INT102
Algorithmic Foundations And Problem Solving
Space and Time Tradeoffs

Dr Pengfei Fan
Department of Intelligent Science

西交利物浦大學
Xi'an Jiaotong-Liverpool University

Learning Outcomes

- The idea for space-for-time tradeoffs
- Two algorithms:
  1. Distribution counting sort
  2. Horspool's algorithm for string searching

Quiz:

Exchange **numeric values** of two variables *u* and *v*.

- Solution 1: using a temp variable.

  temp := u

  u := v

  v := temp

- Solution 2: without using a temp variable.

  u:=u-v

  v:=v+u

  u:= v-u

Space-for-time tradeoffs

Two types of space-for-time algorithms:

- *Input-enhancement* — preprocess the input (or its part) to store some info to be used later in solving the problem

- *Pre-structuring* — preprocess the input using a data structure to make accessing its elements easier

Space-for-time tradeoffs

- *Input enhancement* — preprocess the input (or its part) to store some info to be used later in solving the problem. Two algorithms:

    1. Distribution counting sort

    2. Horspool's algorithm for string searching

Counting sort

Sorting

- Input: a sequence of n numbers $a_0, a_1, \ldots, a_{n-1}$
- Output: arrange the n numbers into ascending order, i.e., from smallest to largest
- There are many sorting algorithms:
  - ✓Insertion sort, $O(n^2)$
  - ✓Selection sort, $O(n^2)$
  - ✓Bubble sort, $O(n^2)$
  - ✓Merge sort, $O(n\log n)$
  - ✓Quick sort, $O(n\log n)$

How fast can we sort?

All the sorting algorithms we have seen so far are ***comparison sorts*** : only use comparisons to determine the relative order of elements. *E.g.*, Selection sort, bubble sort, insertion sort, merge sort.

The best worst-case running time that we've seen for comparison sorting is $O(n \lg n)$.

Q:Is O(nlgn) the best we can do?
A: Yes, as long as we use comparison sorts

In fact, we can prove that any comparison sorting takes at least O(nlogn) time in the worst case.

Counting sort

Counting Sort: No comparisons between elements.

- *Input*: $A[1 \,..\, n]$, where $A[j] \in \{1, 2, \ldots, k\}$.
- *Output*: $B[1 \,..\, n]$, sorted.
- *Auxiliary storage*: $C[1 \,..\, k]$.

Counting sort

- How it works
- Analysis

- Positive Integers
- Counting "occurrences"

Counting sort

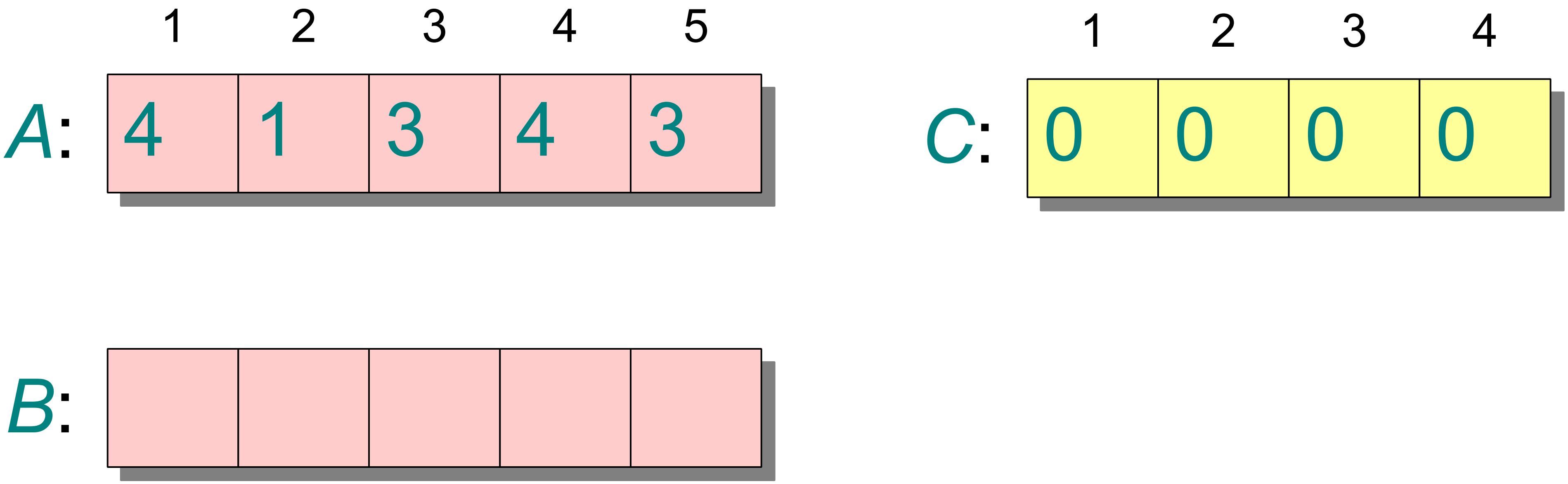
Counting-sort example

	1	2	3	4	5
A:	4	1	3	4	3

	1	2	3	4
C:				

B:					

Loop 1



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

Loop 2

	1	2	3	4	5
A:	4	1	3	4	3

	1	2	3	4
C:	0	0	0	1

B:					

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

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

Loop 2

A:

1	2	3	4	5
4	1	3	4	3

C:

1	2	3	4
1	0	0	1

B:

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

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

Loop 2

A:

1	2	3	4	5
4	1	3	4	3

C:

1	2	3	4
1	0	1	1

B:

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

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

Loop 2

	1	2	3	4	5
A:	4	1	3	4	3

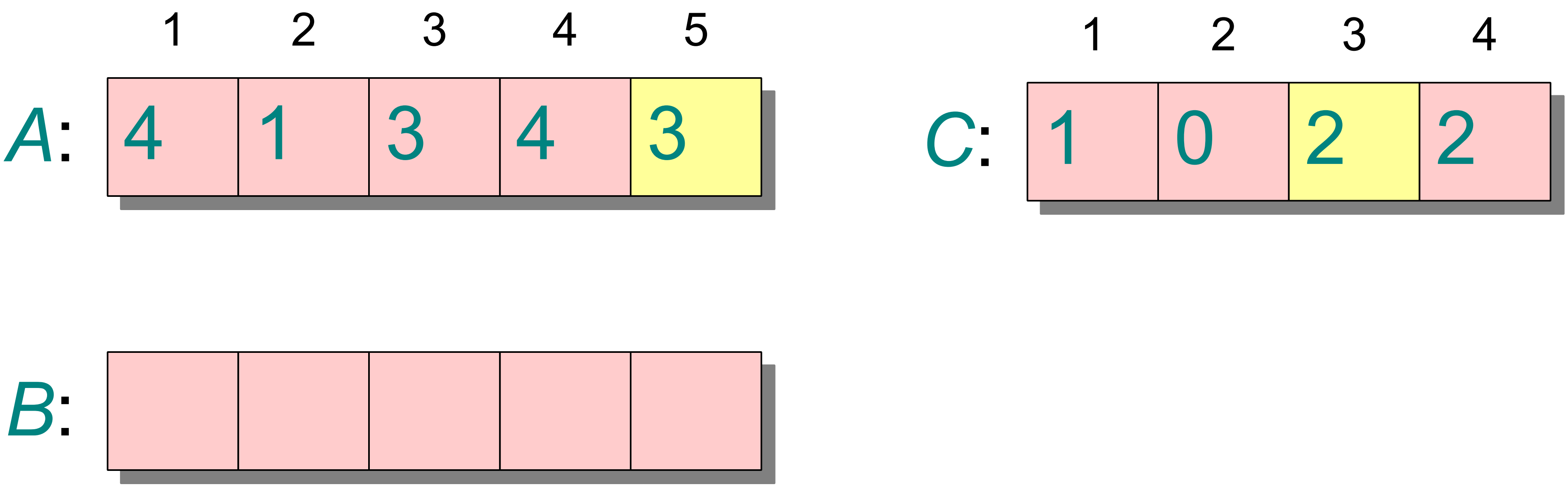
	1	2	3	4
C:	1	0	1	2

B:					

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

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

Loop 2



$\textbf{for } j \leftarrow 1 \textbf{ to } n$

$\quad \textbf{do } C[A[j]] \leftarrow C[A[j]] + 1 \qquad \rhd\ C[i] = |\{\text{key} = i\}|$

Loop 3

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

	1	2	3	4
C:	1	0	2	2
C:	1	1	2	2

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

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

Loop 3

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

	1	2	3	4
C:	1	0	2	2
C:	1	1	3	2

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

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

Loop 3

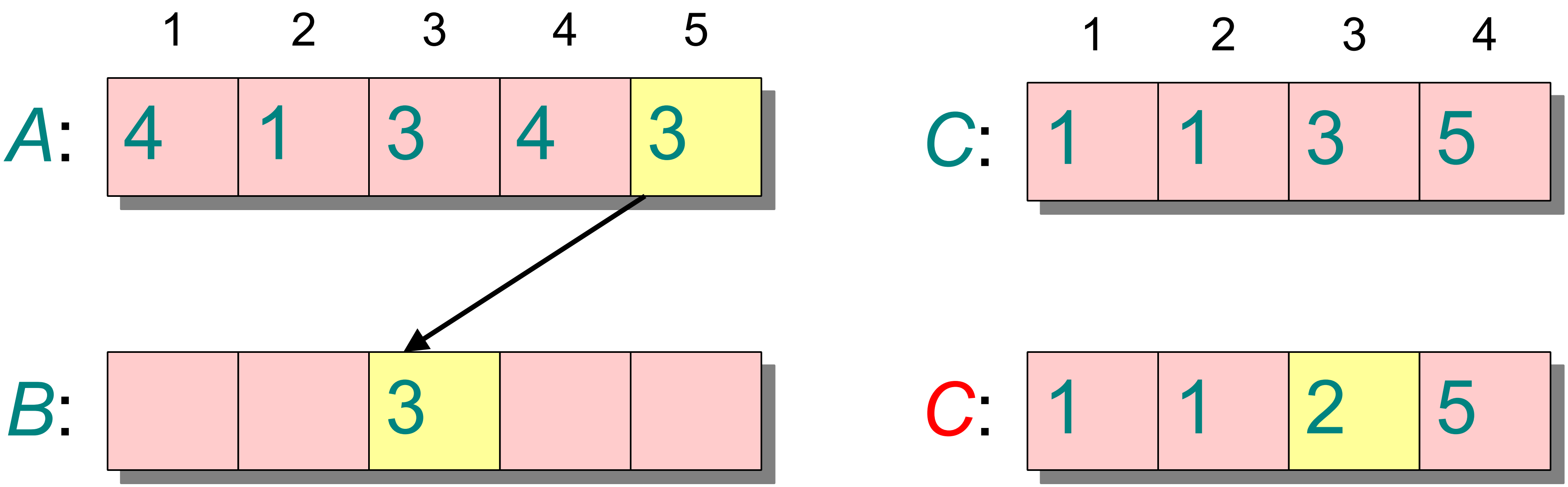
	1	2	3	4	5
A:	4	1	3	4	3
B:					

	1	2	3	4
C:	1	0	2	2
C':	1	1	3	5

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

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

Loop 4

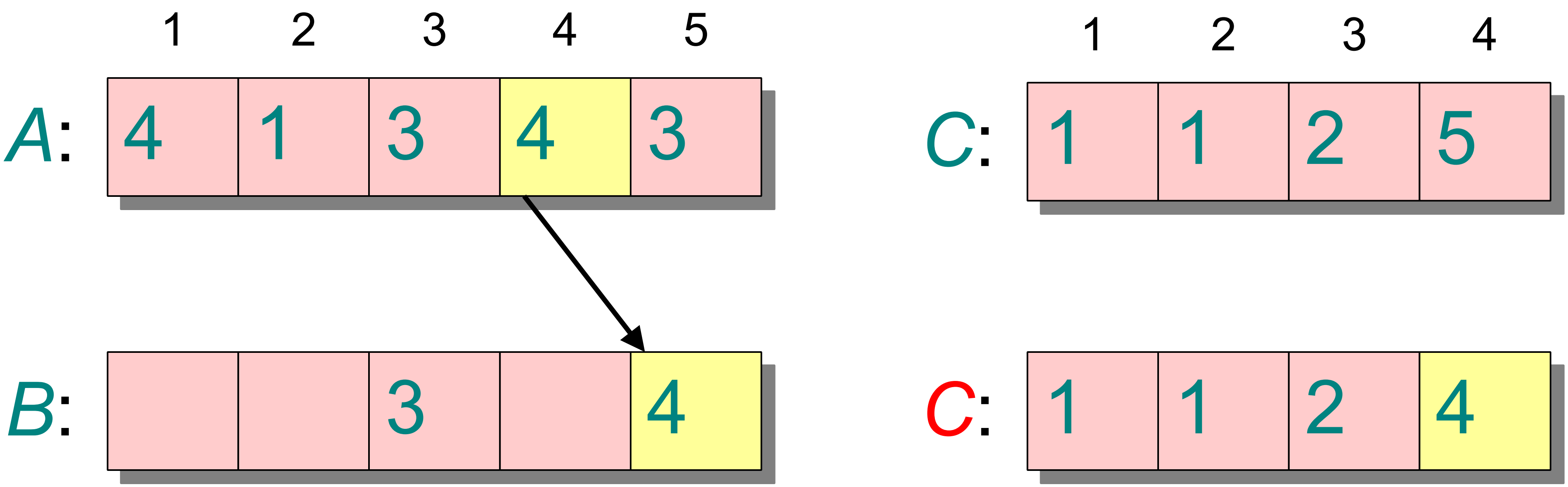


for $j \leftarrow n$ **downto** 1

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

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

Loop 4

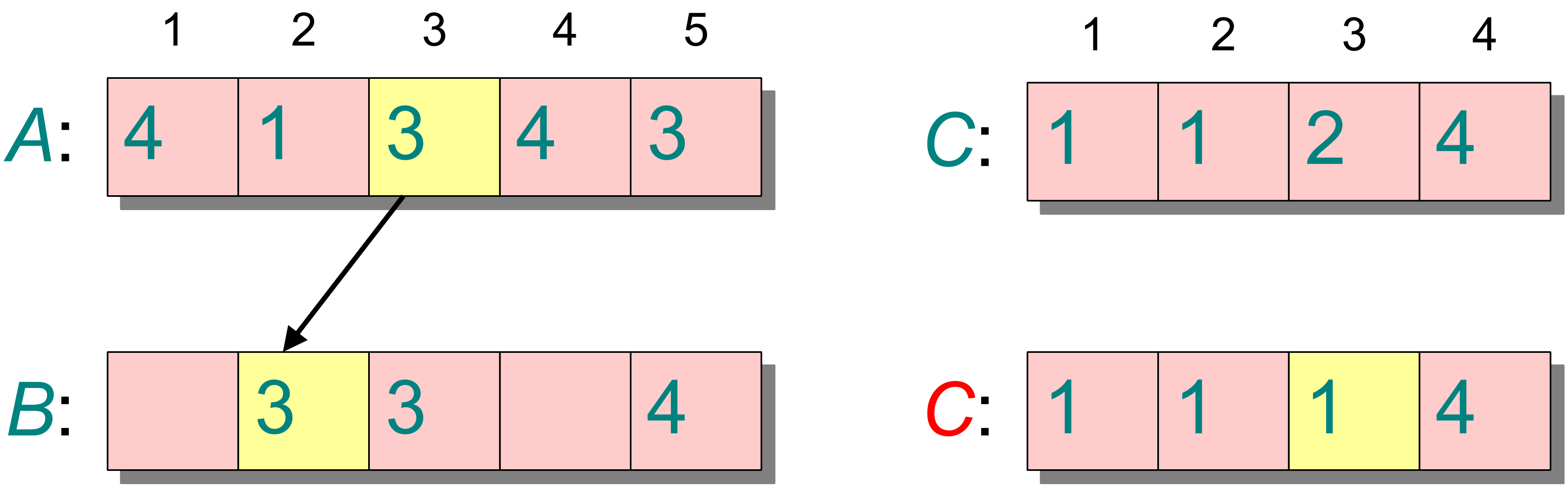


for $j \leftarrow n$ **downto** 1

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

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

Loop 4

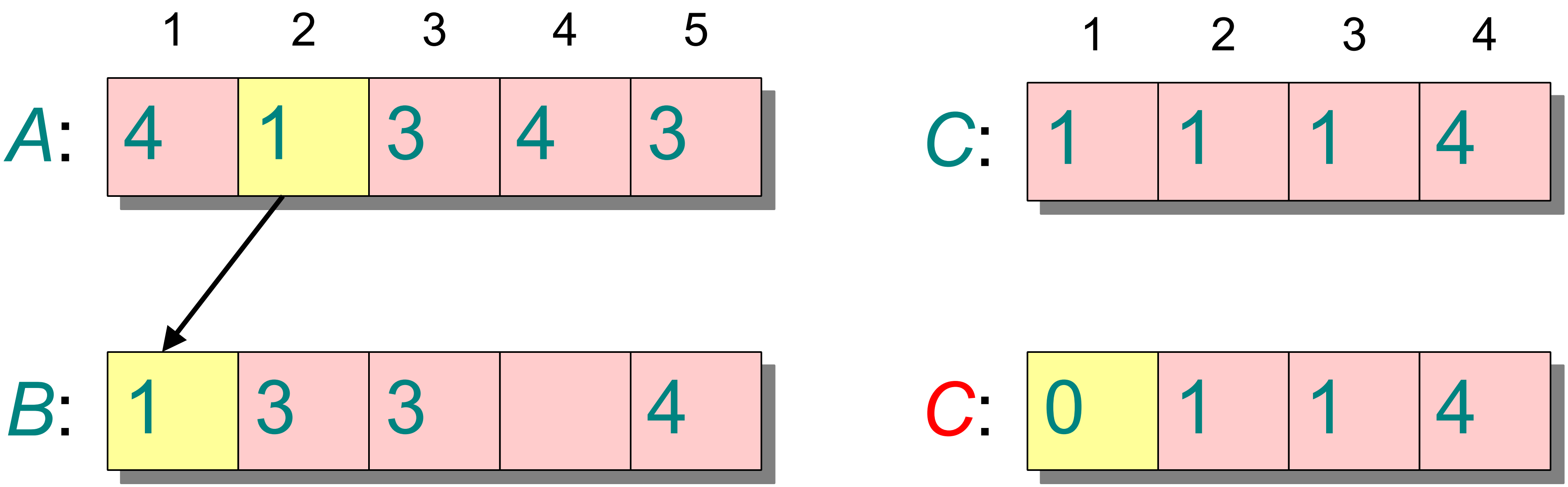


for $j \leftarrow n$ **downto** 1

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

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

Loop 4

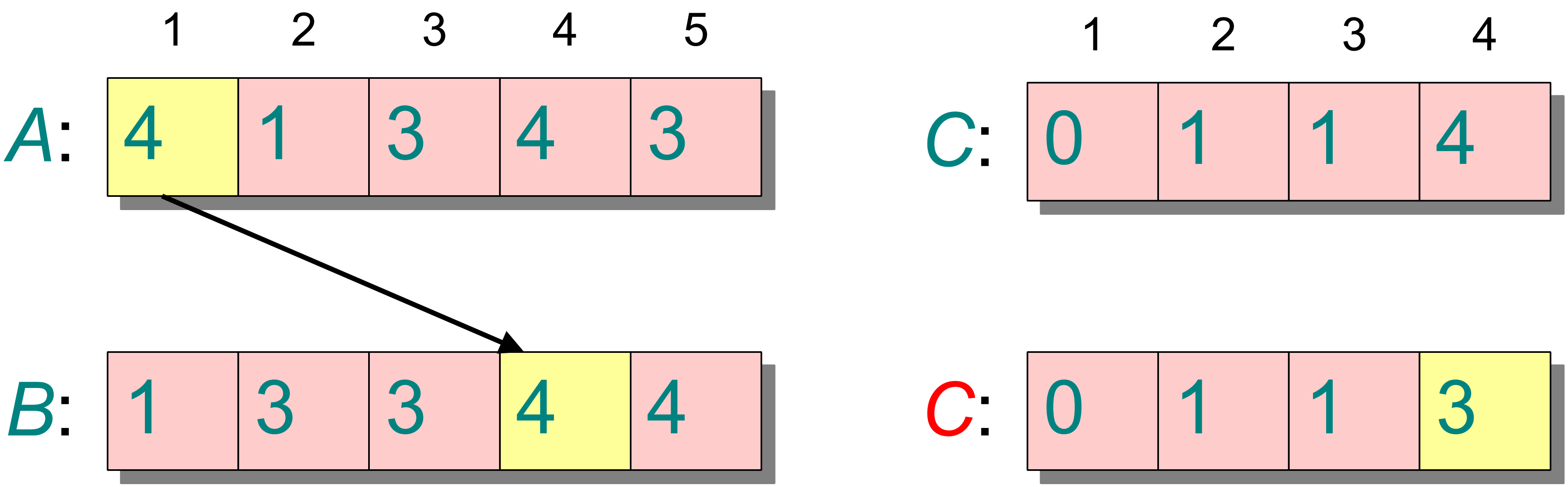


for $j \leftarrow n$ **downto** 1

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

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

Counting sort

for $i \leftarrow 1$ **to** k
  do $C[i] \leftarrow 0$
for $j \leftarrow 1$ **to** n
  do $C[A[j]] \leftarrow C[A[j]] + 1$ # $C[i] = |\{\text{key} = i\}|$
for $i \leftarrow 2$ **to** k
  do $C[i] \leftarrow C[i] + C[i-1]$ # $C[i] = |\{\text{key} \leq i\}|$
for $j \leftarrow n$ **downto** 1
  do $B[C[A[j]]] \leftarrow A[j]$
    $C[A[j]] \leftarrow C[A[j]] - 1$

Analysis

$O(k)$ $\left\{ \begin{array}{l} \textbf{for } i \leftarrow 1 \textbf{ to } k \\ \quad \textbf{do } C[i] \leftarrow 0 \end{array} \right.$

$O(n)$ $\left\{ \begin{array}{l} \textbf{for } j \leftarrow 1 \textbf{ to } n \\ \quad \textbf{do } C[A[j]] \leftarrow C[A[j]] + 1 \end{array} \right.$

$O(k)$ $\left\{ \begin{array}{l} \textbf{for } i \leftarrow 2 \textbf{ to } k \\ \quad \textbf{do } C[i] \leftarrow C[i] + C[i-1] \end{array} \right.$

$O(n)$ $\left\{ \begin{array}{l} \textbf{for } j \leftarrow n \textbf{ downto } 1 \\ \quad \textbf{do } B[C[A[j]]] \leftarrow A[j] \\ \quad\quad C[A[j]] \leftarrow C[A[j]] - 1 \end{array} \right.$

$O(n + k)$

Running time

In the distributed computing setting, the counting sort takes $O(n)$ time.

- But, sorting takes $O(n \lg n)$ time!
- What makes the differences?

Answer:

- ***Comparison sorting*** takes $O(n \lg n)$ time.
- Counting sort is not a ***comparison sort***.
- In fact, not a single comparison between elements occurs!

Exercise

Using counting sort to sort the following sequence consisting of letters in {a, b, c, d}.

b, c, d, c, a

Horspool's Algorithm

Review: String Searching

Pattern : a string of *m* characters to search for

Text : a (long) string of *n* characters to search in

Brute force algorithm

Step 1 Align pattern at beginning of text

Step 2 Moving from left to right, compare each character of pattern to the corresponding character in text until either all characters are found to match (successful search) or a mismatch is detected

Step 3 While a mismatch is detected and the text is not yet exhausted, realign pattern one position to the right and repeat Step 2

Example

```
T[ ]: N O B O D Y _ N O T I C E D _ H I M
P[ ]: N O T
        N O T
          N O T
            N O T
              N O T
                N O T
                  N O T
                    N O T
```

bolded: match
<u>**underlined**</u>: not match
un-bolded: not considered

If a mismatch found, shift one position right.

Horspool's Algorithm

- Horspool's algorithm is a simple string searching algorithms based on the input enhancement idea of preprocessing the pattern.

  – preprocesses pattern to generate a shift table that determines how far to shift the pattern when a mismatch occurs

  – always makes a shift based on the text's character *c* aligned with the last character in the pattern according to the shift table's entry for *c*

How much to shift?

- Look at first (rightmost) character in text that was compared:

The character is not in the pattern

```
.....c......................   (c not in pattern)
BAOBAB
      BAOBAB
```

The character is in the pattern (but not the rightmost)

```
.....O......................   (O occurs once in pattern)
BAOBAB
   BAOBAB
.....A......................   (A occurs twice in pattern)
BAOBAB
 BAOBAB
```

The rightmost characters do match

```
.....B......................
BAOBAB
  BAOBAB
```

Shift table

- For a pattern P[0..m-1], shift size s(c) of a letter c in the text can be precomputed as following:
  - if c is in P[0..m-2]

    s(c) =The number of characters from *c*'s rightmost occurrence in P[0..m-2] to the right end of the pattern P[0..m-1]
  - if c is not in P[0..m-2]

    s(c) =the length m of the pattern P[0..m-1], otherwise
- s(c) is stored in a so-called *shift table* indexed by text and pattern alphabet.

Algorithm ShiftTable(P[0..m-1])

//Fills the shift table used by Horspool's algorithm
//Input: Pattern P[0..m-1] and an alphabet of possible characters
//output: Table[0..size-1] indexed by the alphabet’s characters and filled
//with the shift sizes computed as before.

Initialize all the elements of Table with m

for j=0 **to** m-2 **do** *Table*[P[j]]=m-1-j
Return *Table*

$$\text{Pattern: } \overbrace{\underbrace{P[0], P[1], P[2], \ldots P[j]}_{j+1}, \underbrace{P[j+1], \ldots, P[m-1]}_{m-(j+1)=m-1-j}}^{m}$$

An Example of Shift table

- Let the text and pattern consists of alphabet {A,..., Z}
  Consider the pattern P[0..5] =BAOBAB

- Table

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	-
1	2	6	6	6	6	6	6	6	6	6	6	6	6	3	6	6	6	6	6	6	6	6	6	6	6	6

```
ALGORITHM  HorspoolMatching(P[0..m − 1], T[0..n − 1])
    //Implements Horspool's algorithm for string matching
    //Input: Pattern P[0..m − 1] and text T[0..n − 1]
    //Output: The index of the left end of the first matching substring
    //        or −1 if there are no matches
    ShiftTable(P[0..m − 1])     //generate Table of shifts
    i ← m − 1                   //position of the pattern's right end
    while i ≤ n − 1 do
        k ← 0                   //number of matched characters
        while k ≤ m − 1 and P[m − 1 − k] = T[i − k] do
            k ← k + 1
        if k = m
            return i − m + 1
        else i ← i+Table[T[i]]
    return −1
```

Example of Horspool's alg. application

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	_
1	2	6	6	6	6	6	6	6	6	6	6	6	6	3	6	6	6	6	6	6	6	6	6	6	6	6

```
BARD LOVED BANANAS
BAOBAB
      BAOBAB
        BAOBAB
              BAOBAB
```

(unsuccessful search)

Example of Horspool's alg. application

Boyer Moore Horspool

- Primarily, make 'Bad Match Table'
- Compare pattern to text, starting from rightmost character in the pattern
- If mismatch, move pattern forward corresponding to value in the table.
- Pattern 'tooth'
- Text 'trusthardtoothbrushes'

https://www.youtube.com/watch?v=PHXAOKQk2dw&ab_channel=MikeSlade

Exercise

Assume that all text and patterns consists of letters in A, C, T, G. Create a shift table for the following pattern

TCCTATTCTT

Complexity

- The worst case complexity: $O(mn)$.
- However: for random texts, it is in $O(n)$
- Conclusion: On average, Horspool's algorithm is faster than the brute-force algorithm.

Learning Outcomes

- The idea for space-for-time tradeoffs
- Two algorithms:
  1. Distribution counting sort
  2. Horspool's algorithm for string searching