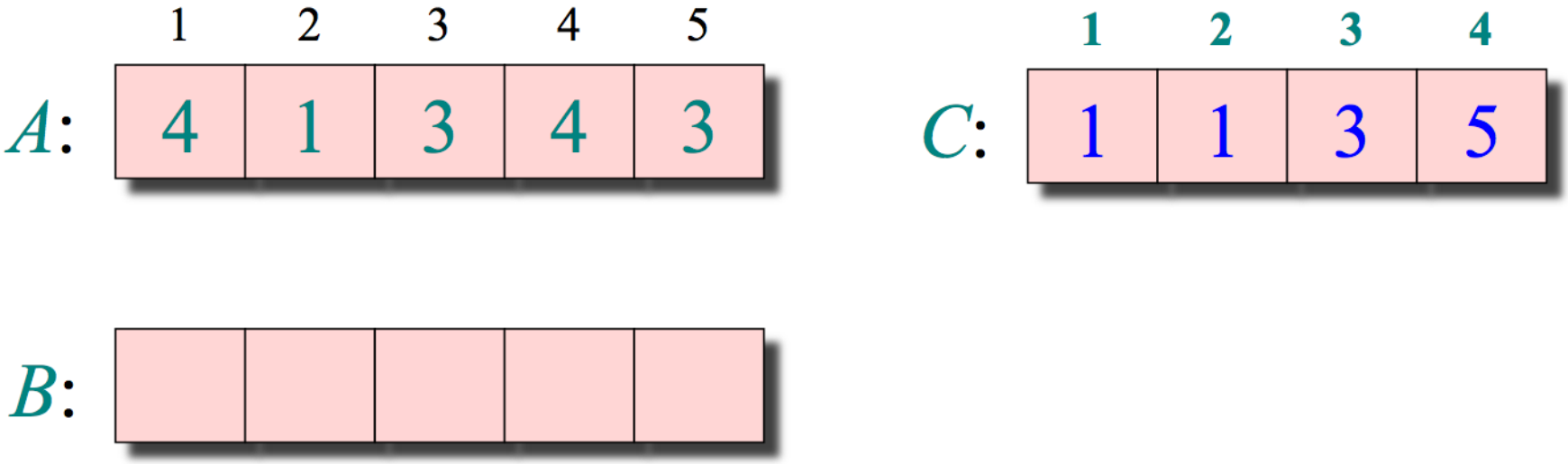
Loop 2: cumulative count



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

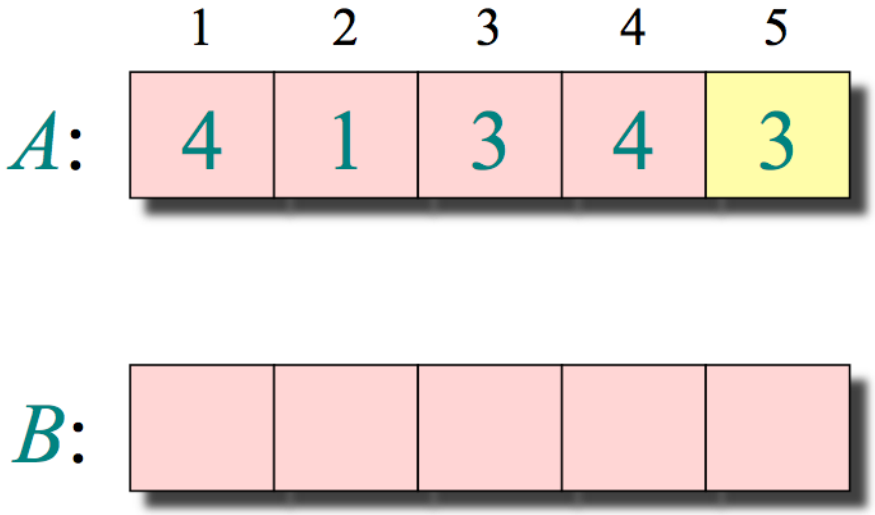
do $C[i] \leftarrow C[i] + C[i-1]$ ▷ $C[i] = |\{\text{key} \le i\}|$

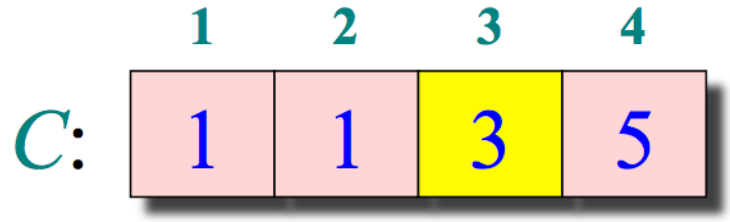
Loop 3: permute the elements of A



for $j \leftarrow n$ **downto** 1
  do $B[C[A[j]]] \leftarrow A[j]$
    $C[A[j]] \leftarrow C[A[j]] - 1$

Loop 3: permute the elements of A

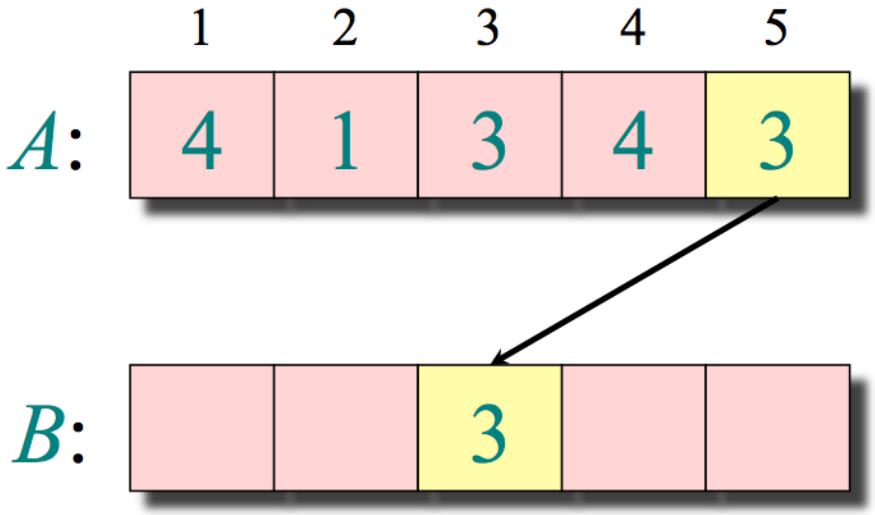


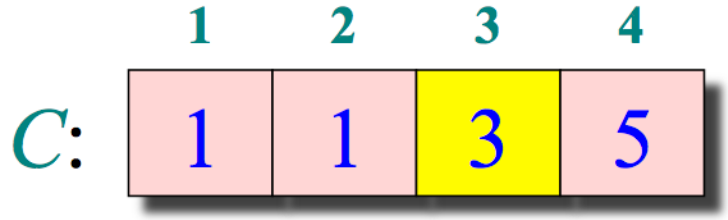


There are exactly 3 elements ≤A[5]; so where should I place A[5]?

for $j \leftarrow n$ **downto** 1
do $B[C[A[j]]] \leftarrow A[j]$
$C[A[j]] \leftarrow C[A[j]] - 1$

Loop 3: permute the elements of A

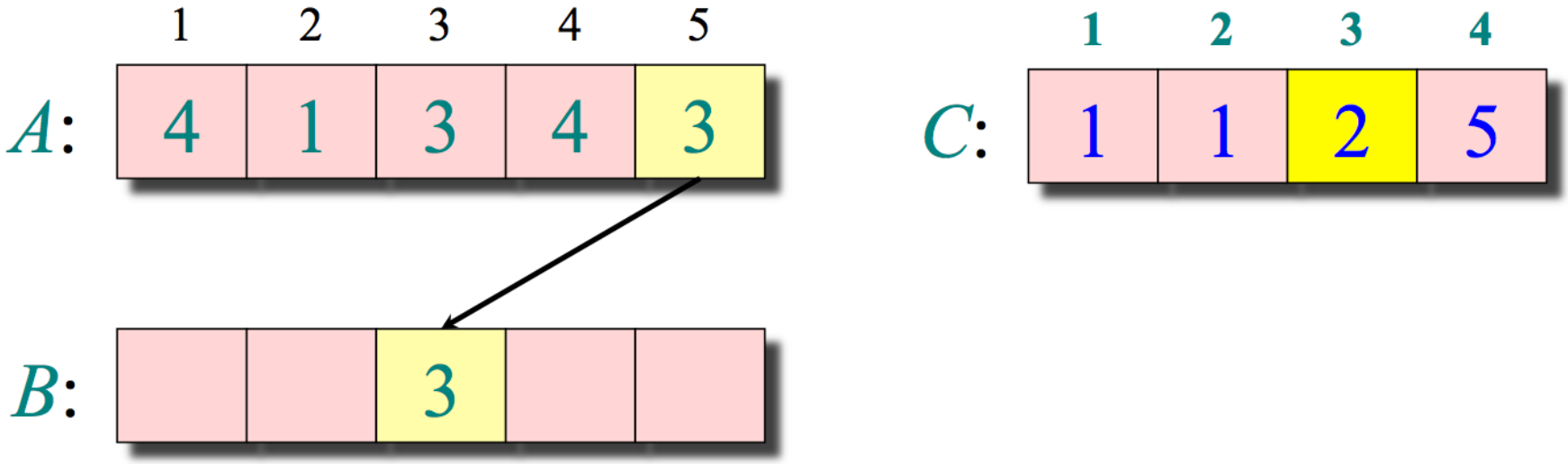




Used-up one 3; update counter.

for $j \leftarrow n$ **downto** 1
do $B[C[A[j]]] \leftarrow A[j]$
$C[A[j]] \leftarrow C[A[j]] - 1$

Loop 3: permute the elements of A

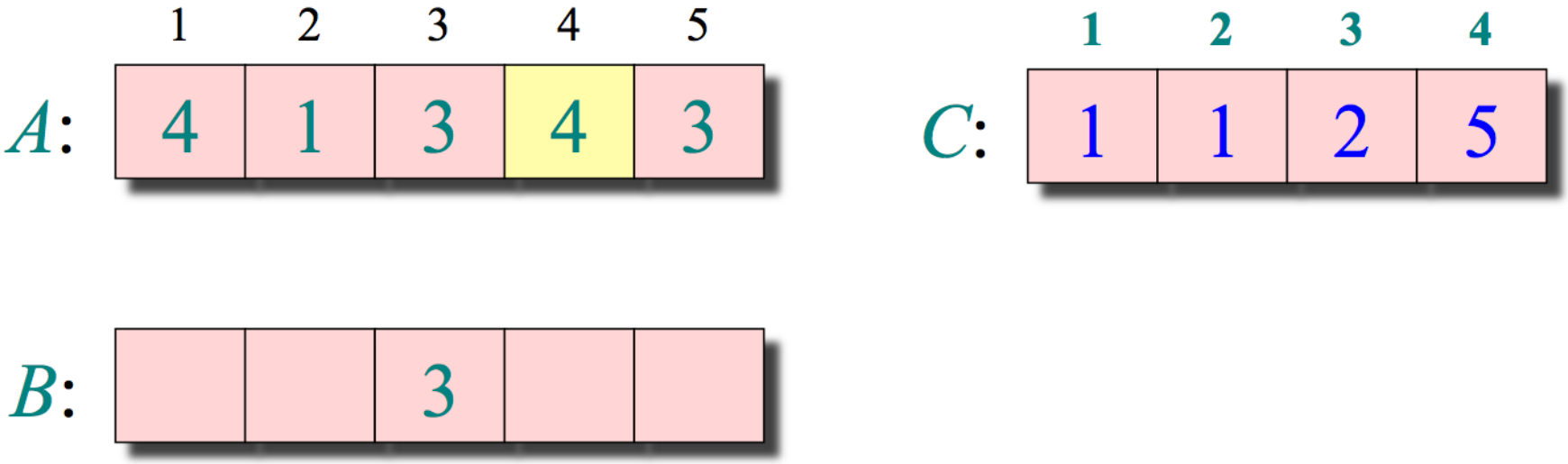


for $j \leftarrow n$ **downto** 1

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

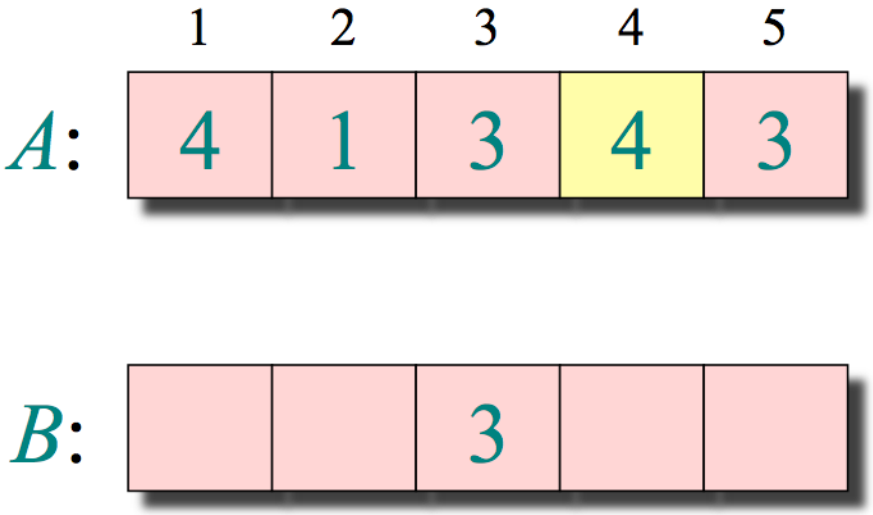
$C[A[j]] \leftarrow C[A[j]] - 1$

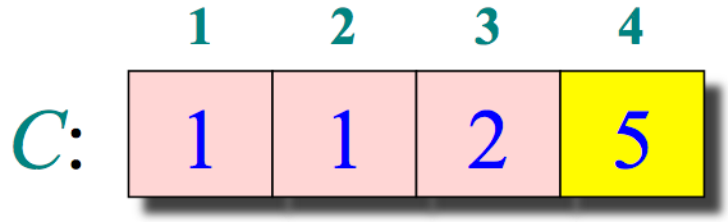
Loop 3: permute the elements of A



for $j \leftarrow n$ **downto** 1
  do $B[C[A[j]]] \leftarrow A[j]$
    $C[A[j]] \leftarrow C[A[j]] - 1$

Loop 3: permute the elements of A

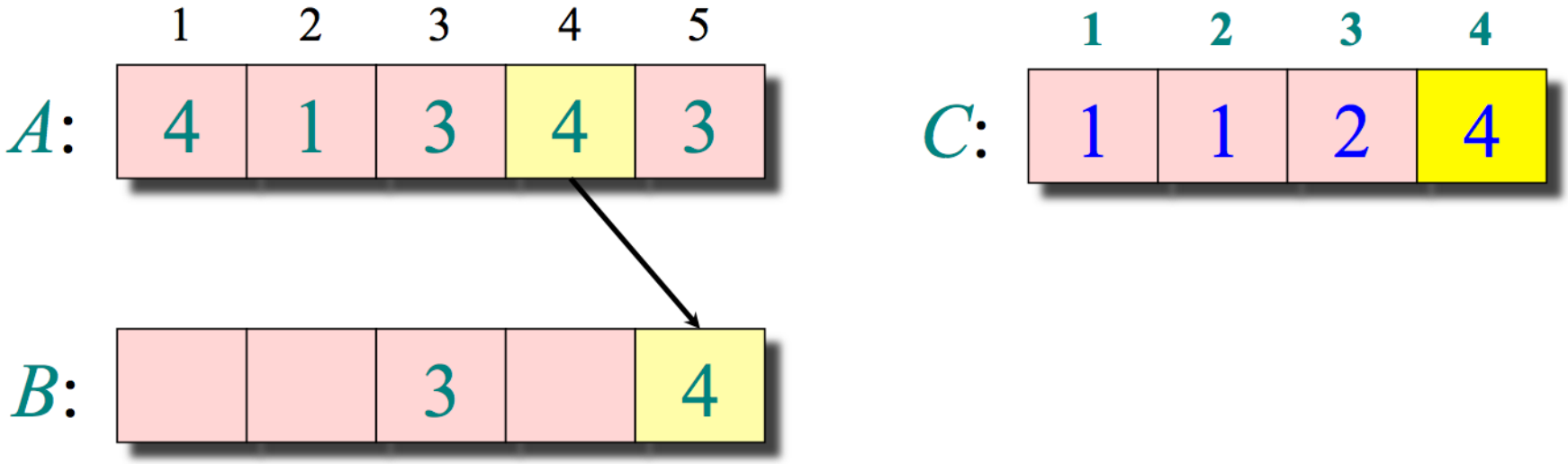




There are exactly 5 elements ≤A[4], so where should I place A[4]?

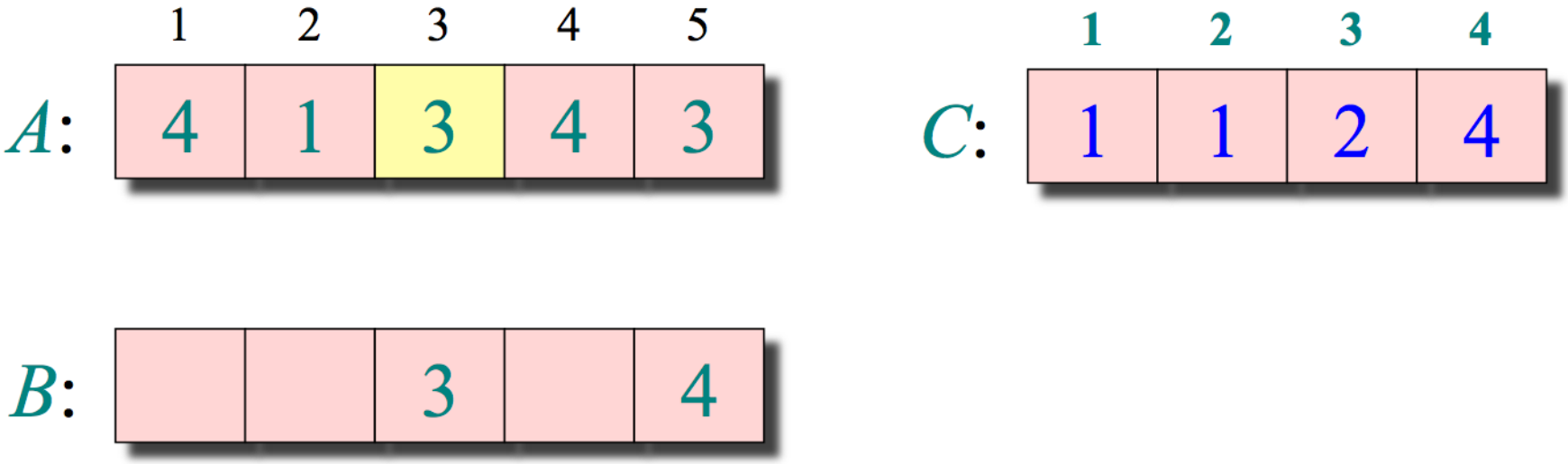
for $j \leftarrow n$ **downto** 1
  do $B[C[A[j]]] \leftarrow A[j]$
    $C[A[j]] \leftarrow C[A[j]] - 1$

Loop 3: permute the elements of A



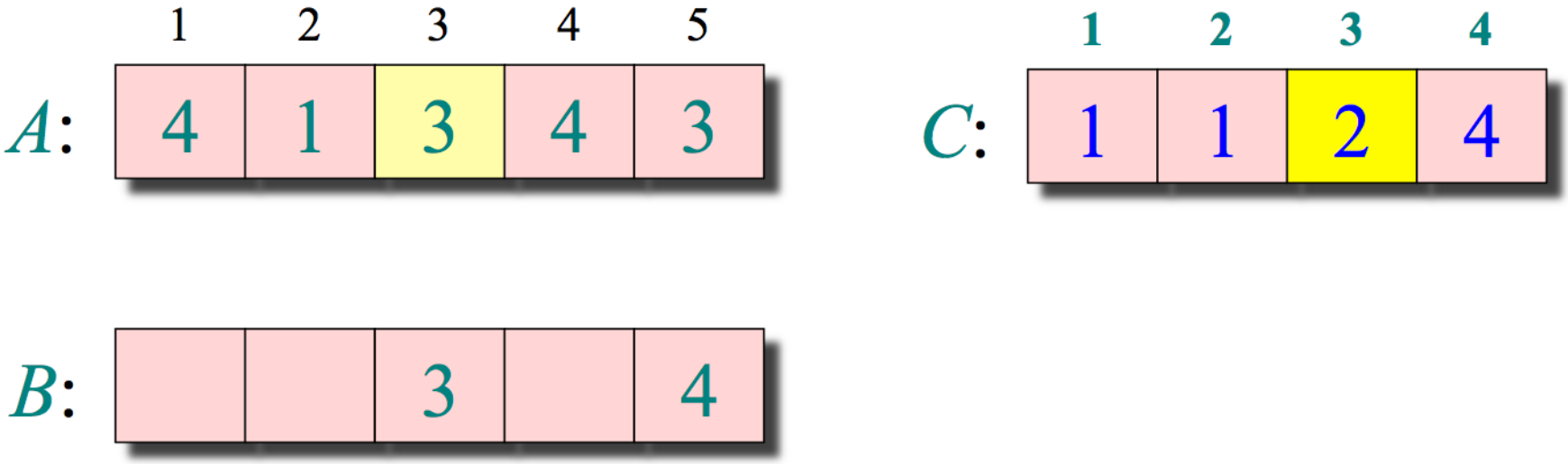
for $j \leftarrow n$ **downto** 1
  do $B[C[A[j]]] \leftarrow A[j]$
    $C[A[j]] \leftarrow C[A[j]] - 1$

Loop 3: permute the elements of A



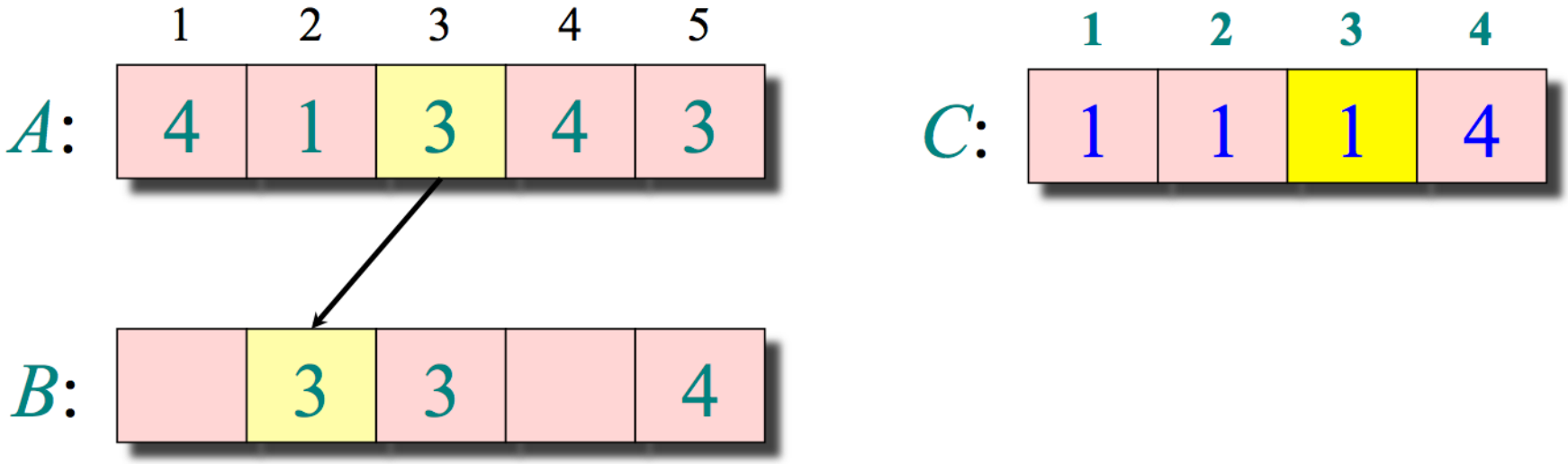
for $j \leftarrow n$ **downto** 1
  do $B[C[A[j]]] \leftarrow A[j]$
    $C[A[j]] \leftarrow C[A[j]] - 1$

Loop 3: permute the elements of A



for $j \leftarrow n$ **downto** 1
  do $B[C[A[j]]] \leftarrow A[j]$
    $C[A[j]] \leftarrow C[A[j]] - 1$

Loop 3: permute the elements of A

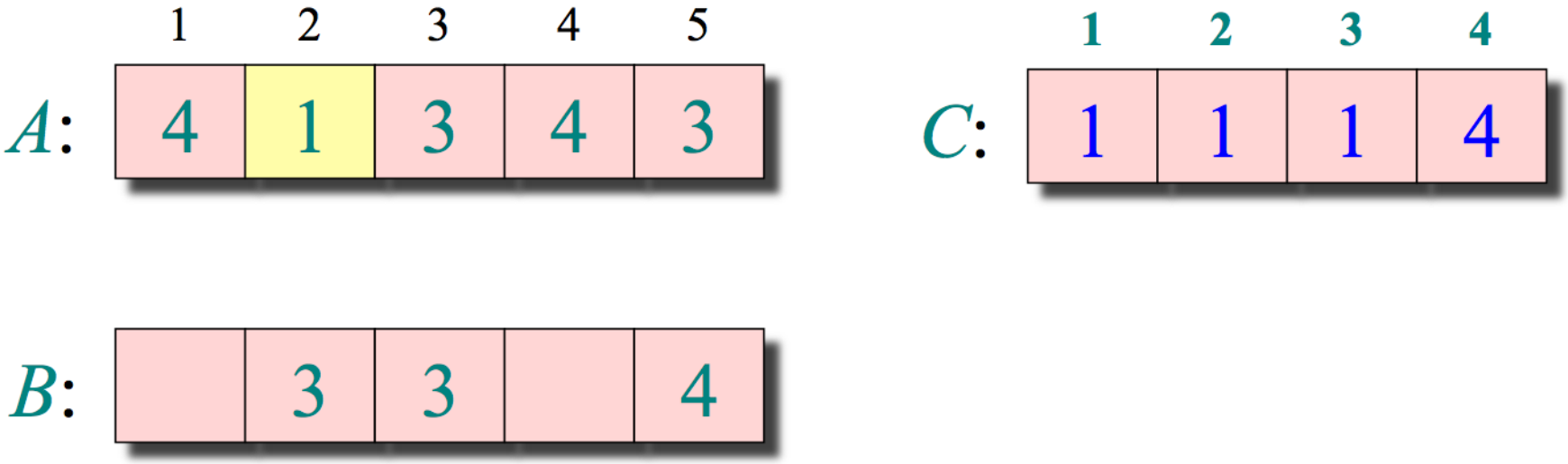


for $j \leftarrow n$ **downto** 1

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

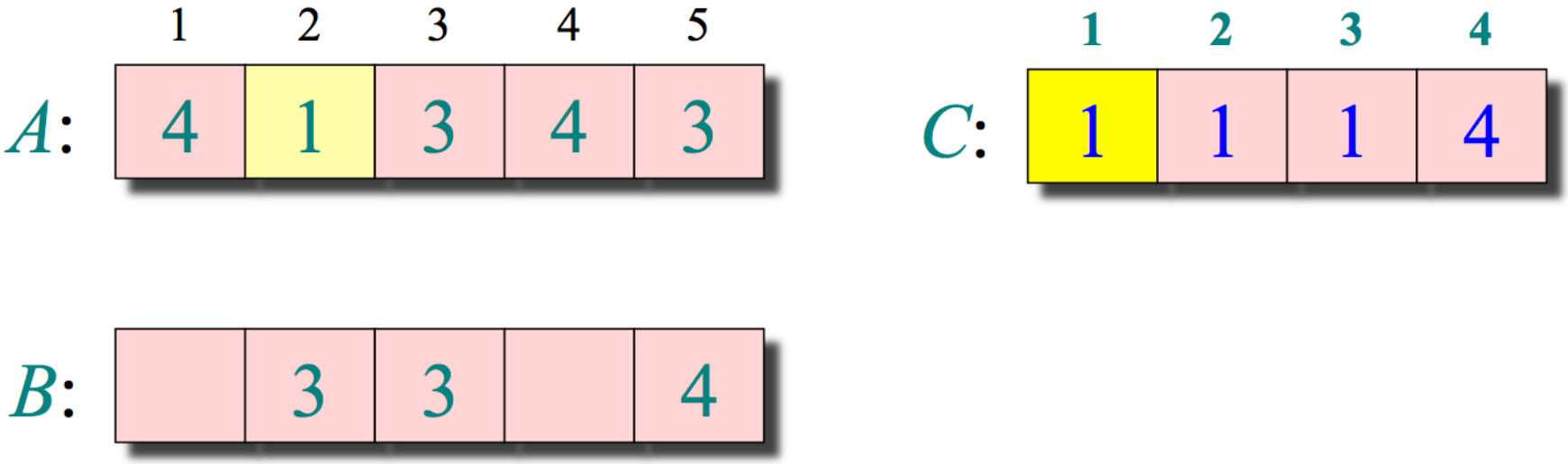
$C[A[j]] \leftarrow C[A[j]] - 1$

Loop 3: permute the elements of A



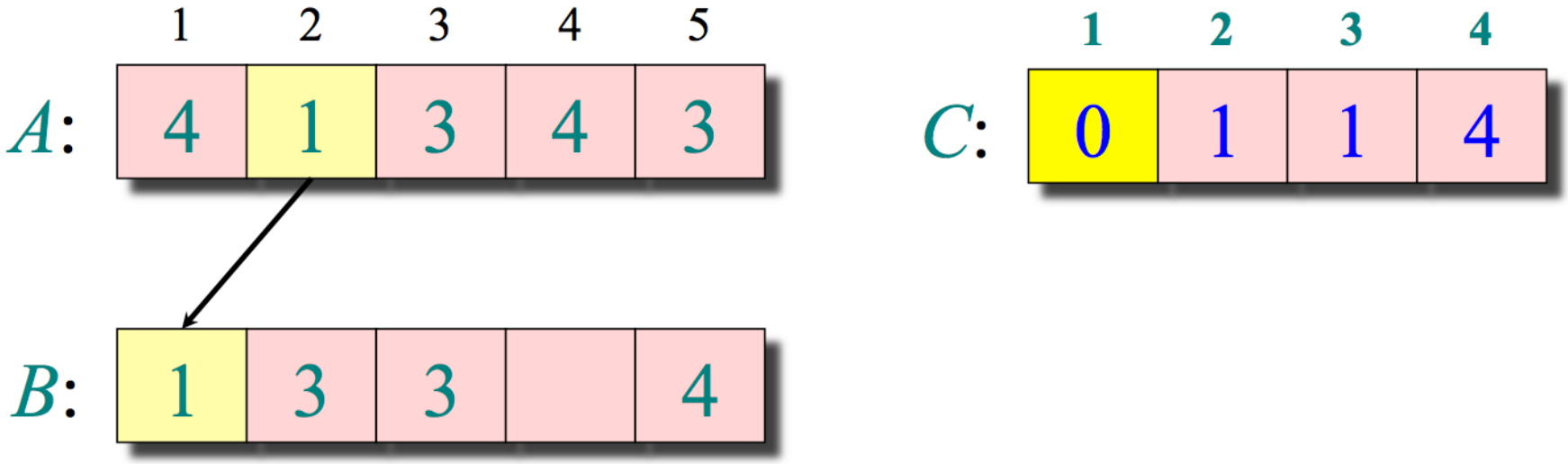
for $j \leftarrow n$ **downto** 1
  do $B[C[A[j]]] \leftarrow A[j]$
    $C[A[j]] \leftarrow C[A[j]] - 1$

Loop 3: permute the elements of A



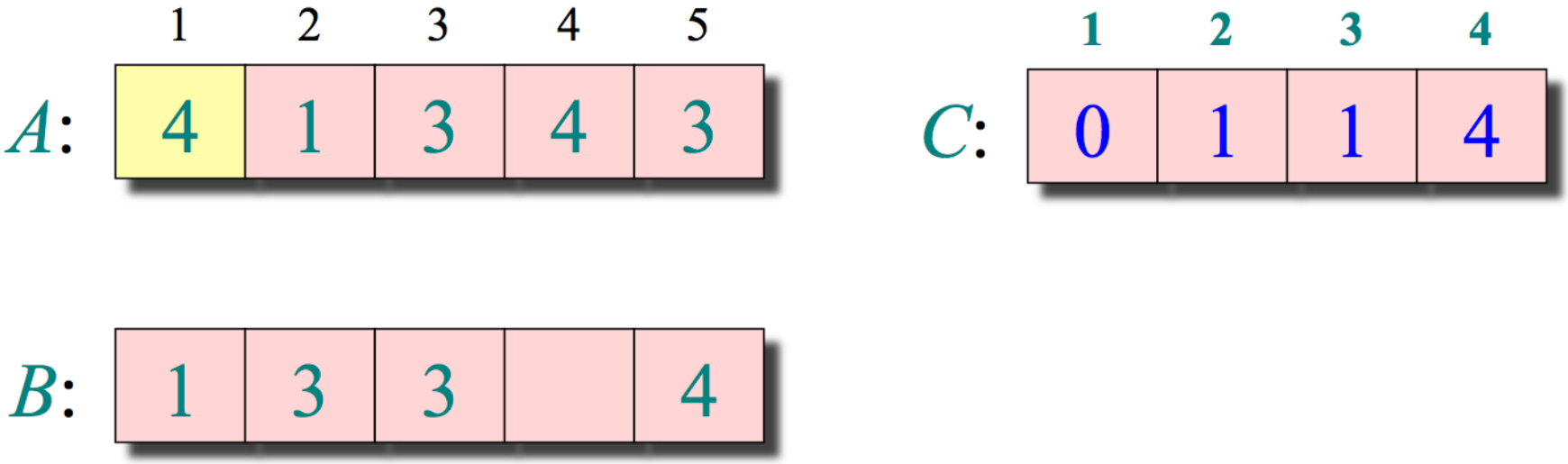
for $j \leftarrow n$ **downto** 1
  do $B[C[A[j]]] \leftarrow A[j]$
    $C[A[j]] \leftarrow C[A[j]] - 1$

Loop 3: permute the elements of A



for $j \leftarrow n$ **downto** 1
    do $B[C[A[j]]] \leftarrow A[j]$
        $C[A[j]] \leftarrow C[A[j]] - 1$

Loop 3: permute the elements of A

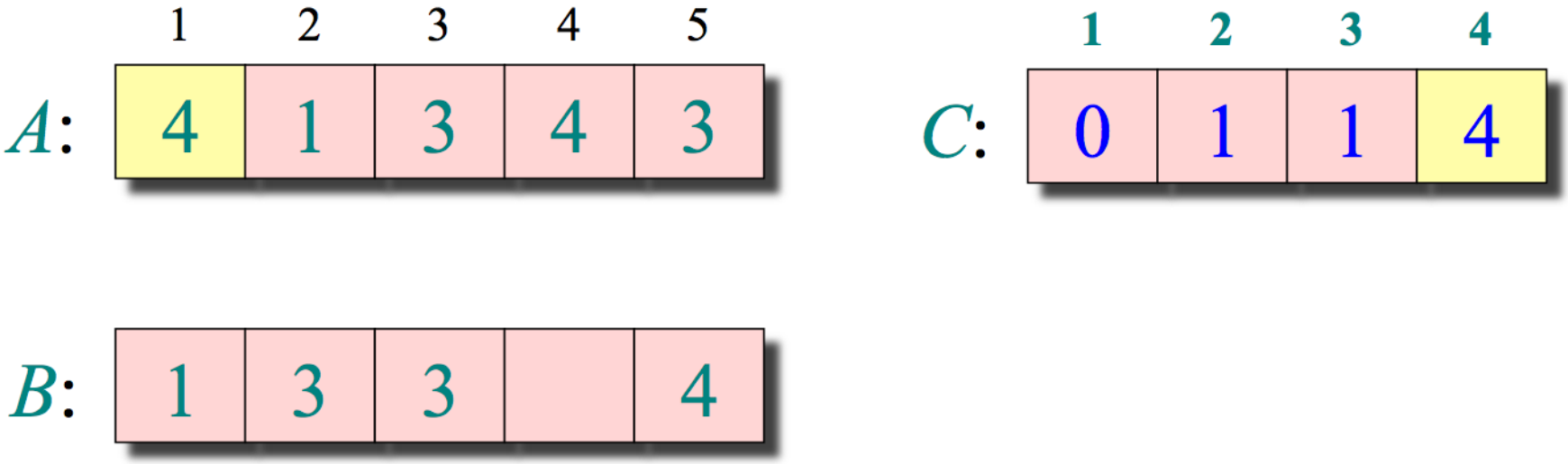


for $j \leftarrow n$ **downto** 1

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

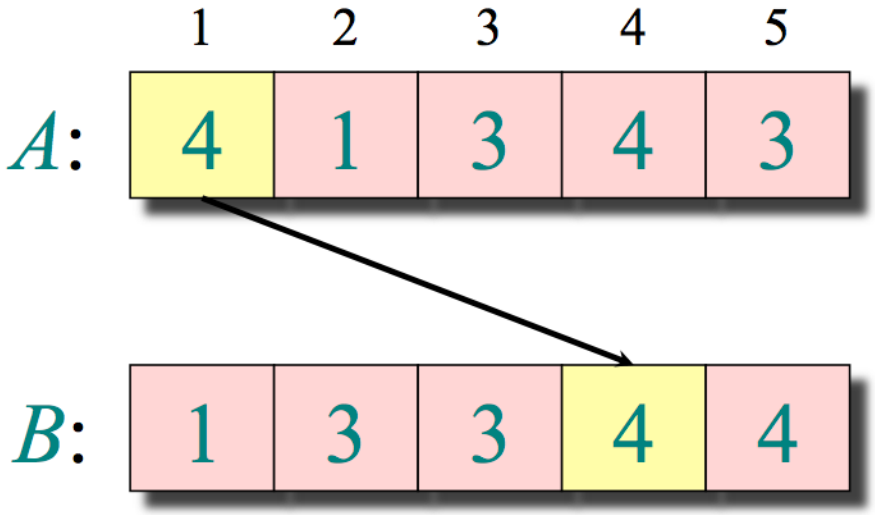
$C[A[j]] \leftarrow C[A[j]] - 1$

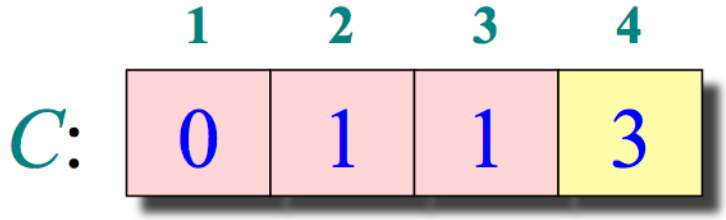
Loop 3: permute the elements of A



for $j \leftarrow n$ **downto** 1
  do $B[C[A[j]]] \leftarrow A[j]$
    $C[A[j]] \leftarrow C[A[j]] - 1$

Loop 3: permute the elements of A





Stable Sort!

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** $B[C[A[j]]] \leftarrow \mathrm{A}[j]$ $C[A[j]] \leftarrow C[A[j]] - 1$
$\Theta(n+k)$	

so $\Theta(n)$ if $k = O(n)$

Radix sort

Reichman University

- **Input** is a list of integers
- sort digit by digit, starting from the least significant
- use stable sort at each stage

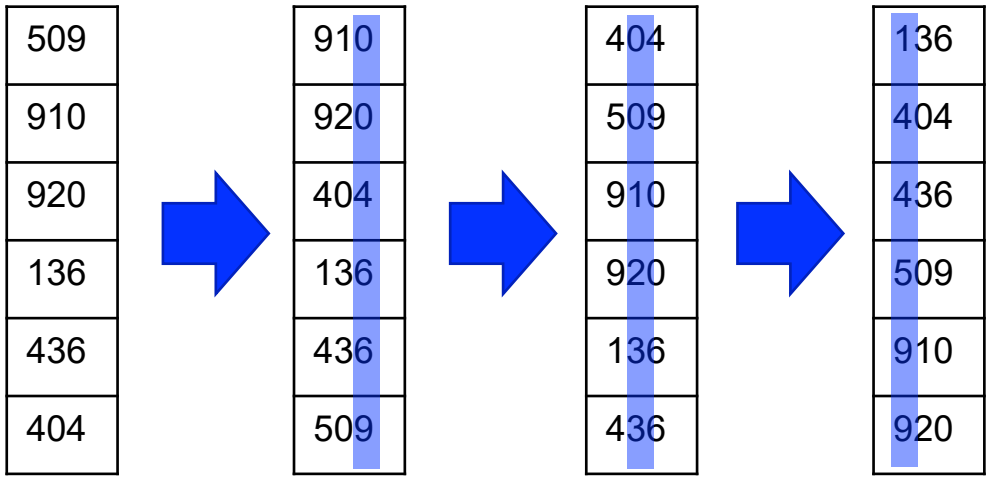
- for example, given n integers in $[0, n^3)$
- treat as 3-digit numbers (base n)
- sort each digit using counting sort in $O(n)$ time each
- total time is $3 \cdot O(n) = O(n)$

Radix sort

Reichman University

- The keys of the elements in the input list can be represented as a list of integers with exactly d digits at some base b
  - For example: binary (0-1), decimal (0-9), hexadecimal(0-9,a-f)
- The algorithm:

  Sorts in stages:
  - Starting from the least significant digit (LSD) up to the most significant digit (MSD)
  - At each stage apply a stable sort algorithm (such as count sort)



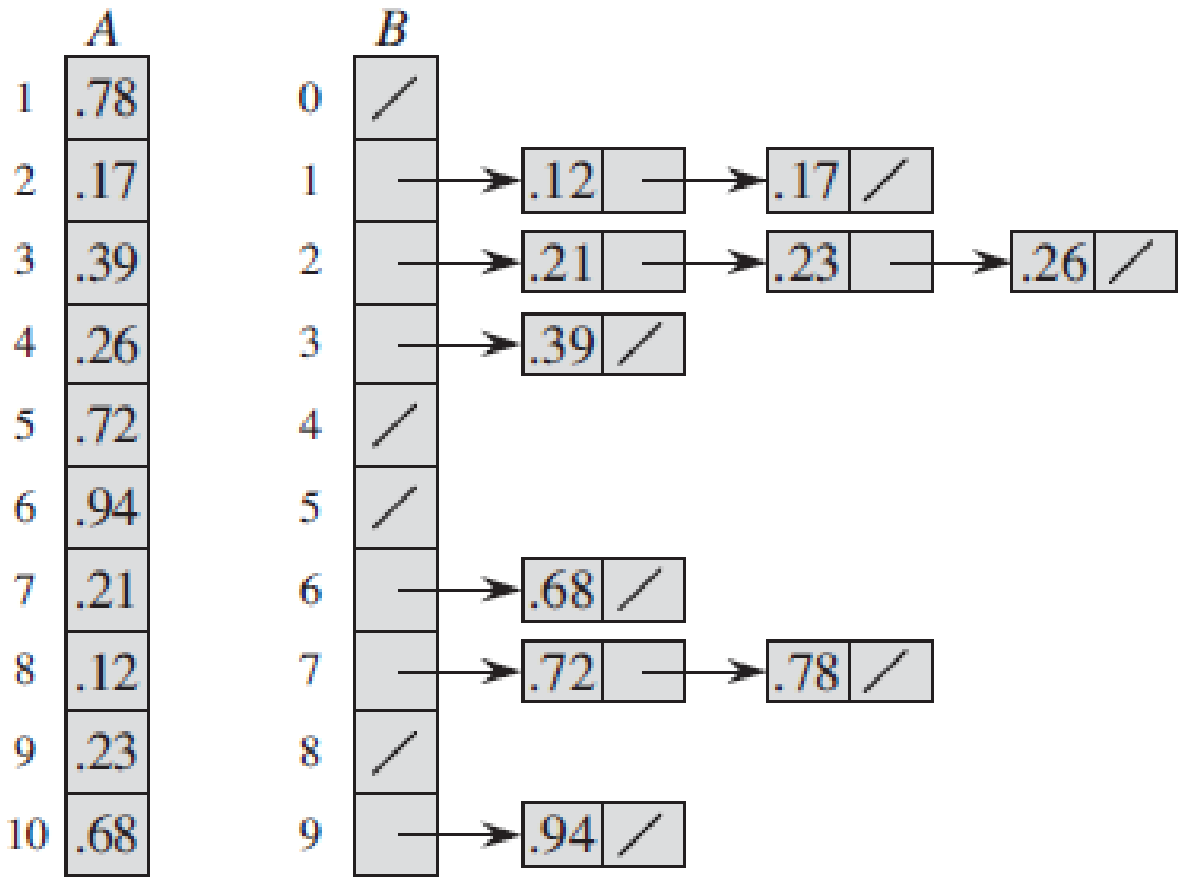
Bucket sort

Reichman University

- not comparison based
- **input:** array A[1…n] numbers uniformly distributed in [0,1)
- **output:** sorted list of elements

- create an array of n empty lists (buckets)
- for all i, insert element A[i] to list B[$\lfloor n$A[i]$\rfloor$]
- naively sort the elements in each list separately
- return the concatenation of all the lists

Bucket sort



(Figure from CLRS)

How can we make this sort stable?

Bucket sort: Complexity

Reichman University

- create an array of n empty lists (buckets) — $O(n)$ time
- for all i, insert element A[i] to list B[$\lfloor n\mathrm{A}[\mathrm{i}] \rfloor$] — $O(n)$ time
- naively sort the elements in each list separately — $\sum_{i=1}^{n} O(n_i^2)$ time
- return the concatenation of all the lists — $O(n)$ time

- worst case: all elements fall in a single bucket
- average case?

Summary

Reichman University

10 Sorting Algorithms Easily Explained:

Each sorting algorithm is explained in surface level, then the mathematical formula is explained and ending with a real life example.

https://www.youtube.com/watch?v=rbbTd-gkajw

Visualization of 15 Sorting Algorithms in 6 Minutes:

Sorts random shuffles of integers, with both speed and the number of items adapted to each algorithm's complexity.

https://www.youtube.com/watch?v=kPRA0W1kECg