CS3230 Lecture 4

NUS National University of Singapore
School of Computing

"A Review of Sorting, Quicksort Analysis, and Linear Time Sorting Algorithms"

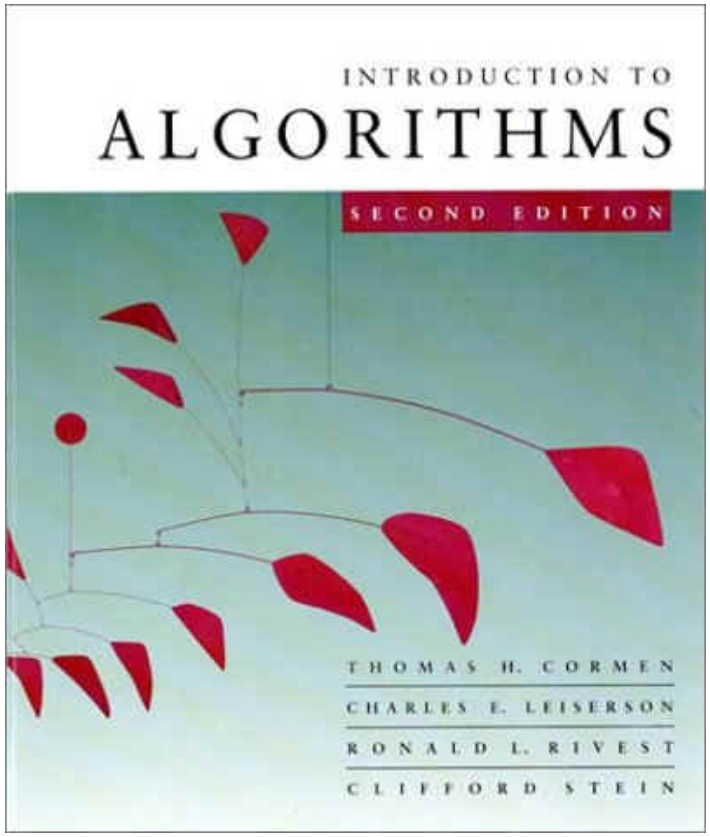
❑ Lecture Topics and Readings

- ❖ (Quick Review) of Sorting Methods [CLRS]-C?
- ❖ Quicksort and Randomized QS [CLRS]-C7
- ❖ Lower Bound for Sorting [CLRS]-C8.1
- ❖ Linear Time Sorting Algorithms [CLRS]-C8.2,8.3

Creative View of Sorting Methods (Gives new insights)
Randomized Quicksort (only 38.6% sub-optimal)
Lower Bound also important. How to break barriers!

Hon Wai Leong, NUS

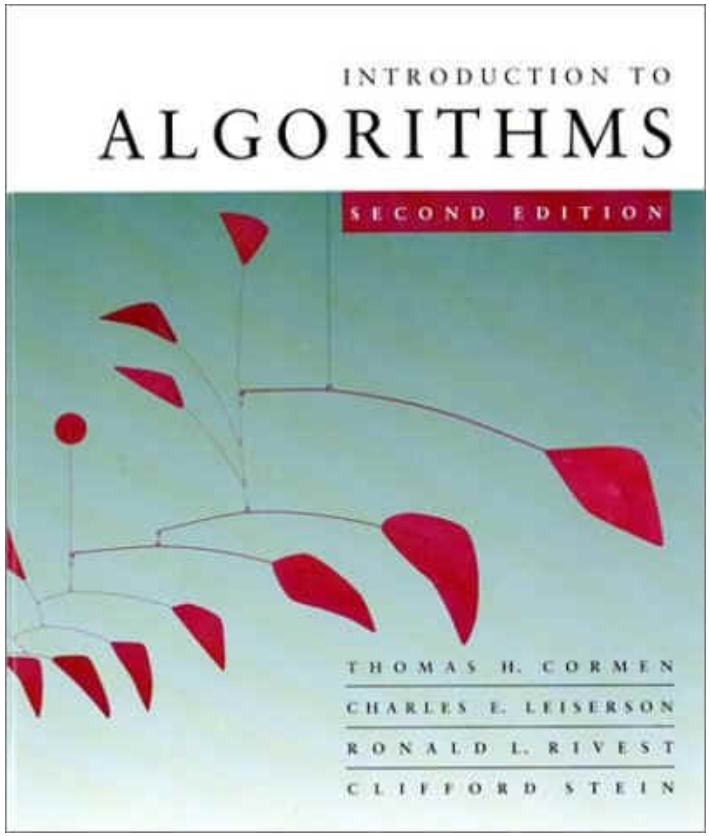
[CLRS]...



Sabbatical leave at NUS
Computer Science Dept 1995/96

[CLRS] & Charles Leiserson.

[CLRS] @500K



[CLRS]-90, [CLRS]-01, [CLRS]-09
Celebrating 500,000 copies sold

[HH2013]… 3rd edition

Steven Halim

Felix Halim

[HH13] *Competitive Programming*, (3rd edition)
by Steven Halim and Felix Halim, 2013.

Antony Hoare (1934 –)

Invented Quicksort (at age 26)

Developed Hoare's Logic (for program correctness)

Developed CSP (including dining philisophers' problem)

Quote: (about difficulties of creating software systems)

"There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult."

Tony Hoare, Singapore 2008 Computing in the 21st Century

- **Turing Award, 1980**
- **Knighted, 2000**

Thank you.

Q & A

CS3230 Lecture 4

NUS National University of Singapore
School of Computing

"A Review of Sorting, Quicksort Analysis, and Linear Time Sorting Algorithms"

❑ Lecture Topics and Readings

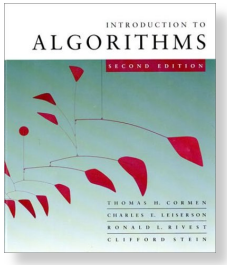
- ❖ (Quick Review) of Sorting Methods [CLRS]-C?
- ❖ Quicksort and Randomized QS [CLRS]-C7
- ❖ Lower Bound for Sorting [CLRS]-C8.1
- ❖ Linear Time Sorting Algorithms [CLRS]-C8.2,8.3

Creative View of Sorting Methods (Gives new insights)
Randomized Quicksort (only 38.6% sub-optimal)
Lower Bound also important. How to break barriers!

Hon Wai Leong, NUS

Sorting Animation: by Steven Halim & students

http://www.comp.nus.edu.sg/~stevenha/visualization/sorting.html



The problem of sorting

Input: sequence $\langle a_1, a_2, \ldots, a_n \rangle$ of numbers.

Output: permutation $\langle a'_1, a'_2, \ldots, a'_n \rangle$ such that $a'_1 \le a'_2 \le \cdots \le a'_n$.

Example: ***Input:*** 8 2 4 9 3 6

Output: 2 3 4 6 8 9

Sorting Animation: by Steven Halim & students

http://www.comp.nus.edu.sg/~stevenha/visualization/sorting.html

Sorting: Problem and Algorithms

Problem: Sorting

Given a list of n numbers, sort them

Algorithms:

- Selection Sort $\Theta(n^2)$
- Insertion Sort $\Theta(n^2)$
- Bubble Sort $\Theta(n^2)$
- Merge Sort $\Theta(n \lg n)$
- Quicksort $\Theta(n \lg n)$*

* *average case*

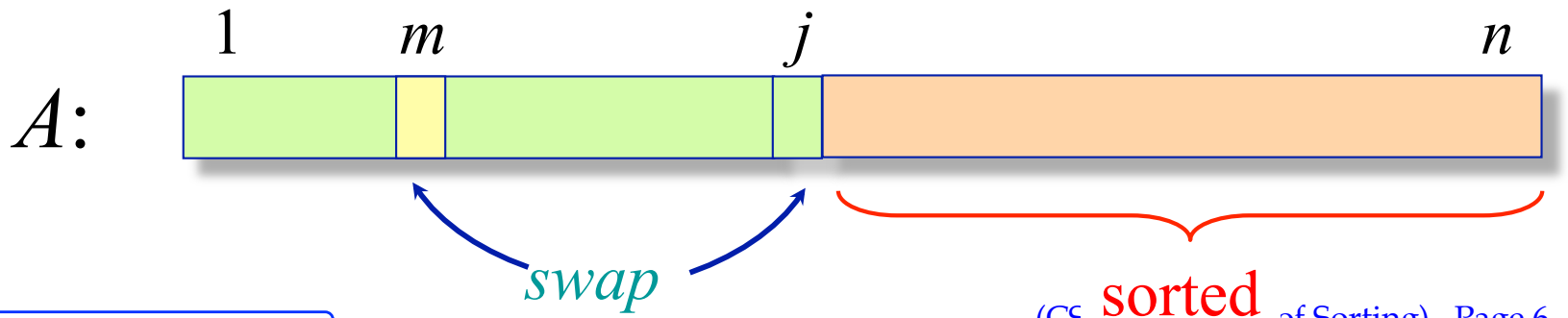
Selection Sort Algorithm

Recall from Lecture 2

```
SELECTION-SORT (A, n)      ▷ A[1 . . n]
  j ← n
  while j > 1
        do m ← Find-Max (A, j)
           Swap (A[m], A[j])
           j ← j – 1
```

Let's make this recursive.

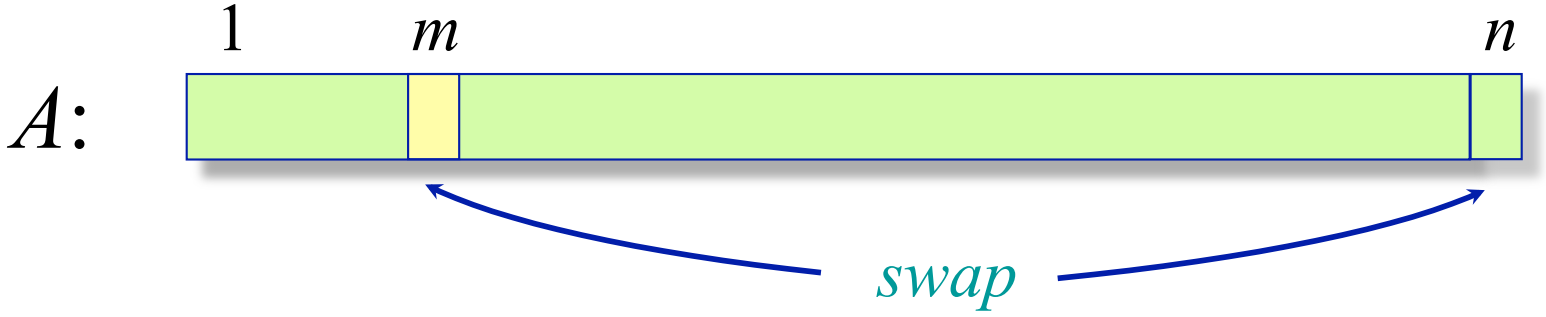
$A[m]$ is largest among $A[1..j]$



Recursive Selection Sort

```
SELECTION-SORT-R (A, n)   ▷ A[1 . . n]
    if n = 1 then return
    m ← Find-Max (A, n)
    Swap (A[m], A[n])
    SELECTION-SORT-R (A, n–1)
```

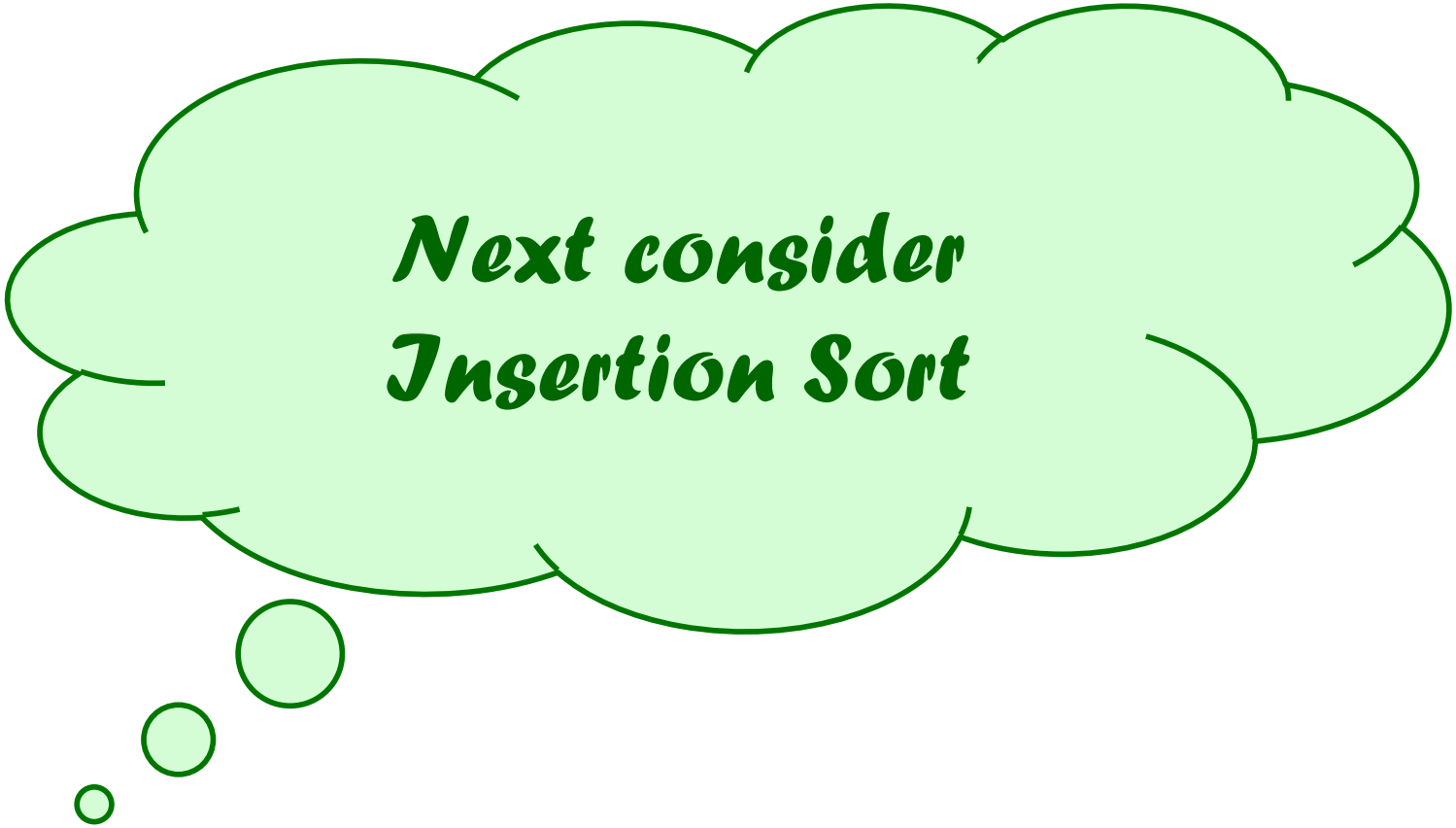
$A[m]$ is largest among $A[1..n]$

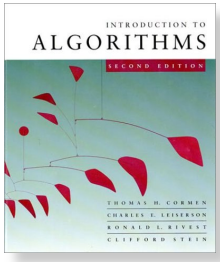


Recursive Selection Sort

```
SELECTION-SORT-R (A, n)   ▷ A[1 . . n]
    if n = 1 then return
    m ← Find-Max (A, n)
    Swap (A[m], A[n])
    SELECTION-SORT-R (A, n–1)
```

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1; \\ T(n-1) + \Theta(n) & \text{if } n > 1. \end{cases}$$



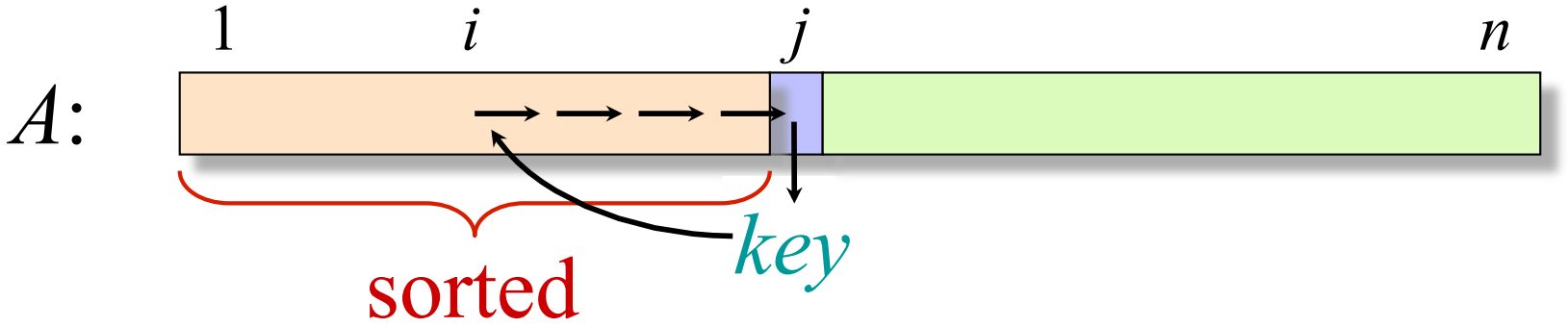


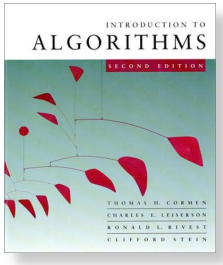
Insertion sort

Recall from Lecture 2

"pseudocode"

```
INSERTION-SORT (A, n)      ▷ A[1 . . n]
    for j ← 2 to n
        do key ← A[ j]
           i ← j – 1
           while i > 0 and A[i] > key
               do A[i+1] ← A[i]
                  i ← i – 1
           A[i+1] = key
```





Insertion sort

INSERTION-SORT (A, n) ▷ $A[1 \,..\, n]$

for $j \leftarrow 2$ **to** n

do $key \leftarrow A[j]$

$i \leftarrow j - 1$

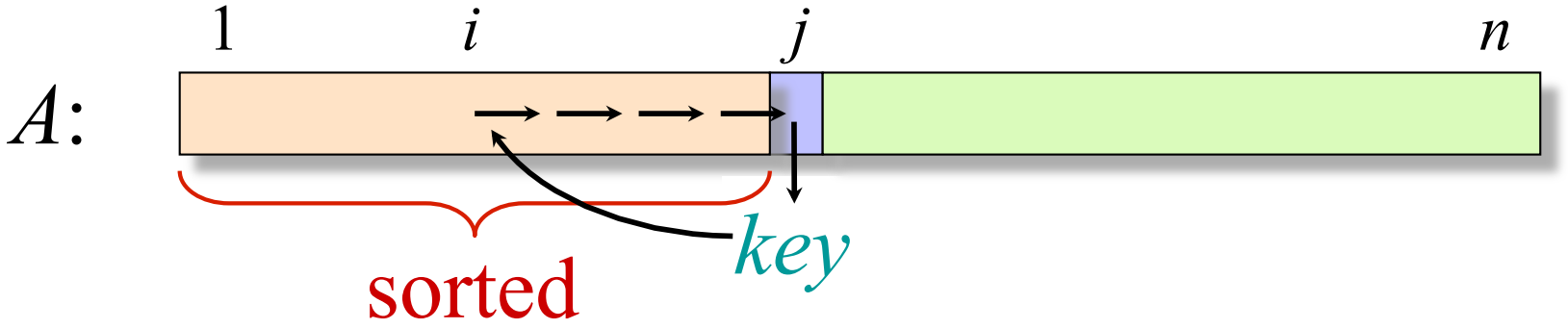
while $i > 0$ and $A[i] > key$

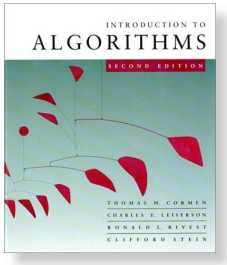
do $A[i+1] \leftarrow A[i]$

$i \leftarrow i - 1$

$A[i+1] = key$

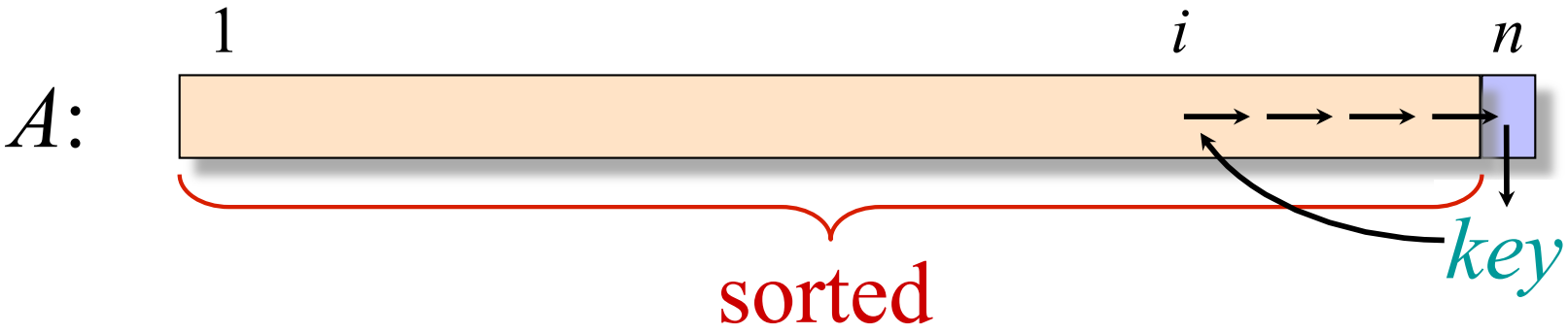
"inserts" $A[j]$ into sorted $A[1 \,..\, (j-1)]$

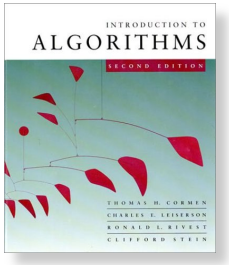




Recursive Insertion sort

```
INSERTION-SORT-R (A, n)   ▷ A[1 . . n]
    if n = 1 then return
    INSERTION-SORT-R (A, n–1)
    insert A[n] into sorted A[1 .. n–1]
```

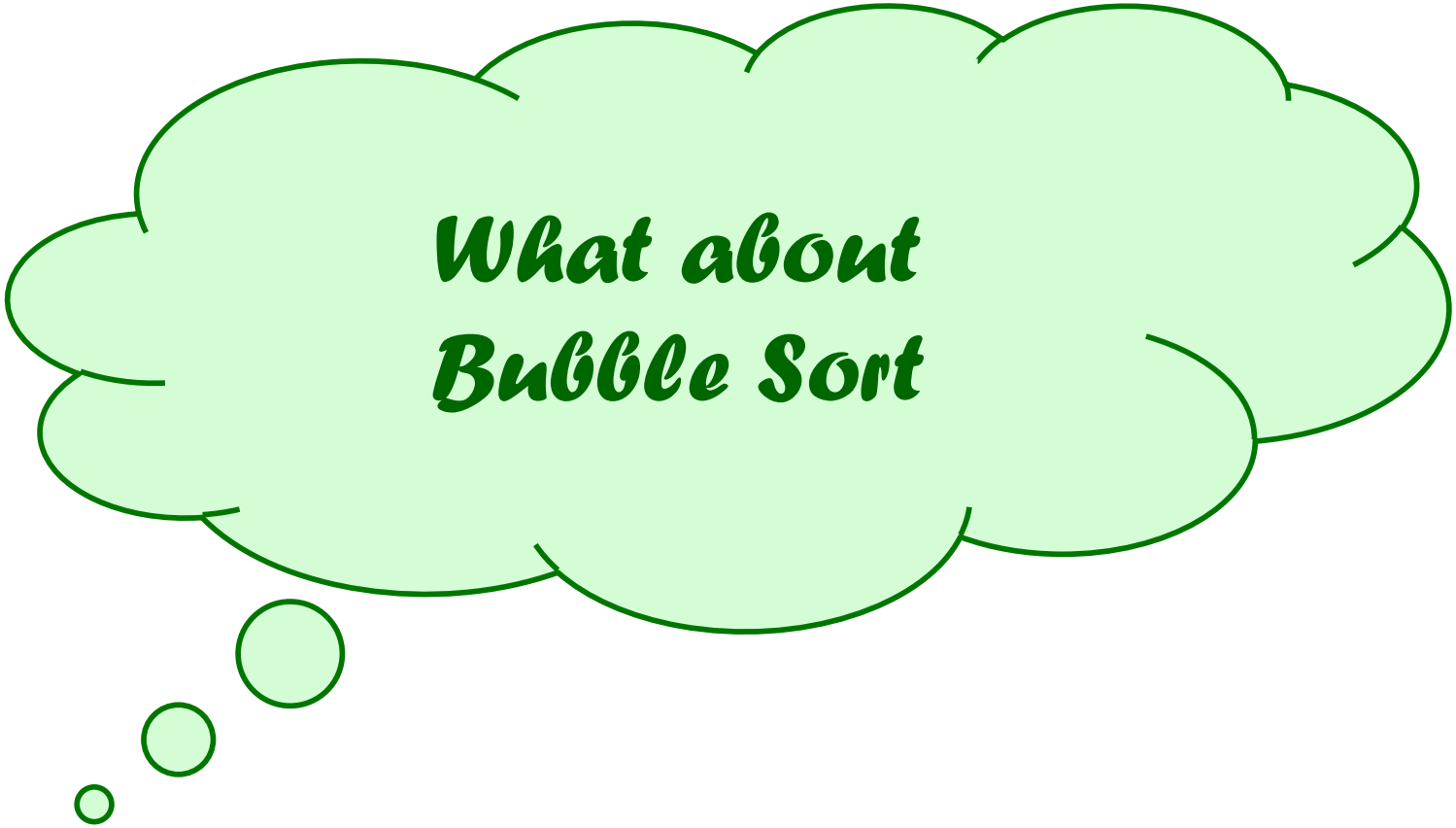




Recursive Insertion sort

INSERTION-SORT-R (A, n) $\triangleright$ $A[1 \,..\, n]$
  if $n = 1$ **then return**
  INSERTION-SORT-R $(A, n\text{–}1)$
  insert $A[n]$ into sorted $A[1 \,..\, n\text{–}1]$

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1; \\ T(n\text{–}1) + \Theta(n) & \text{if } n > 1. \end{cases}$$



Recursive Bubble Sort

```
BUBBLE-SORT-R (A, n) ▷ A[1 . . n]
    if n = 1 then return
    One bubble-phase on A[1 .. n]
    BUBBLE-SORT-R (A, n–1)
```

Recursive Bubble Sort

BUBBLE-SORT-R (A, n) $\triangleright$ $A[1 \,..\, n]$
  if $n = 1$ **then return**
  One bubble-phase on $A[1 \,..\, n]$
  BUBBLE-SORT-R $(A, n-1)$

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1; \\ T(n-1) + \Theta(n) & \text{if } n > 1. \end{cases}$$

Hon Wai Leong, NUS

All have the *same* recurrence

Selection Sort, Insertion Sort, Bubble Sort

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1; \\ T(n-1) + \Theta(n) & \text{if } n > 1. \end{cases}$$

$(n-1), 0$

Extreme imbalance

All have the *same* recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1; \\ T(n-1) + \Theta(n) & \text{if } n > 1. \end{cases}$$

How to "solve" this recurrence

Answer: Use TELESCOPING

Hon Wai Leong, NUS

How to "solve" this recurrence

Answer: Use TELESCOPING

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

$$T(n) = \Theta(n^2)$$

$$= cn + c(n-1) + T(n-2)$$

$$= cn + c(n-1) + c(n-2) + T(n-3)$$

$$= cn + c(n-1) + c(n-2) + \ldots c2 + T(1)$$

$$= cn + c(n-1) + c(n-2) + \ldots c2 + c$$

$$= c\,(n + (n-1) + (n-2) + \ldots 2 + 1)$$

Observation:

Selection Sort, Insertion Sort, Bubble Sort

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1; \\ T(n-1) + \Theta(n) & \text{if } n > 1. \end{cases}$$

They all have running time: $T(n) = \Theta(n^2)$

Imbalance in Divide &Conquer algorithms produces inefficient algorithms

Merge sort (Perfect balance)

MERGE-SORT $A[1 \ldots n]$

1. If $n = 1$, done.
2. Recursively sort $A[\,1 \ldots \lceil n/2 \rceil\,]$ and $A[\,\lceil n/2 \rceil + 1 \ldots n\,]$.
3. "***Merge***" the 2 sorted lists.

$$T(n) = \begin{cases} \Theta(1) \text{ if } n = 1; \\ 2T(n/2) + \Theta(n) \text{ if } n > 1. \end{cases}$$

M-Thm: $a = 2, b = 2 \Rightarrow n^{\log_b a} = n^{\log_2 2} = n$

$\Rightarrow$ CASE 2 $(k = 0) \Rightarrow T(n) = \Theta(n \lg n)$.

What about Heapsort, Quicksort

Heapsort $\Theta(n \lg n)$

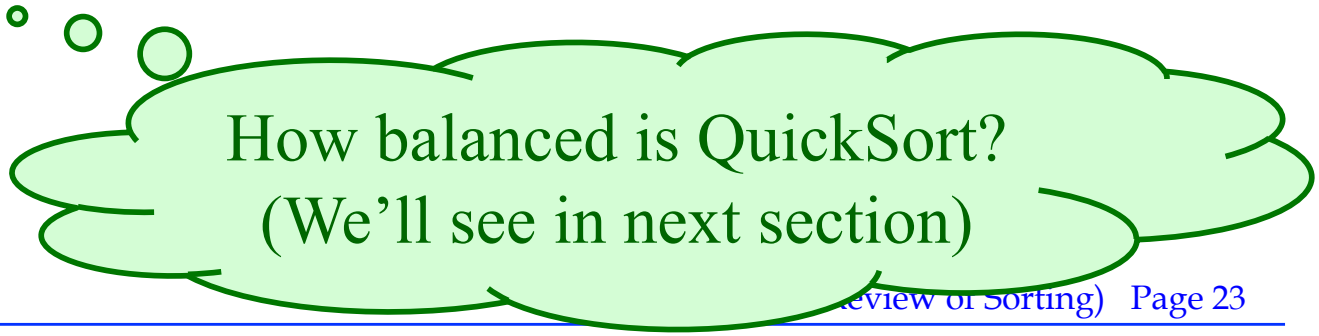
builds a data structure – Heap $\Theta(n)$

sort efficiently using the Heap $\Theta(n \lg n)$

Quicksort

Partitions array about a pivot $\Theta(n)$

Recursively sort each partition $O(??)$



Thank you.

Q & A

CS3230 Lecture 4

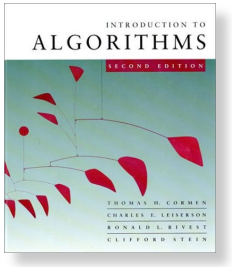
NUS National University of Singapore
School of Computing

"A Review of Sorting, Quicksort Analysis, and Linear Time Sorting Algorithms"

- Lecture Topics and Readings
  - (Quick Review) of Sorting Methods [CLRS]-C?
  - Quicksort and Randomized QS [CLRS]-C7
  - Lower Bound for Sorting [CLRS]-C8.1
  - Linear Time Sorting Algorithms [CLRS]-C8.2,8.3

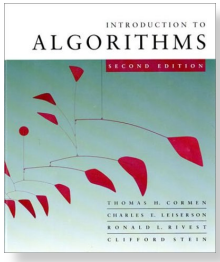
Creative View of Sorting Methods (Gives new insights)
Randomized Quicksort (only 38.6% sub-optimal)
Lower Bound also important. How to break barriers!

Hon Wai Leong, NUS



Quicksort

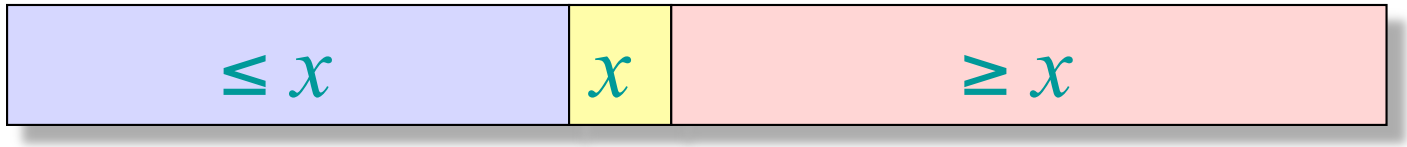
- Proposed by C.A.R. Hoare in 1962.
- Divide-and-conquer algorithm.
- Sorts "in place" (like insertion sort, but not like merge sort).
- Very practical (with tuning).



Divide and conquer

Quicksort an n-element array:

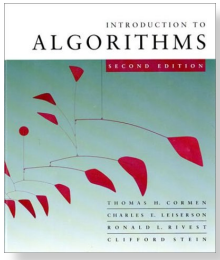
1. Divide: Partition the array into two subarrays around a ***pivot*** x such that elements in lower subarray $\leq x \leq$ elements in upper subarray.



2. Conquer: Recursively sort the two subarrays.

3. Combine: Trivial.

Key: *Linear-time partitioning subroutine.*

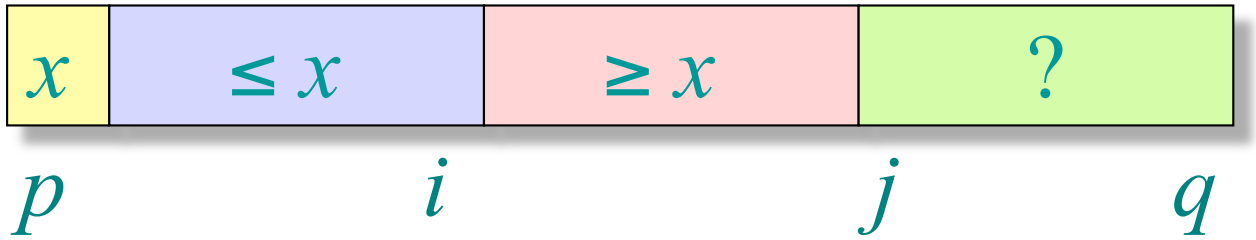


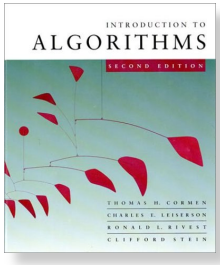
Partitioning subroutine

```
PARTITION(A, p, q)   ▷ A[p . . q]
    x ← A[p]         ▷ pivot = A[p]
    i ← p
    for j ← p + 1 to q
        do if A[j] ≤ x
                then  i ← i + 1
                      exchange A[i] ↔ A[j]
    exchange A[p] ↔ A[i]
    return i
```

Running time $= O(n)$ for n elements.

Invariant:

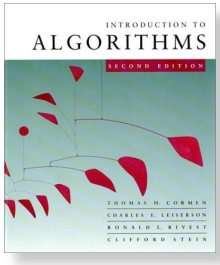




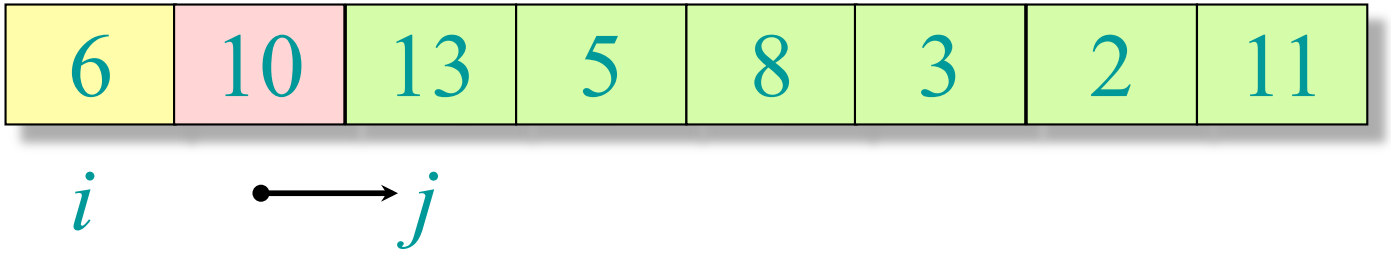
Example of partitioning

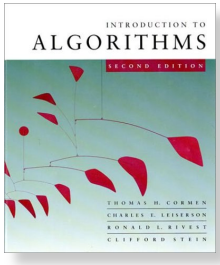
6	10	13	5	8	3	2	11

i *j*

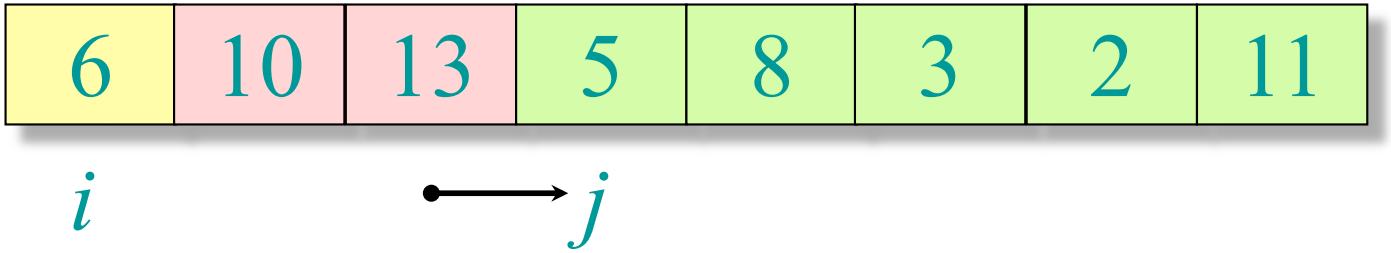


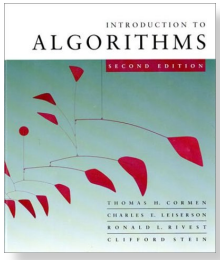
Example of partitioning



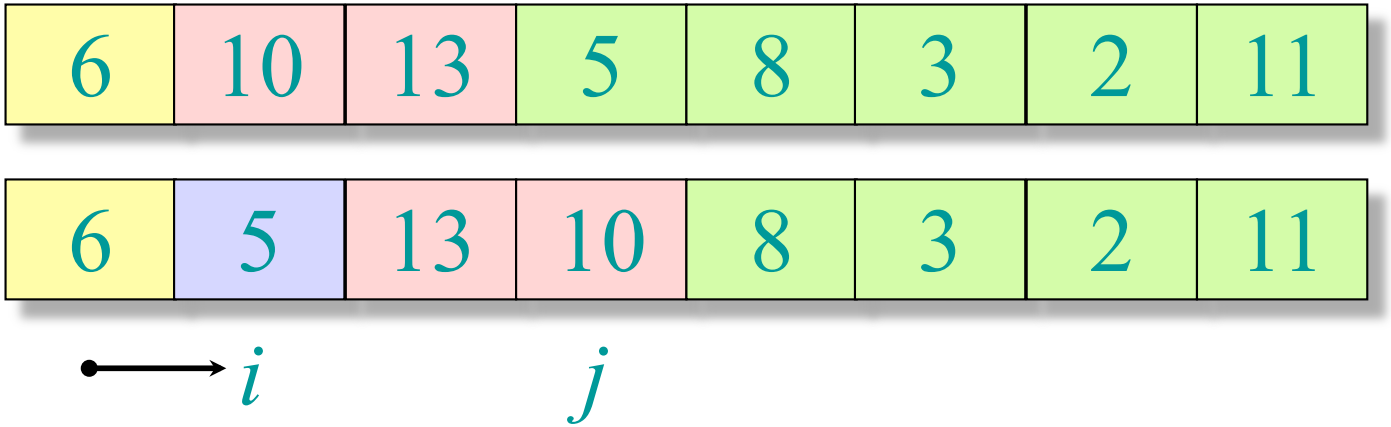


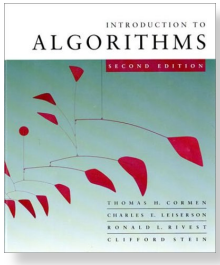
Example of partitioning



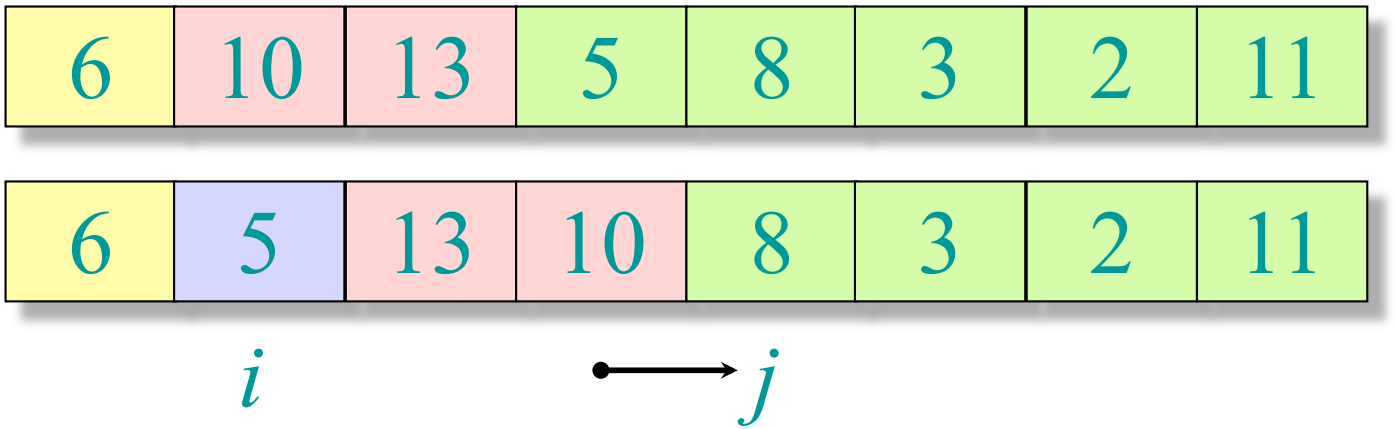


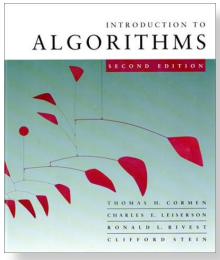
Example of partitioning



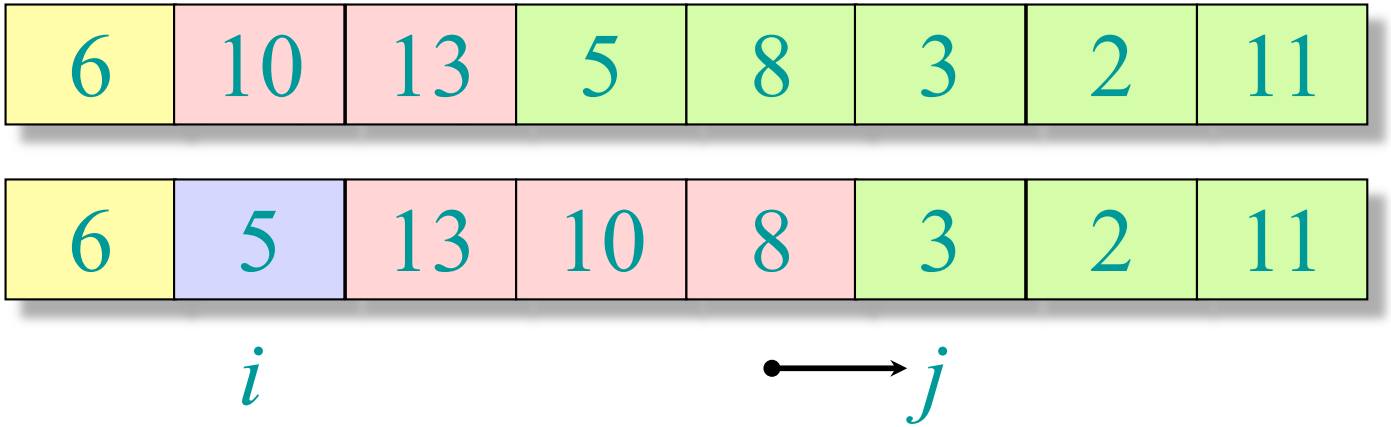


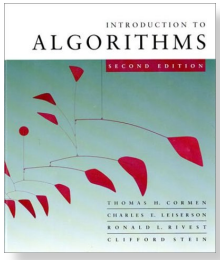
Example of partitioning



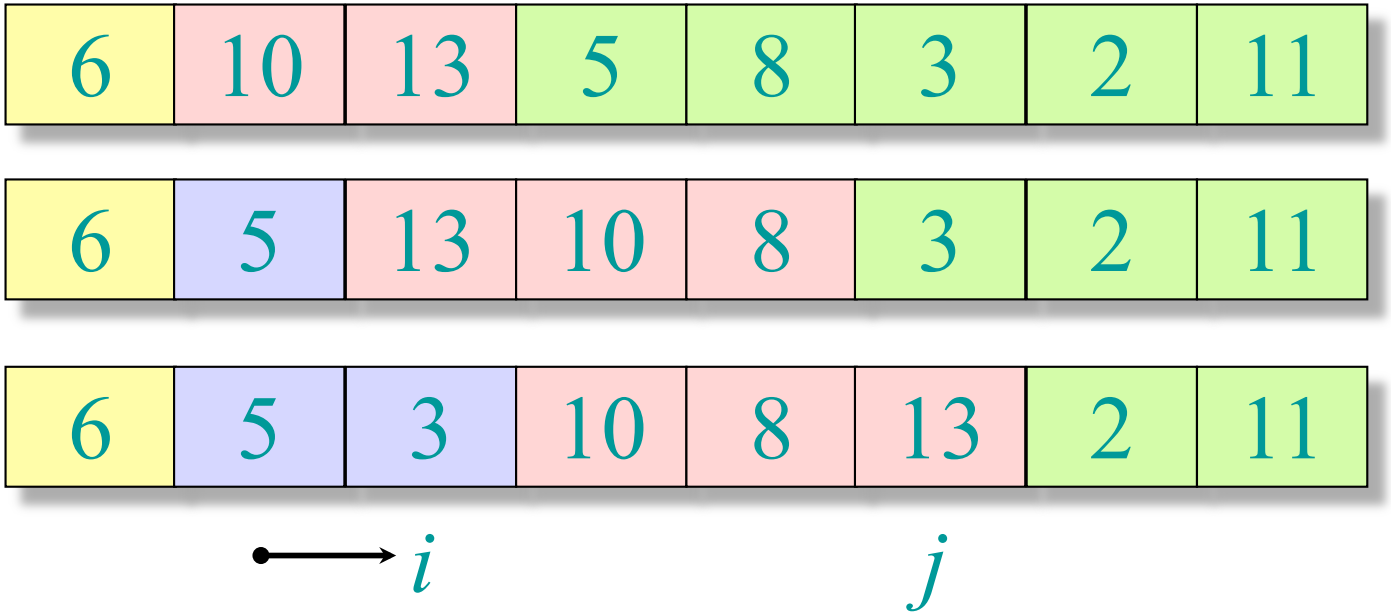


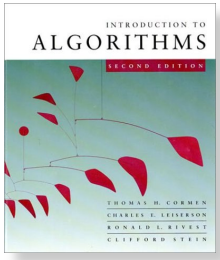
Example of partitioning



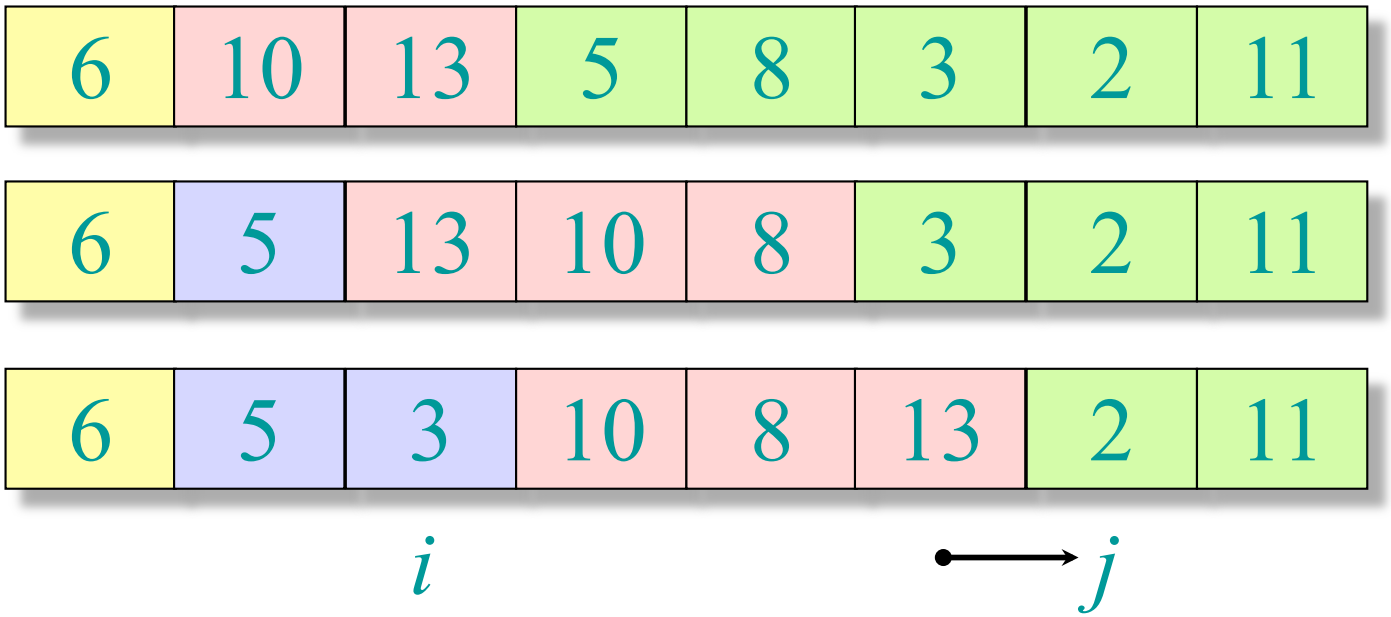


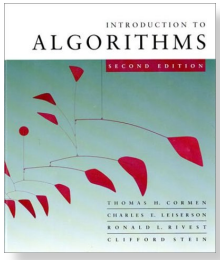
Example of partitioning



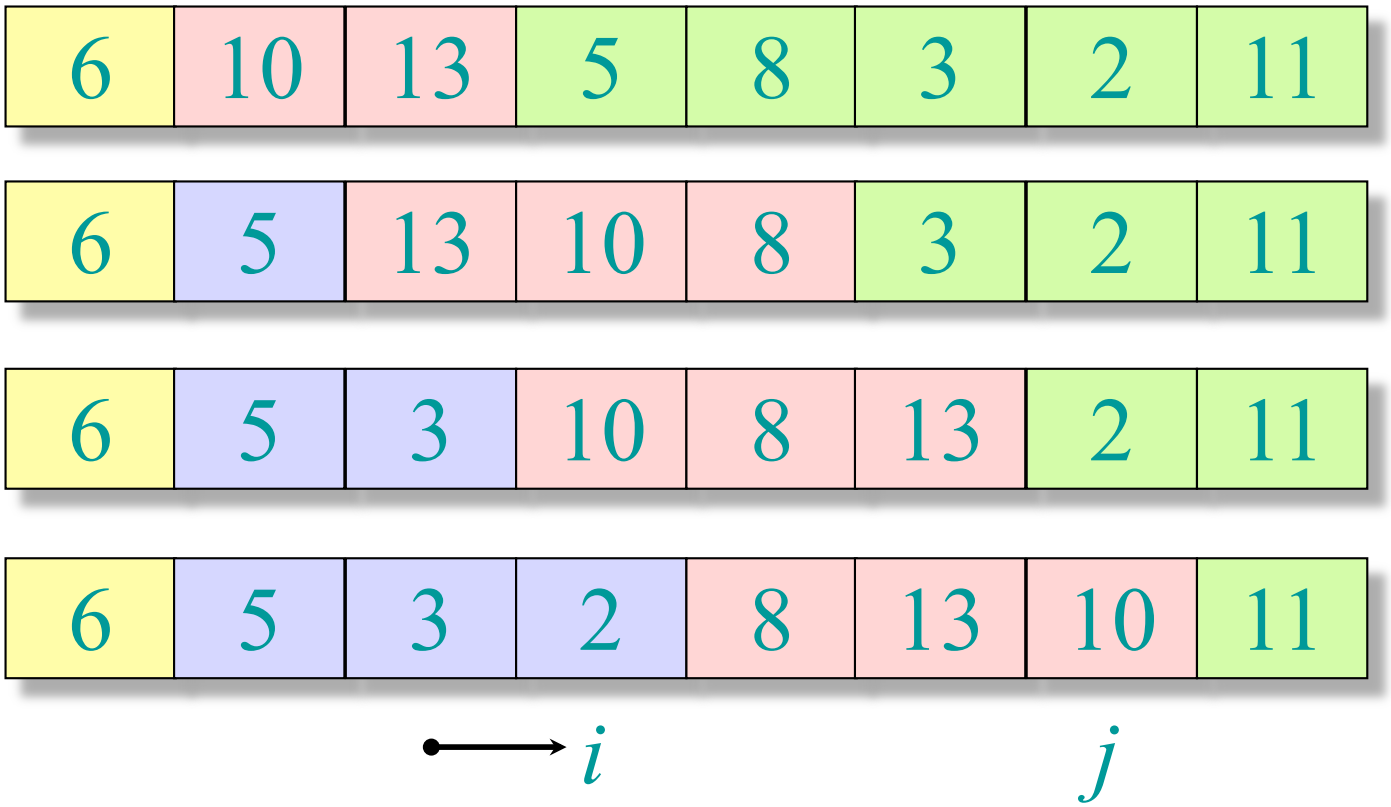


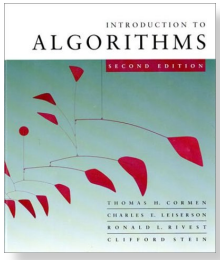
Example of partitioning



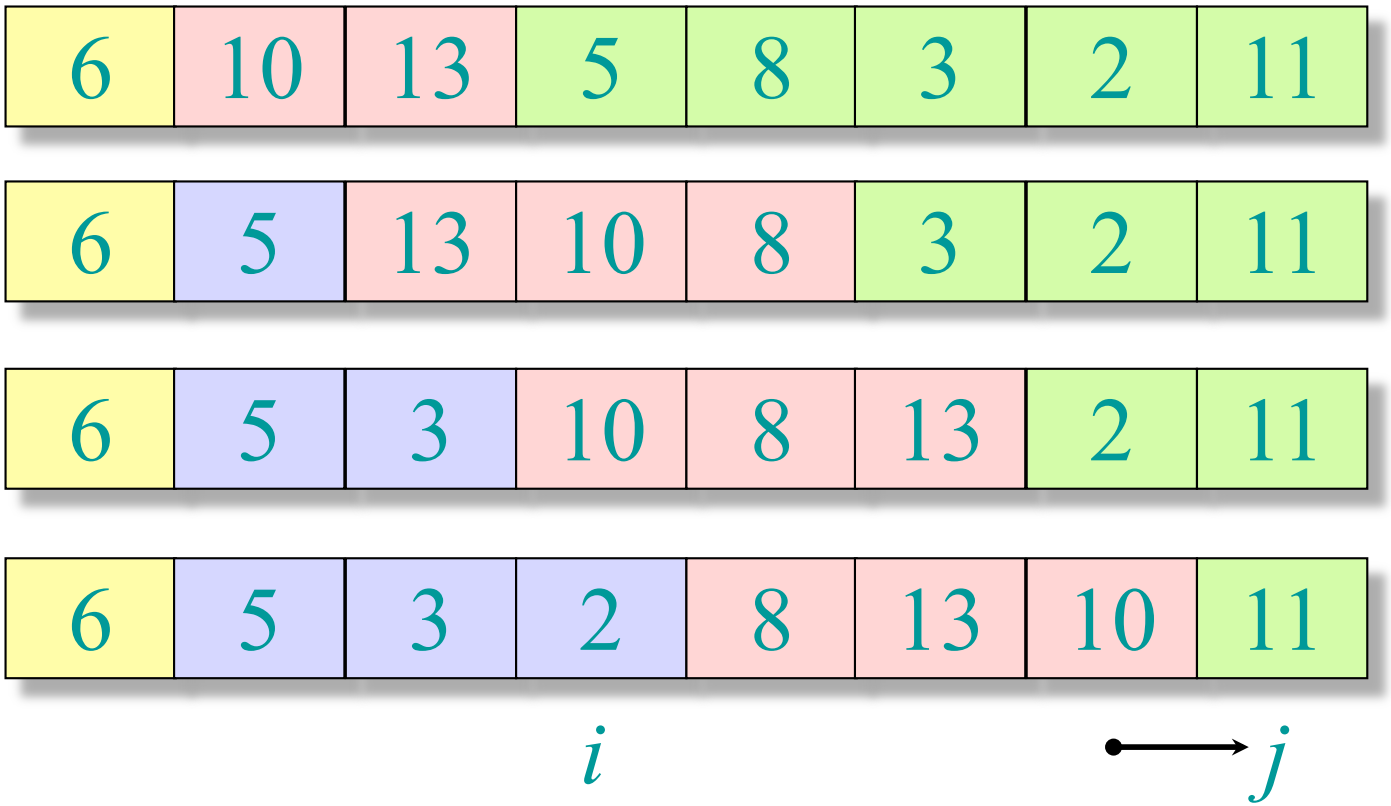


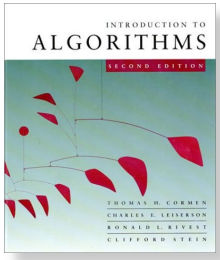
Example of partitioning



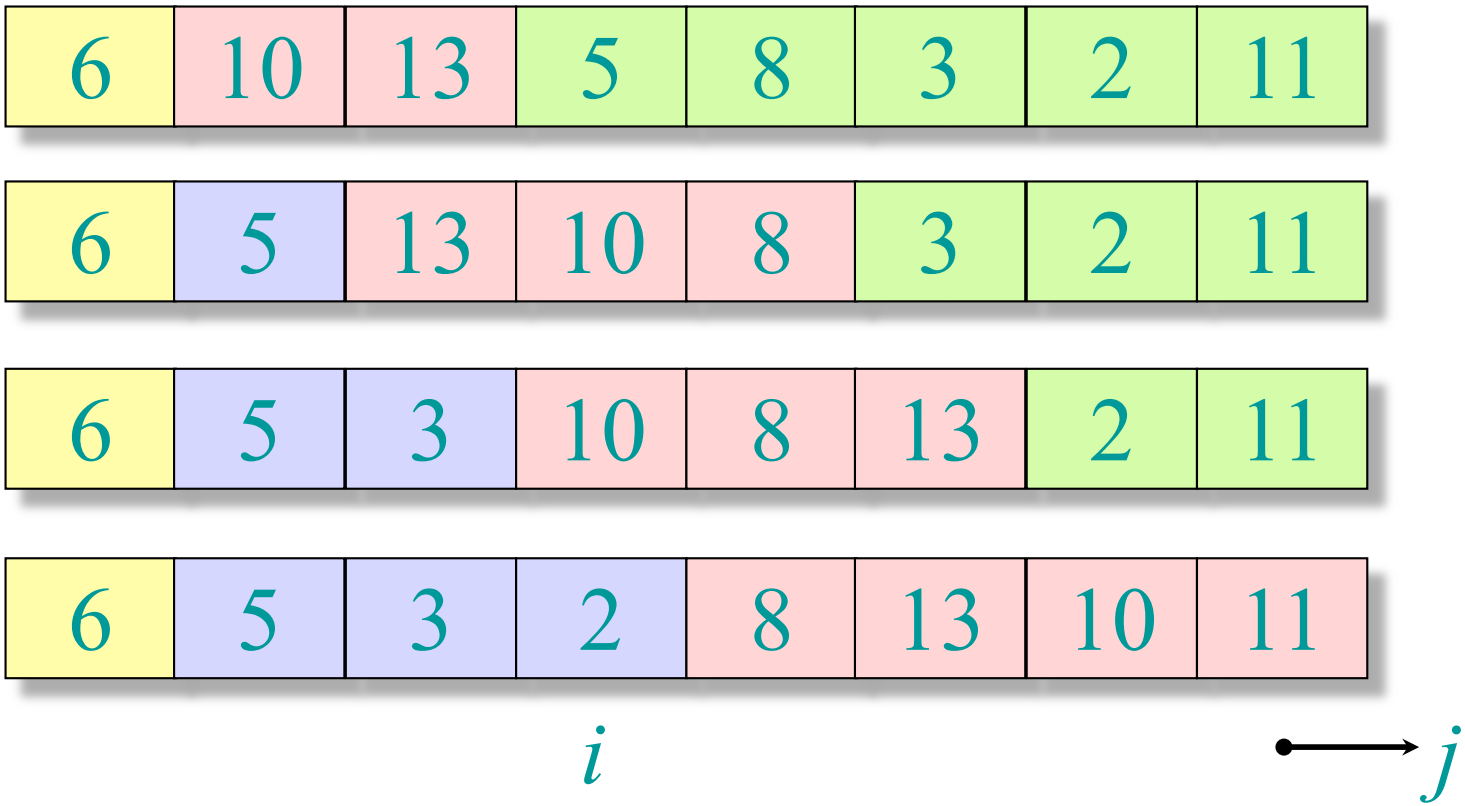


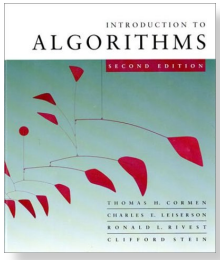
Example of partitioning





Example of partitioning

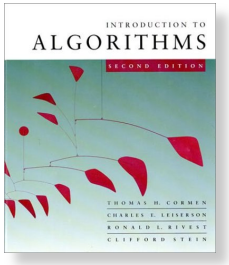




Example of partitioning

6	10	13	5	8	3	2	11
6	5	13	10	8	3	2	11
6	5	3	10	8	13	2	11
6	5	3	2	8	13	10	11
2	5	3	6	8	13	10	11

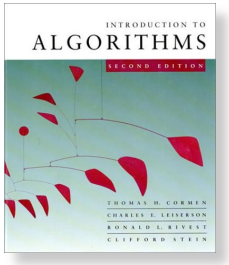
i



Pseudocode for quicksort

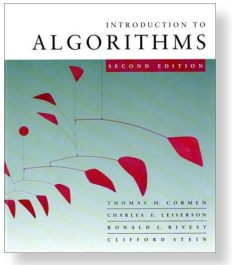
```
QUICKSORT(A, p, r)
    if p < r
        then q ← PARTITION(A, p, r)
             QUICKSORT(A, p, q–1)
             QUICKSORT(A, q+1, r)
```

Initial call: QUICKSORT$(A, 1, n)$



Analysis of quicksort

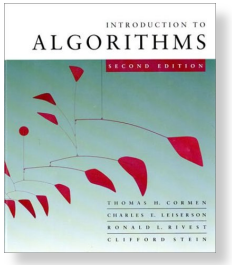
- Assume all input elements are distinct.
- In practice, there are better partitioning algorithms for when duplicate input elements may exist.
- Let $T(n)$ = worst-case running time on an array of n elements.



Worst-case of quicksort

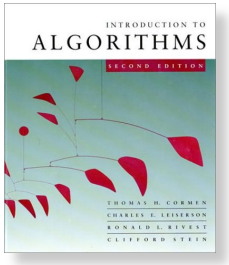
- Input sorted or reverse sorted.
- Partition around min or max element.
- One side of partition always has no elements.

$$\begin{aligned} T(n) &= T(0) + T(n-1) + \Theta(n) \\ &= \Theta(1) + T(n-1) + \Theta(n) \\ &= T(n-1) + \Theta(n) \\ &= \Theta(n^2) \qquad \textit{(arithmetic series)} \end{aligned}$$



Worst-case recursion tree

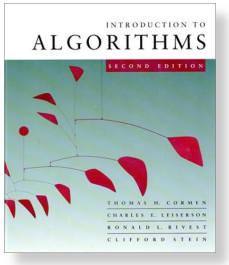
$$T(n) = T(0) + T(n-1) + cn$$



Worst-case recursion tree

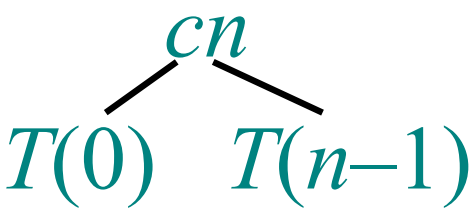
$$T(n) = T(0) + T(n-1) + cn$$

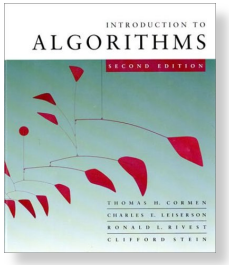
$T(n)$



Worst-case recursion tree

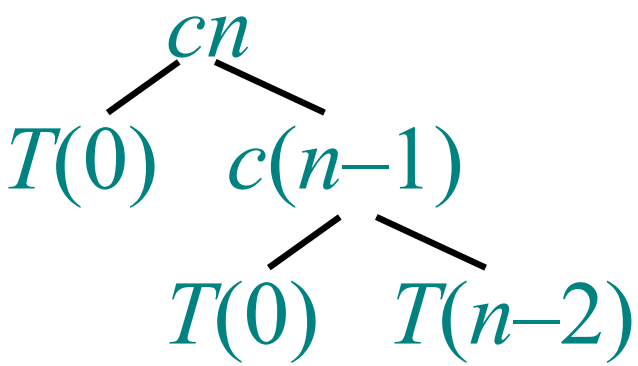
$$T(n) = T(0) + T(n-1) + cn$$

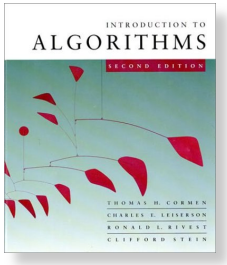




Worst-case recursion tree

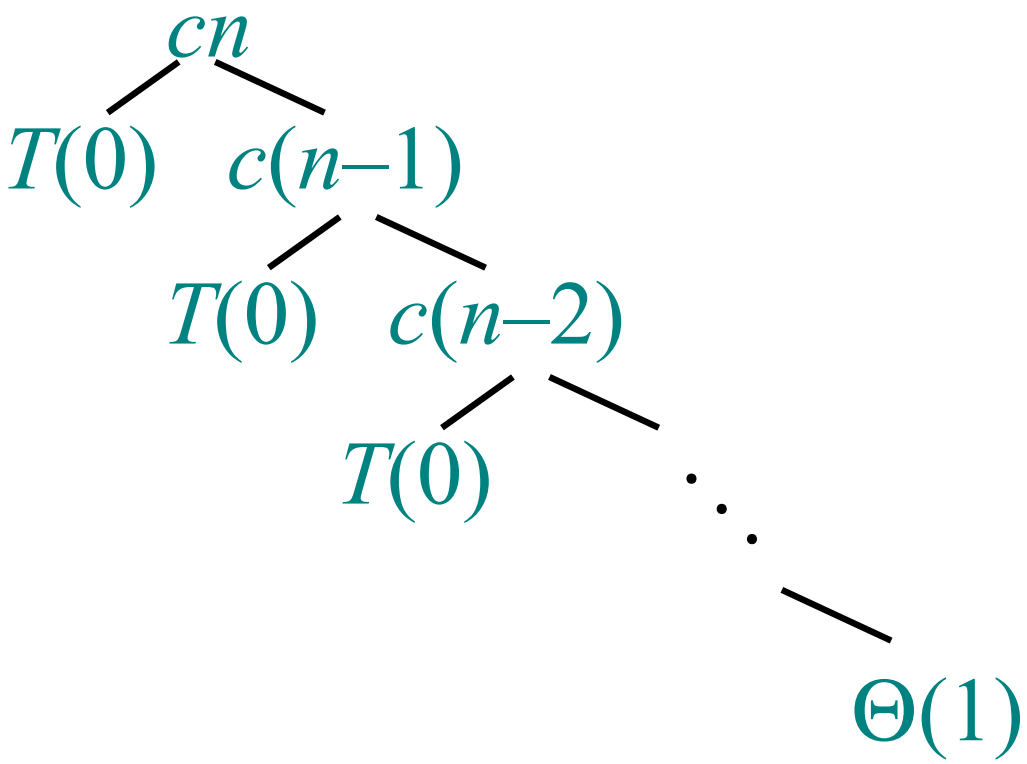
$$T(n) = T(0) + T(n–1) + cn$$

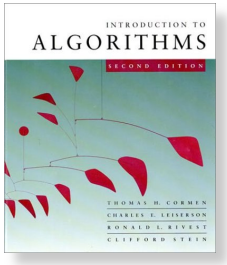




Worst-case recursion tree

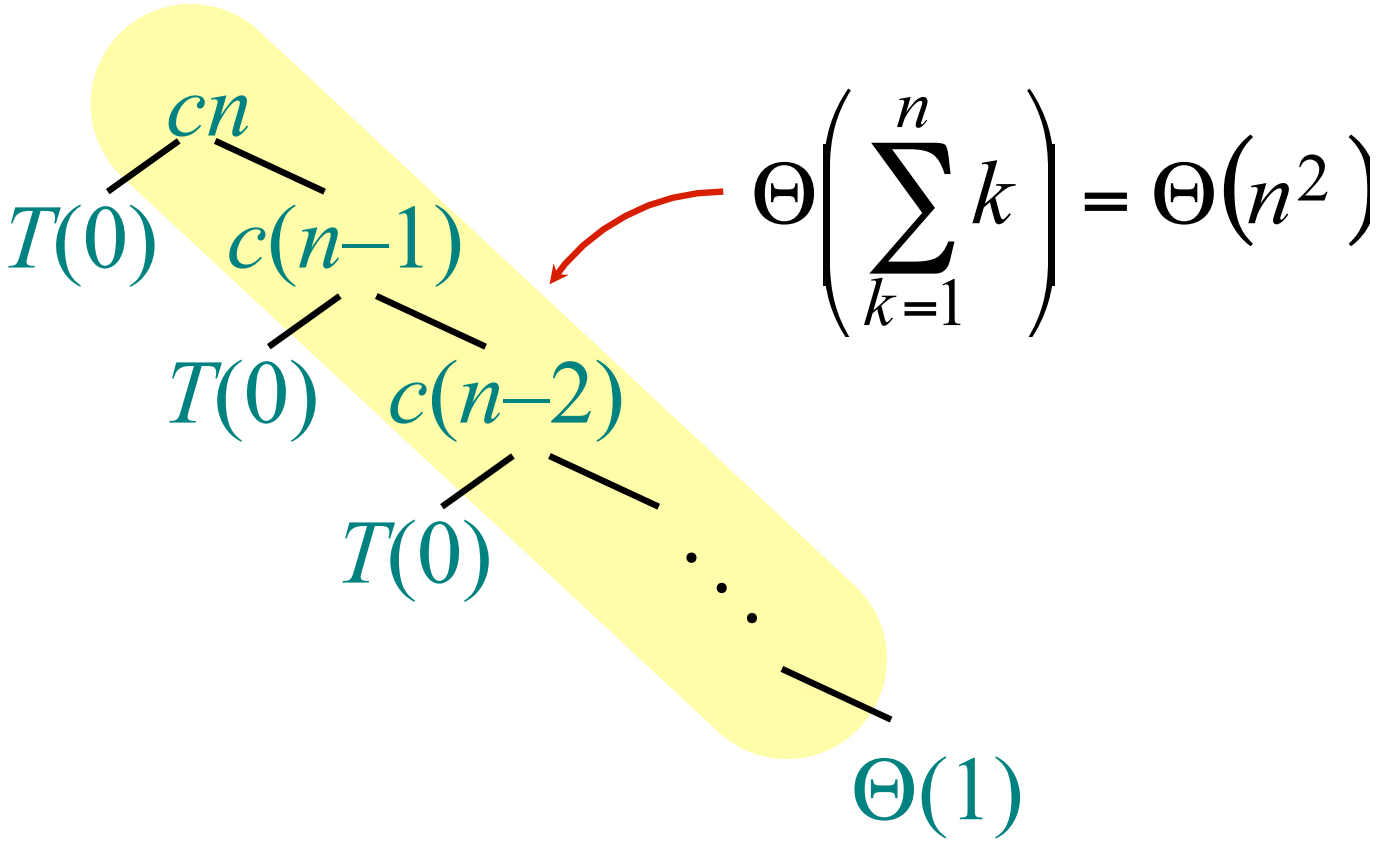
$$T(n) = T(0) + T(n-1) + cn$$

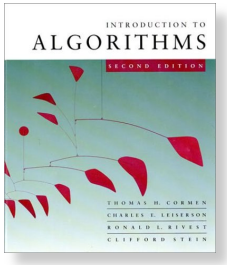




Worst-case recursion tree

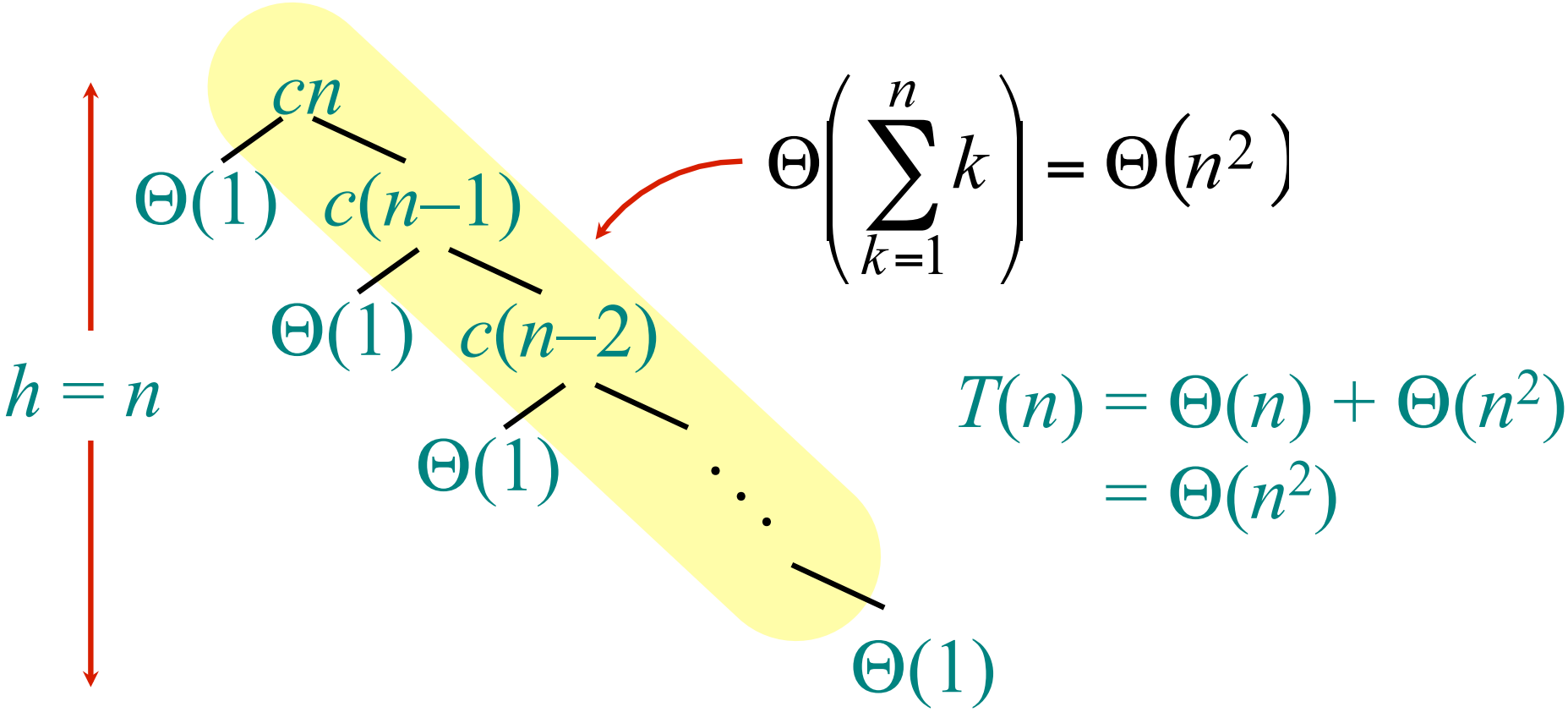
$$T(n) = T(0) + T(n-1) + cn$$





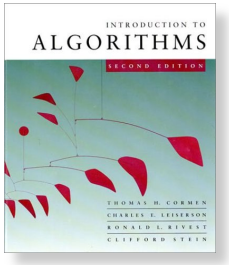
Worst-case recursion tree

$$T(n) = T(0) + T(n-1) + cn$$



$$\Theta\left(\sum_{k=1}^{n} k\right) = \Theta(n^2)$$

$$T(n) = \Theta(n) + \Theta(n^2)$$
$$= \Theta(n^2)$$



Best-case analysis

(For intuition only!)

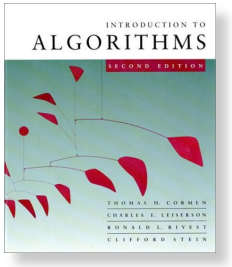
If we're lucky, PARTITION splits the array evenly:

$$T(n) = 2T(n/2) + \Theta(n)$$
$$= \Theta(n \lg n) \qquad \text{(same as merge sort)}$$

What if the split is always $\frac{1}{10} : \frac{9}{10}$?

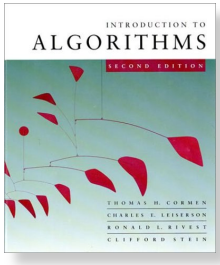
$$T(n) = T\left(\tfrac{1}{10} n\right) + T\left(\tfrac{9}{10} n\right) + \Theta(n)$$

What is the solution to this recurrence?

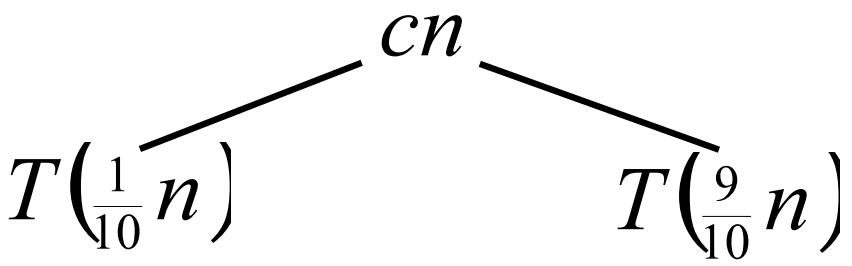


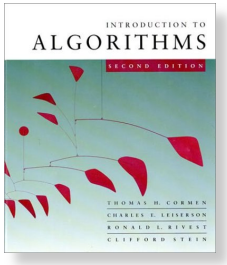
Analysis of "almost-best" case

$$T(n)$$

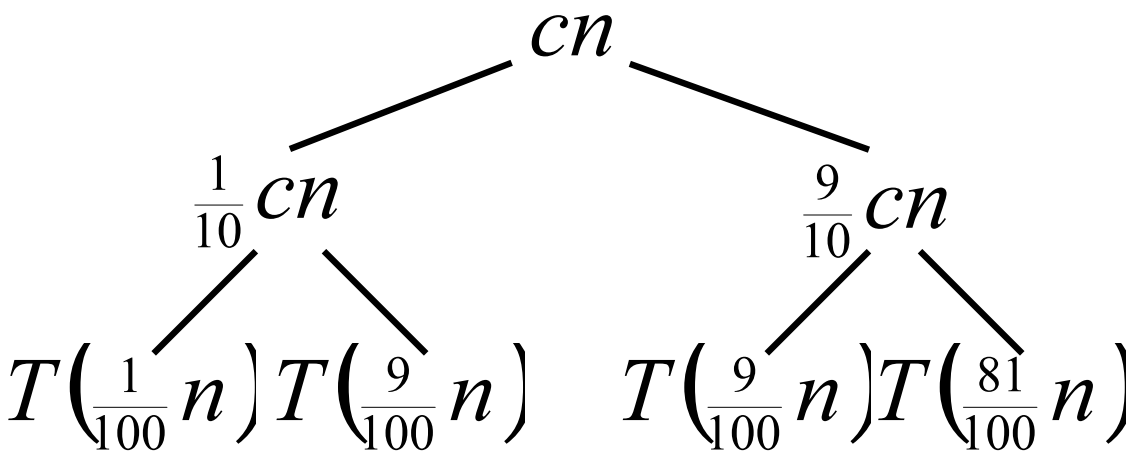


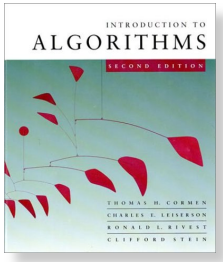
Analysis of "almost-best" case



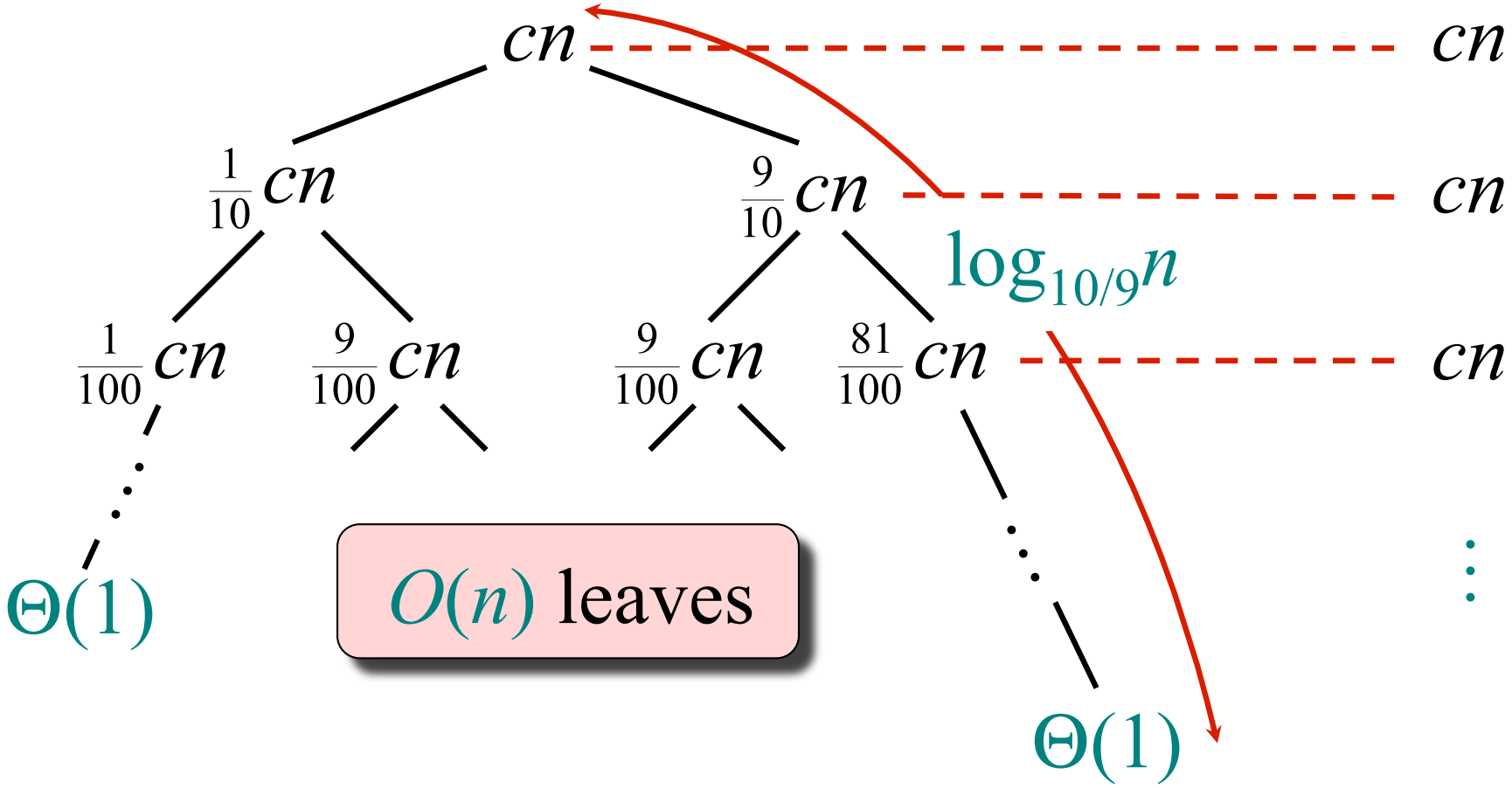


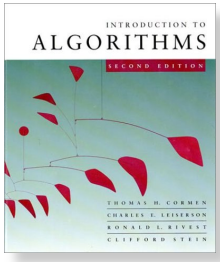
Analysis of "almost-best" case



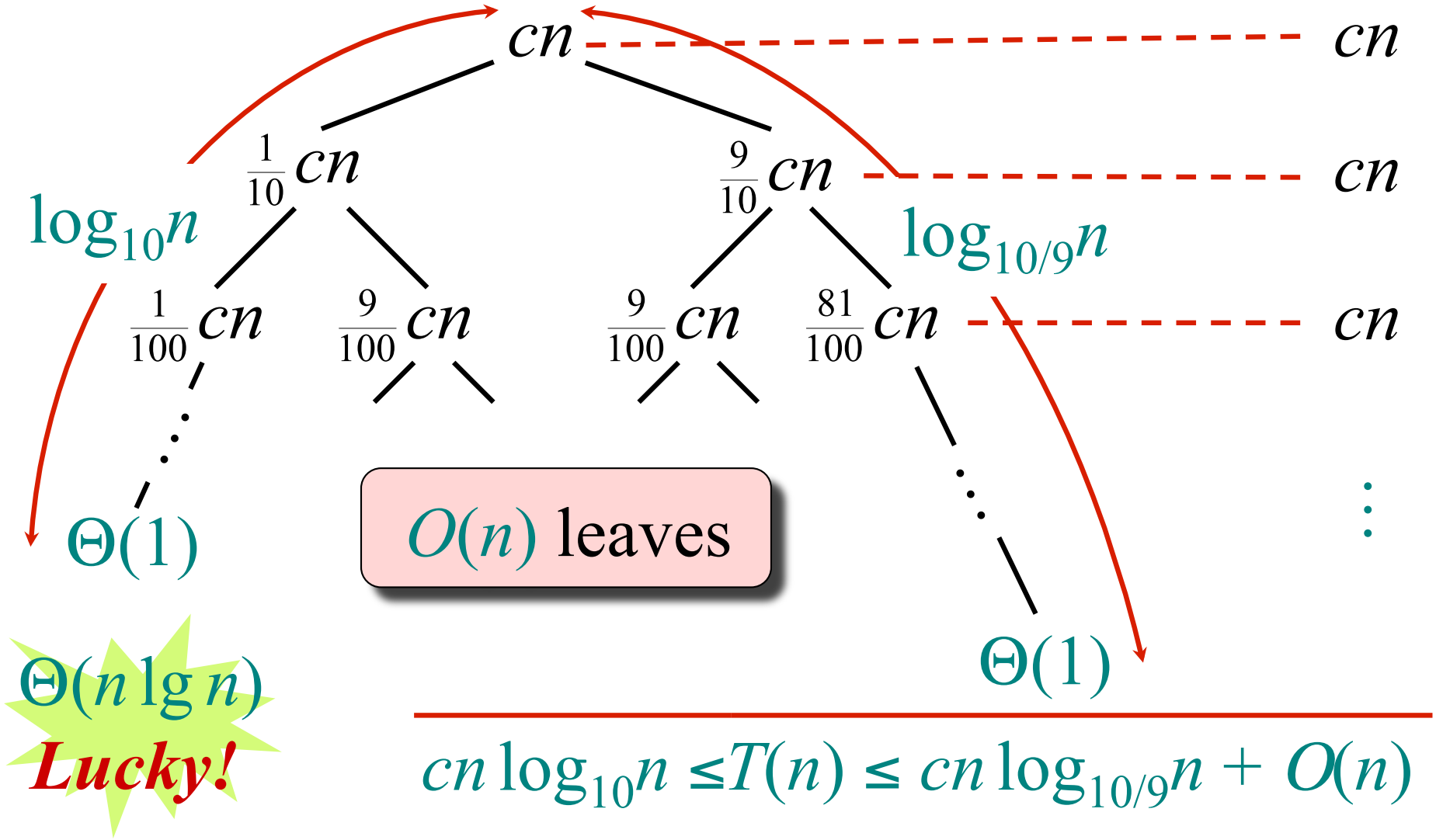


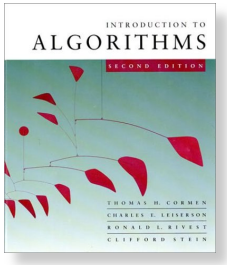
Analysis of "almost-best" case





Analysis of "almost-best" case





More intuition

Suppose we alternate lucky, unlucky, lucky, unlucky, lucky, ….

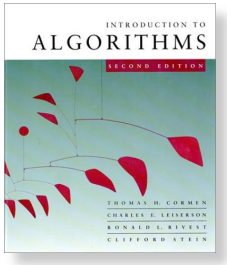
$$L(n) = 2U(n/2) + \Theta(n) \quad \textbf{\textit{lucky}}$$
$$U(n) = L(n-1) + \Theta(n) \quad \textbf{\textit{unlucky}}$$

Solving:

$$\begin{aligned} L(n) &= 2(L(n/2-1) + \Theta(n/2)) + \Theta(n) \\ &= 2L(n/2-1) + \Theta(n) \\ &= \Theta(n \lg n) \end{aligned}$$

Lucky!

How can we make sure we are usually lucky?



Randomized quicksort

IDEA: Partition around a ***random*** element.

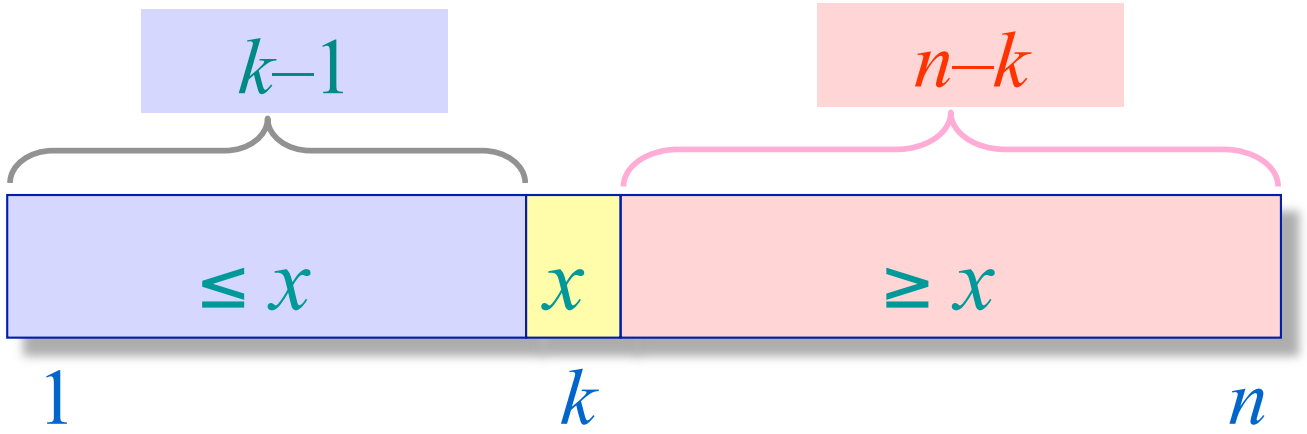
- Running time is independent of the input order.
- No assumptions need to be made about the input distribution.
- No specific input elicits the worst-case behavior.
- The worst case is determined only by the output of a random-number generator.

Analysis of Randomized Quicksort

Let $T(n)$ = the *average* time taken to sort an array of size n using Quicksort

If pivot x ends up in position k,

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



Prob(pivot is at pos k) = $1/n$ *for all* k

Analysis of Randomized Quicksort

Then, we have

$$T(n) = \begin{cases} T(0) + T(n-1) + (n+1) & \text{if } 0 : n-1 \text{ split} \\ T(1) + T(n-2) + (n+1) & \text{if } 1 : n-2 \text{ split} \\ T(2) + T(n-3) + (n+1) & \text{if } 2 : n-3 \text{ split} \\ \vdots & \vdots \\ T(n-1) + T(0) + (n+1) & \text{if } n-1 : 0 \text{ split} \end{cases}$$

Prob(pivot is at pos k) = $1/n$ *for all* k

$$T(n) = \sum_{k=1}^{n} \frac{1}{n} \cdot \left[T(k-1) + T(n-k) + (n+1) \right]$$

Analysis of Randomized Quicksort

Then, we have the following recurrence:

$$T(n) = \sum_{k=1}^{n} \frac{1}{n} \cdot \left[T(k-1) + T(n-k) + (n+1) \right]$$

→*Expand the summations*

Analysis of Randomized Quicksort

Then, we have the following recurrence:

$$T(n) = \sum_{k=1}^{n} \frac{1}{n} \cdot \left[T(k-1) + T(n-k) + (n+1) \right]$$

$$nT(n) = 2\sum_{k=0}^{n-1} T(k) + n(n+1)$$

$$nT(n) = 2\big(T(0) + T(1) + \ldots + T(n-1)\big) + n(n+1)$$

→Get rid of dependence on "full history"

Analysis of Randomized Quicksort

Then, we get rid of "full history":

$$T(n) = \sum_{k=1}^{n} \frac{1}{n} \cdot \left[T(k-1) + T(n-k) + (n+1)\right]$$

$$nT(n) = 2\left[T(0) + T(1) + \ldots + T(n-2) + T(n-1)\right] + n(n+1)$$

$$(n-1)T(n-1) = 2\left[T(0) + T(1) + \ldots + T(n-2)\right] + (n-1)n$$

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

→ ***Divide by n(n+1)…*** (make it telescopic)

Hon Wai Leong, NUS

Analysis of Randomized Quicksort

Divide by $n(n+1)$… (make it telescopic)

$$T(n) = \sum_{k=1}^{n} \frac{1}{n} \cdot \left[T(k-1) + T(n-k) + (n+1)\right]$$

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

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

→ ***Now "telescope"…***

Analysis of Randomized Quicksort

Now, telescope…

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

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

$$= \frac{2}{(n+1)} + \frac{2}{(n)} + \frac{2}{(n-1)} + \frac{T(n-3)}{(n-2)}$$

$$= \frac{2}{(n+1)} + \frac{2}{(n)} + \frac{2}{(n-1)} + \ldots + \frac{2}{(3)} + \frac{T(1)}{(2)}$$

$$\frac{T(n)}{(n+1)} = \frac{T(1)}{(2)} + 2\left[\frac{1}{(n+1)} + \frac{1}{(n)} + \frac{1}{(n-1)} + \ldots + \frac{1}{(3)}\right]$$

Analysis of Randomized Quicksort

Then, we have the following recurrence:

$$T(n) = \sum_{k=1}^{n} \frac{1}{n} \cdot \left[T(k-1) + T(n-k) + (n+1) \right]$$

$$\frac{T(n)}{(n+1)} = \frac{T(1)}{(2)} + 2\left[\frac{1}{(n+1)} + \frac{1}{(n)} + \frac{1}{(n-1)} + \ldots + \frac{1}{(3)} \right]$$

$$T(n) = 2(n+1)H(n+1) + O(n)$$

$$H(n) = \sum_{k=1}^{n} \frac{1}{k} \text{ is the Harmonic series}$$

Analysis of Randomized Quicksort

Avg running time of Randomized Quicksort:

$$T(n) = 2(n+1)H(n+1) + O(n)$$

$$H(n) = \ln n + O(1) \quad \text{[CLRS] - App.A}$$

$$T(n) = 2(n+1)\ln n + O(n)$$

$$T(n) = 1.386 n \lg n + O(n)$$

Randomized Quicksort is ***only* 38.6% *from optimal***.

Optimal sorting is $T^*(n) = (n \lg n)$ [See L.B. for Sorting]

Hon Wai Leong, NUS

Recap…

Beautiful analysis of
Randomized Quicksort to get…

$$T(n) = 1.386 n \lg n + O(n)$$

Not that difficult, *right?*

Where are the key steps?

- Get rid of full history
- Telescope

Recap: The Key Steps

This recurrence depends on *full history*

$$n \cdot T(n) = 2\sum_{k=0}^{n} T(k) + n(n+1)$$

Step 1: *Get rid of full history*… to get

$$n \cdot T(n) = (n+1)T(n-1) + 2n$$

Step 2: Get to a form that can *telescope*…

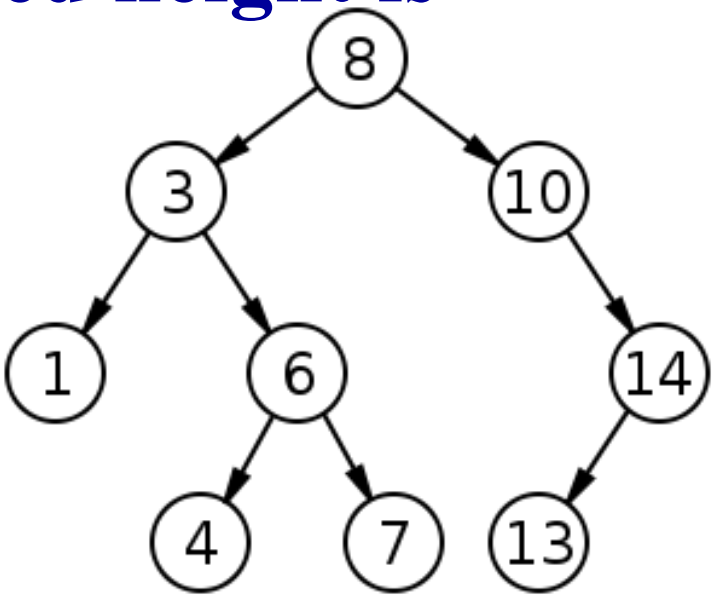
$$\frac{T(n)}{(n+1)} = \frac{T(n-1)}{(n)} + \frac{2}{(n+1)}$$

Hon Wai Leong, NUS

Using the result…

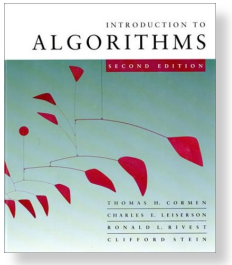
Using a similar analysis, we can show…

- **For a randomly built *n*-node BST (binary search tree), the expected height is**
  - **1.386 lg *n***



- ***Try it out yourself…***

  Or read [CLRS]-C12.4

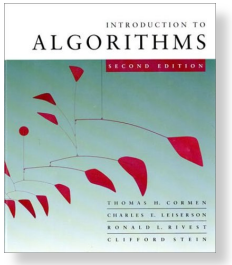


Randomized quicksort analysis [by CLRS]

[CLRS] uses a slightly different analysis …

Let $T(n)$ = the random variable for the running time of randomized quicksort on an input of size n, assuming random numbers are independent.

For $k = 0, 1, \ldots, n-1$, define the ***indicator random variable***



Quicksort in practice

- Quicksort is a great general-purpose sorting algorithm.
- Quicksort is typically over twice as fast as merge sort.
- Quicksort can benefit substantially from ***code tuning***.
- Quicksort behaves well even with caching and virtual memory.

Thank you.

Q & A

CS3230 Lecture 4(c)

NUS National University of Singapore
School of Computing

"Lower Bound for Sorting, Optimal Sorting & Linear-Time Sorting"

❑ Lecture Topics and Readings

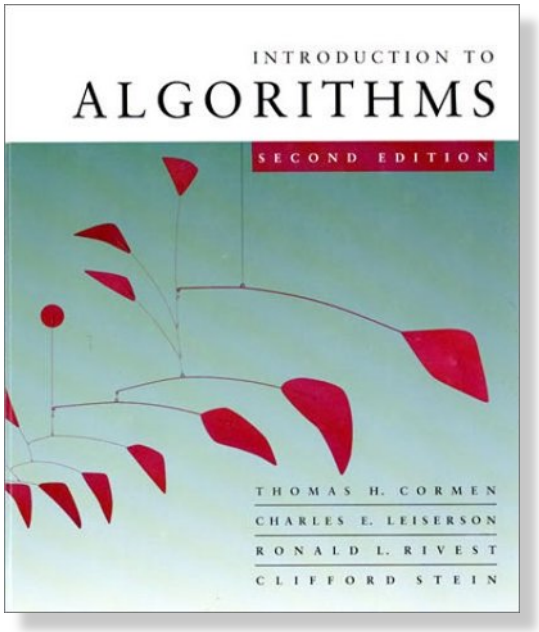
- ❖ Lower Bound for Sorting [CLRS]-C8.1
- ❖ Linear Time Sorting Algorithms [CLRS]-C8.2,8.3

Lower Bound for Sorting,
Optimal Sorting,
Thinking outside the Box,
Busting the Lower Bound

Hon Wai Leong, NUS

Introduction to Algorithms

6.046J/18.401J



LECTURE 5

Sorting Lower Bounds

- Decision trees

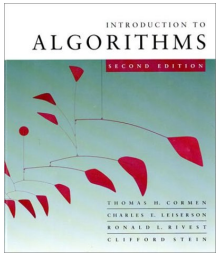
Linear-Time Sorting

- Counting sort
- Radix sort

Appendix: Punched cards

Prof. Erik Demaine

(Modified slightly by LeongHW, 2013/14)



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.

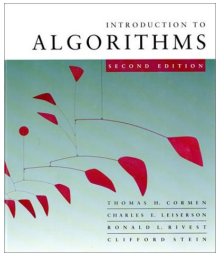
- *E.g.*, insertion sort, merge sort, quicksort, heapsort.

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

Is $O(n \lg n)$ the best we can do?

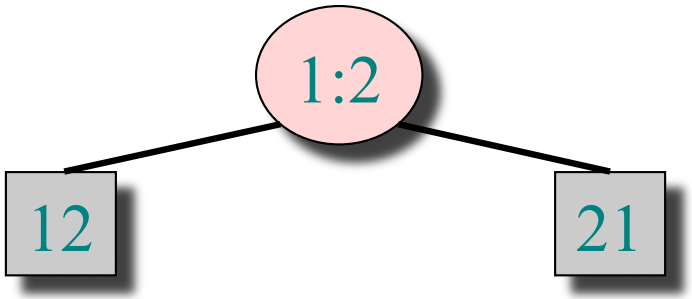
Decision trees can help us answer this question.

(Modified slightly by LeongHW, 2013/14)



To sort 2 numbers

Sort $\langle a_1, a_2 \rangle$

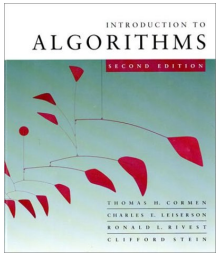


Just one comparison is needed. Trivial

Decision Tree Model

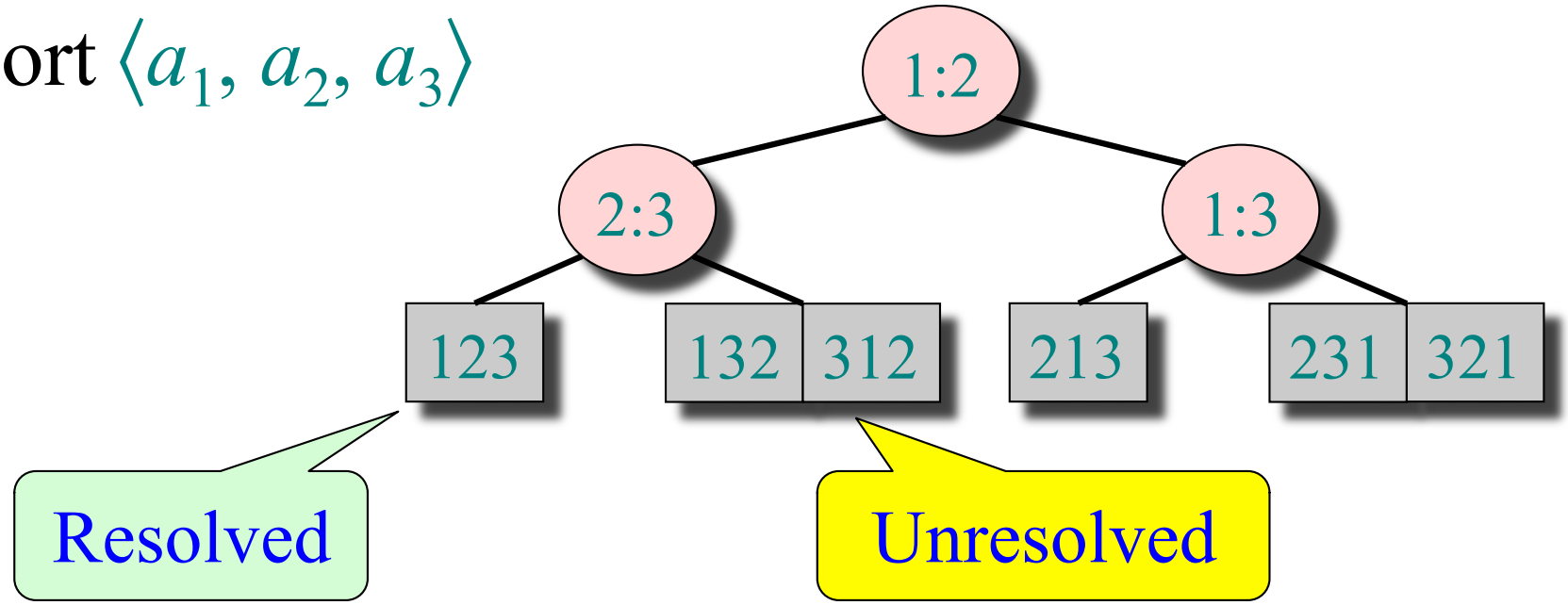
Each internal node is labeled $i{:}j$ for $i, j \in \{1, 2, \ldots, n\}$.

- The left subtree shows subsequent comparisons if $a_i \leq a_j$.
- The right subtree shows subsequent comparisons if $a_i \geq a_j$.



Can we sort 3 numbers with only 2 comparisons?

Sort $\langle a_1, a_2, a_3 \rangle$

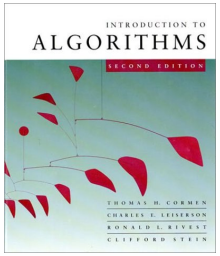


Some (2 of 6) input cases are resolved.

Some (4 of 6) input cases are not fully resolved.

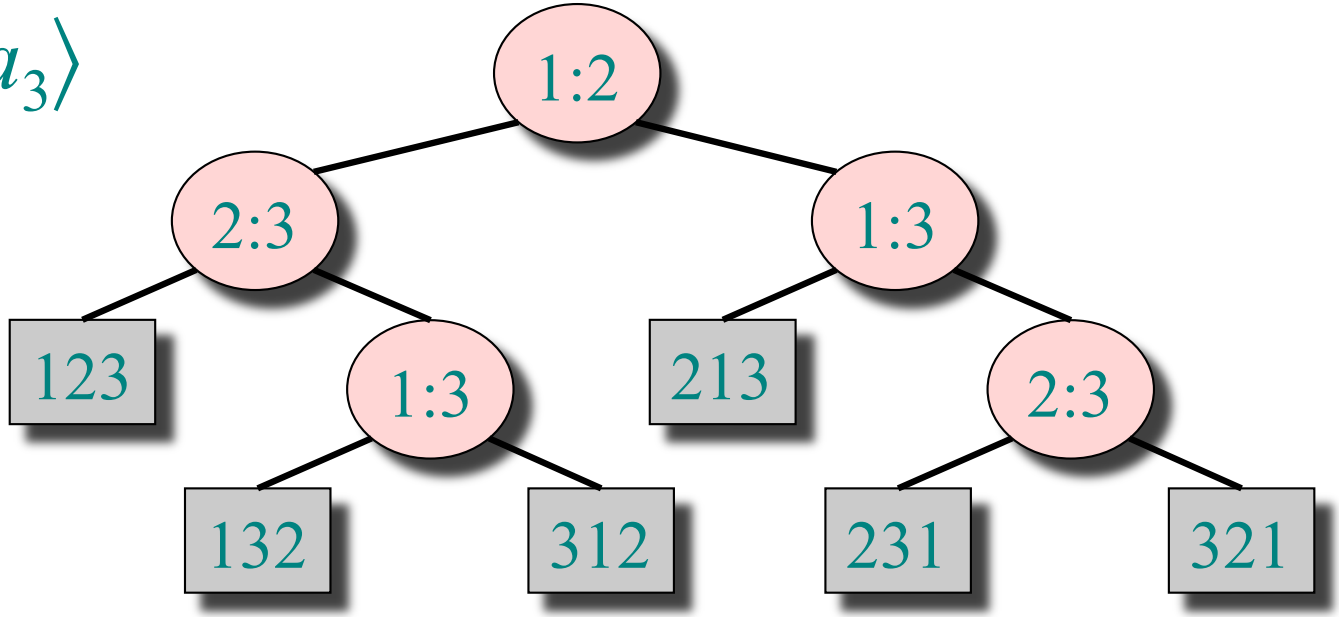
- Need one more comparison to resolve all input cases.

(Modified slightly by LeongHW, 2013/14)



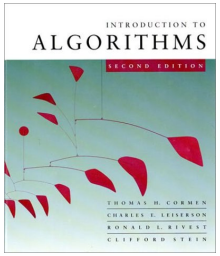
Sorting 3 numbers with 3 comparisons.

Sort $\langle a_1, a_2, a_3 \rangle$



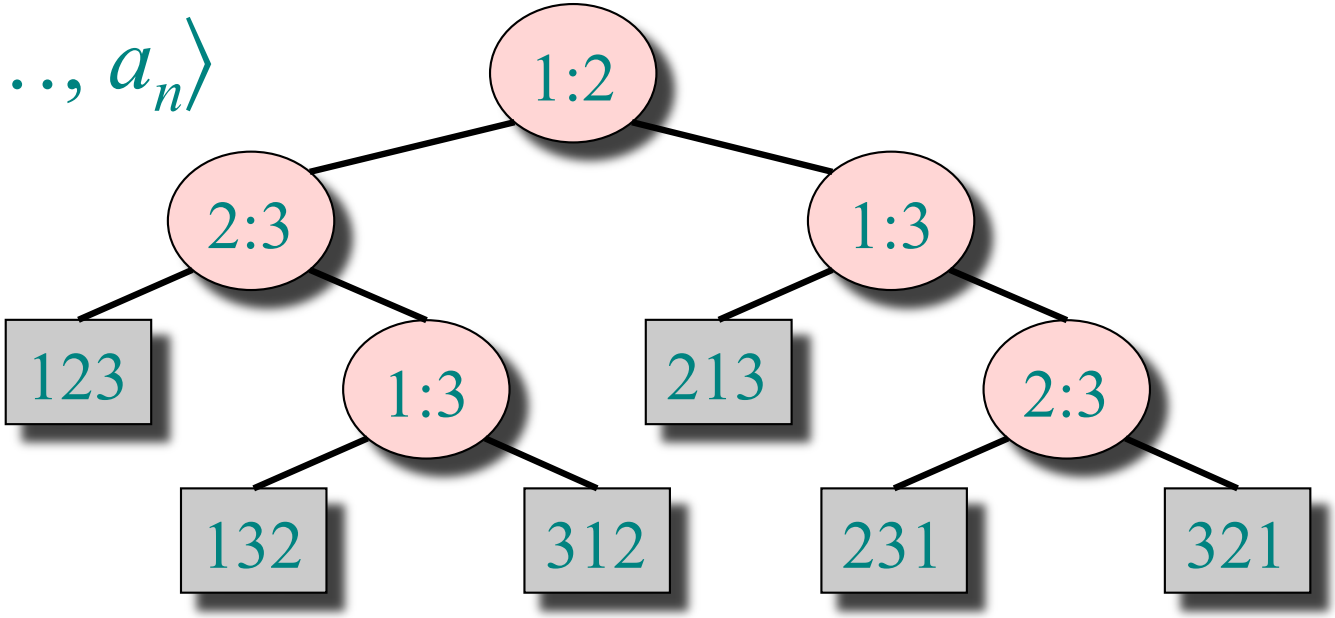
All (6 of 6) input cases are resolved.

There are 6 = 3! possible input cases, all resolved.



Decision-tree example

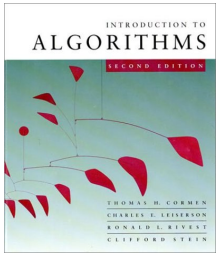
Sort $\langle a_1, a_2, \ldots, a_n \rangle$



Each internal node is labeled $i{:}j$ for $i, j \in \{1, 2, \ldots, n\}$.

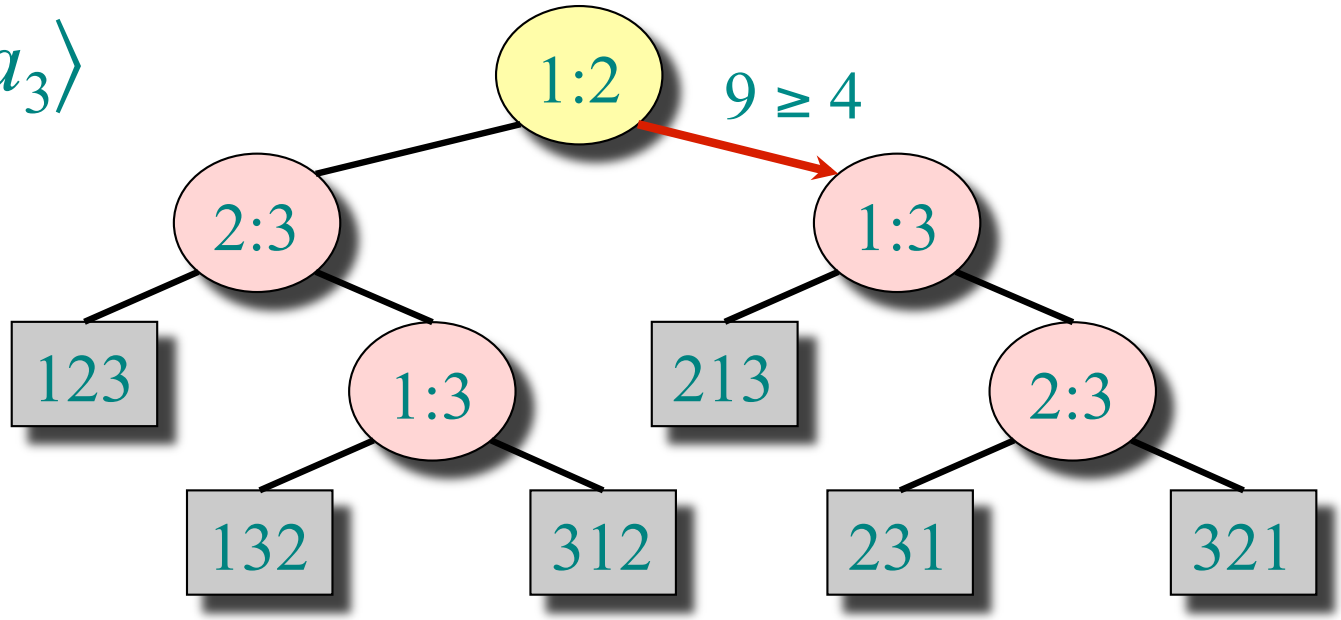
- The left subtree shows subsequent comparisons if $a_i \leq a_j$.
- The right subtree shows subsequent comparisons if $a_i \geq a_j$.

(Modified slightly by LeongHW, 2013/14)



Decision-tree example

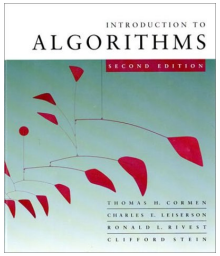
Sort $\langle a_1, a_2, a_3 \rangle$ $= \langle 9, 4, 6 \rangle$:



Each internal node is labeled $i{:}j$ for $i, j \in \{1, 2, \ldots, n\}$.

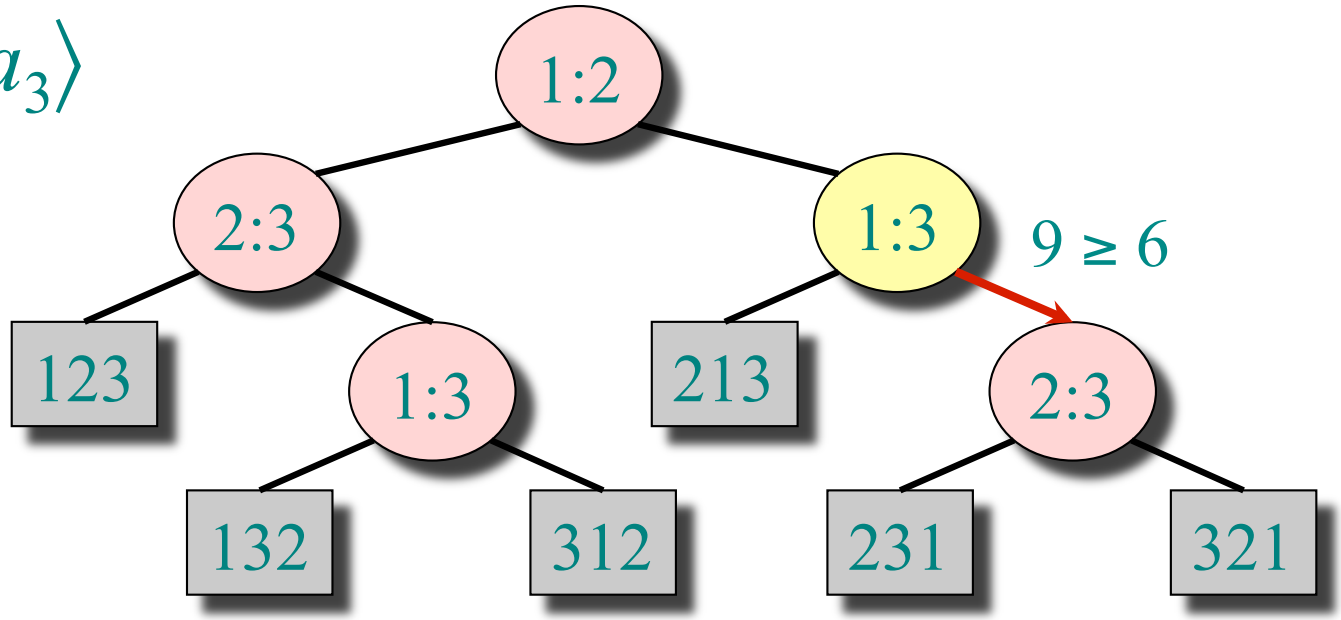
- The left subtree shows subsequent comparisons if $a_i \le a_j$.
- The right subtree shows subsequent comparisons if $a_i \ge a_j$.

(Modified slightly by LeongHW, 2013/14)



Decision-tree example

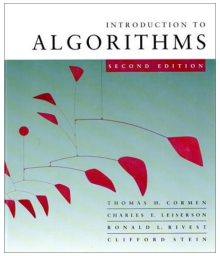
Sort $\langle a_1, a_2, a_3 \rangle$ $= \langle 9, 4, 6 \rangle$:



Each internal node is labeled $i{:}j$ for $i, j \in \{1, 2, \ldots, n\}$.

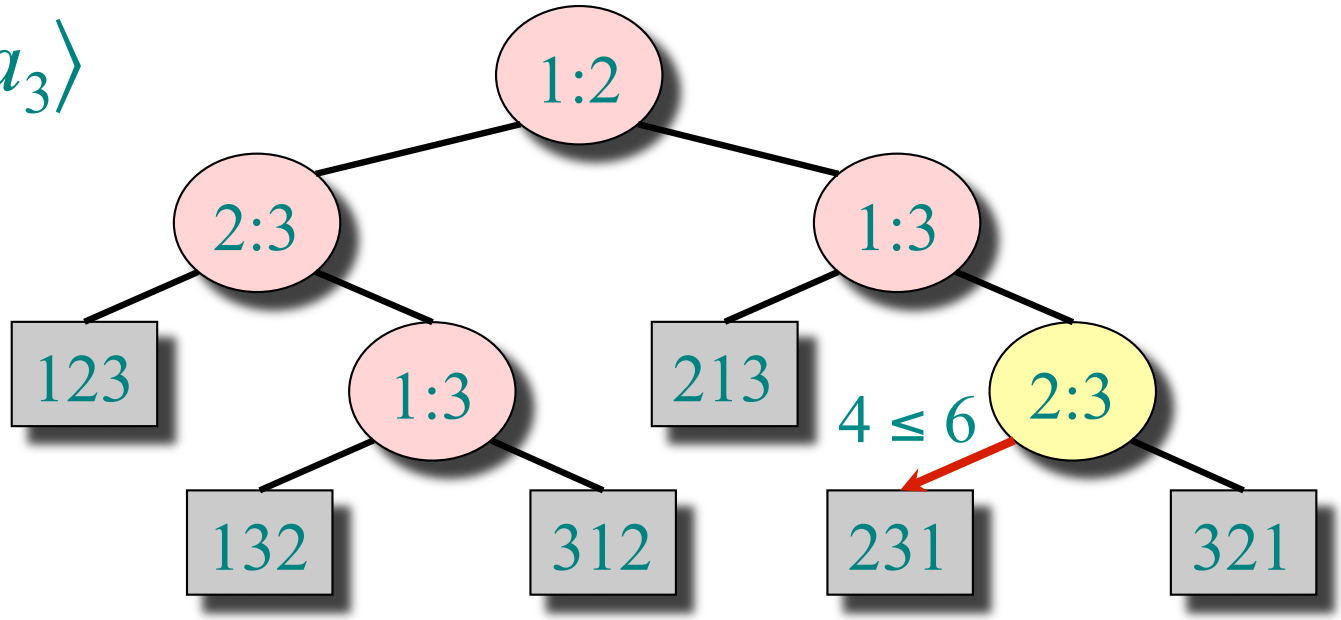
- The left subtree shows subsequent comparisons if $a_i \leq a_j$.
- The right subtree shows subsequent comparisons if $a_i \geq a_j$.

(Modified slightly by LeongHW, 2013/14)



Decision-tree example

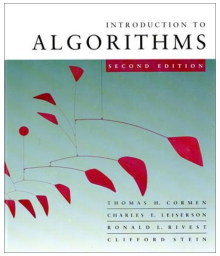
Sort $\langle a_1, a_2, a_3 \rangle$ $= \langle 9, 4, 6 \rangle$:



Each internal node is labeled $i{:}j$ for $i, j \in \{1, 2, \ldots, n\}$.

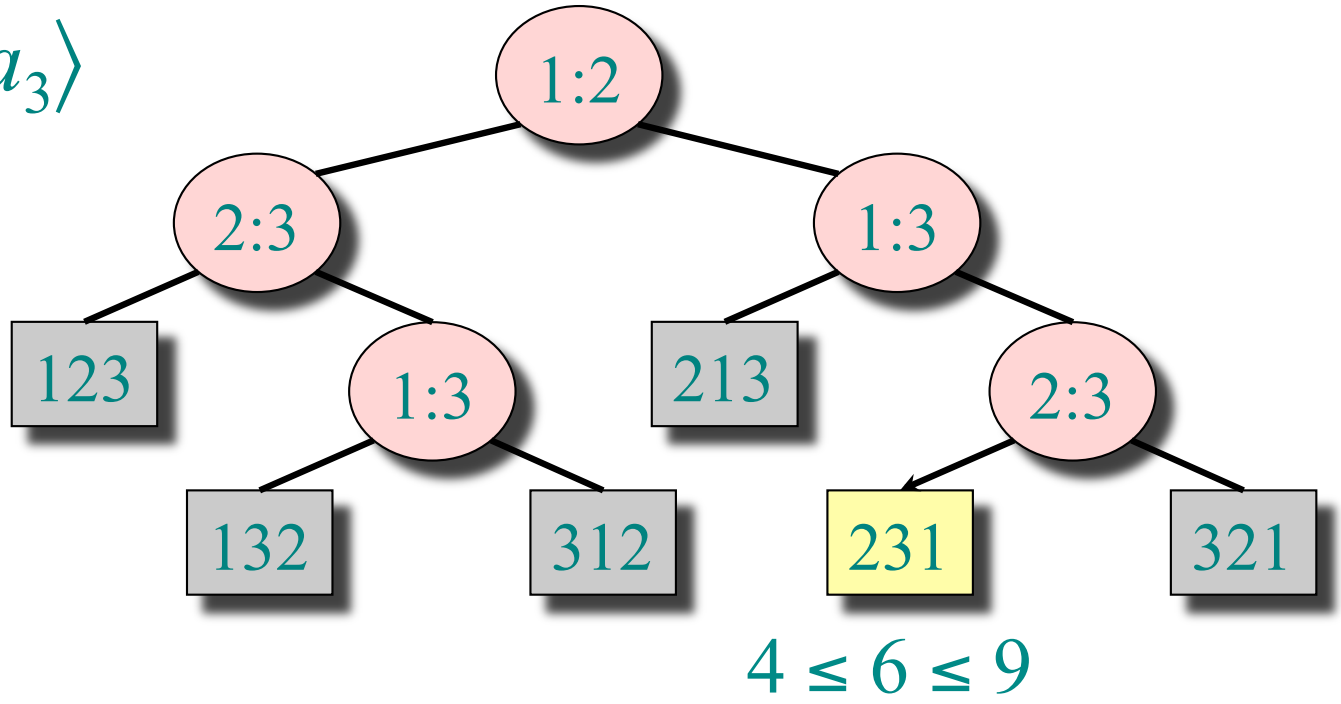
- The left subtree shows subsequent comparisons if $a_i \le a_j$.
- The right subtree shows subsequent comparisons if $a_i \ge a_j$.

(Modified slightly by LeongHW, 2013/14)



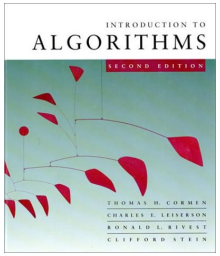
Decision-tree example

Sort $\langle a_1, a_2, a_3 \rangle$ $= \langle\ 9, 4, 6\ \rangle$:



Each leaf contains a permutation $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$ to indicate that the ordering $a_{\pi(1)} \le a_{\pi(2)} \le \cdots \le a_{\pi(n)}$ has been established.

(Modified slightly by LeongHW, 2013/14)

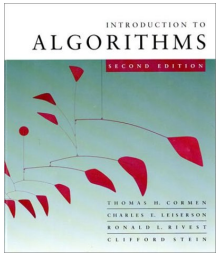


Decision-tree model

A decision tree can model the execution of any comparison sort:

- One tree for each input size n.
- View the algorithm as splitting whenever it compares two elements.
- The tree contains the comparisons along all possible instruction traces.
- The running time of the algorithm = the length of the path taken.
- Worst-case running time = height of tree.

(Modified slightly by LeongHW, 2013/14)



Lower bound for decision-tree sorting

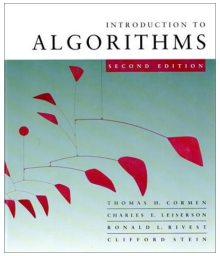
Theorem. Any decision tree that can sort n elements must have height $\Omega(n \lg n)$.

Proof. The tree must contain $\geq n!$ leaves, since there are $n!$ possible permutations. A height-h binary tree has $\leq 2^h$ leaves. Thus, $n! \leq 2^h$.

$$\begin{aligned} \therefore\ h &\geq \lg(n!) && \text{(lg is mono. increasing)} \\ &\geq \lg\left((n/e)^n\right) && \text{(Stirling's formula)} \\ &= n \lg n - n \lg e \\ &= \Omega(n \lg n)\,. \end{aligned}$$

□

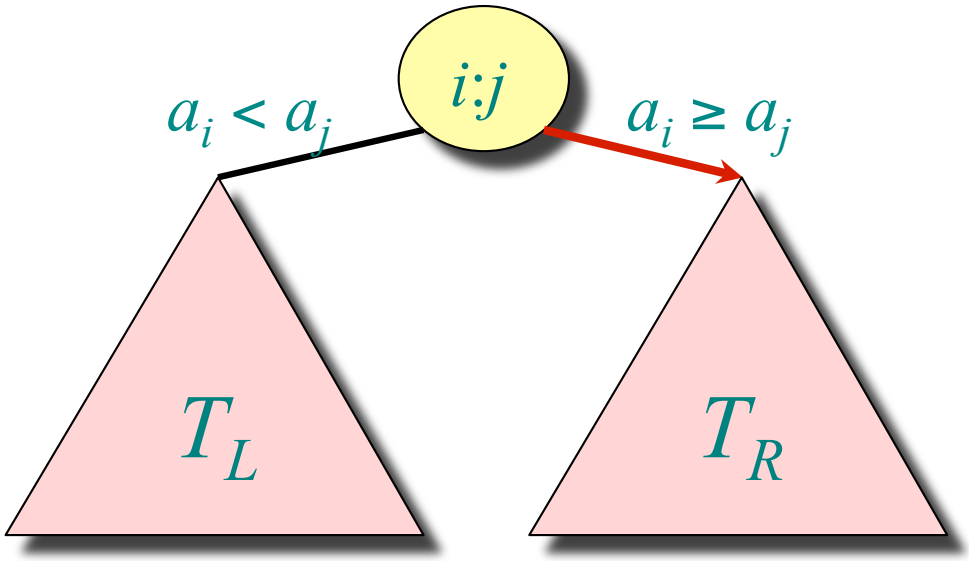
(Modified slightly by LeongHW, 2013/14)



Decision-tree for 4 numbers?

Sort $\langle a_1, a_2, a_3, a_4 \rangle$ $= \langle 9, 4, 6, 7 \rangle$:

Q: How many leaves are there in total?

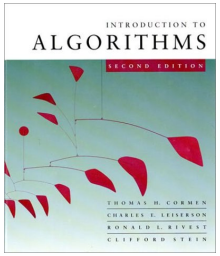


If we compare a_i and a_j

- Left subtree T_L has all perms with $a_i < a_j$.
- Right subtree T_R has all perms with $a_i \geq a_j$.

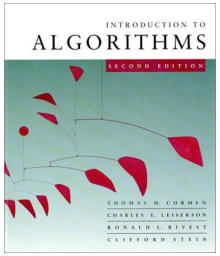
For minimum height, try to balance size of T_L and T_R

(Modified slightly by LeongHW, 2013/14)



Lower bound for comparison sorting

Corollary. Heapsort and merge sort are asymptotically optimal comparison sorting algorithms. □

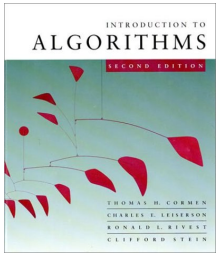


Fun with Sorting Networks

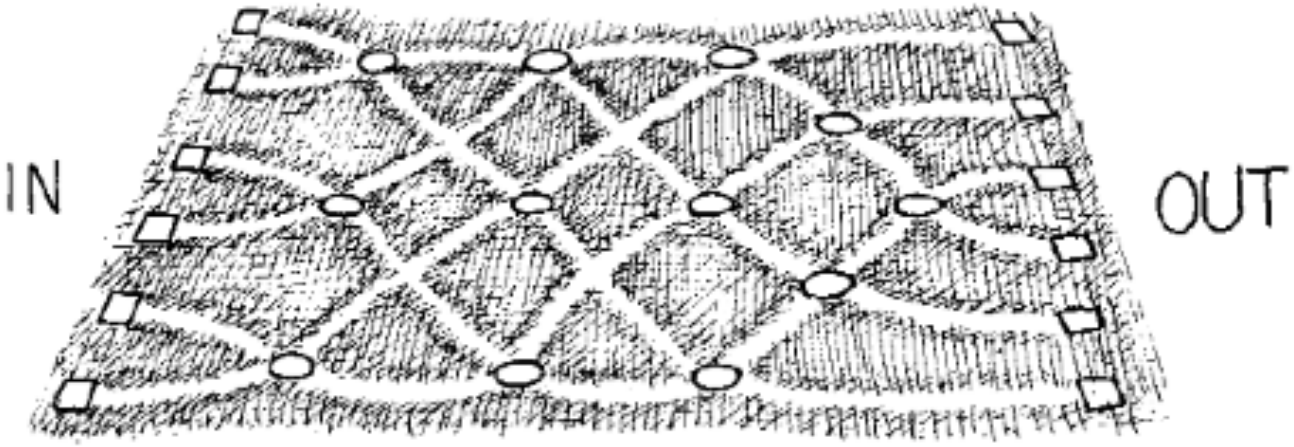
from CS-UnPlugged

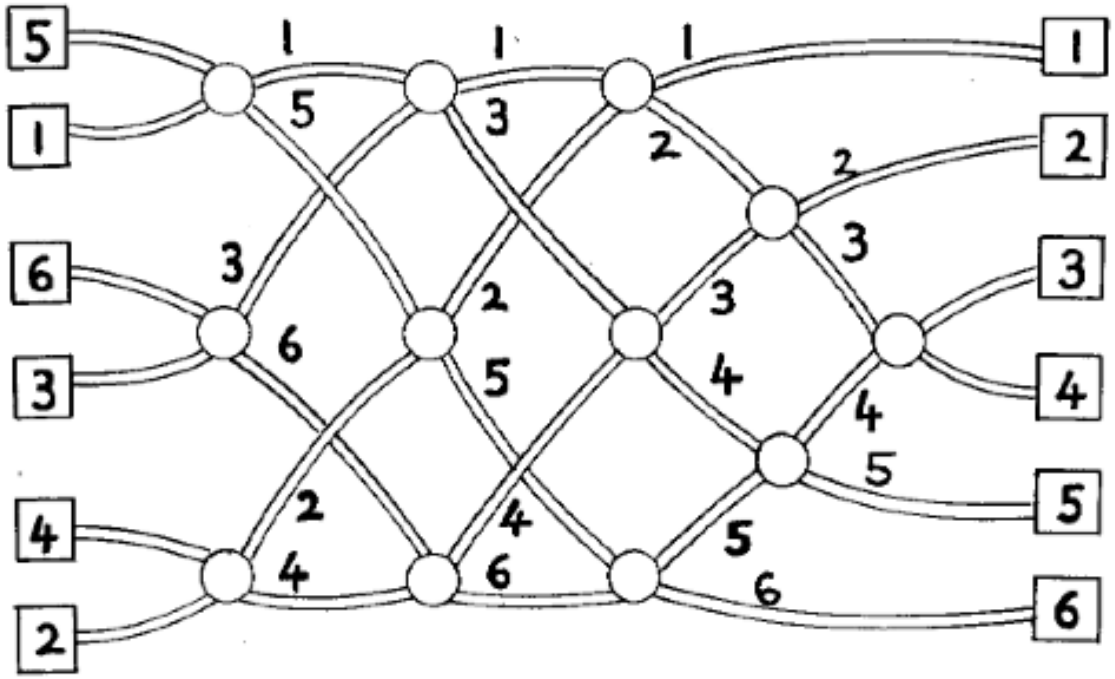
Tim Bell, Mike Fellows, Ian Witten, "CS UnPlugged"
http://csunplugged.org/
Youtube: http://www.youtube.com/watch?v=30WcPnvfiKE#t=58 (1:43 min)

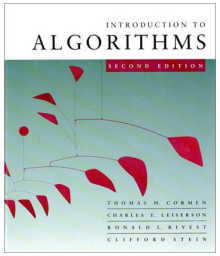
(*Modified slightly by LeongHW, 2013/14*)



Sorting networks…



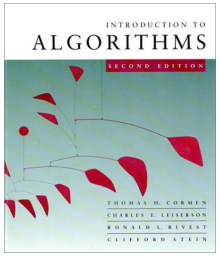




Sorting Network (with Mike Fellows)

Youtube: http://www.youtube.com/watch?v=30WcPnvfiKE#t=58 (1:43 min)

(Modified slightly by LeongHW, 2013/14)



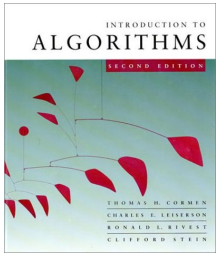
Lessons fr. Sorting Networks

Q1. Build the fastest sorting network for sorting 4 numbers.

Q2. Build the sorting network for *bubble sort algorithm* on 4 numbers.

Q3. Modify the sorting network (for 6 #'s) so that it only *finds the largest*, i.e., moves the largest to the end.

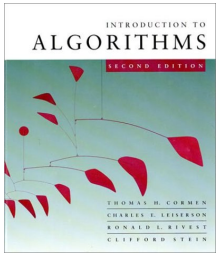
(Modified slightly by LeongHW, 2013/14)



Breaking the (n lg n) Barrier

To Linear Time Sort

(*Modified slightly by LeongHW, 2013/14*)

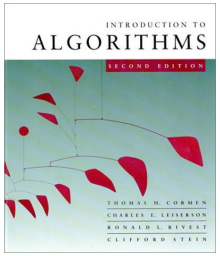


Sorting in linear time

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***: $C[1 \,..\, k]$.

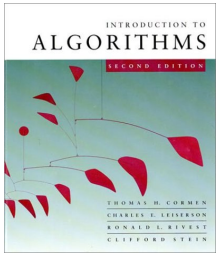
(Modified slightly by LeongHW, 2013/14)



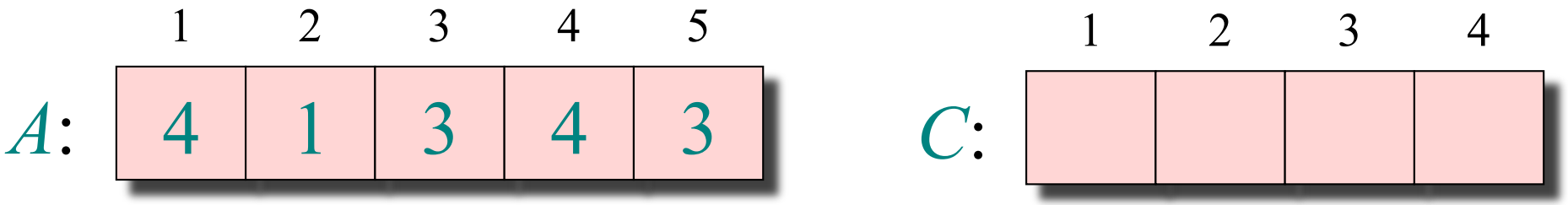
Counting sort

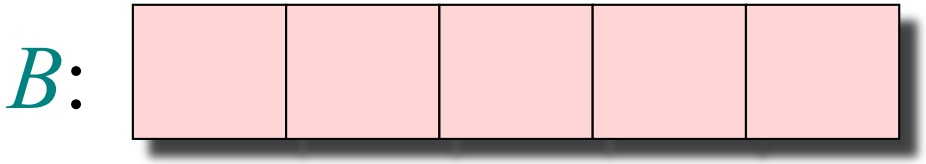
```
for i ← 1 to k
    do C[i] ← 0
for j ← 1 to n
    do C[A[j]] ← C[A[j]] + 1      ▷ C[i] = |{key = i}|
for i ← 2 to k
    do C[i] ← C[i] + C[i–1]       ▷ C[i] = |{key ≤ i}|
for j ← n downto 1
    do B[C[A[j]]] ← A[j]
       C[A[j]] ← C[A[j]] – 1
```

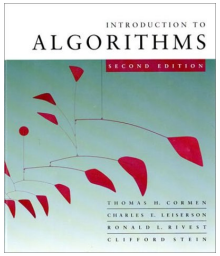
(Modified slightly by LeongHW, 2013/14)



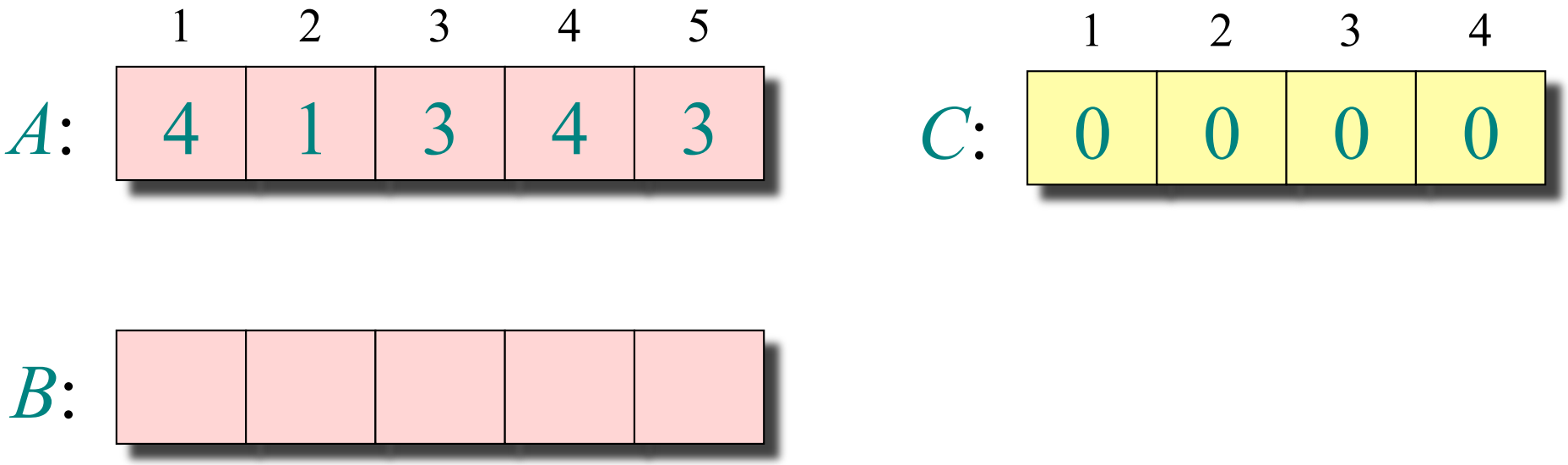
Counting-sort example







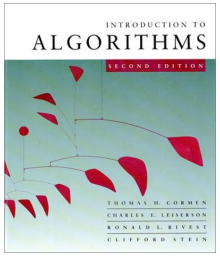
Loop 1



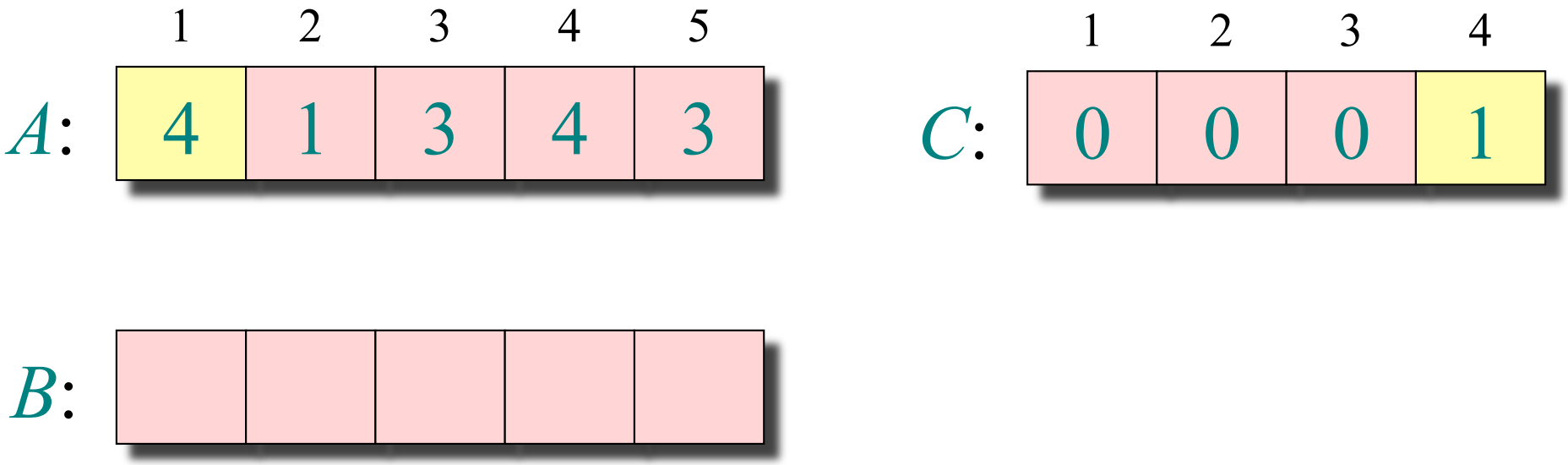
for $i \leftarrow 1$ **to** k

do $C[i] \leftarrow 0$

(Modified slightly by LeongHW, 2013/14)



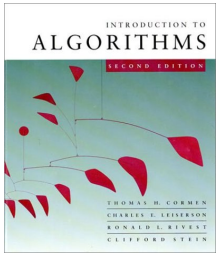
Loop 2



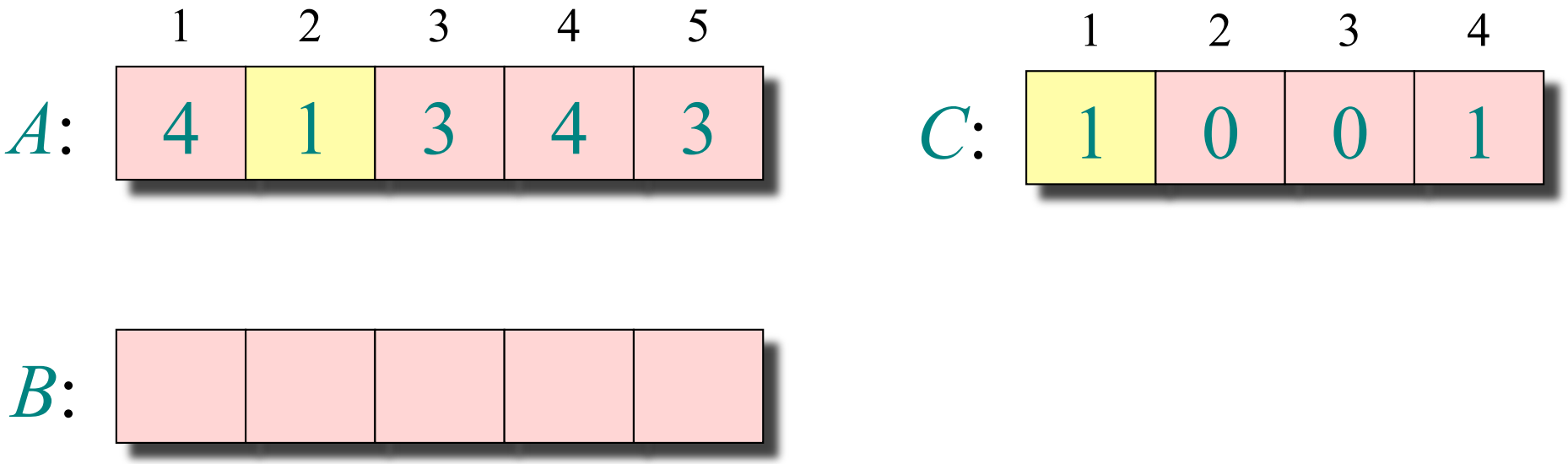
for $j \leftarrow 1$ **to** n

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

(Modified slightly by LeongHW, 2013/14)



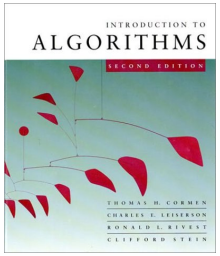
Loop 2



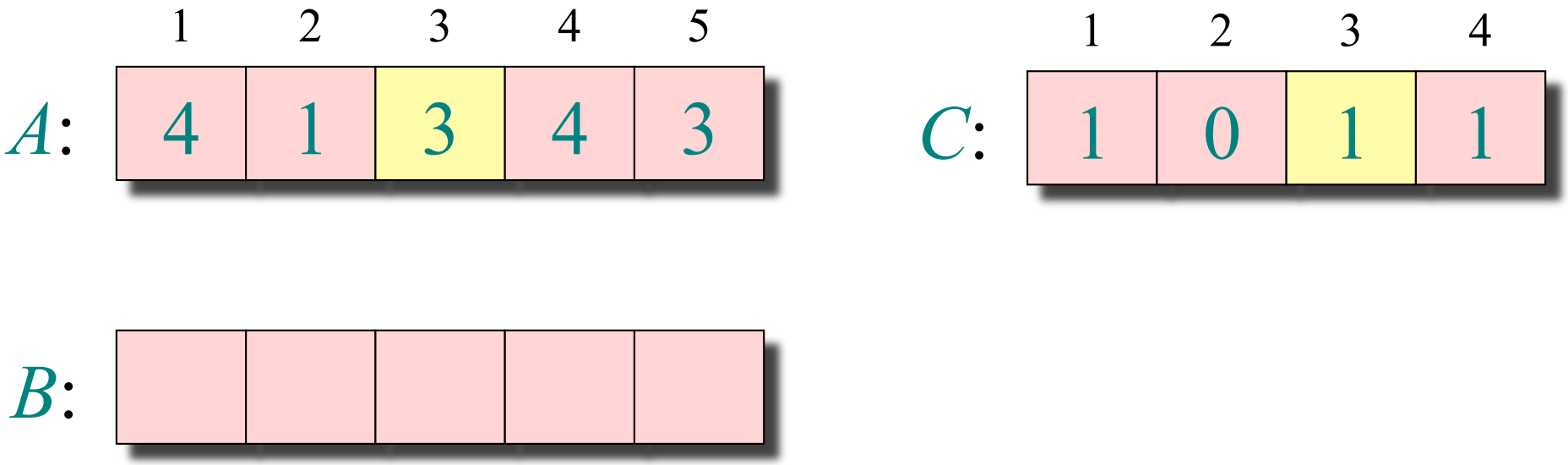
for $j \leftarrow 1$ **to** n

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

(Modified slightly by LeongHW, 2013/14)



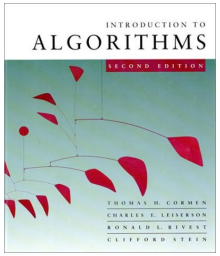
Loop 2



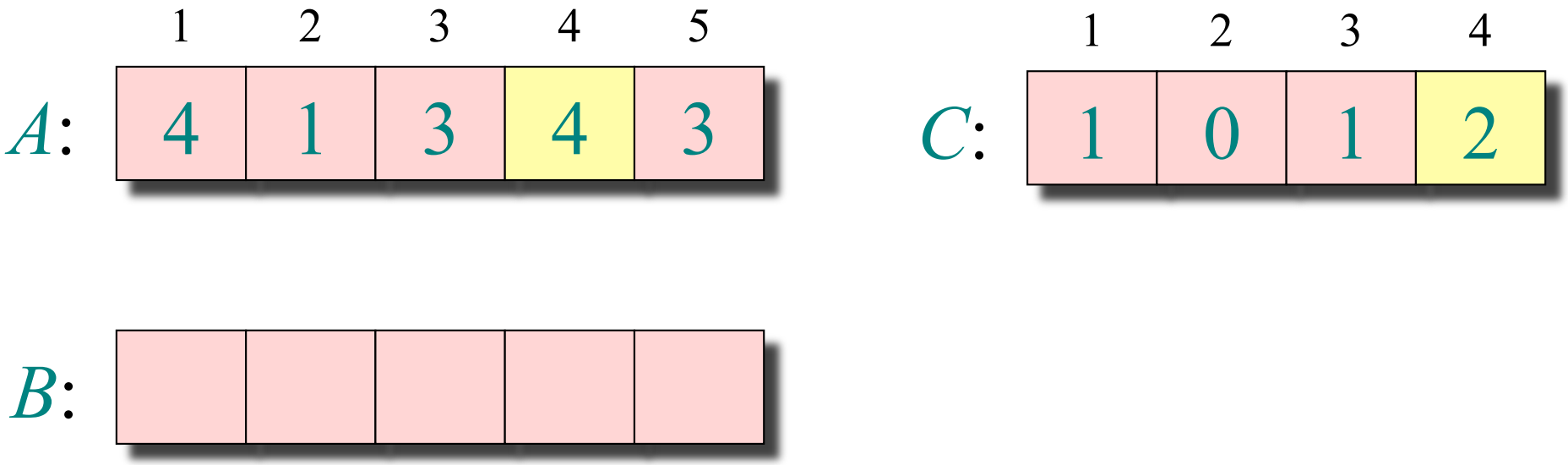
for $j \leftarrow 1$ **to** n

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

(Modified slightly by LeongHW, 2013/14)



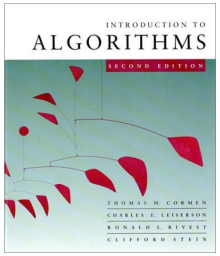
Loop 2



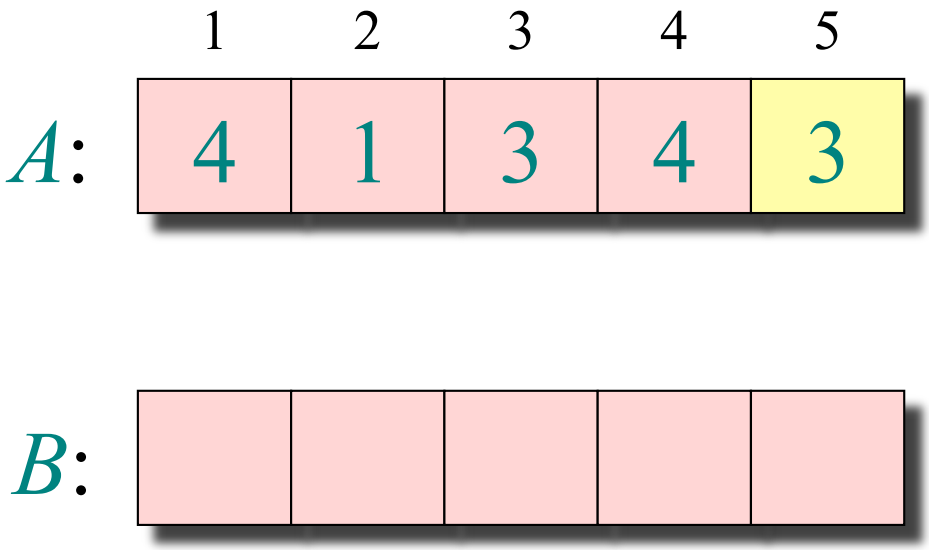
for $j \leftarrow 1$ **to** n

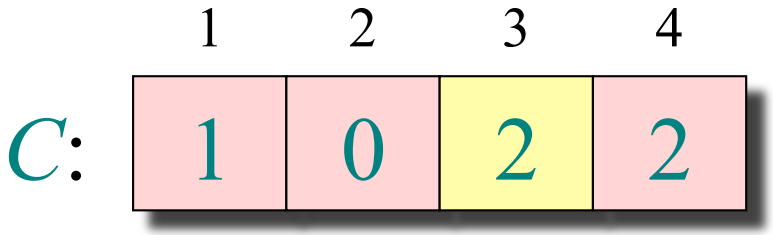
do $C[A[j]] \leftarrow C[A[j]] + 1$ ▷ $C[i] = |\{\text{key} = i\}|$

(Modified slightly by LeongHW, 2013/14)



Loop 2

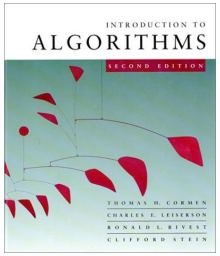




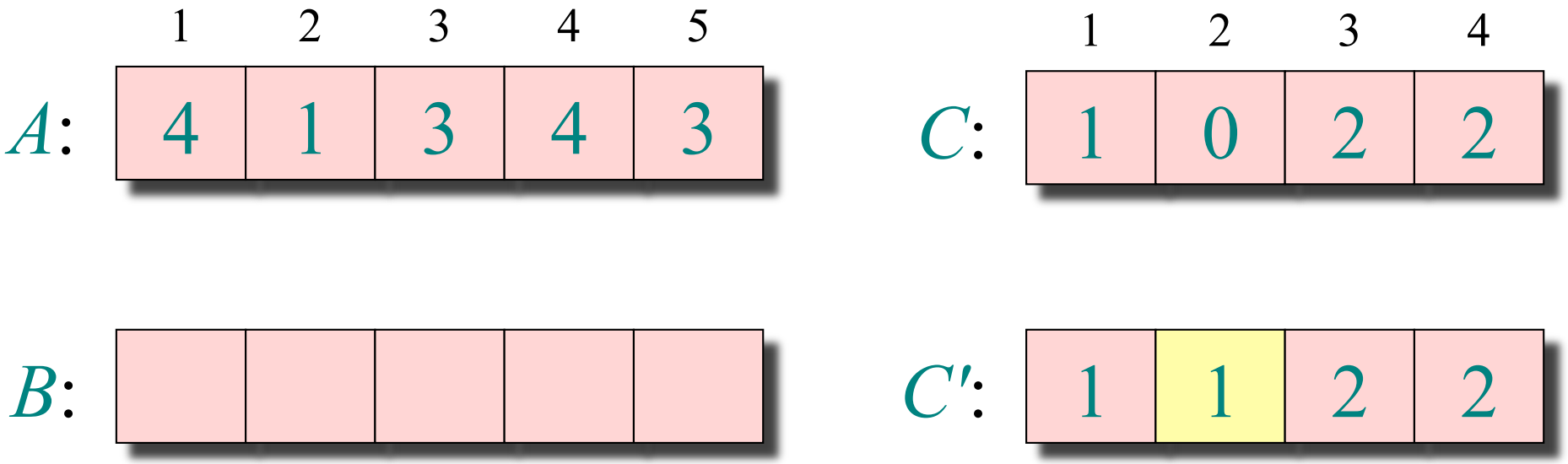
for $j \leftarrow 1$ **to** n

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

(Modified slightly by LeongHW, 2013/14)



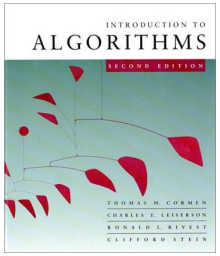
Loop 3



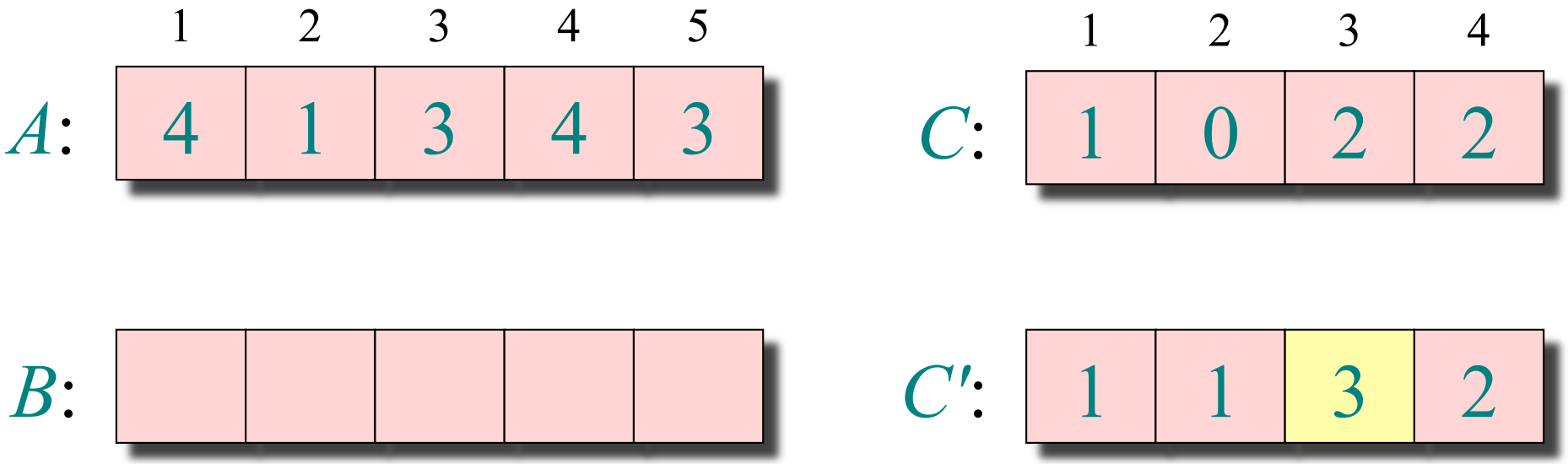
for $i \leftarrow 2$ **to** k

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

(Modified slightly by LeongHW, 2013/14)



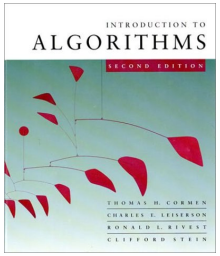
Loop 3



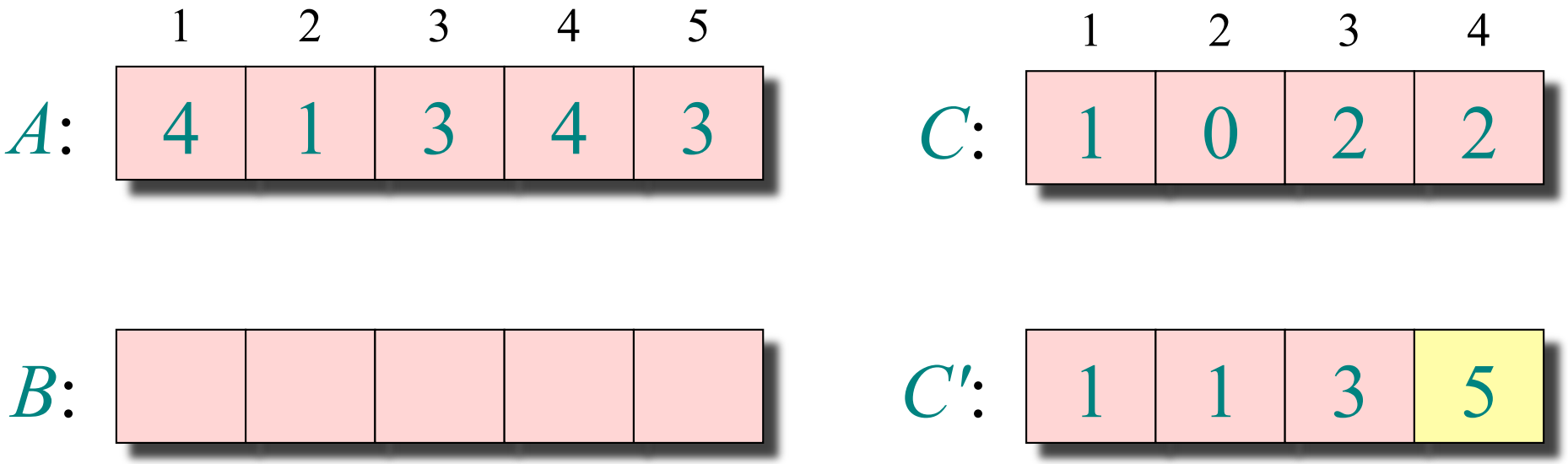
for $i \leftarrow 2$ **to** k

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

(Modified slightly by LeongHW, 2013/14)



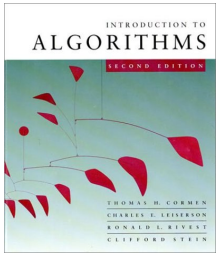
Loop 3



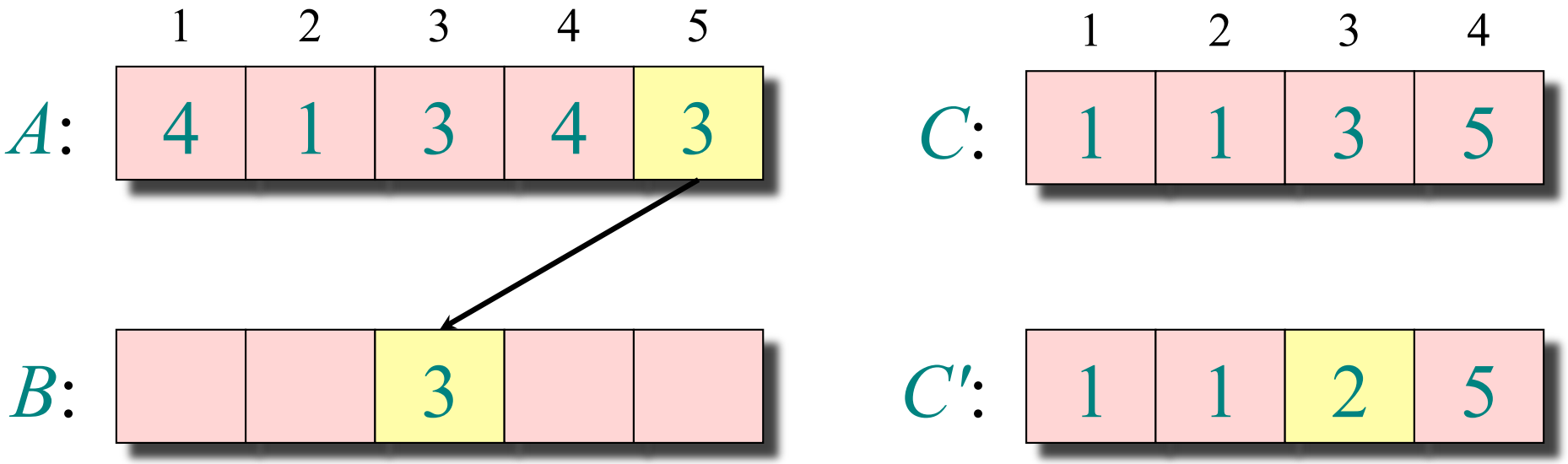
for $i \leftarrow 2$ **to** k

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

(Modified slightly by LeongHW, 2013/14)

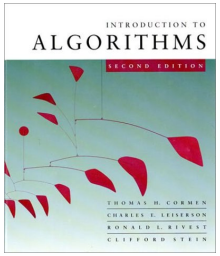


Loop 4

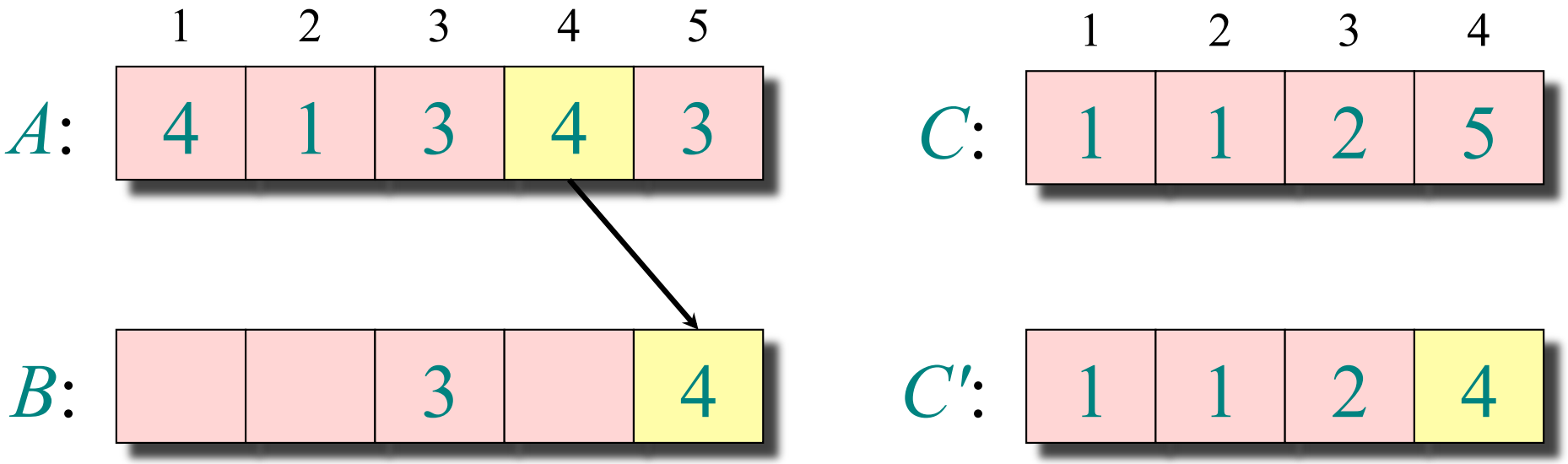


for $j \leftarrow n$ **downto** 1
  do $B[C[A[j]]] \leftarrow A[j]$
    $C[A[j]] \leftarrow C[A[j]] - 1$

(Modified slightly by LeongHW, 2013/14)

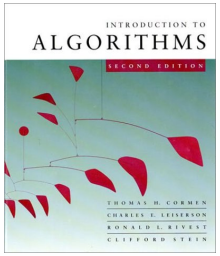


Loop 4

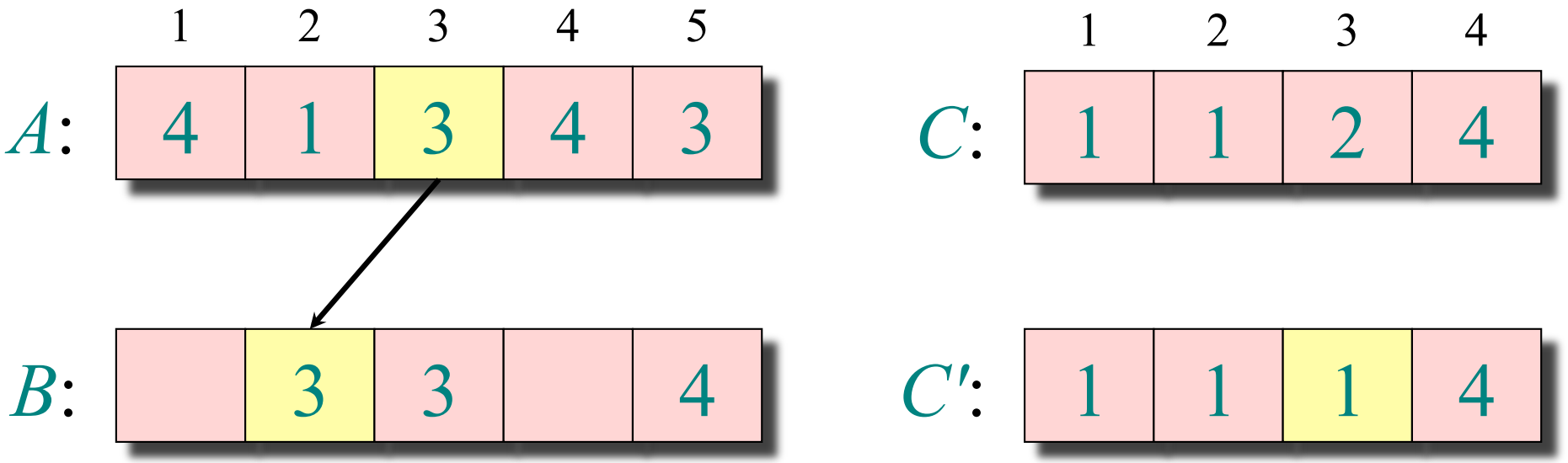


for $j \leftarrow n$ **downto** 1
  do $B[C[A[j]]] \leftarrow A[j]$
    $C[A[j]] \leftarrow C[A[j]] - 1$

(Modified slightly by LeongHW, 2013/14)

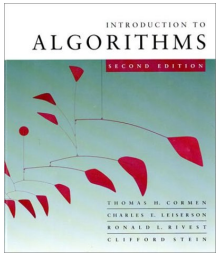


Loop 4

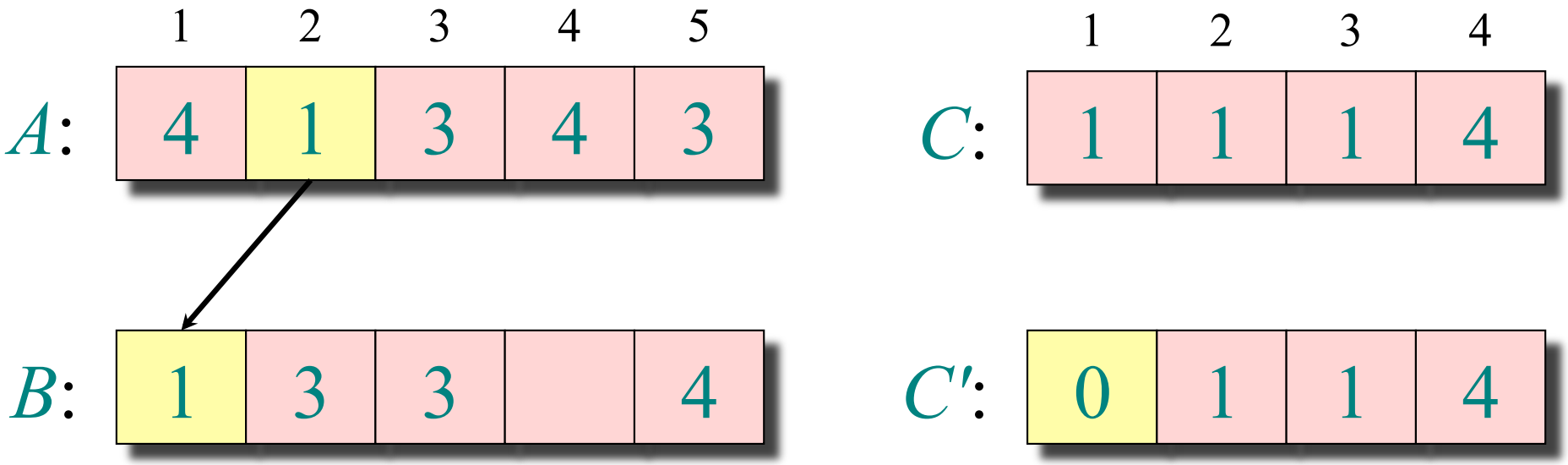


for $j \leftarrow n$ **downto** 1
  do $B[C[A[j]]] \leftarrow A[j]$
    $C[A[j]] \leftarrow C[A[j]] - 1$

(Modified slightly by LeongHW, 2013/14)

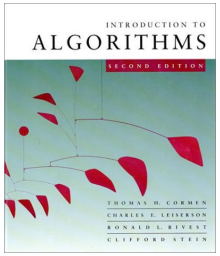


Loop 4

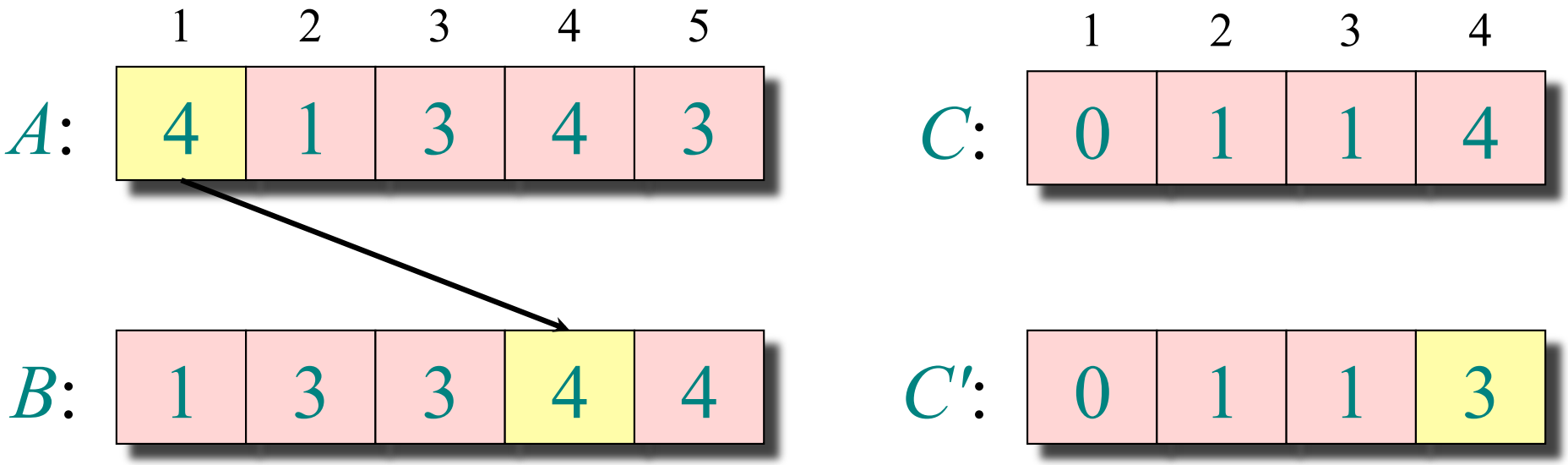


for $j \leftarrow n$ **downto** 1
  do $B[C[A[j]]] \leftarrow A[j]$
    $C[A[j]] \leftarrow C[A[j]] - 1$

(Modified slightly by LeongHW, 2013/14)



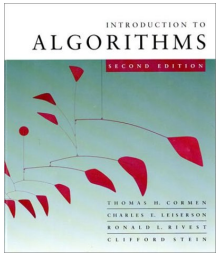
Loop 4



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

What if we go
for $j \leftarrow 1$ **to** n ?
Will it still sort?

(Modified slightly by LeongHW, 2013/14)



Analysis

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

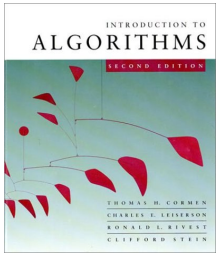
$\Theta(n)$
for $j \leftarrow 1$ **to** n
  do $C[A[j]] \leftarrow C[A[j]] + 1$

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

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

$\Theta(n + k)$

(Modified slightly by LeongHW, 2013/14)



Running time

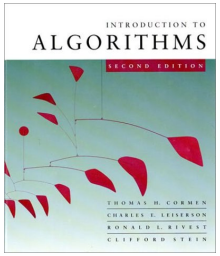
If $k = O(n)$, then counting sort takes $\Theta(n)$ time.

- But, sorting takes $\Omega(n \lg n)$ time!
- Where's the fallacy?

Answer:

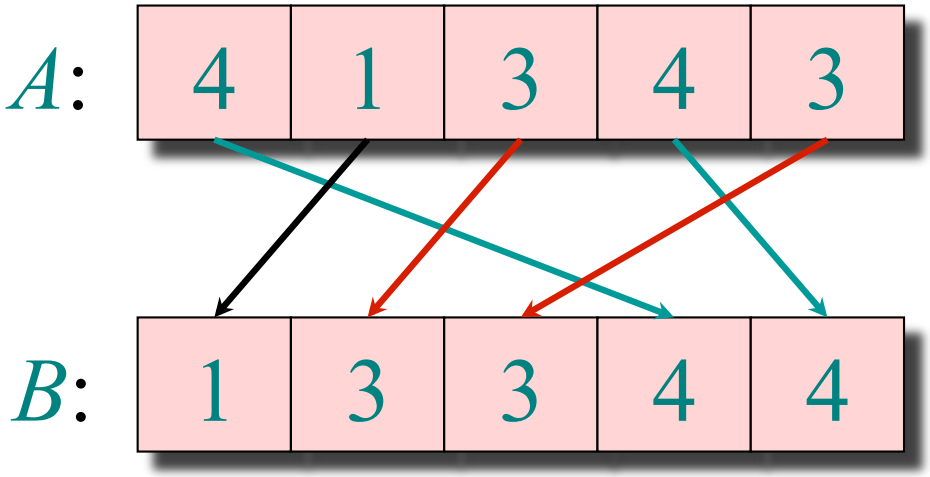
- ***Comparison sorting*** takes $\Omega(n \lg n)$ time.
- Counting sort is not a ***comparison sort***.
- In fact, not a single comparison between elements occurs!

(Modified slightly by LeongHW, 2013/14)



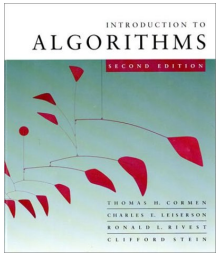
Stable sorting

Counting sort is a ***stable*** sort: it preserves the input order among equal elements.



Exercise: What other sorts have this property?

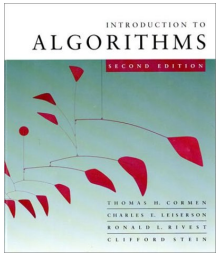
(Modified slightly by LeongHW, 2013/14)



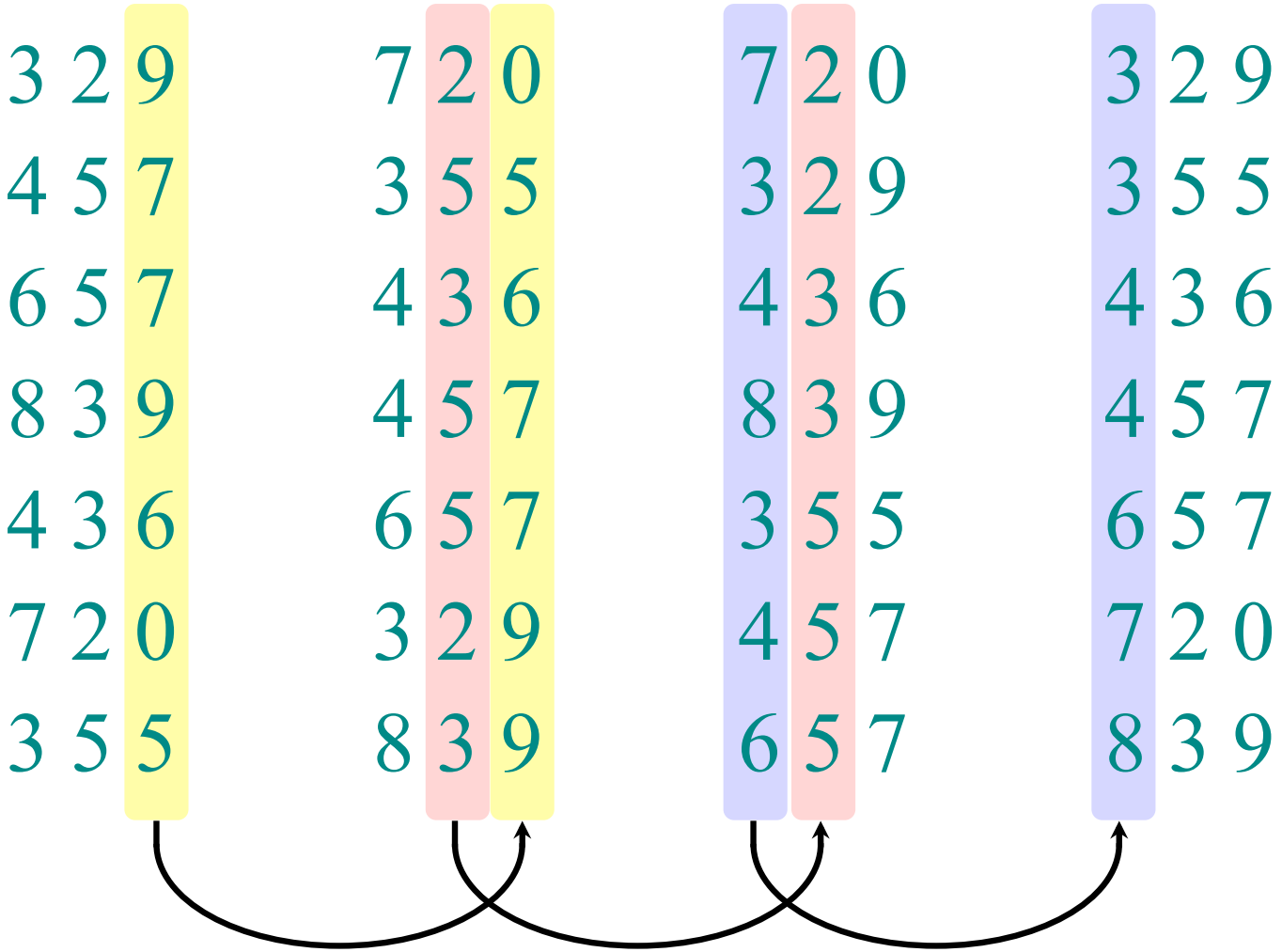
Radix sort

- ***Origin***: Herman Hollerith's card-sorting machine for the 1890 U.S. Census. (See Appendix  .)
- Digit-by-digit sort.
- Hollerith's original (bad) idea: sort on most-significant digit first.
- Good idea: Sort on ***least-significant digit first*** with auxiliary ***stable*** sort.

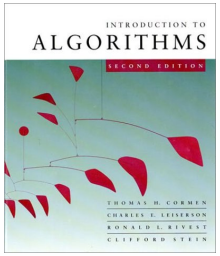
(Modified slightly by LeongHW, 2013/14)



Operation of radix sort



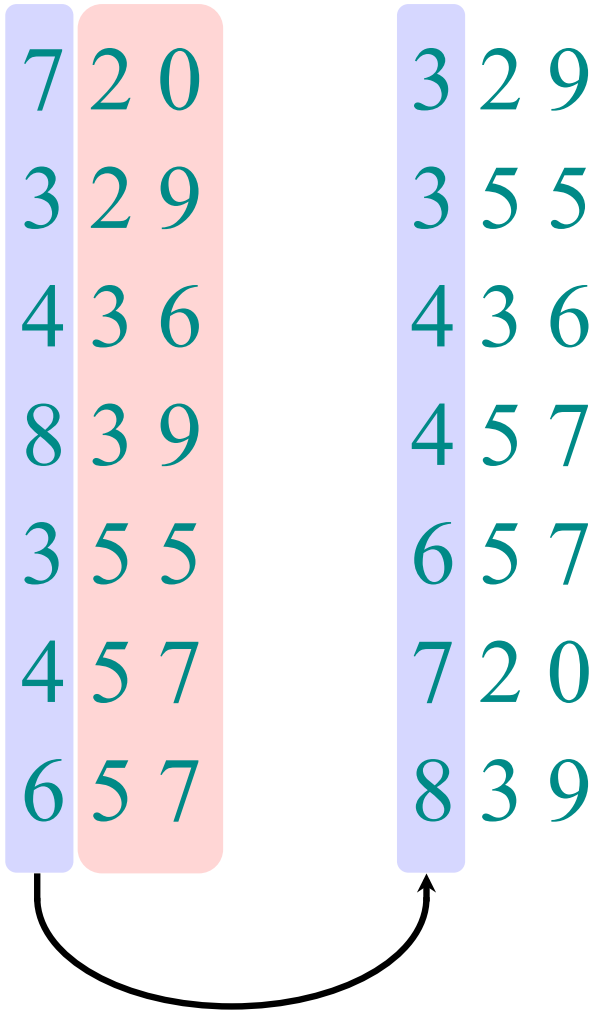
(Modified slightly by LeongHW, 2013/14)



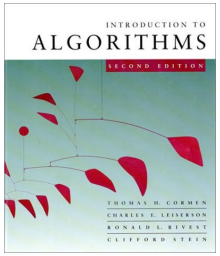
Correctness of radix sort

Induction on digit position

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



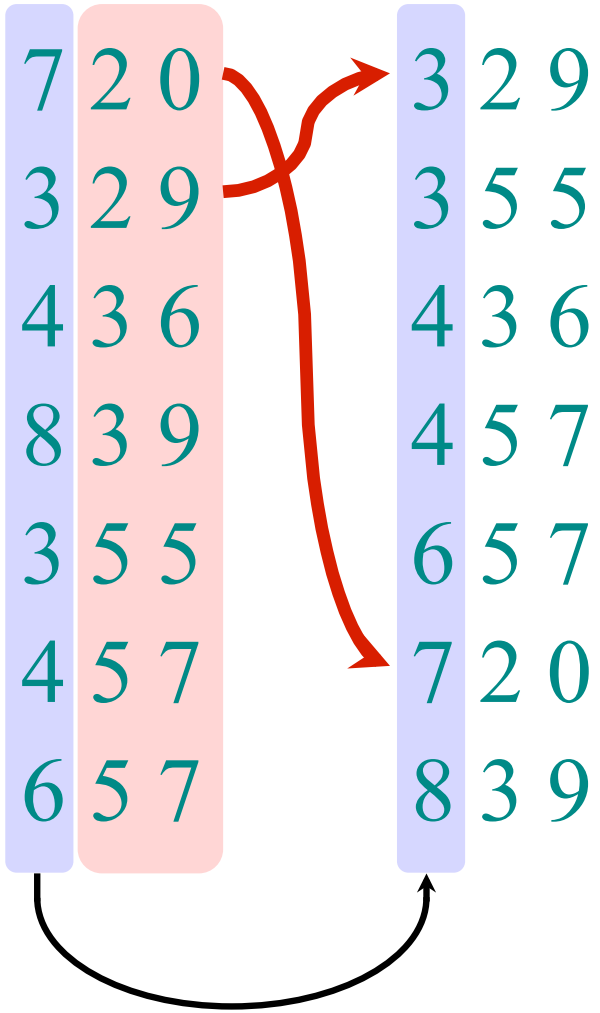
(Modified slightly by LeongHW, 2013/14)



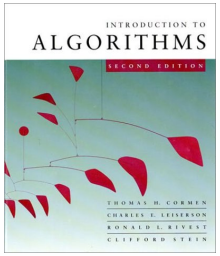
Correctness of radix sort

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.



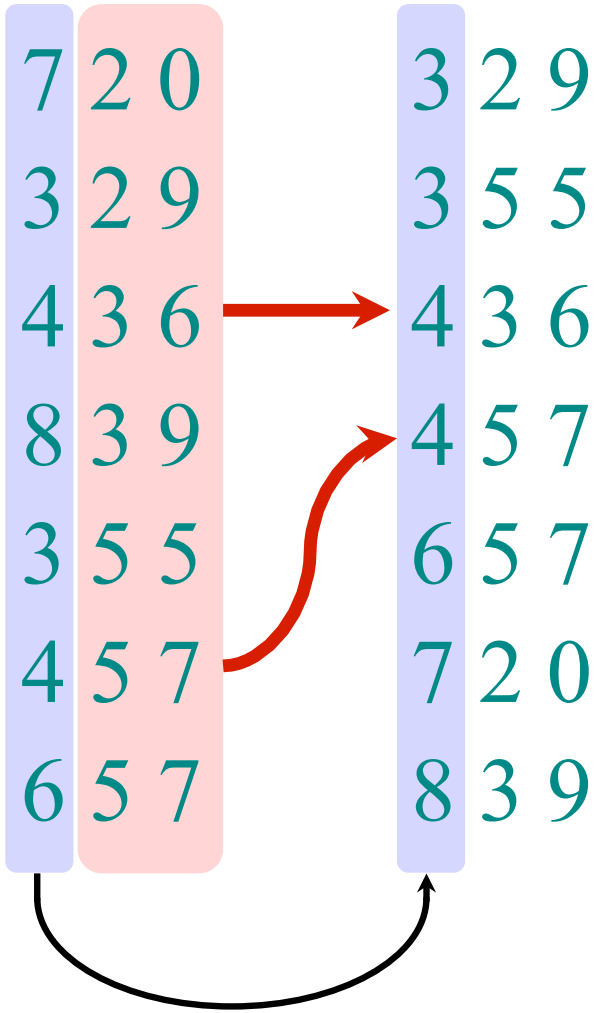
(Modified slightly by LeongHW, 2013/14)



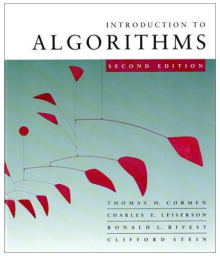
Correctness of radix sort

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.



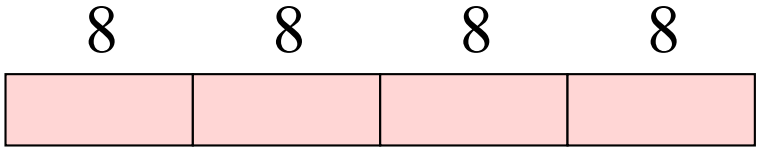
(Modified slightly by LeongHW, 2013/14)



Analysis of radix sort

- Assume counting sort is the auxiliary stable sort.
- Sort n computer words of b bits each.
- Each word can be viewed as having b/r base-2^r digits.

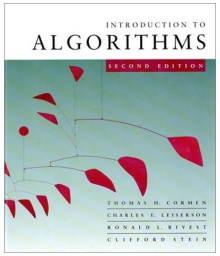
Example: 32-bit word



$r = 8 \Rightarrow b/r = 4$ passes of counting sort on base-2^8 digits; or $r = 16 \Rightarrow b/r = 2$ passes of counting sort on base-2^{16} digits.

How many passes should we make?

(Modified slightly by LeongHW, 2013/14)



Analysis (continued)

Recall: Counting sort takes $\Theta(n + k)$ time to sort n numbers in the range from 0 to $k - 1$.

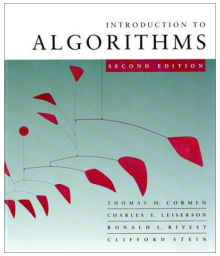
If each b-bit word is broken into r-bit pieces, each pass of counting sort takes $\Theta(n + 2^r)$ time. Since there are b/r passes, we have

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

Choose r to minimize $T(n,b)$:

- Increasing r means fewer passes, but as $r \gg \lg n$, the time grows exponentially.

(Modified slightly by LeongHW, 2013/14)



Choosing r

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

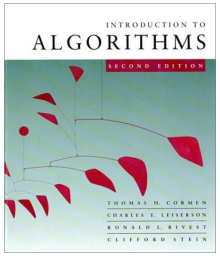
Minimize $T(n,b)$ by differentiating and setting to 0.

Or, just observe that we don't want $2^r \gg n$, and there's 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 $b = d \lg n \Rightarrow$ radix sort runs in $\Theta(dn)$ time.

(Modified slightly by LeongHW, 2013/14)



Conclusions

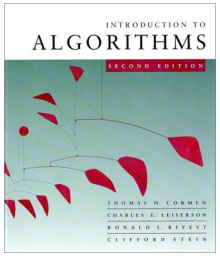
In practice, radix sort is fast for large inputs, as well as simple to code and maintain.

Example (32-bit numbers):

- At most 3 passes when sorting ≥ 2000 numbers.
- Merge sort and quicksort do at least $\lceil \lg 2000 \rceil = 11$ passes.

Downside: Unlike quicksort, radix sort displays little locality of reference, and thus a well-tuned quicksort fares better on modern processors, which feature steep memory hierarchies.

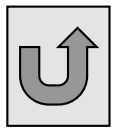
(Modified slightly by LeongHW, 2013/14)



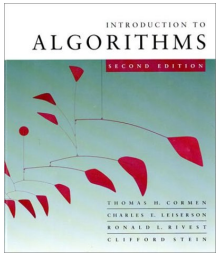
Appendix: Punched-card technology

- Herman Hollerith (1860-1929)
- Punched cards
- Hollerith's tabulating system
- Operation of the sorter
- Origin of radix sort
- "Modern" IBM card
- Web resources on punched-card technology

Return to last slide viewed.



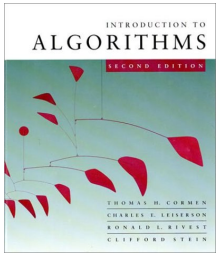
(Modified slightly by LeongHW, 2013/14)



Herman Hollerith (1860-1929)

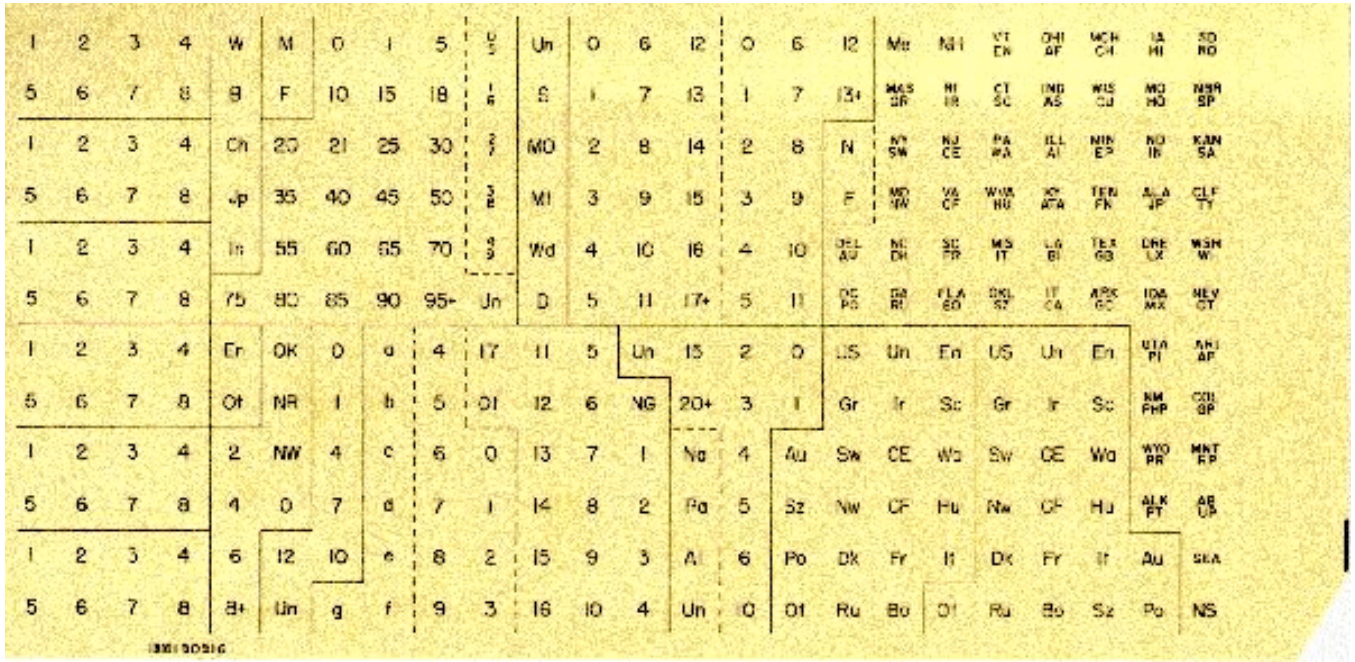
- The 1880 U.S. Census took almost 10 years to process.
- While a lecturer at MIT, Hollerith prototyped punched-card technology.
- His machines, including a "card sorter," allowed the 1890 census total to be reported in 6 weeks.
- He founded the Tabulating Machine Company in 1911, which merged with other companies in 1924 to form International Business Machines.

(Modified slightly by LeongHW, 2013/14)



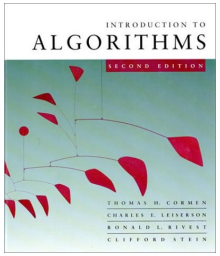
Punched cards

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



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

(Modified slightly by LeongHW, 2013/14)



Hollerith's tabulating system

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

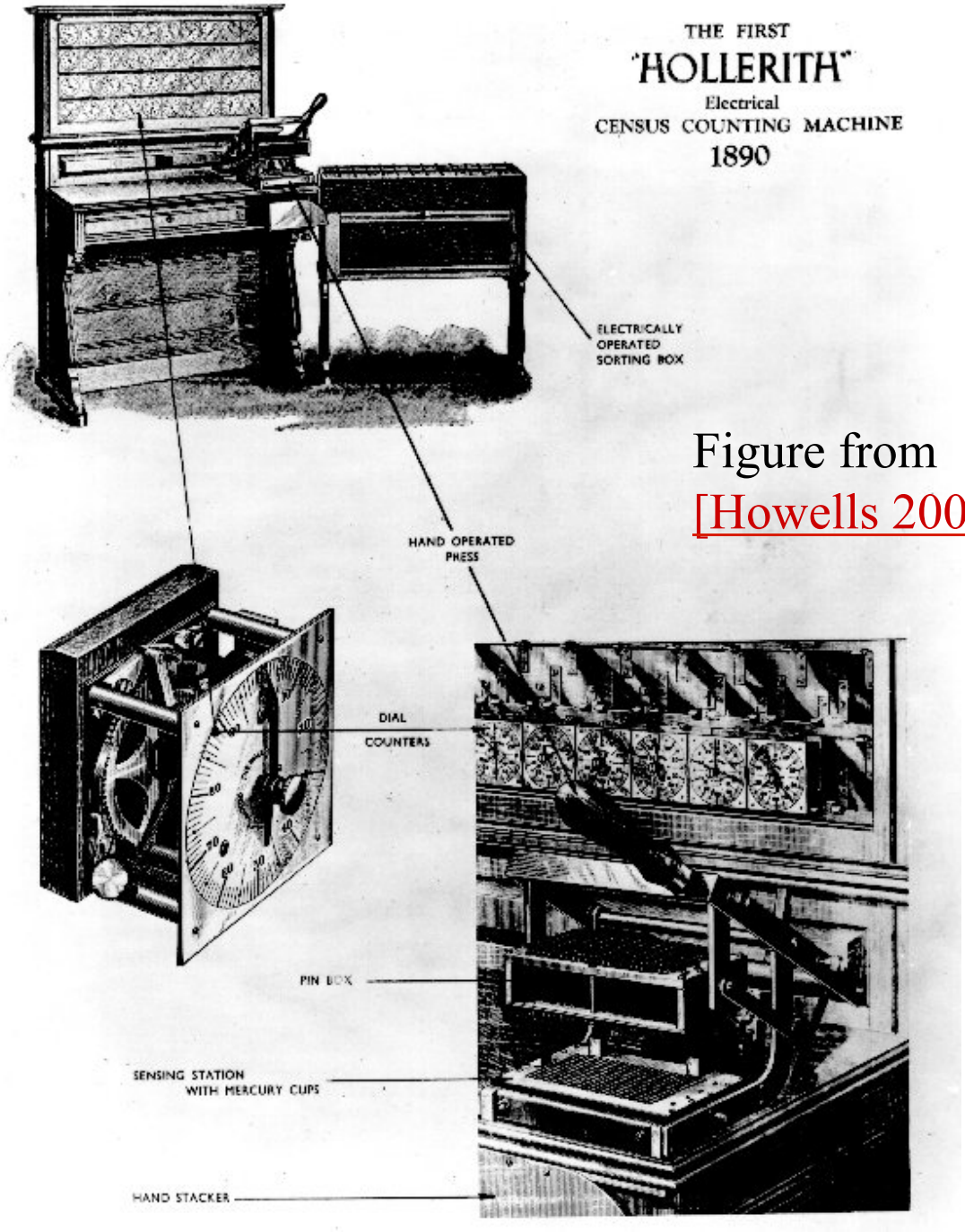
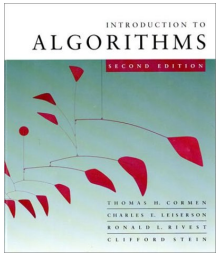


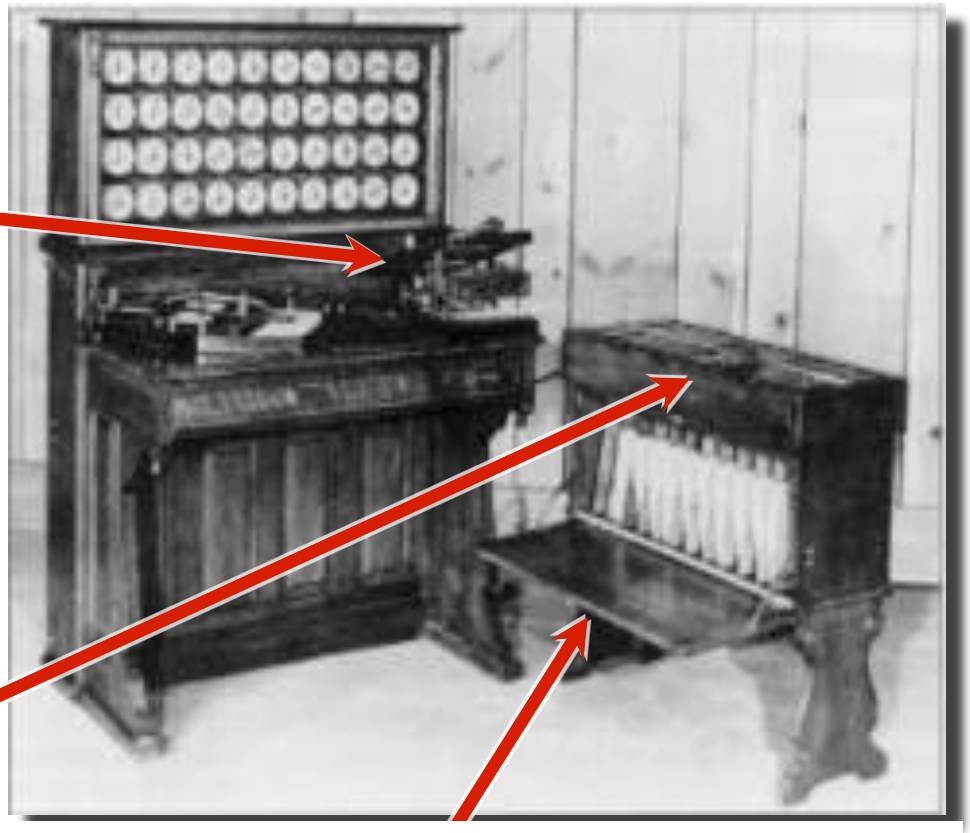
Figure from [Howells 2000].

(Modified slightly by LeongHW, 2013/14)



Operation of the sorter

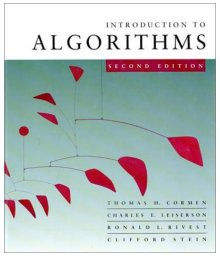
- An operator inserts a card into the press.
- Pins on the press reach through the punched holes to make electrical contact with mercury-filled cups beneath the card.
- Whenever a particular digit value is punched, the lid of the corresponding sorting bin lifts.
- The operator deposits the card into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



Hollerith Tabulator, Pantograph, Press, and Sorter

- 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.

(Modified slightly by LeongHW, 2013/14)



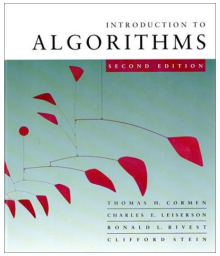
Origin of radix sort

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.

(Modified slightly by LeongHW, 2013/14)



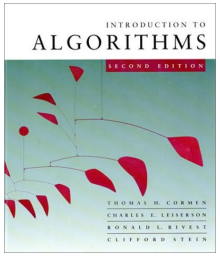
"Modern" IBM card

- One character per column.

Produced by the WWW Virtual Punch-Card Server.

So, that's why text windows have 80 columns!

(Modified slightly by LeongHW, 2013/14)



Web resources on punched-card technology

- Doug Jones's punched card index
- Biography of Herman Hollerith
- The 1890 U.S. Census
- Early history of IBM
- Pictures of Hollerith's inventions
- Hollerith's patent application (borrowed from Gordon Bell's CyberMuseum)
- Impact of punched cards on U.S. history

(Modified slightly by LeongHW, 2013/14)