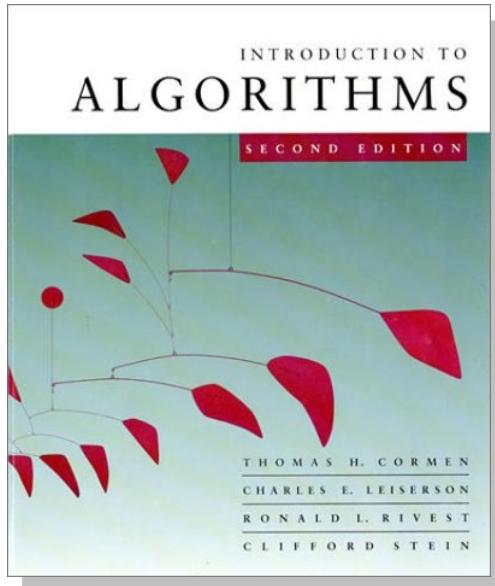


# *Introduction to Algorithms*

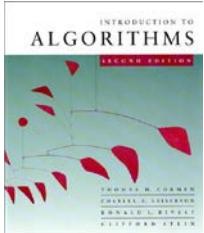
**6.046J/18.401J**



## **LECTURE 1**

### **Analysis of Algorithms**

- Insertion sort
- Asymptotic analysis
- Merge sort
- Recurrences

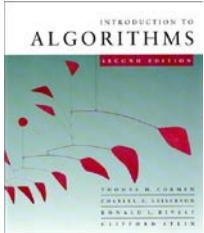


# Analysis of algorithms

*The theoretical study of computer-program performance and resource usage.*

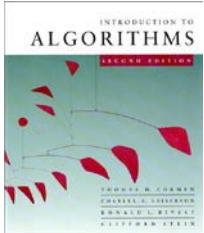
What's more important than performance?

- modularity
- correctness
- maintainability
- functionality
- robustness
- user-friendliness
- programmer time
- simplicity
- extensibility
- reliability



# Why study algorithms and performance?

- Algorithms help us to understand *scalability*.
- Performance often draws the line between what is feasible and what is impossible.
- Algorithmic mathematics provides a *language* for talking about program behavior.
- Performance is the *currency* of computing.
- The lessons of program performance generalize to other computing resources.
- Speed is fun!



# The problem of sorting

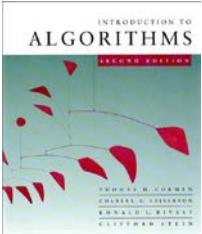
***Input:*** sequence  $\langle a_1, a_2, \dots, a_n \rangle$  of numbers.

***Output:*** permutation  $\langle a'_1, a'_2, \dots, a'_n \rangle$  such that  $a'_1 \leq a'_2 \leq \dots \leq a'_n$ .

**Example:**

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

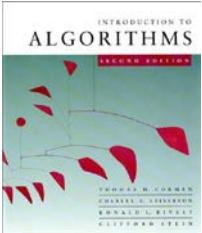
***Output:*** 2 3 4 6 8 9



# Insertion sort

“pseudocode”

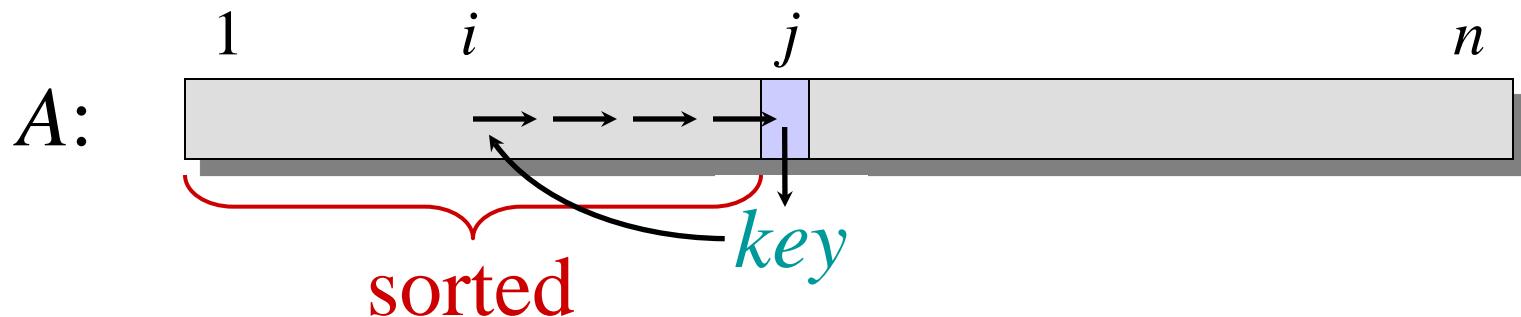
```
INSERTION-SORT ( $A, n$ )      ▷  $A[1 \dots n]$ 
  for  $j \leftarrow 2$  to  $n$ 
    do  $key \leftarrow A[j]$ 
         $i \leftarrow j - 1$ 
        while  $i > 0$  and  $A[i] > key$ 
          do  $A[i+1] \leftarrow A[i]$ 
               $i \leftarrow i - 1$ 
     $A[i+1] = key$ 
```

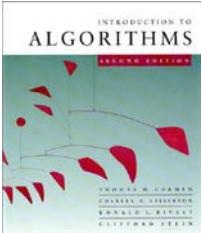


# Insertion sort

“pseudocode”

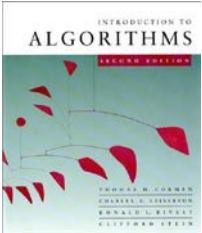
```
INSERTION-SORT ( $A, n$ )      ▷  $A[1 \dots n]$ 
  for  $j \leftarrow 2$  to  $n$ 
    do  $key \leftarrow A[j]$ 
         $i \leftarrow j - 1$ 
        while  $i > 0$  and  $A[i] > key$ 
          do  $A[i+1] \leftarrow A[i]$ 
               $i \leftarrow i - 1$ 
     $A[i+1] = key$ 
```





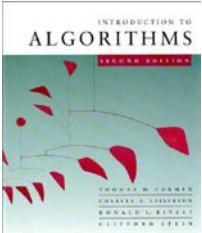
# Example of insertion sort

8      2      4      9      3      6



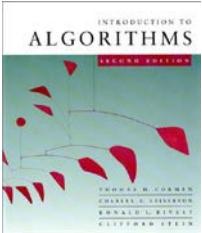
# Example of insertion sort





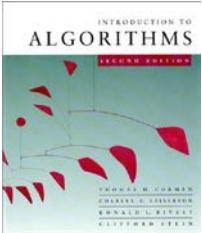
# Example of insertion sort



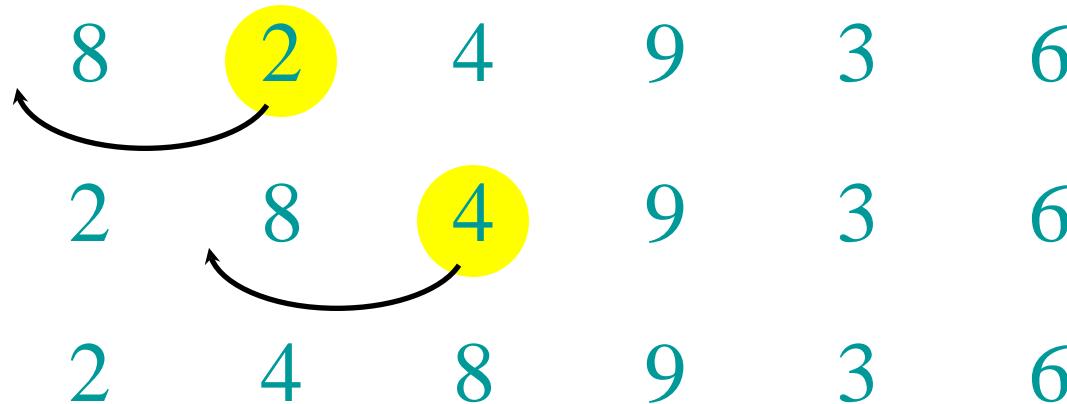


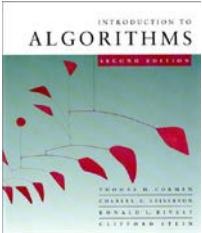
# Example of insertion sort



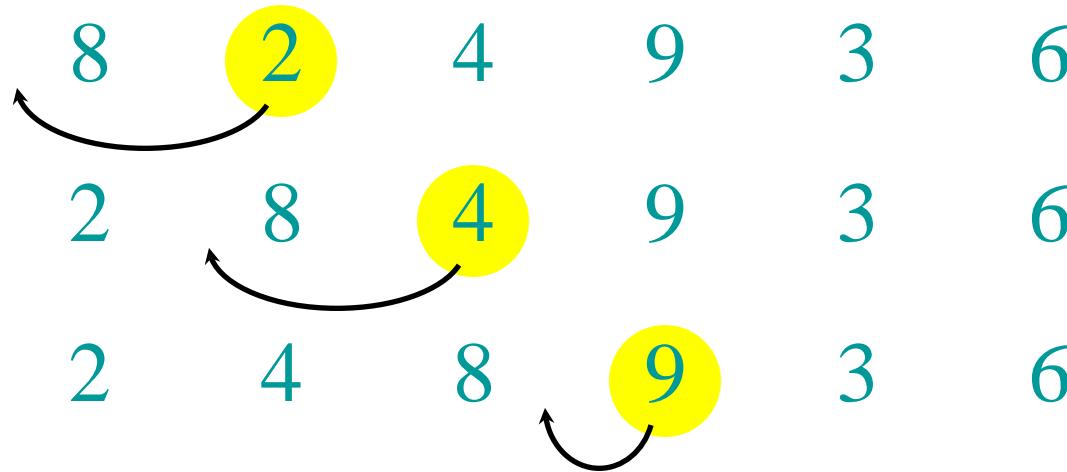


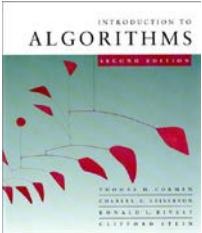
# Example of insertion sort



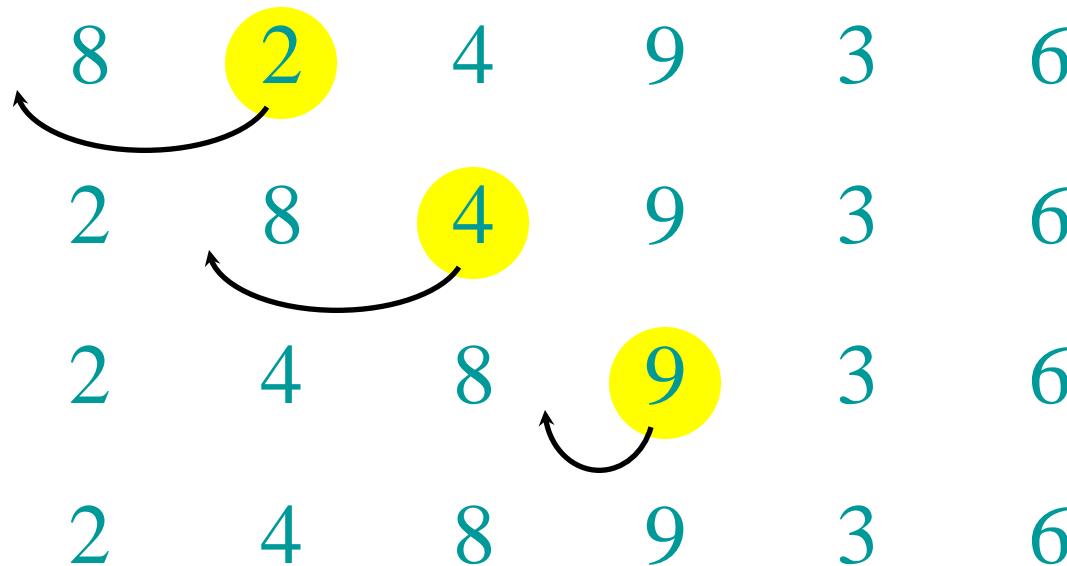


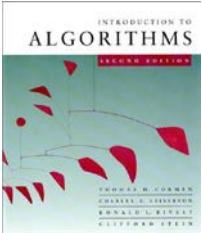
# Example of insertion sort



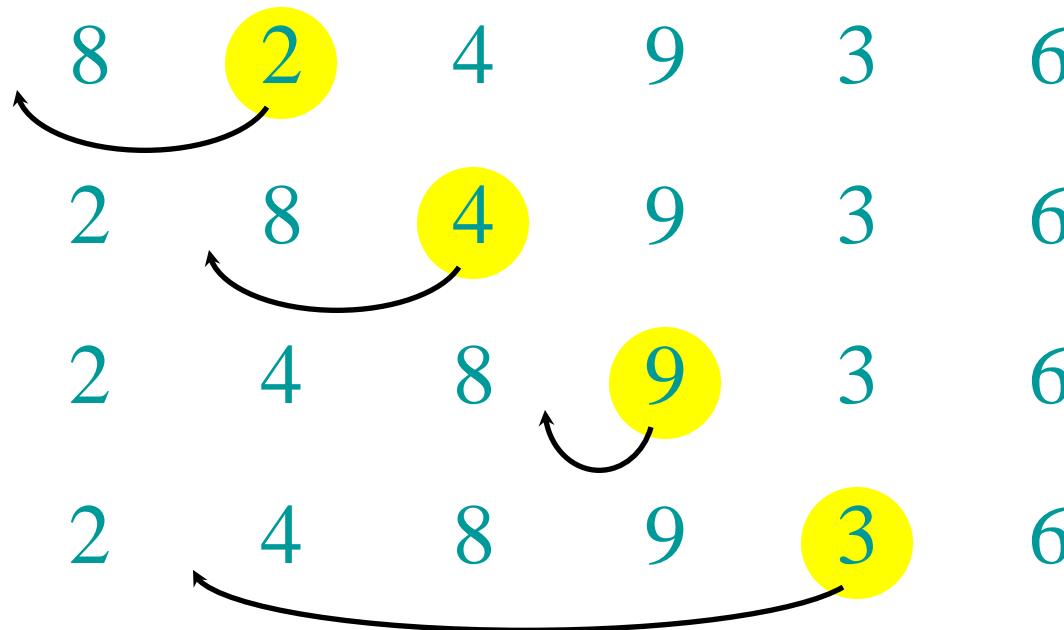


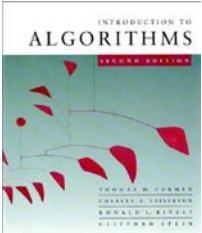
# Example of insertion sort



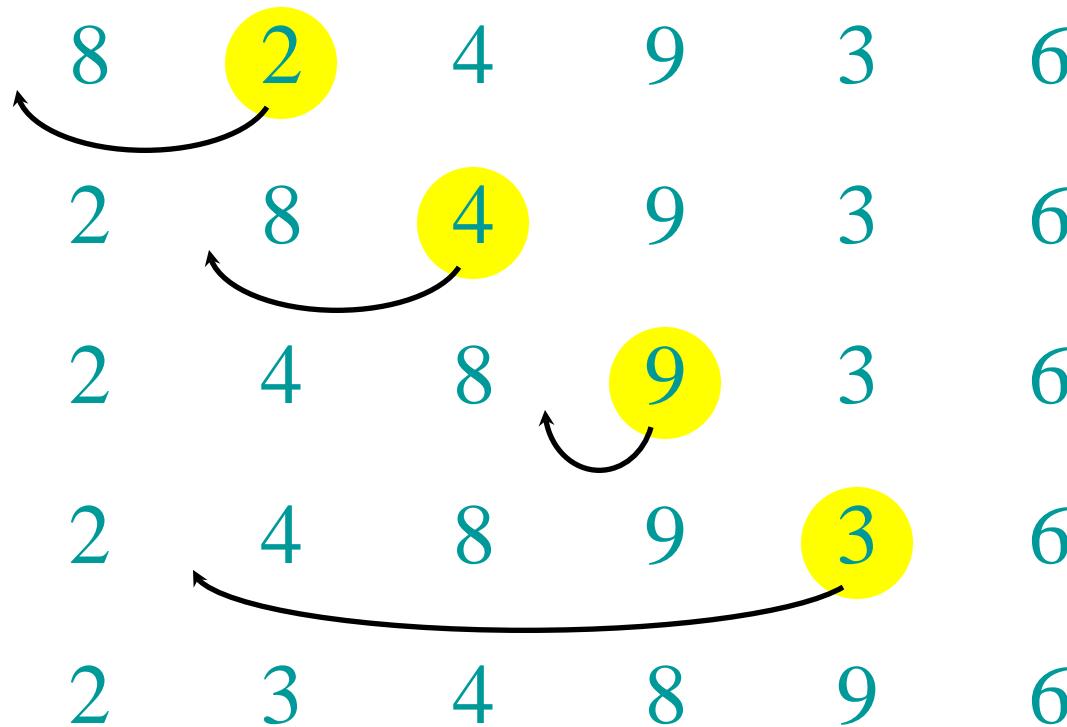


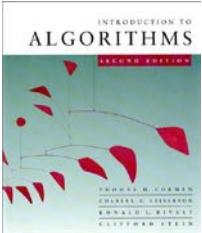
# Example of insertion sort



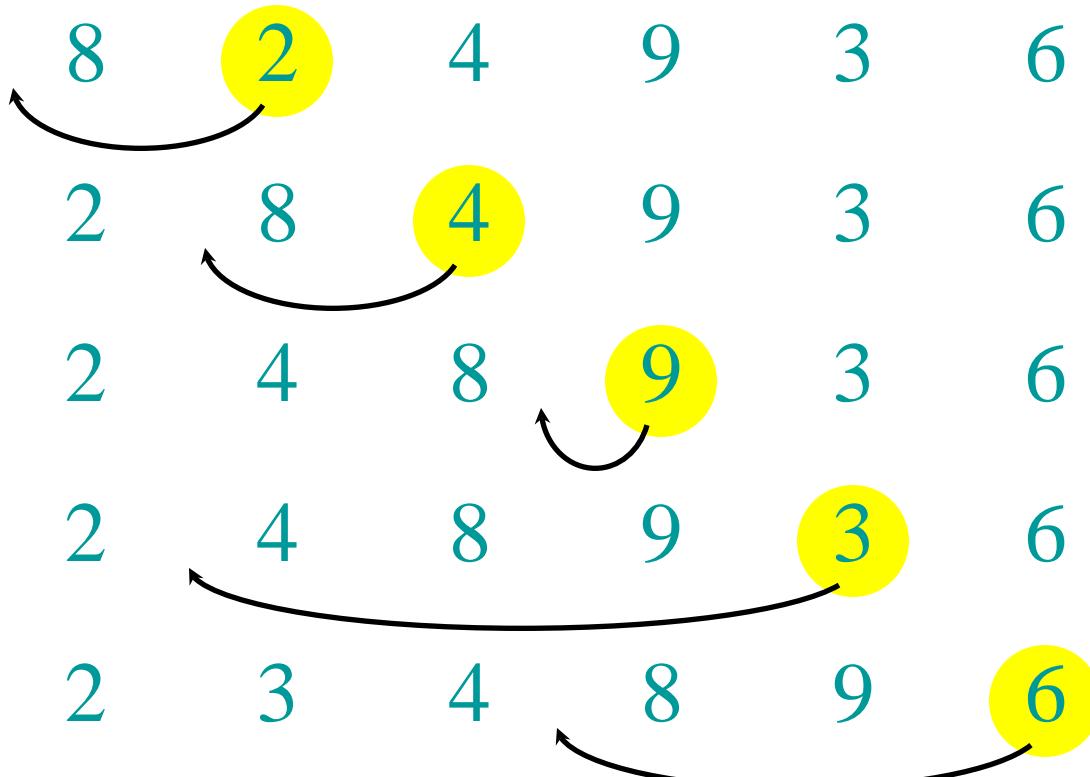


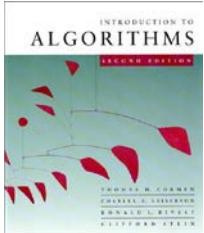
# Example of insertion sort



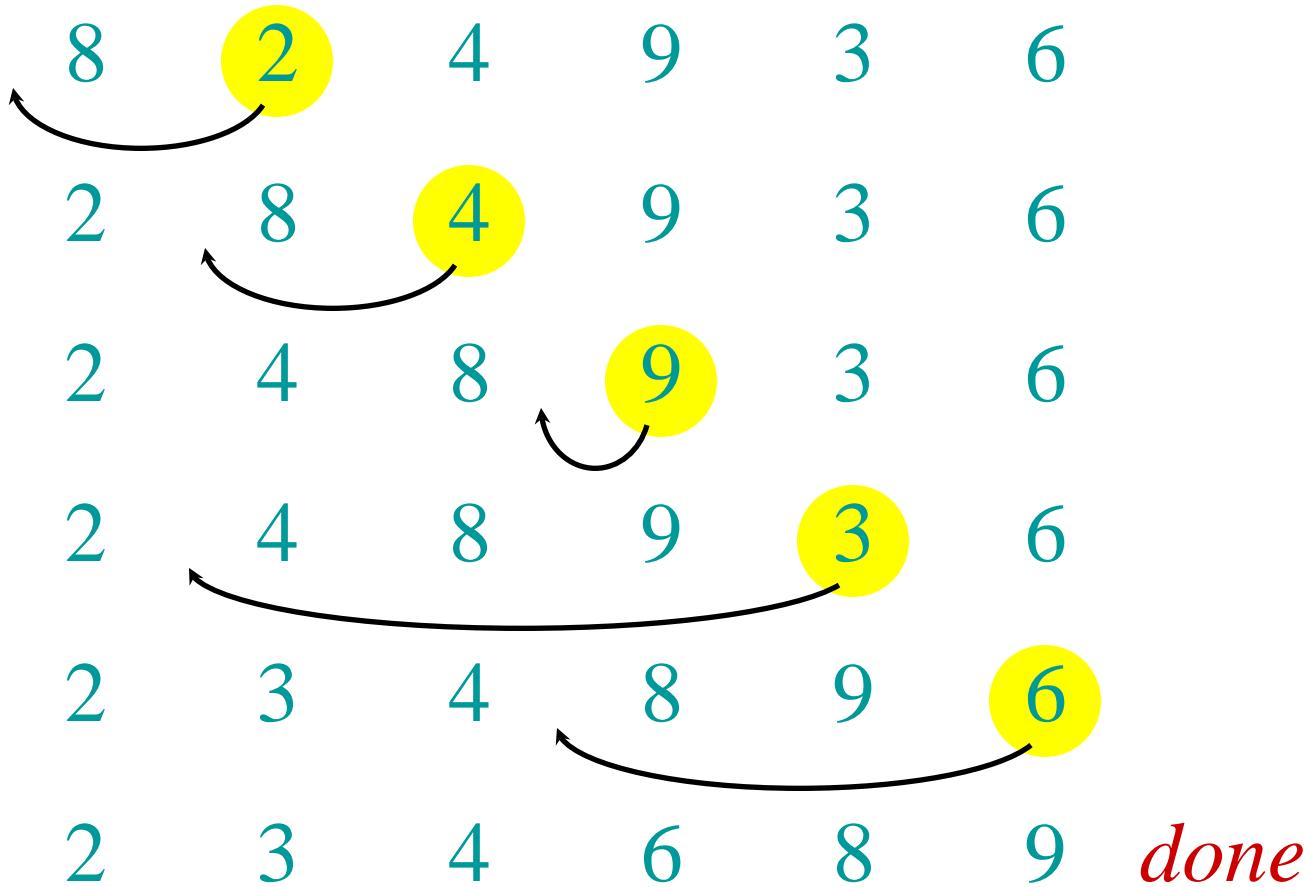


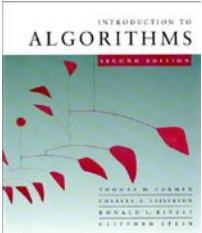
# Example of insertion sort





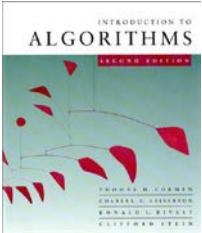
# Example of insertion sort





# Running time

- The running time depends on the input: an already sorted sequence is easier to sort.
- Parameterize the running time by the size of the input, since short sequences are easier to sort than long ones.
- Generally, we seek upper bounds on the running time, because everybody likes a guarantee.



# Kinds of analyses

**Worst-case:** (usually)

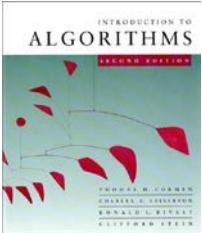
- $T(n)$  = maximum time of algorithm on any input of size  $n$ .

**Average-case:** (sometimes)

- $T(n)$  = expected time of algorithm over all inputs of size  $n$ .
- Need assumption of statistical distribution of inputs.

**Best-case:** (bogus)

- Cheat with a slow algorithm that works fast on *some* input.



# Machine-independent time

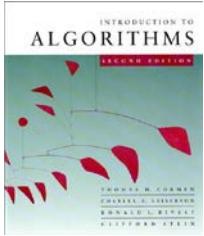
*What is insertion sort's worst-case time?*

- It depends on the speed of our computer:
  - relative speed (on the same machine),
  - absolute speed (on different machines).

**BIG IDEA:**

- Ignore machine-dependent constants.
- Look at *growth* of  $T(n)$  as  $n \rightarrow \infty$ .

**“Asymptotic Analysis”**



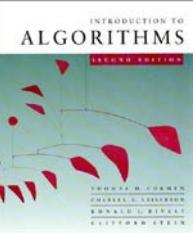
# $\Theta$ -notation

## **Math:**

$\Theta(g(n)) = \{ f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0 \}$

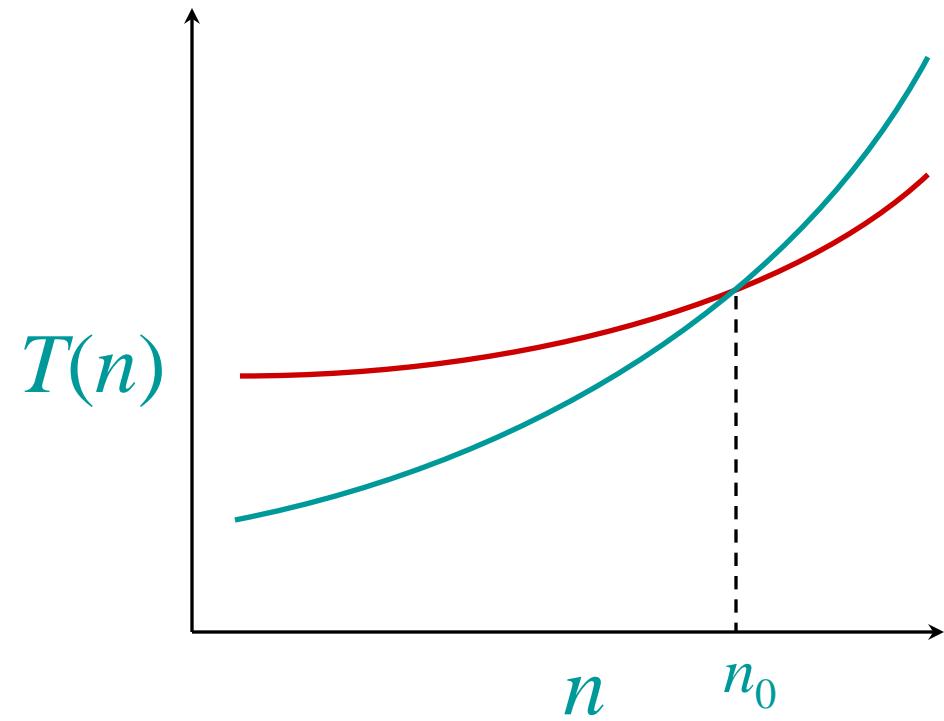
## **Engineering:**

- Drop low-order terms; ignore leading constants.
- Example:  $3n^3 + 90n^2 - 5n + 6046 = \Theta(n^3)$

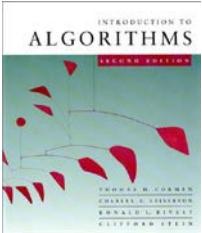


# Asymptotic performance

When  $n$  gets large enough, a  $\Theta(n^2)$  algorithm *always* beats a  $\Theta(n^3)$  algorithm.



- We shouldn't ignore asymptotically slower algorithms, however.
- Real-world design situations often call for a careful balancing of engineering objectives.
- Asymptotic analysis is a useful tool to help to structure our thinking.



# Insertion sort analysis

**Worst case:** Input reverse sorted.

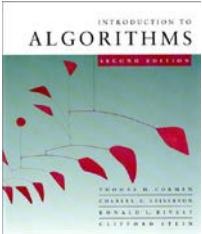
$$T(n) = \sum_{j=2}^n \Theta(j) = \Theta(n^2) \quad [\text{arithmetic series}]$$

**Average case:** All permutations equally likely.

$$T(n) = \sum_{j=2}^n \Theta(j/2) = \Theta(n^2)$$

*Is insertion sort a fast sorting algorithm?*

- Moderately so, for small  $n$ .
- Not at all, for large  $n$ .

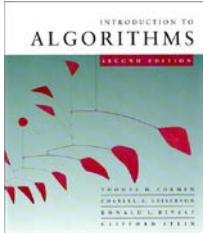


# Merge sort

**MERGE-SORT  $A[1 \dots n]$**

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

***Key subroutine:* MERGE**



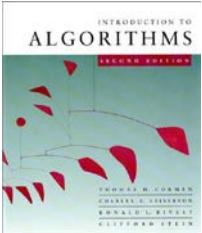
# Merging two sorted arrays

20 12

13 11

7 9

2 1

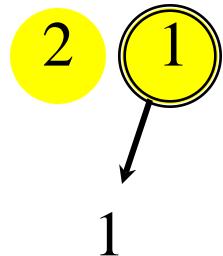


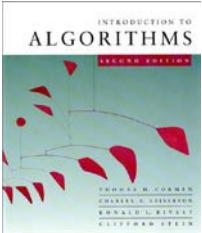
# Merging two sorted arrays

20 12

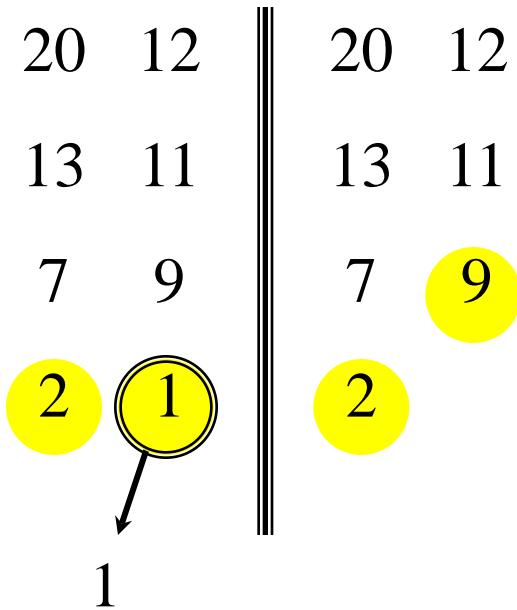
13 11

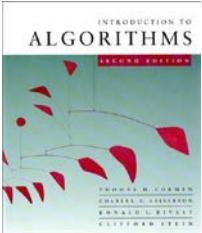
7 9



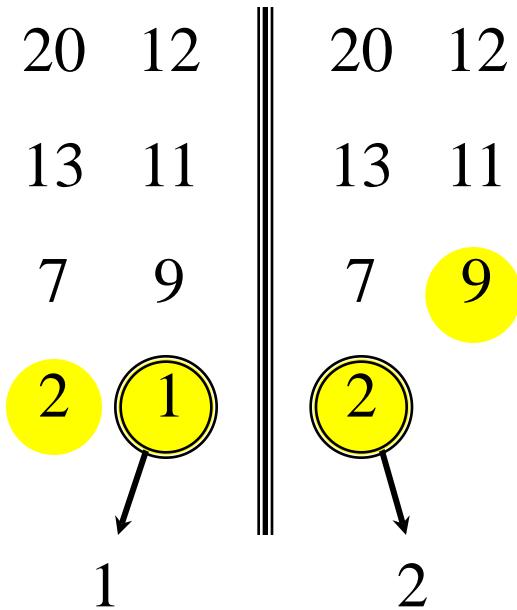


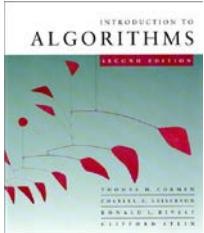
# Merging two sorted arrays



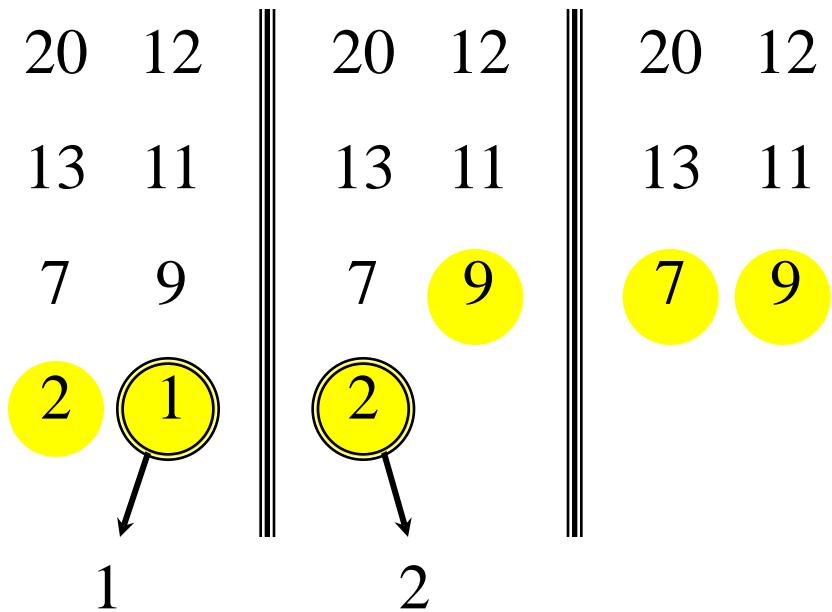


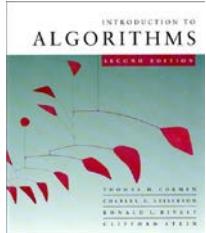
# Merging two sorted arrays



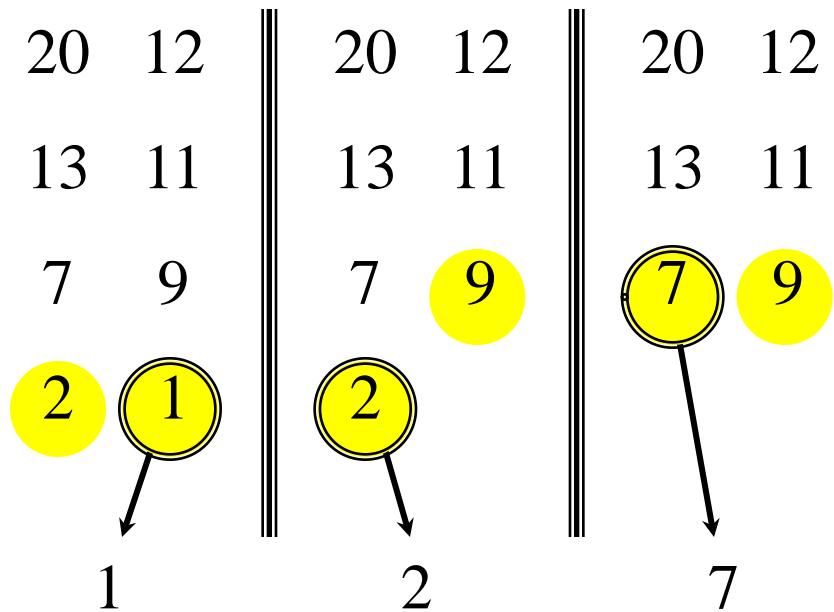


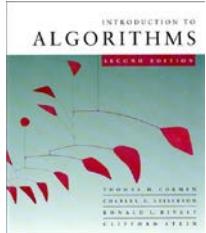
# Merging two sorted arrays



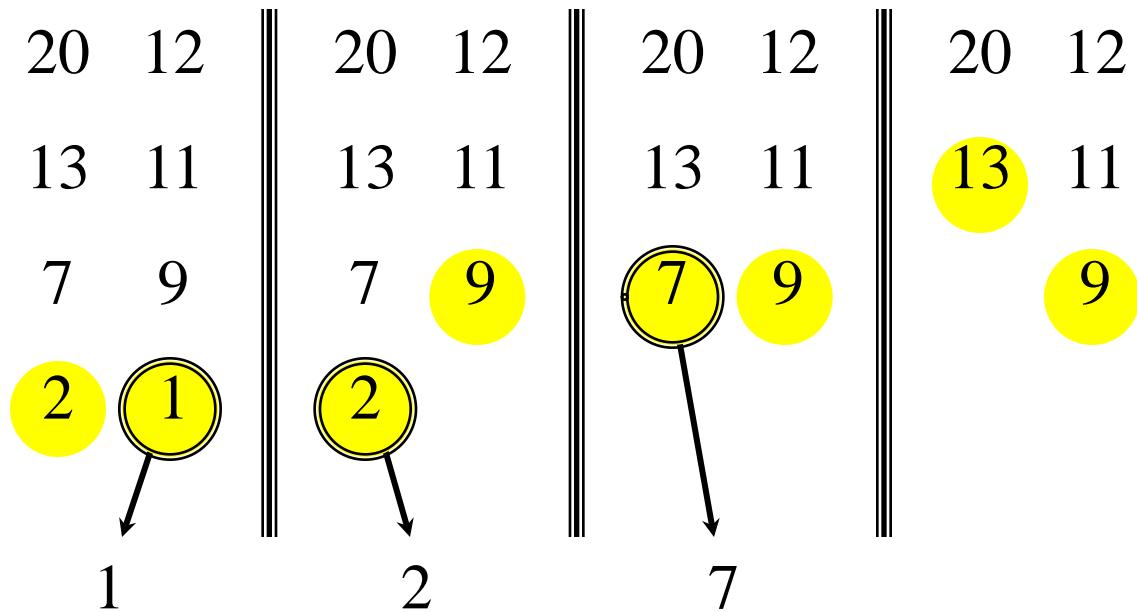


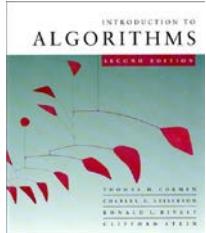
# Merging two sorted arrays



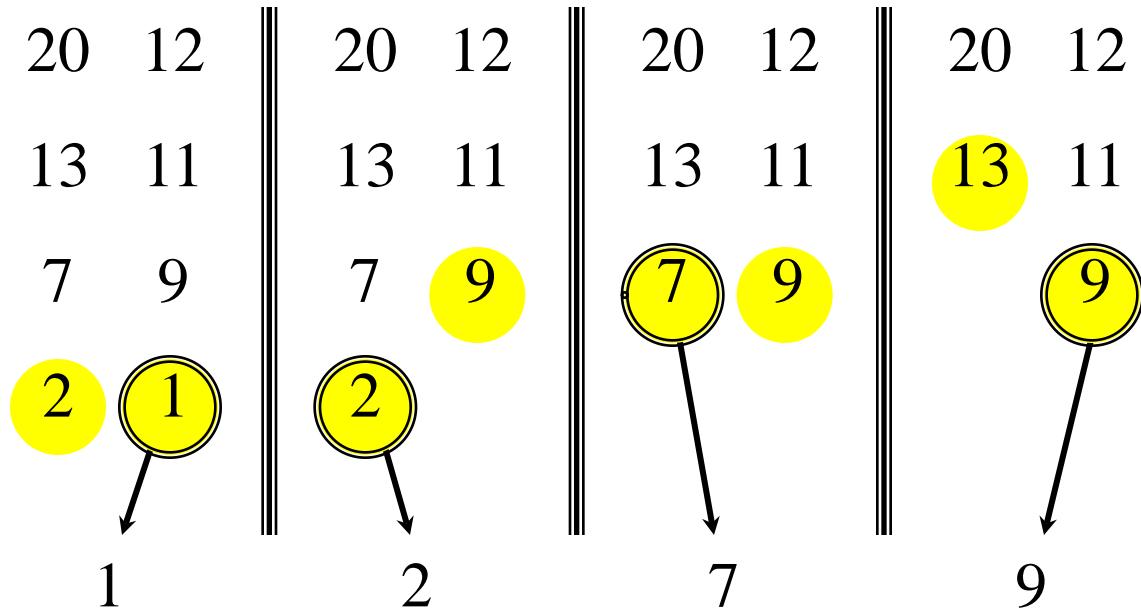


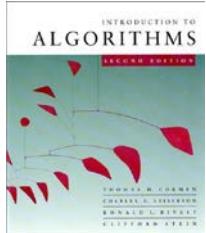
# Merging two sorted arrays



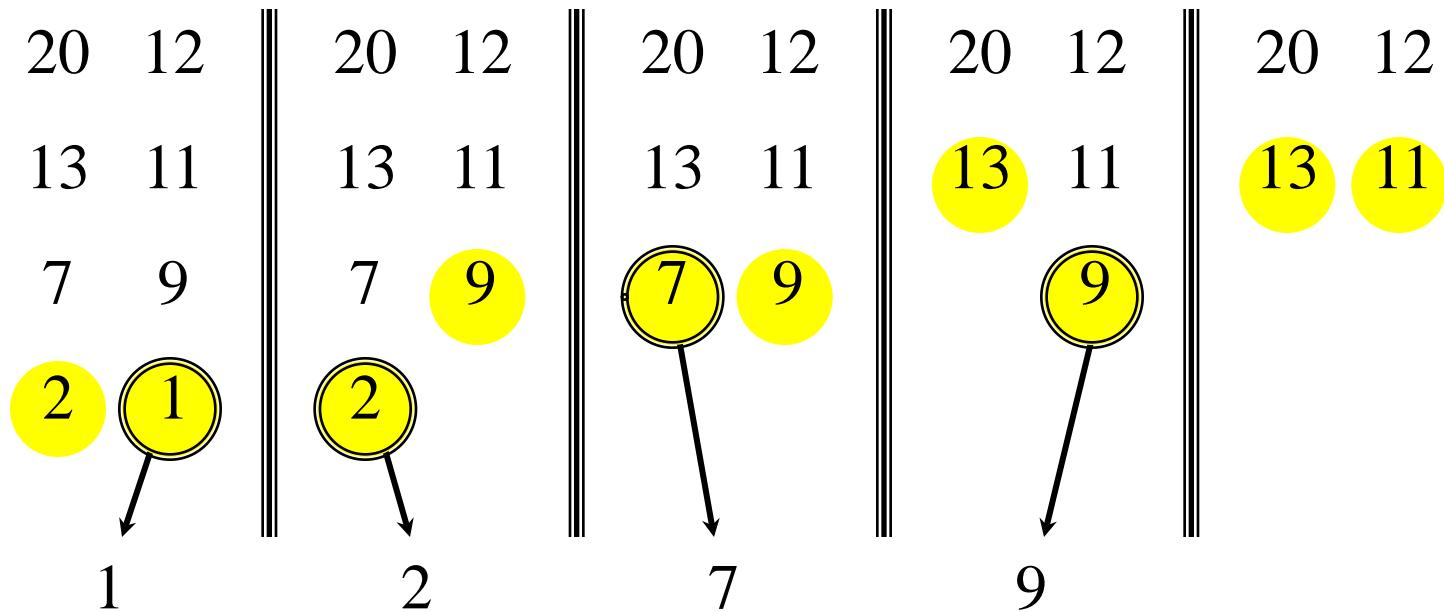


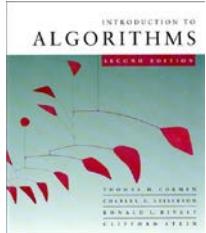
# Merging two sorted arrays



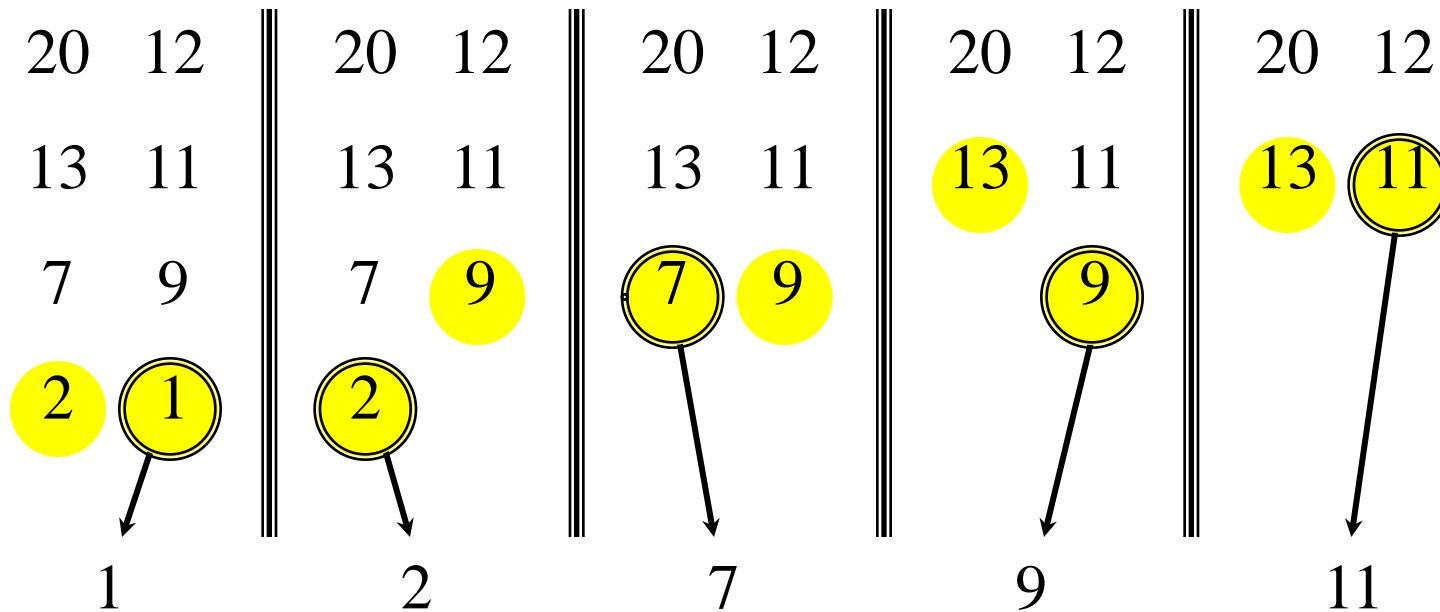


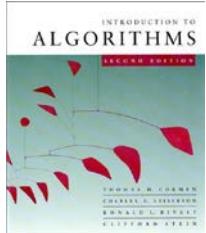
# Merging two sorted arrays



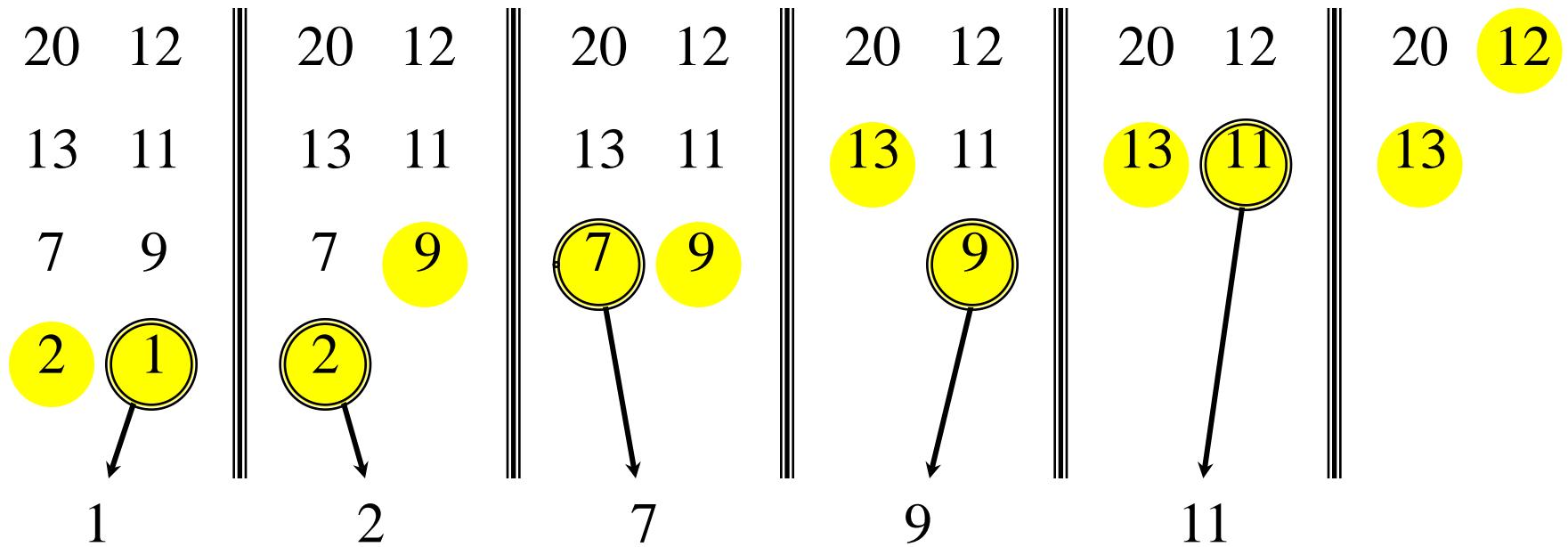


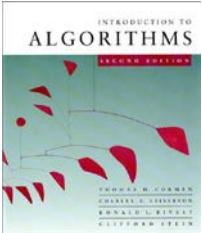
# Merging two sorted arrays



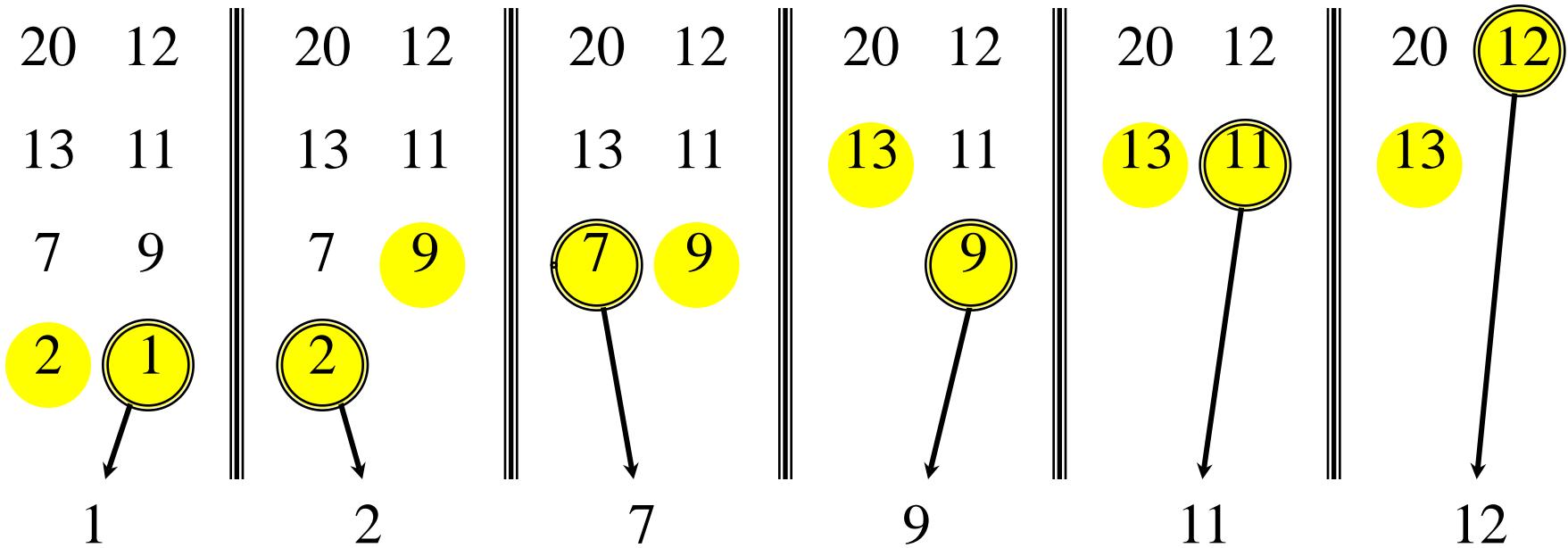


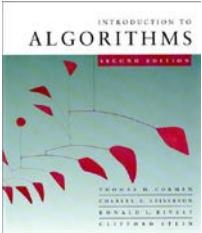
# Merging two sorted arrays



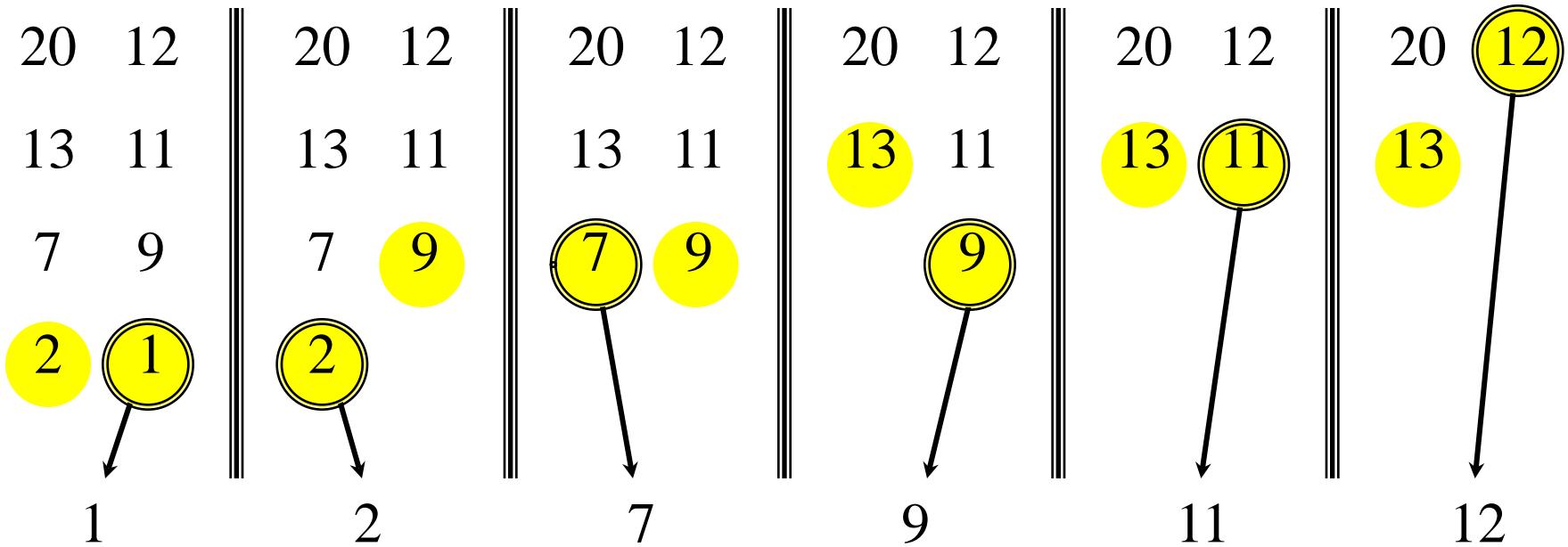


# Merging two sorted arrays

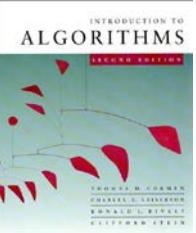




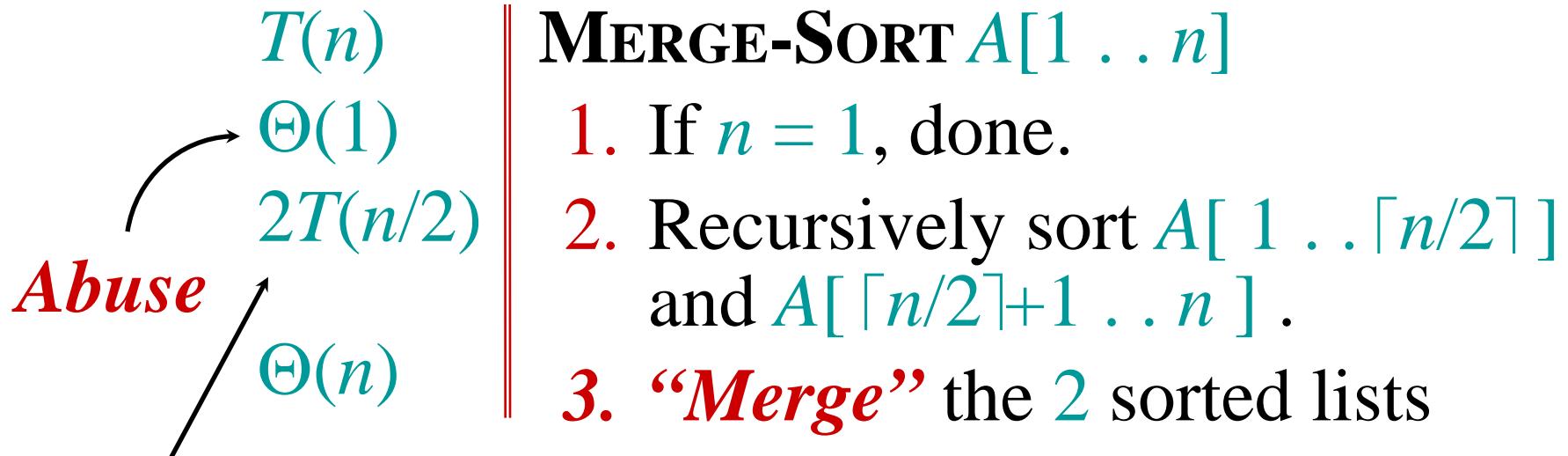
# Merging two sorted arrays



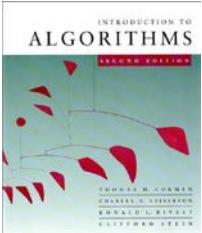
Time =  $\Theta(n)$  to merge a total  
of  $n$  elements (linear time).



# Analyzing merge sort



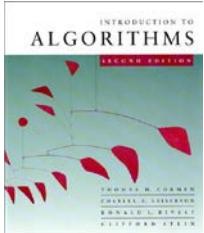
**Sloppiness:** Should be  $T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor)$ , but it turns out not to matter asymptotically.



# Recurrence for merge sort

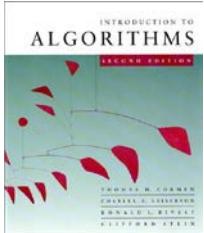
$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1; \\ 2T(n/2) + \Theta(n) & \text{if } n > 1. \end{cases}$$

- We shall usually omit stating the base case when  $T(n) = \Theta(1)$  for sufficiently small  $n$ , but only when it has no effect on the asymptotic solution to the recurrence.
- CLRS and Lecture 2 provide several ways to find a good upper bound on  $T(n)$ .



# Recursion tree

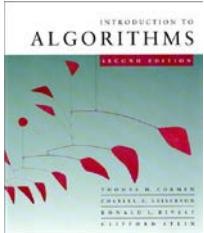
Solve  $T(n) = 2T(n/2) + cn$ , where  $c > 0$  is constant.



# Recursion tree

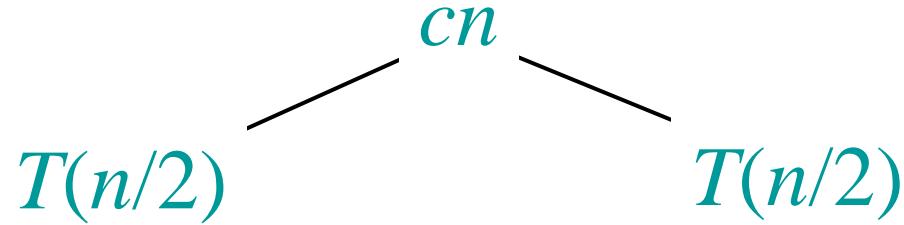
Solve  $T(n) = 2T(n/2) + cn$ , where  $c > 0$  is constant.

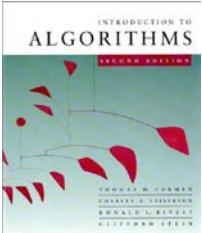
$$T(n)$$



# Recursion tree

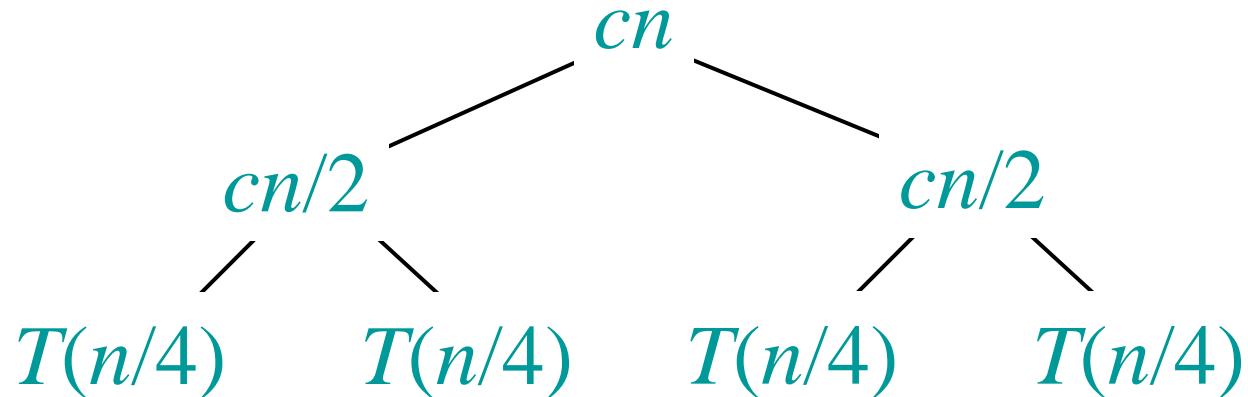
Solve  $T(n) = 2T(n/2) + cn$ , where  $c > 0$  is constant.

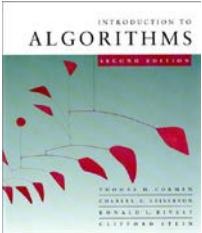




# Recursion tree

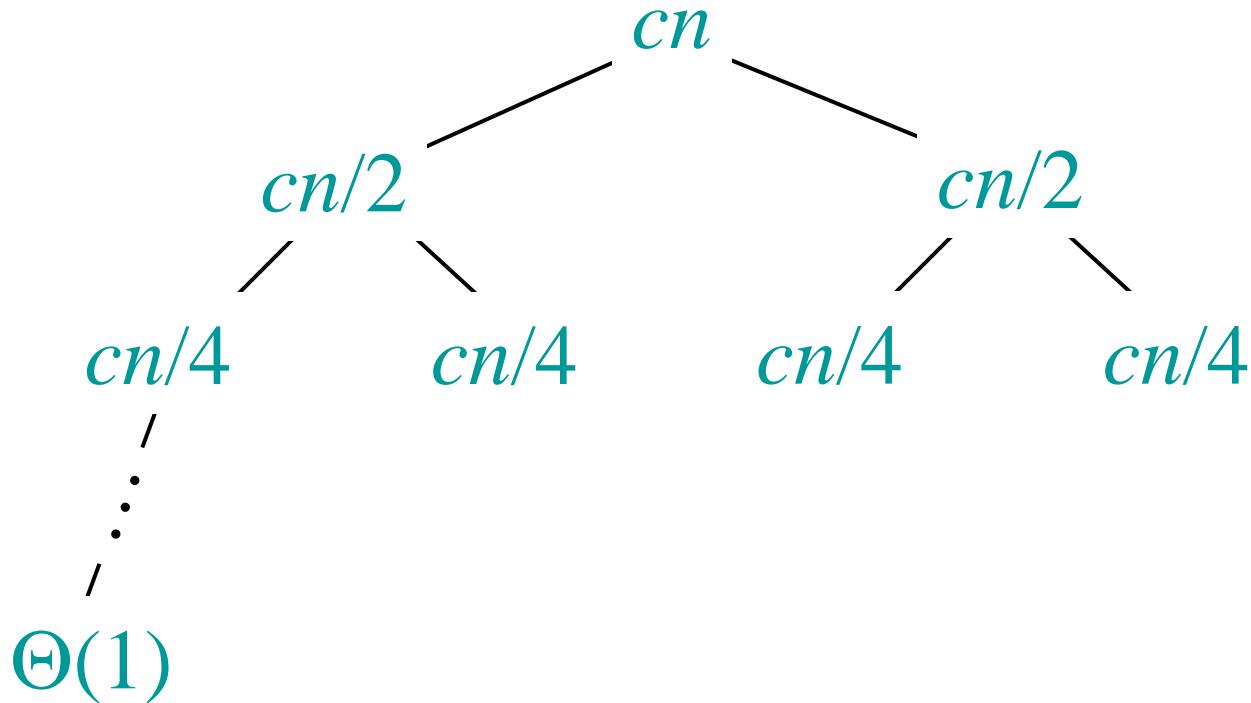
Solve  $T(n) = 2T(n/2) + cn$ , where  $c > 0$  is constant.

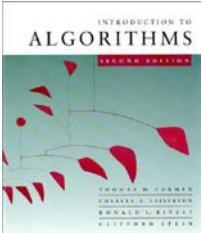




# Recursion tree

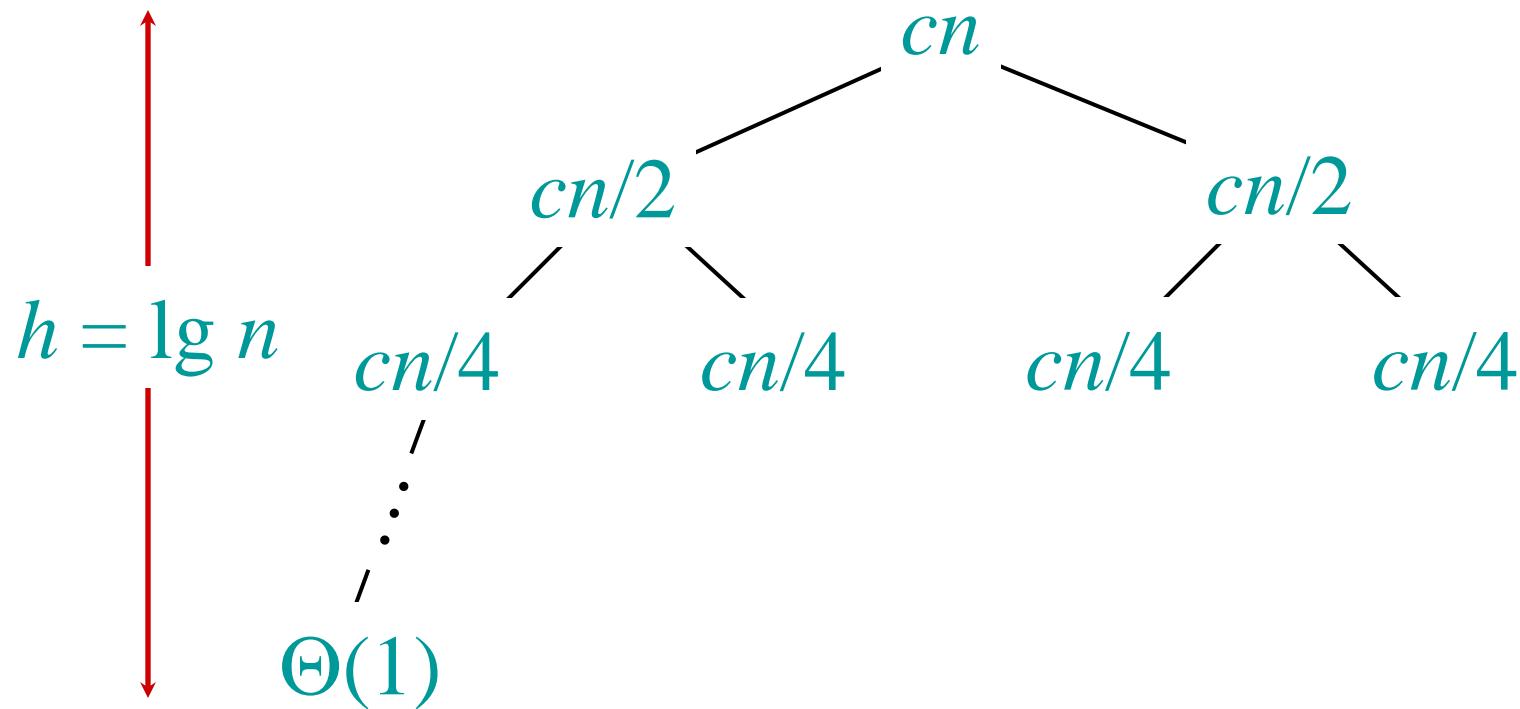
Solve  $T(n) = 2T(n/2) + cn$ , where  $c > 0$  is constant.

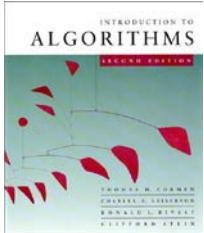




# Recursion tree

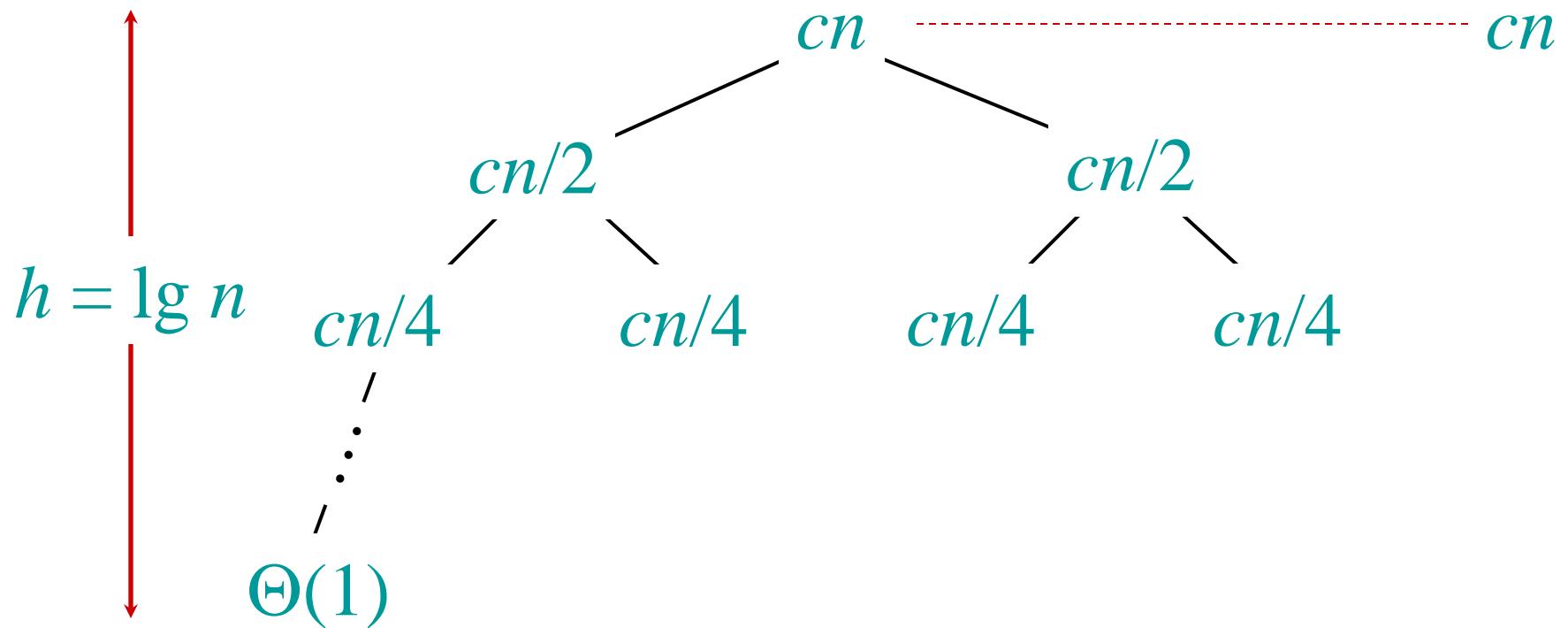
Solve  $T(n) = 2T(n/2) + cn$ , where  $c > 0$  is constant.

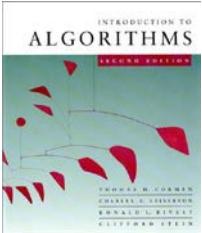




# Recursion tree

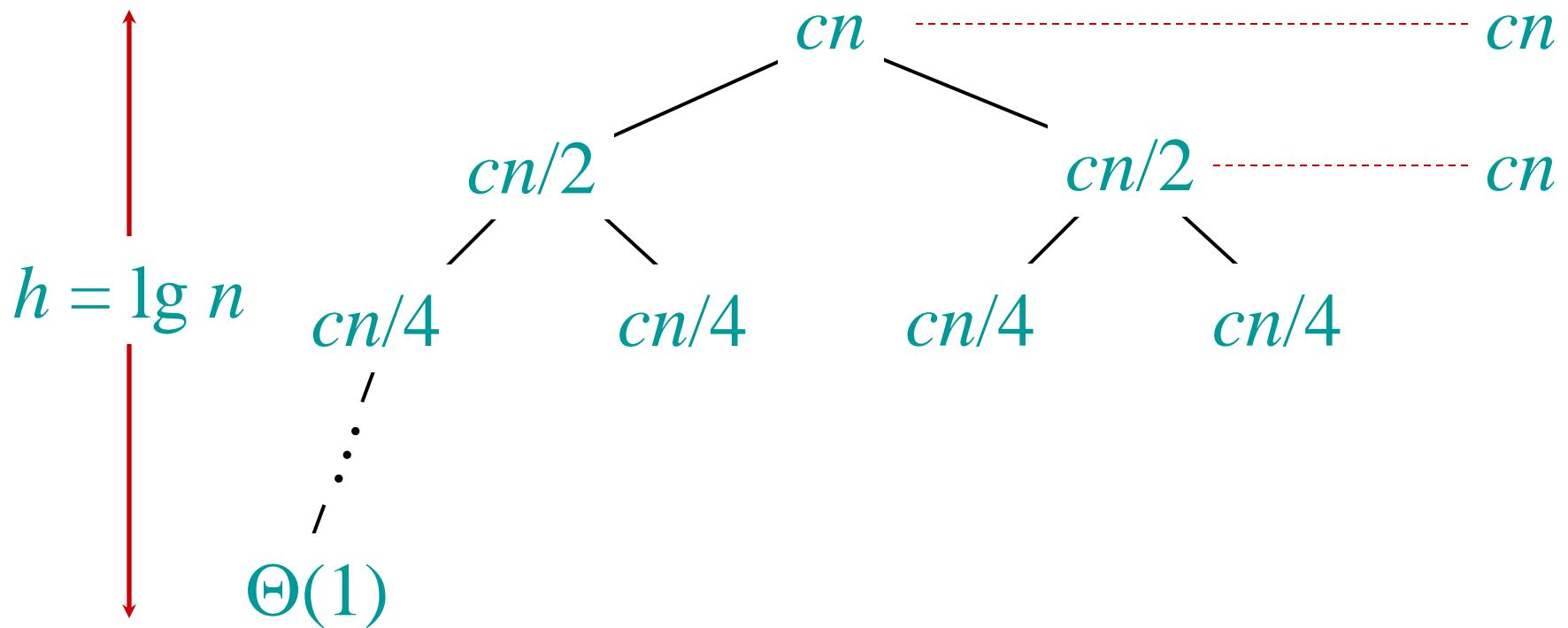
Solve  $T(n) = 2T(n/2) + cn$ , where  $c > 0$  is constant.

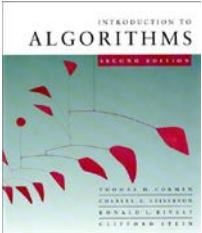




# Recursion tree

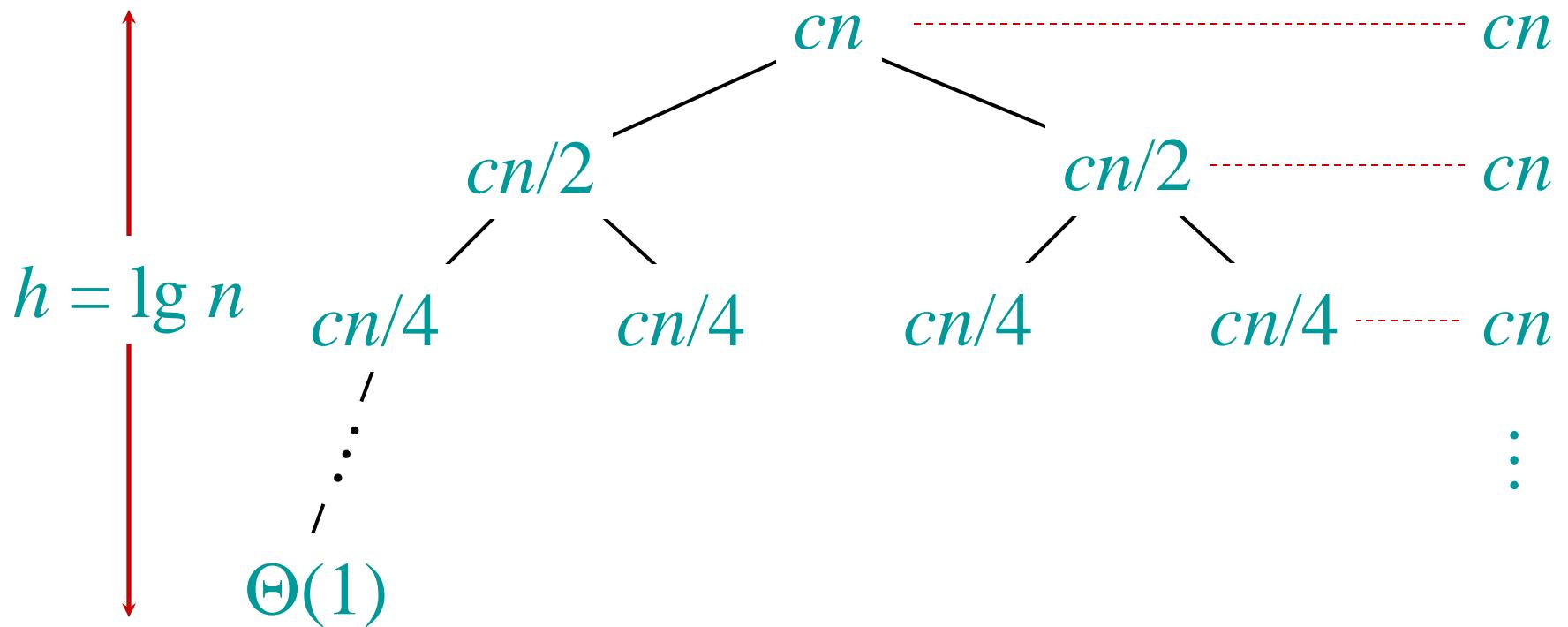
Solve  $T(n) = 2T(n/2) + cn$ , where  $c > 0$  is constant.

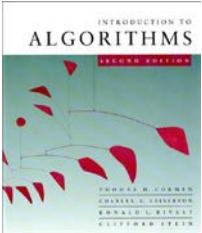




# Recursion tree

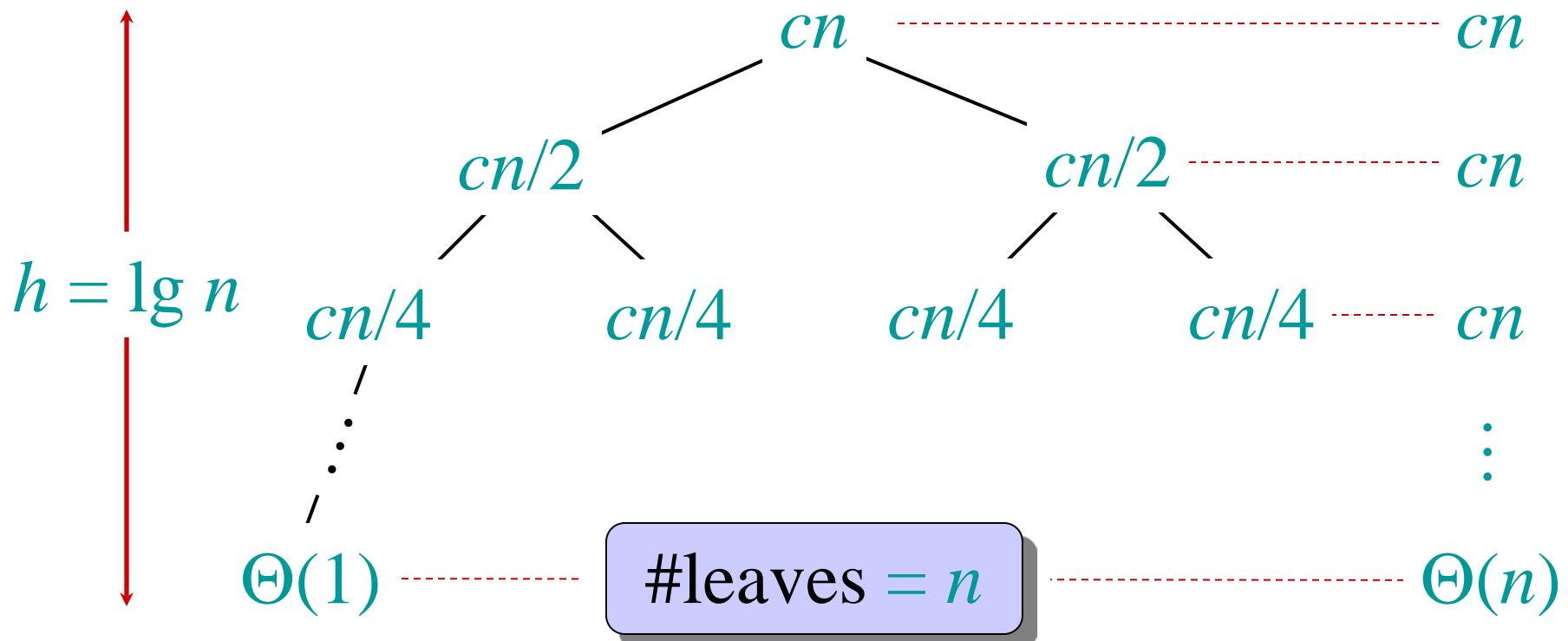
Solve  $T(n) = 2T(n/2) + cn$ , where  $c > 0$  is constant.

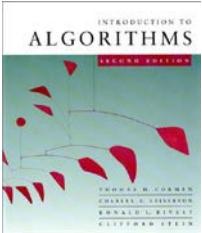




# Recursion tree

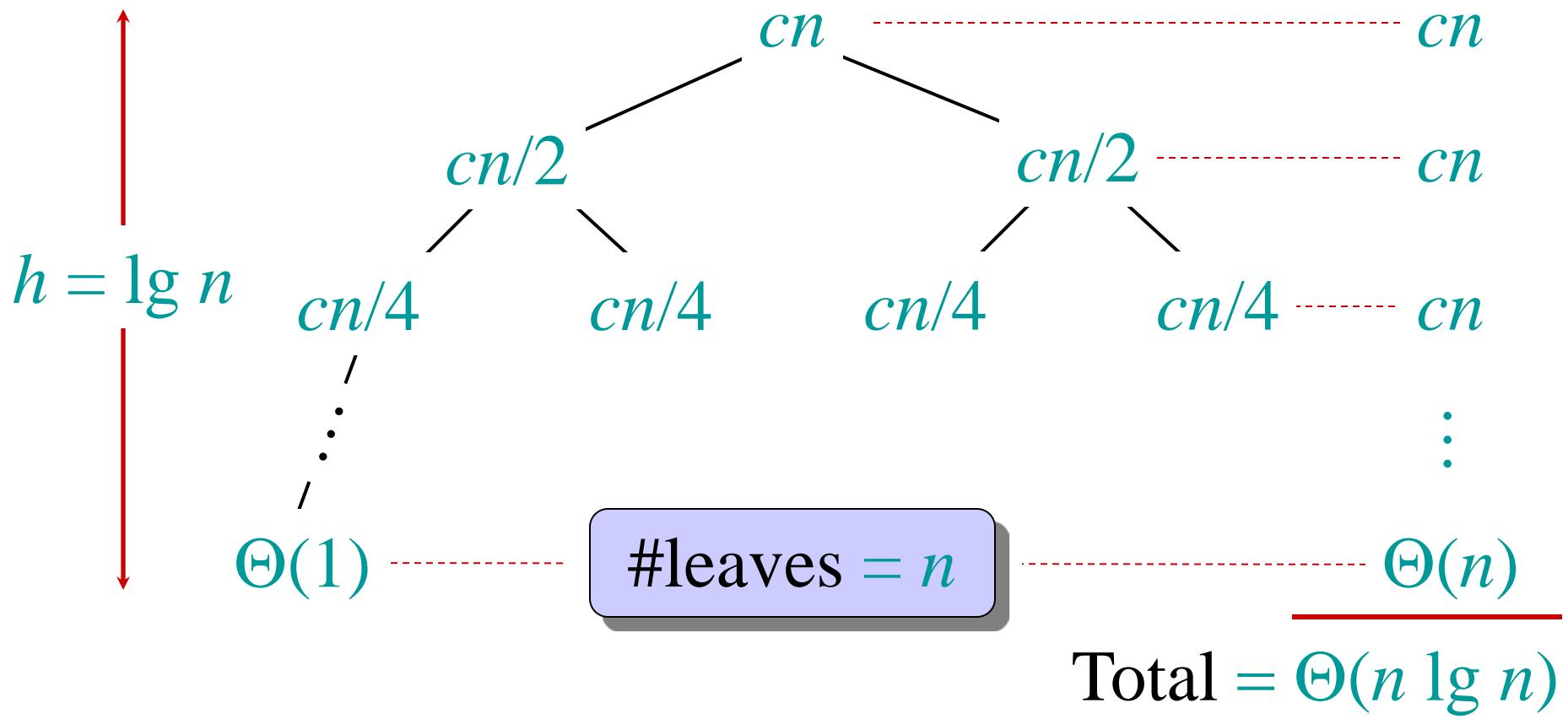
Solve  $T(n) = 2T(n/2) + cn$ , where  $c > 0$  is constant.

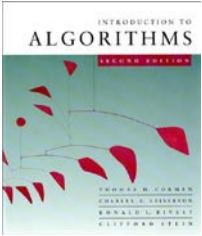




# Recursion tree

Solve  $T(n) = 2T(n/2) + cn$ , where  $c > 0$  is constant.



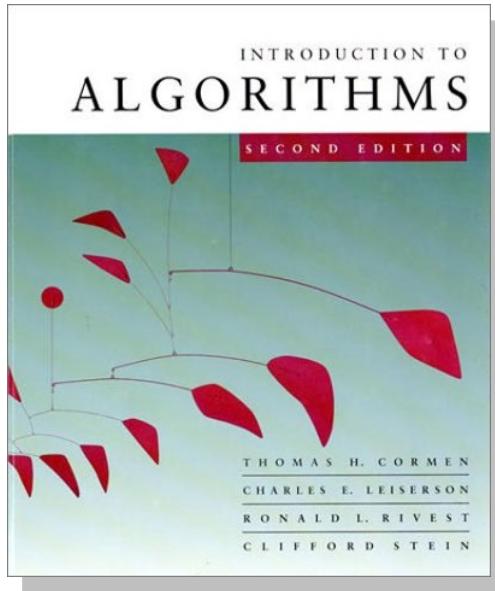


# Conclusions

- $\Theta(n \lg n)$  grows more slowly than  $\Theta(n^2)$ .
- Therefore, merge sort asymptotically beats insertion sort in the worst case.
- In practice, merge sort beats insertion sort for  $n > 30$  or so.
- Go test it out for yourself!

# *Introduction to Algorithms*

**6.046J/18.401J**



## **LECTURE 2**

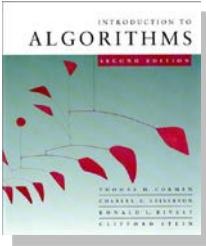
### **Asymptotic Notation**

- $O$ -,  $\Omega$ -, and  $\Theta$ -notation

### **Recurrences**

- Substitution method
- Iterating the recurrence
- Recursion tree
- Master method

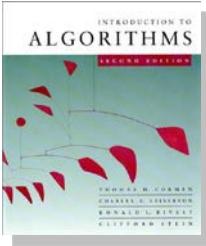
**Prof. Erik Demaine**



# Asymptotic notation

*O*-notation (upper bounds):

We write  $f(n) = O(g(n))$  if there exist constants  $c > 0$ ,  $n_0 > 0$  such that  $0 \leq f(n) \leq cg(n)$  for all  $n \geq n_0$ .

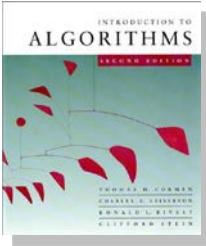


# Asymptotic notation

*O*-notation (upper bounds):

We write  $f(n) = O(g(n))$  if there exist constants  $c > 0$ ,  $n_0 > 0$  such that  $0 \leq f(n) \leq cg(n)$  for all  $n \geq n_0$ .

**EXAMPLE:**  $2n^2 = O(n^3)$     ( $c = 1$ ,  $n_0 = 2$ )



# Asymptotic notation

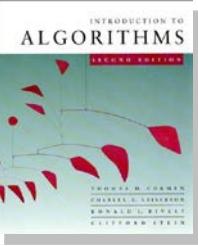
$O$ -notation (upper bounds):

We write  $f(n) = O(g(n))$  if there exist constants  $c > 0$ ,  $n_0 > 0$  such that  $0 \leq f(n) \leq cg(n)$  for all  $n \geq n_0$ .

**EXAMPLE:**  $2n^2 = O(n^3)$     ( $c = 1$ ,  $n_0 = 2$ )

*functions,  
not values*

Two red arrows originate from the italicized text "functions, not values" below. One arrow points upwards towards the first 'n' in the term  $n^3$ . The other arrow points upwards towards the first 'n' in the term  $2n^2$ .

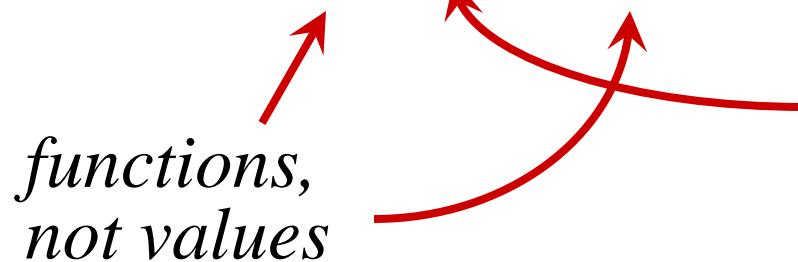


# Asymptotic notation

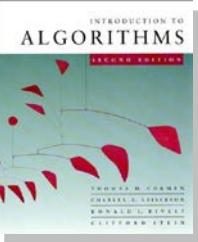
$O$ -notation (upper bounds):

We write  $f(n) = O(g(n))$  if there exist constants  $c > 0$ ,  $n_0 > 0$  such that  $0 \leq f(n) \leq cg(n)$  for all  $n \geq n_0$ .

**EXAMPLE:**  $2n^2 = O(n^3)$  ( $c = 1$ ,  $n_0 = 2$ )

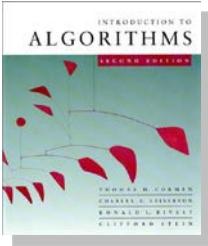


*funny, “one-way”  
equality*



# Set definition of O-notation

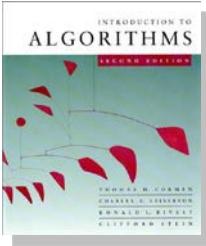
$O(g(n)) = \{ f(n) : \text{there exist constants } c > 0, n_0 > 0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \}$



# Set definition of O-notation

$O(g(n)) = \{ f(n) : \text{there exist constants } c > 0, n_0 > 0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \}$

**EXAMPLE:**  $2n^2 \in O(n^3)$

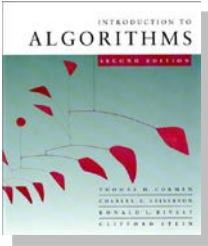


# Set definition of O-notation

$O(g(n)) = \{ f(n) : \text{there exist constants } c > 0, n_0 > 0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \}$

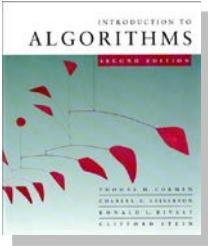
**EXAMPLE:**  $2n^2 \in O(n^3)$

(*Logicians:*  $\lambda n. 2n^2 \in O(\lambda n. n^3)$ , but it's convenient to be sloppy, as long as we understand what's *really* going on.)



# Macro substitution

***Convention:*** A set in a formula represents an anonymous function in the set.



# Macro substitution

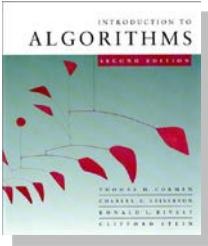
**Convention:** A set in a formula represents an anonymous function in the set.

**EXAMPLE:**  $f(n) = n^3 + O(n^2)$

means

$$f(n) = n^3 + h(n)$$

for some  $h(n) \in O(n^2)$ .



# Macro substitution

**Convention:** A set in a formula represents an anonymous function in the set.

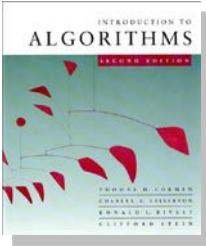
**EXAMPLE:**  $n^2 + O(n) = O(n^2)$

means

for any  $f(n) \in O(n)$ :

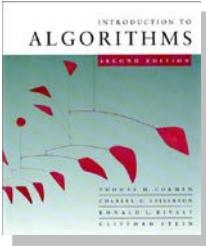
$$n^2 + f(n) = h(n)$$

for some  $h(n) \in O(n^2)$ .



# $\Omega$ -notation (lower bounds)

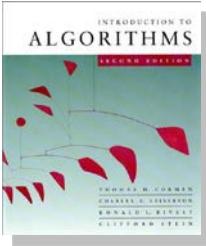
$O$ -notation is an *upper-bound* notation. It makes no sense to say  $f(n)$  is at least  $O(n^2)$ .



# $\Omega$ -notation (lower bounds)

$O$ -notation is an *upper-bound* notation. It makes no sense to say  $f(n)$  is at least  $O(n^2)$ .

$\Omega(g(n)) = \{ f(n) : \text{there exist constants } c > 0, n_0 > 0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0 \}$

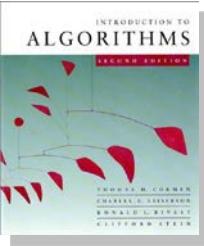


# $\Omega$ -notation (lower bounds)

$O$ -notation is an *upper-bound* notation. It makes no sense to say  $f(n)$  is at least  $O(n^2)$ .

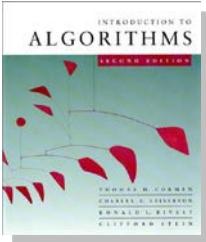
$\Omega(g(n)) = \{ f(n) : \text{there exist constants } c > 0, n_0 > 0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0 \}$

**EXAMPLE:**  $\sqrt{n} = \Omega(\lg n)$  ( $c = 1, n_0 = 16$ )



# $\Theta$ -notation (tight bounds)

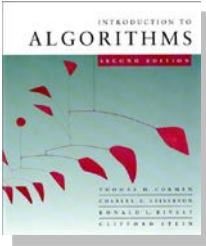
$$\Theta(g(n)) = \mathcal{O}(g(n)) \cap \Omega(g(n))$$



# $\Theta$ -notation (tight bounds)

$$\Theta(g(n)) = \mathcal{O}(g(n)) \cap \Omega(g(n))$$

**EXAMPLE:**  $\frac{1}{2}n^2 - 2n = \Theta(n^2)$

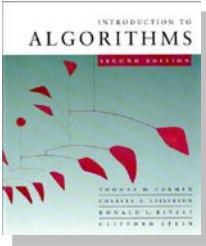


# $O$ -notation and $\omega$ -notation

$O$ -notation and  $\Omega$ -notation are like  $\leq$  and  $\geq$ .  
 $o$ -notation and  $\omega$ -notation are like  $<$  and  $>$ .

$o(g(n)) = \{ f(n) : \text{for any constant } c > 0,$   
there is a constant  $n_0 > 0$   
such that  $0 \leq f(n) < cg(n)$   
for all  $n \geq n_0 \}$

**EXAMPLE:**  $2n^2 = o(n^3)$     ( $n_0 = 2/c$ )

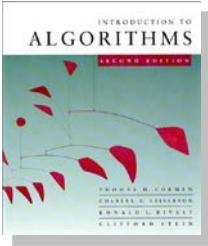


# $\Theta$ -notation and $\omega$ -notation

$O$ -notation and  $\Omega$ -notation are like  $\leq$  and  $\geq$ .  
 $\omega$ -notation and  $\omega$ -notation are like  $<$  and  $>$ .

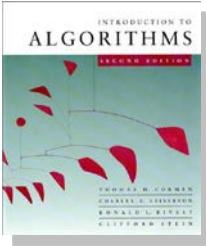
$\omega(g(n)) = \{ f(n) : \text{for any constant } c > 0,$   
there is a constant  $n_0 > 0$   
such that  $0 \leq cg(n) < f(n)$   
for all  $n \geq n_0 \}$

**EXAMPLE:**  $\sqrt{n} = \omega(\lg n)$     ( $n_0 = 1 + 1/c$ )



# Solving recurrences

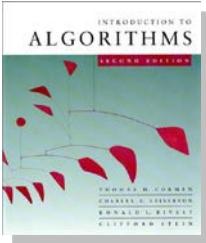
- The analysis of merge sort from **Lecture 1** required us to solve a recurrence.
- Recurrences are like solving integrals, differential equations, etc.
  - Learn a few tricks.
- **Lecture 3**: Applications of recurrences to divide-and-conquer algorithms.



# Substitution method

*The most general method:*

1. *Guess* the form of the solution.
2. *Verify* by induction.
3. *Solve* for constants.



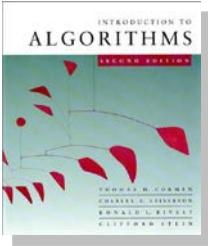
# Substitution method

*The most general method:*

1. **Guess** the form of the solution.
2. **Verify** by induction.
3. **Solve** for constants.

**EXAMPLE:**  $T(n) = 4T(n/2) + n$

- [Assume that  $T(1) = \Theta(1)$ .]
- Guess  $O(n^3)$ . (Prove  $O$  and  $\Omega$  separately.)
- Assume that  $T(k) \leq ck^3$  for  $k < n$ .
- Prove  $T(n) \leq cn^3$  by induction.

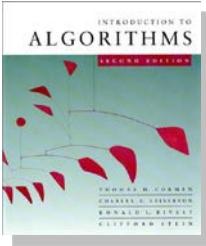


# Example of substitution

$$\begin{aligned} T(n) &= 4T(n/2) + n \\ &\leq 4c(n/2)^3 + n \\ &= (c/2)n^3 + n \\ &= cn^3 - ((c/2)n^3 - n) \leftarrow \textit{desired} - \textit{residual} \\ &\leq cn^3 \leftarrow \textit{desired} \end{aligned}$$

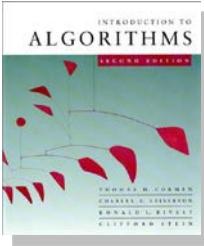
whenever  $(c/2)n^3 - n \geq 0$ , for example, if  $c \geq 2$  and  ~~$n \geq 1$~~ .

*residual*



# Example (continued)

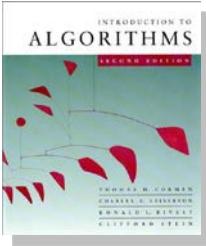
- We must also handle the initial conditions, that is, ground the induction with base cases.
- **Base:**  $T(n) = \Theta(1)$  for all  $n < n_0$ , where  $n_0$  is a suitable constant.
- For  $1 \leq n < n_0$ , we have “ $\Theta(1)$ ”  $\leq cn^3$ , if we pick  $c$  big enough.



# Example (continued)

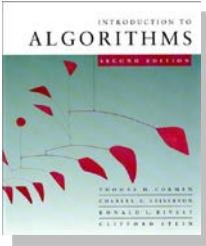
- We must also handle the initial conditions, that is, ground the induction with base cases.
  - **Base:**  $T(n) = \Theta(1)$  for all  $n < n_0$ , where  $n_0$  is a suitable constant.
  - For  $1 \leq n < n_0$ , we have “ $\Theta(1)$ ”  $\leq cn^3$ , if we pick  $c$  big enough.
- 
- 

*This bound is not tight!*



# A tighter upper bound?

We shall prove that  $\textcolor{teal}{T}(n) = O(n^2)$ .

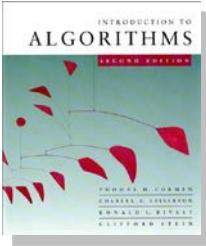


# A tighter upper bound?

We shall prove that  $T(n) = O(n^2)$ .

Assume that  $T(k) \leq ck^2$  for  $k < n$ :

$$\begin{aligned} T(n) &= 4T(n/2) + n \\ &\leq 4c(n/2)^2 + n \\ &= cn^2 + n \\ &= O(n^2) \end{aligned}$$



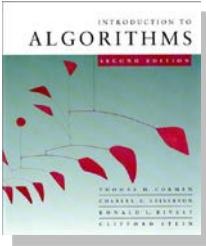
# A tighter upper bound?

We shall prove that  $T(n) = O(n^2)$ .

Assume that  $T(k) \leq ck^2$  for  $k < n$ :

$$\begin{aligned} T(n) &= 4T(n/2) + n \\ &\leq 4c(n/2)^2 + n \\ &= cn^2 + n \\ &= O(n^2) \end{aligned}$$

~~O(n<sup>2</sup>)~~ *Wrong!* We must prove the I.H.



# A tighter upper bound?

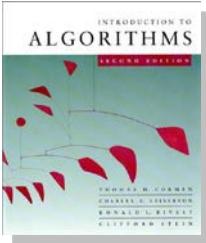
We shall prove that  $T(n) = O(n^2)$ .

Assume that  $T(k) \leq ck^2$  for  $k < n$ :

$$\begin{aligned} T(n) &= 4T(n/2) + n \\ &\leq 4c(n/2)^2 + n \\ &= cn^2 + n \end{aligned}$$

~~$O(n^2)$~~  **Wrong!** We must prove the I.H.

$$\begin{aligned} &= cn^2 - (-n) \quad [ \text{desired} - \text{residual} ] \\ &\leq cn^2 \quad \text{for } \mathbf{no} \text{ choice of } c > 0. \text{ Lose!} \end{aligned}$$

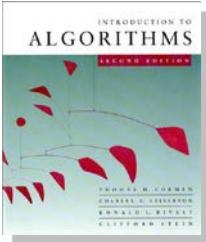


# A tighter upper bound!

**IDEA:** Strengthen the inductive hypothesis.

- *Subtract* a low-order term.

*Inductive hypothesis:*  $T(k) \leq c_1k^2 - c_2k$  for  $k < n$ .



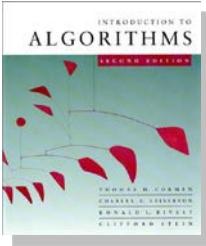
# A tighter upper bound!

**IDEA:** Strengthen the inductive hypothesis.

- *Subtract* a low-order term.

*Inductive hypothesis:*  $T(k) \leq c_1k^2 - c_2k$  for  $k < n$ .

$$\begin{aligned} T(n) &= 4T(n/2) + n \\ &= 4(c_1(n/2)^2 - c_2(n/2)) + n \\ &= c_1n^2 - 2c_2n + n \\ &= c_1n^2 - c_2n - (c_2n - n) \\ &\leq c_1n^2 - c_2n \text{ if } c_2 \geq 1. \end{aligned}$$



# A tighter upper bound!

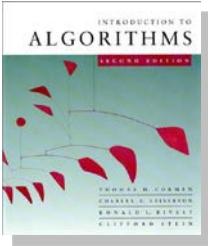
**IDEA:** Strengthen the inductive hypothesis.

- *Subtract* a low-order term.

*Inductive hypothesis:*  $T(k) \leq c_1k^2 - c_2k$  for  $k < n$ .

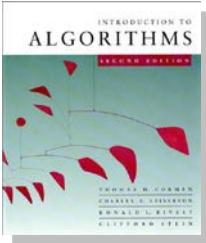
$$\begin{aligned} T(n) &= 4T(n/2) + n \\ &= 4(c_1(n/2)^2 - c_2(n/2)) + n \\ &= c_1n^2 - 2c_2n + n \\ &= c_1n^2 - c_2n - (c_2n - n) \\ &\leq c_1n^2 - c_2n \text{ if } c_2 \geq 1. \end{aligned}$$

Pick  $c_1$  big enough to handle the initial conditions.



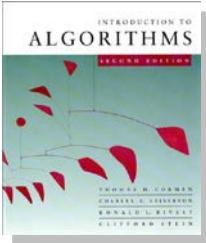
# Recursion-tree method

- A recursion tree models the costs (time) of a recursive execution of an algorithm.
- The recursion-tree method can be unreliable, just like any method that uses ellipses (...).
- The recursion-tree method promotes intuition, however.
- The recursion tree method is good for generating guesses for the substitution method.



# Example of recursion tree

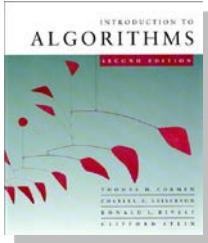
Solve  $T(n) = T(n/4) + T(n/2) + n^2$ :



# Example of recursion tree

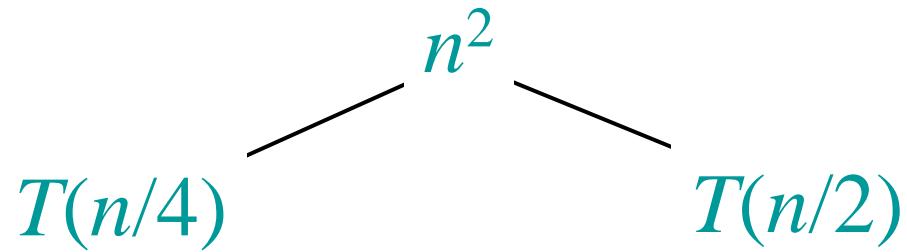
Solve  $T(n) = T(n/4) + T(n/2) + n^2$ :

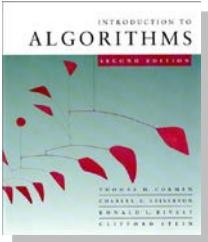
$$T(n)$$



# Example of recursion tree

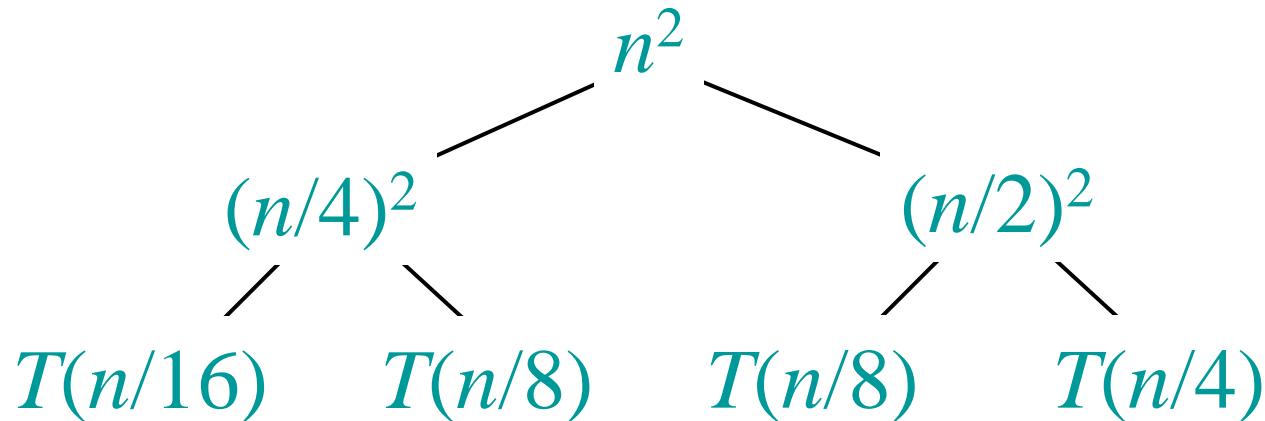
Solve  $T(n) = T(n/4) + T(n/2) + n^2$ :

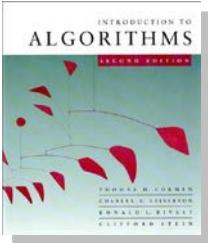




# Example of recursion tree

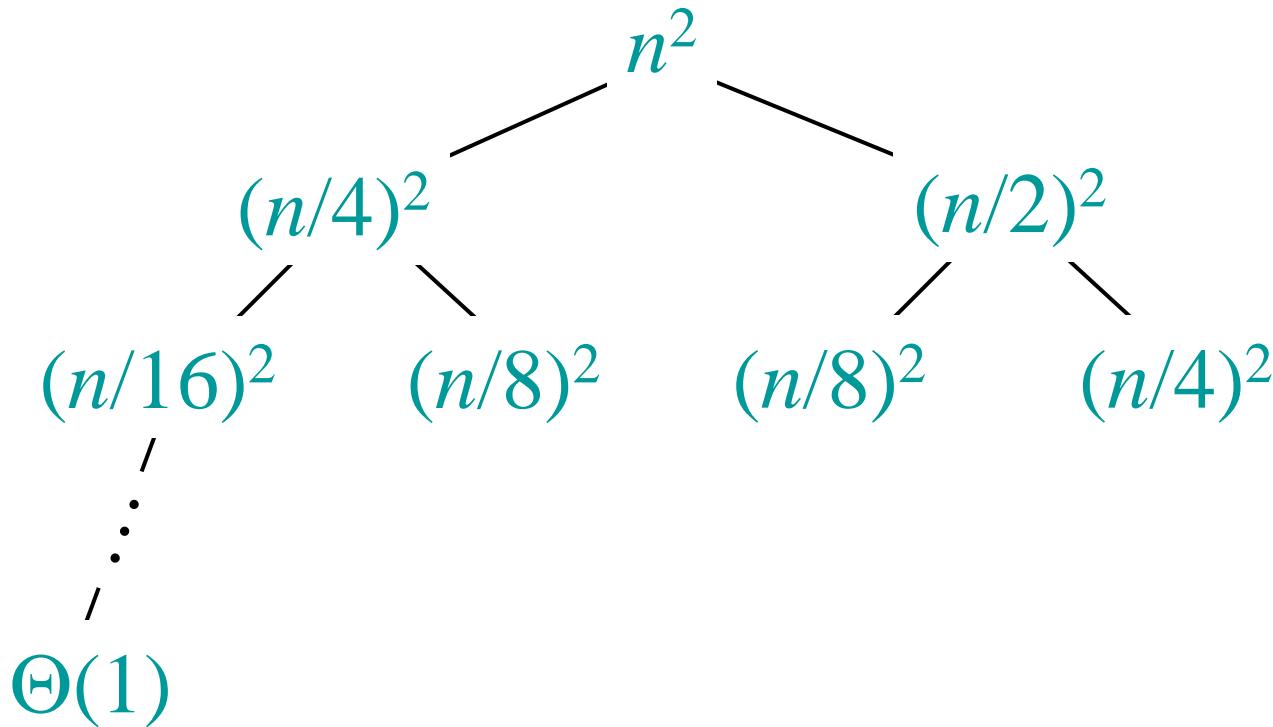
Solve  $T(n) = T(n/4) + T(n/2) + n^2$ :

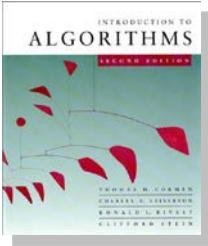




# Example of recursion tree

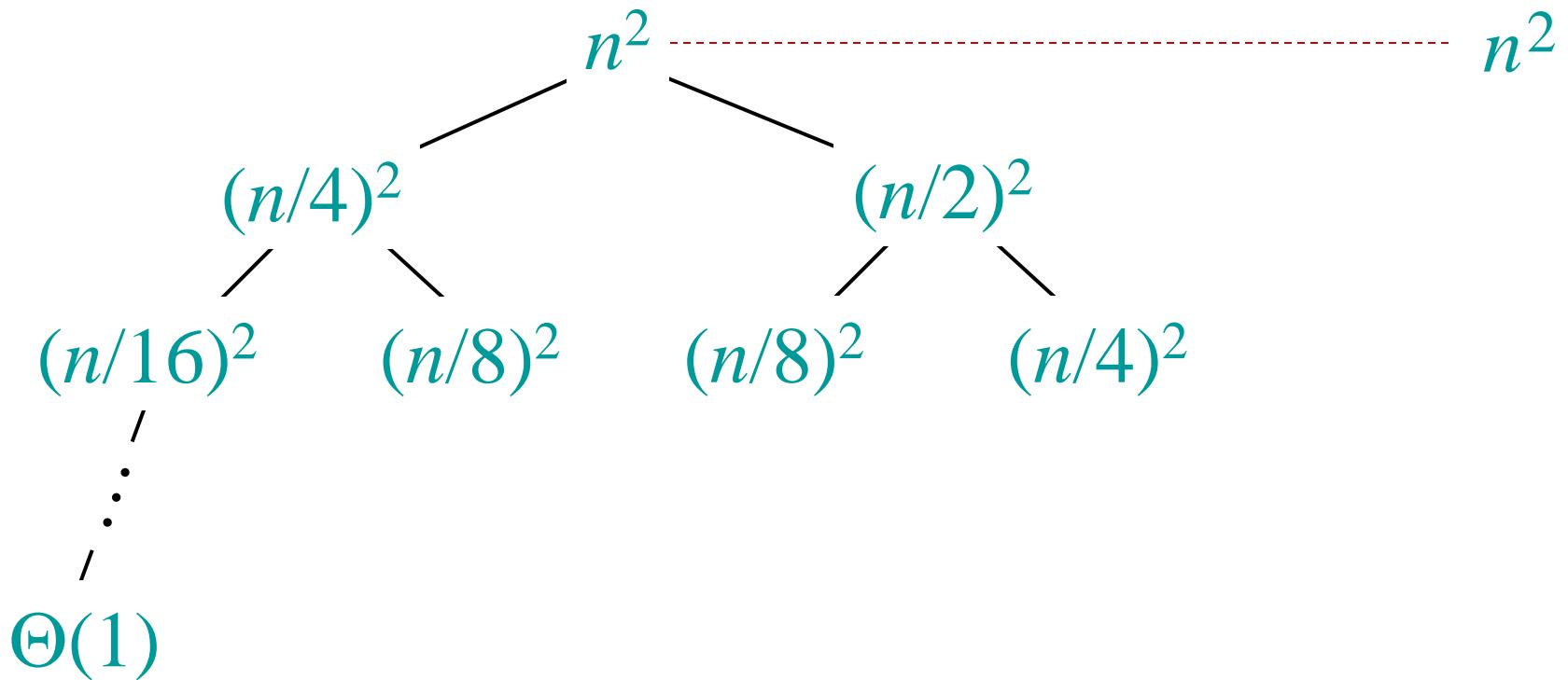
Solve  $T(n) = T(n/4) + T(n/2) + n^2$ :

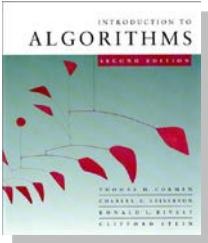




# Example of recursion tree

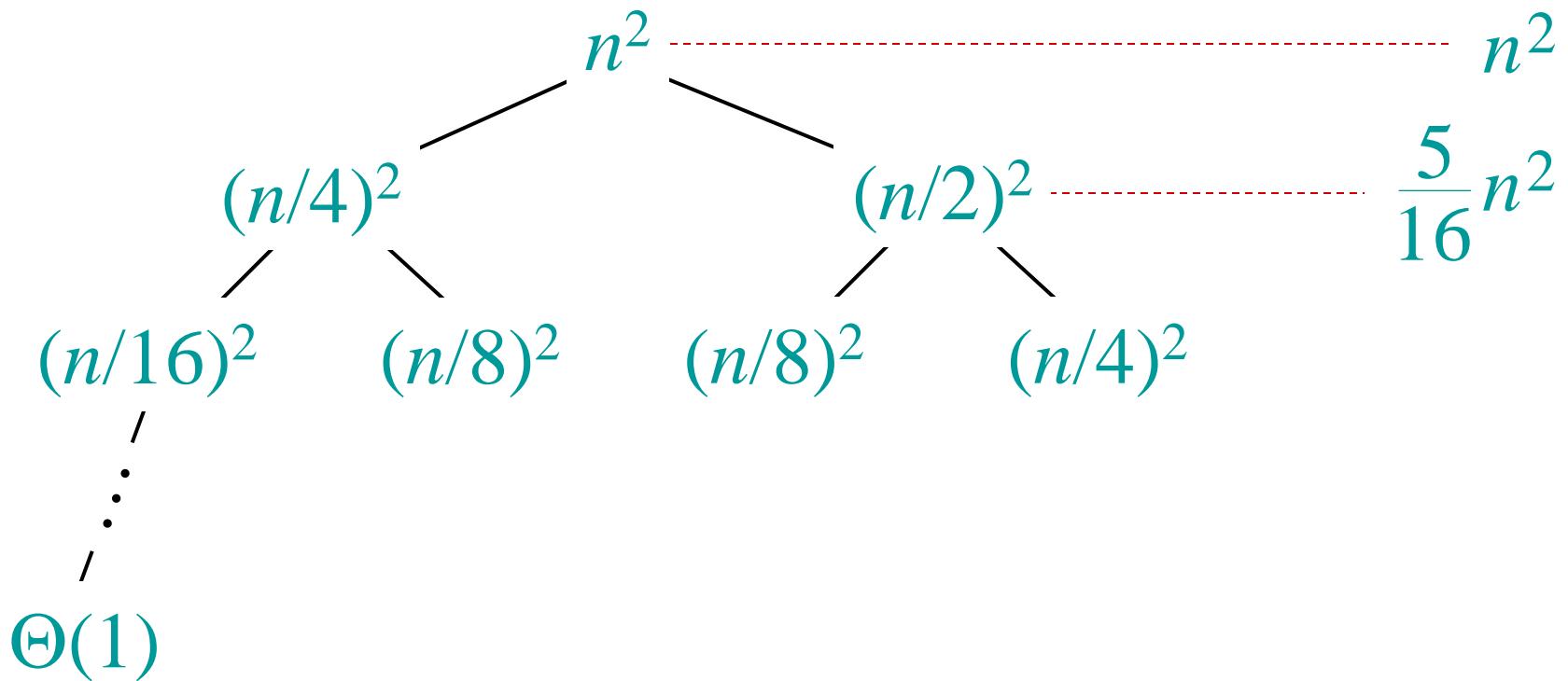
Solve  $T(n) = T(n/4) + T(n/2) + n^2$ :

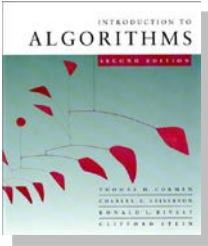




# Example of recursion tree

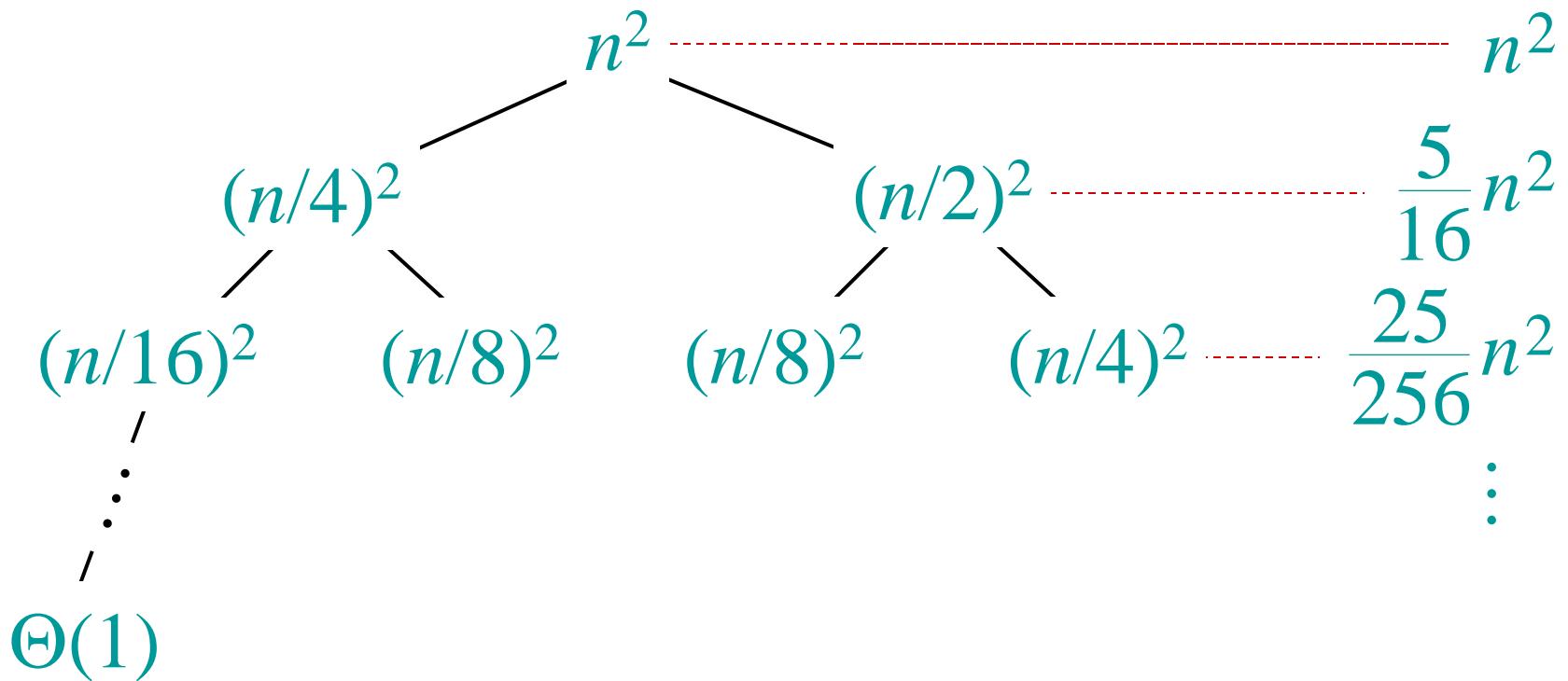
Solve  $T(n) = T(n/4) + T(n/2) + n^2$ :

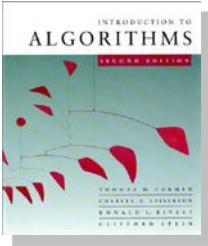




# Example of recursion tree

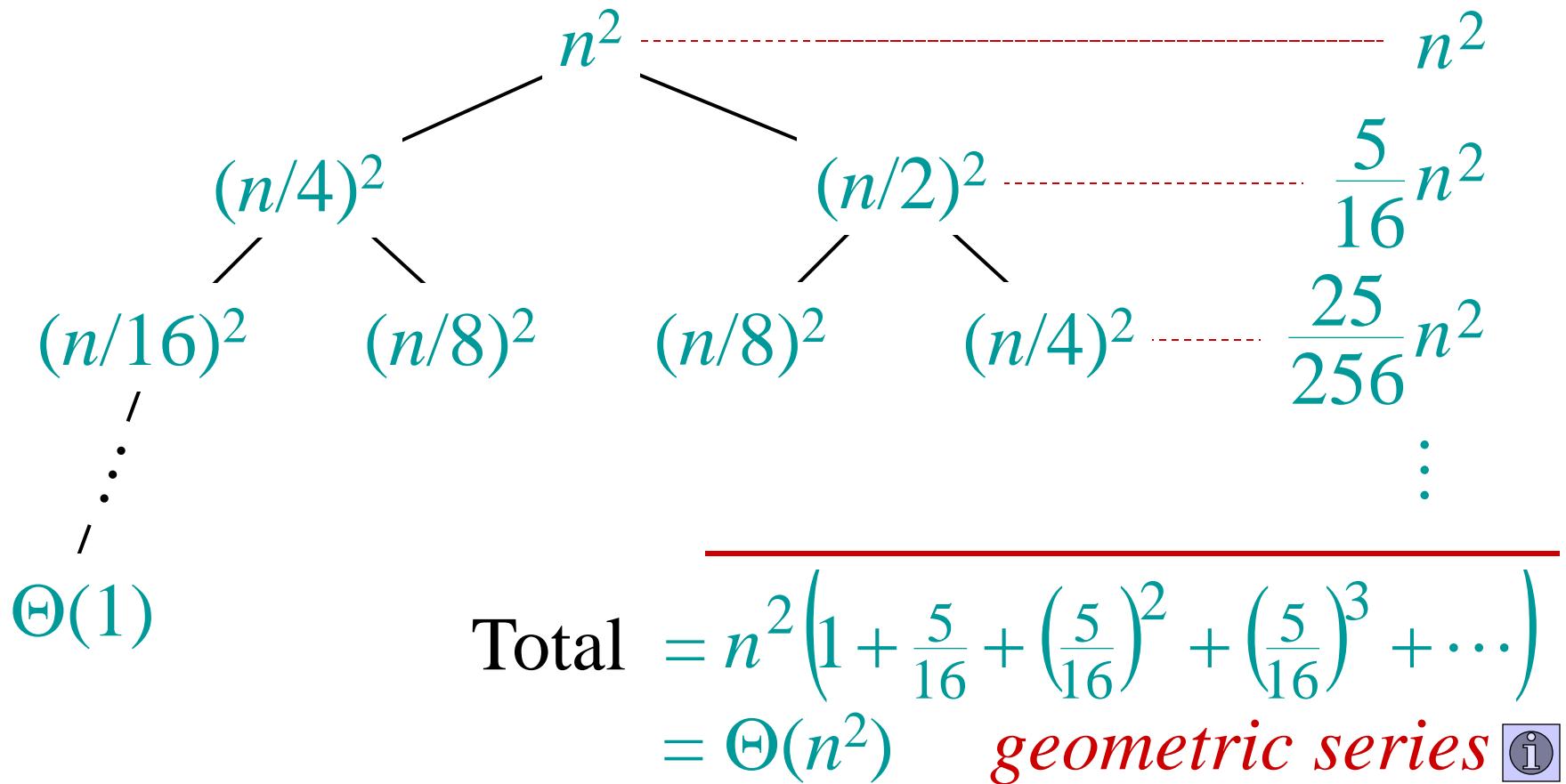
Solve  $T(n) = T(n/4) + T(n/2) + n^2$ :

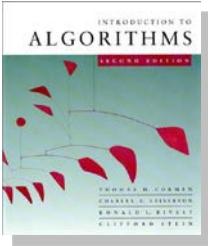




# Example of recursion tree

Solve  $T(n) = T(n/4) + T(n/2) + n^2$ :



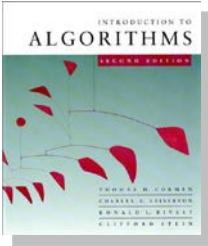


# The master method

The master method applies to recurrences of the form

$$T(n) = a T(n/b) + f(n) ,$$

where  $a \geq 1$ ,  $b > 1$ , and  $f$  is asymptotically positive.



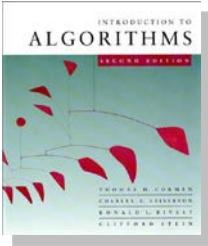
# Three common cases

Compare  $f(n)$  with  $n^{\log b a}$ :

1.  $f(n) = O(n^{\log b a - \varepsilon})$  for some constant  $\varepsilon > 0$ .

- $f(n)$  grows polynomially slower than  $n^{\log b a}$  (by an  $n^\varepsilon$  factor).

**Solution:**  $T(n) = \Theta(n^{\log b a})$ .



# Three common cases

Compare  $f(n)$  with  $n^{\log b a}$ :

1.  $f(n) = O(n^{\log b a - \varepsilon})$  for some constant  $\varepsilon > 0$ .

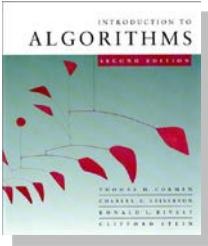
- $f(n)$  grows polynomially slower than  $n^{\log b a}$  (by an  $n^\varepsilon$  factor).

**Solution:**  $T(n) = \Theta(n^{\log b a})$ .

2.  $f(n) = \Theta(n^{\log b a} \lg^k n)$  for some constant  $k \geq 0$ .

- $f(n)$  and  $n^{\log b a}$  grow at similar rates.

**Solution:**  $T(n) = \Theta(n^{\log b a} \lg^{k+1} n)$ .



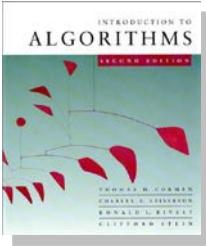
# Three common cases (cont.)

Compare  $f(n)$  with  $n^{\log_b a}$ :

3.  $f(n) = \Omega(n^{\log_b a + \varepsilon})$  for some constant  $\varepsilon > 0$ .
  - $f(n)$  grows polynomially faster than  $n^{\log_b a}$  (by an  $n^\varepsilon$  factor),

and  $f(n)$  satisfies the ***regularity condition*** that  $af(n/b) \leq cf(n)$  for some constant  $c < 1$ .

***Solution:***  $T(n) = \Theta(f(n))$ .



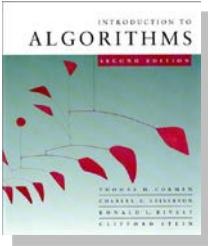
# Examples

**Ex.**  $T(n) = 4T(n/2) + n$

$a = 4, b = 2 \Rightarrow n^{\log_b a} = n^2; f(n) = n.$

**CASE 1:**  $f(n) = O(n^{2-\varepsilon})$  for  $\varepsilon = 1$ .

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



# Examples

**Ex.**  $T(n) = 4T(n/2) + n$

$$a = 4, b = 2 \Rightarrow n^{\log_b a} = n^2; f(n) = n.$$

**CASE 1:**  $f(n) = O(n^{2-\varepsilon})$  for  $\varepsilon = 1$ .

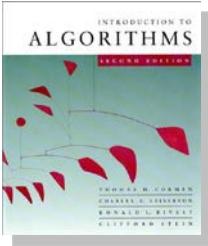
$$\therefore T(n) = \Theta(n^2).$$

**Ex.**  $T(n) = 4T(n/2) + n^2$

$$a = 4, b = 2 \Rightarrow n^{\log_b a} = n^2; f(n) = n^2.$$

**CASE 2:**  $f(n) = \Theta(n^2 \lg^0 n)$ , that is,  $k = 0$ .

$$\therefore T(n) = \Theta(n^2 \lg n).$$



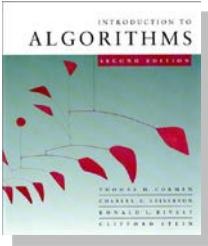
# Examples

**Ex.**  $T(n) = 4T(n/2) + n^3$

$a = 4, b = 2 \Rightarrow n^{\log_b a} = n^2; f(n) = n^3.$

CASE 3:  $f(n) = \Omega(n^{2+\varepsilon})$  for  $\varepsilon = 1$

and  $4(n/2)^3 \leq cn^3$  (reg. cond.) for  $c = 1/2.$   
 $\therefore T(n) = \Theta(n^3).$



# Examples

**Ex.**  $T(n) = 4T(n/2) + n^3$

$a = 4, b = 2 \Rightarrow n^{\log_b a} = n^2; f(n) = n^3.$

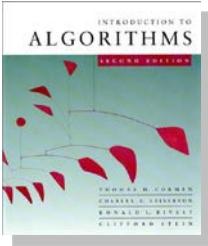
CASE 3:  $f(n) = \Omega(n^{2+\varepsilon})$  for  $\varepsilon = 1$

and  $4(n/2)^3 \leq cn^3$  (reg. cond.) for  $c = 1/2$ .  
 $\therefore T(n) = \Theta(n^3).$

**Ex.**  $T(n) = 4T(n/2) + n^2/\lg n$

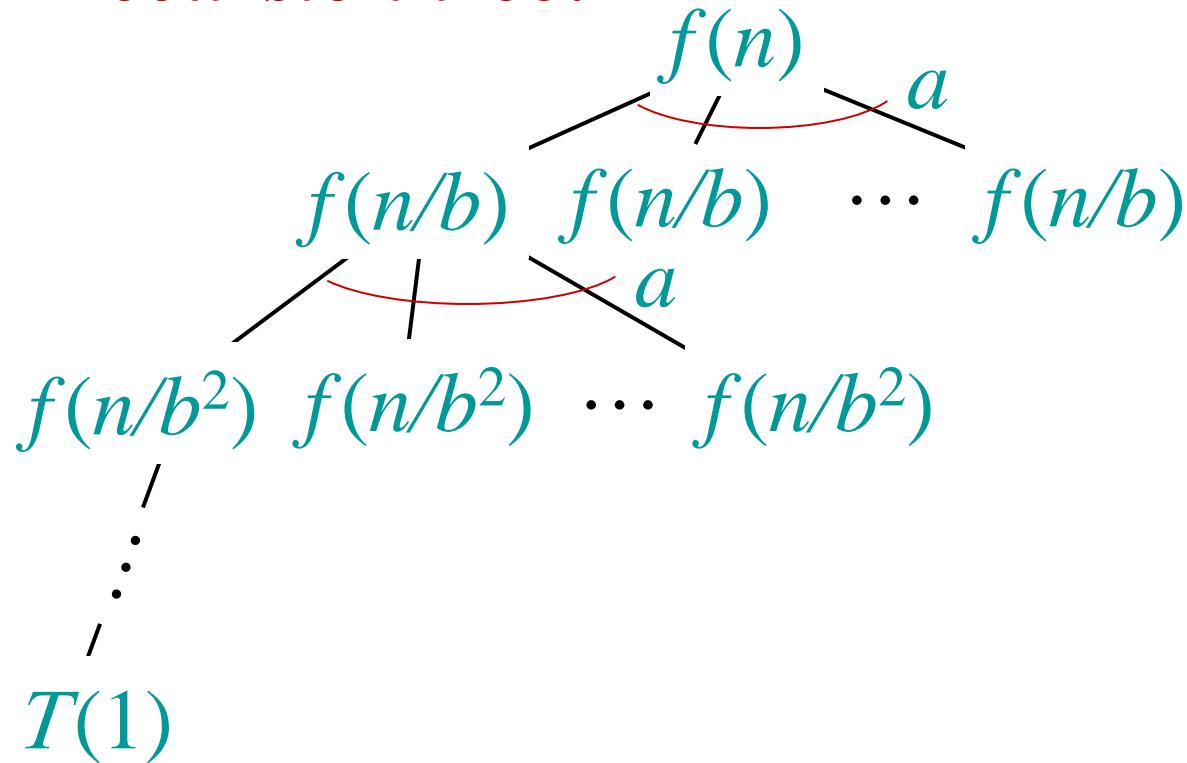
$a = 4, b = 2 \Rightarrow n^{\log_b a} = n^2; f(n) = n^2/\lg n.$

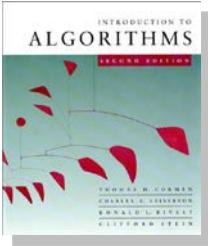
Master method does not apply. In particular, for every constant  $\varepsilon > 0$ , we have  $n^\varepsilon = \omega(\lg n)$ .



# Idea of master theorem

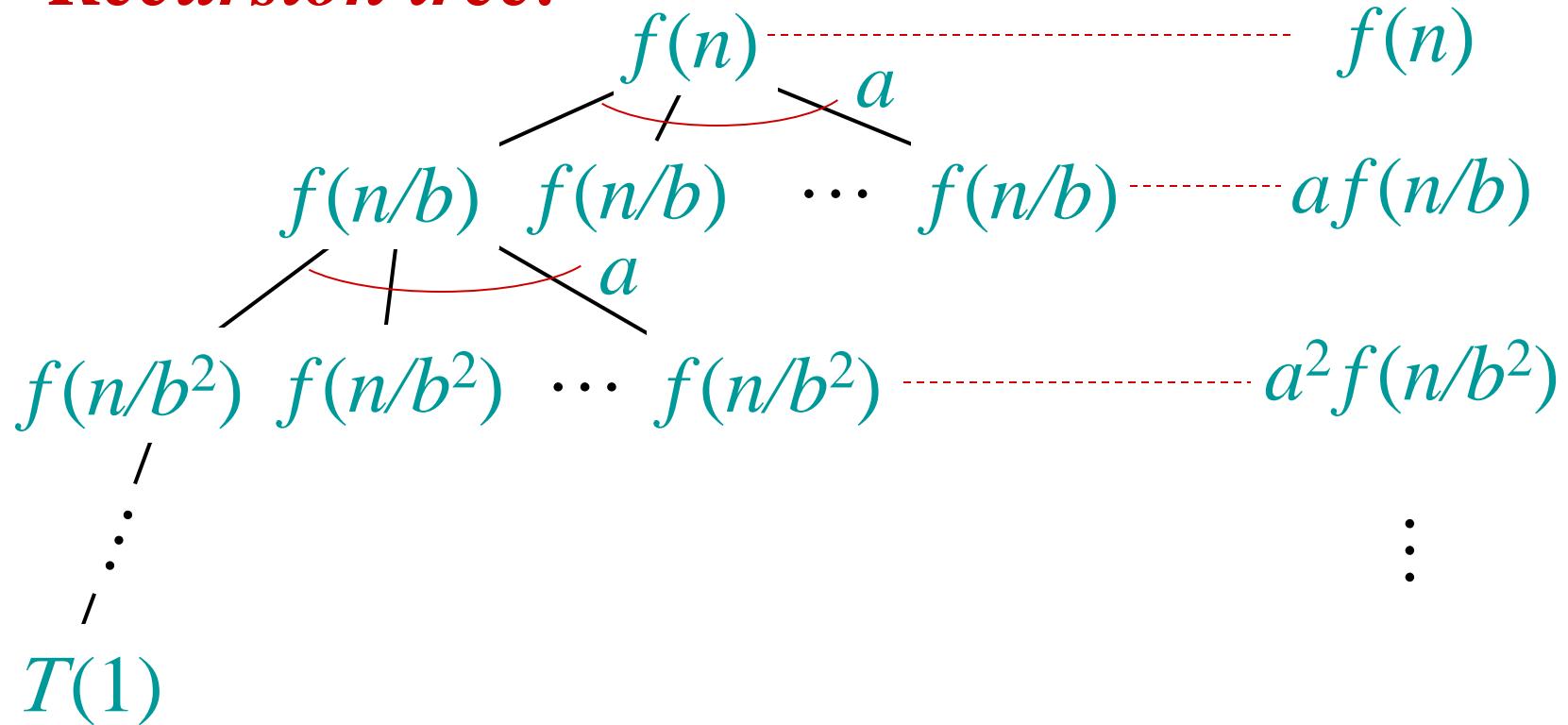
*Recursion tree:*

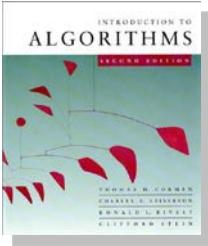




# Idea of master theorem

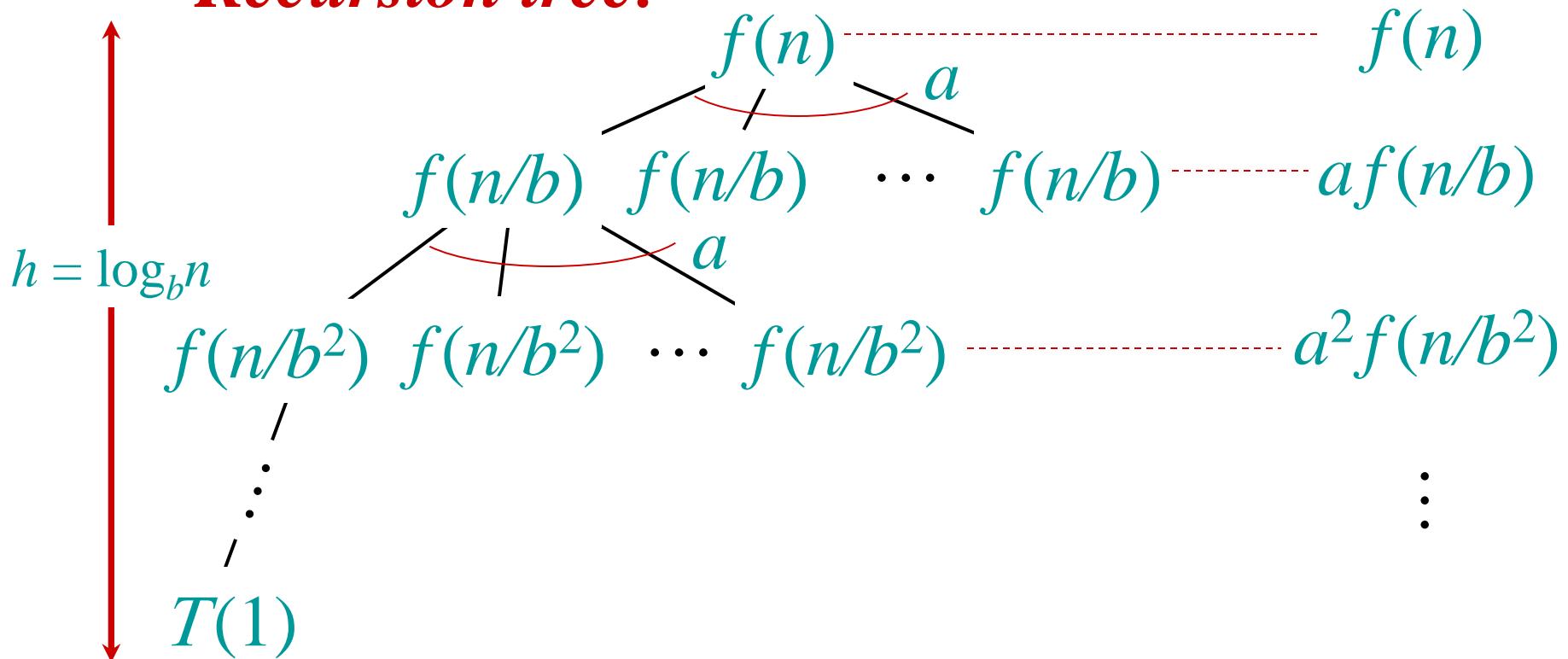
*Recursion tree:*

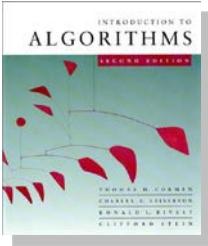




# Idea of master theorem

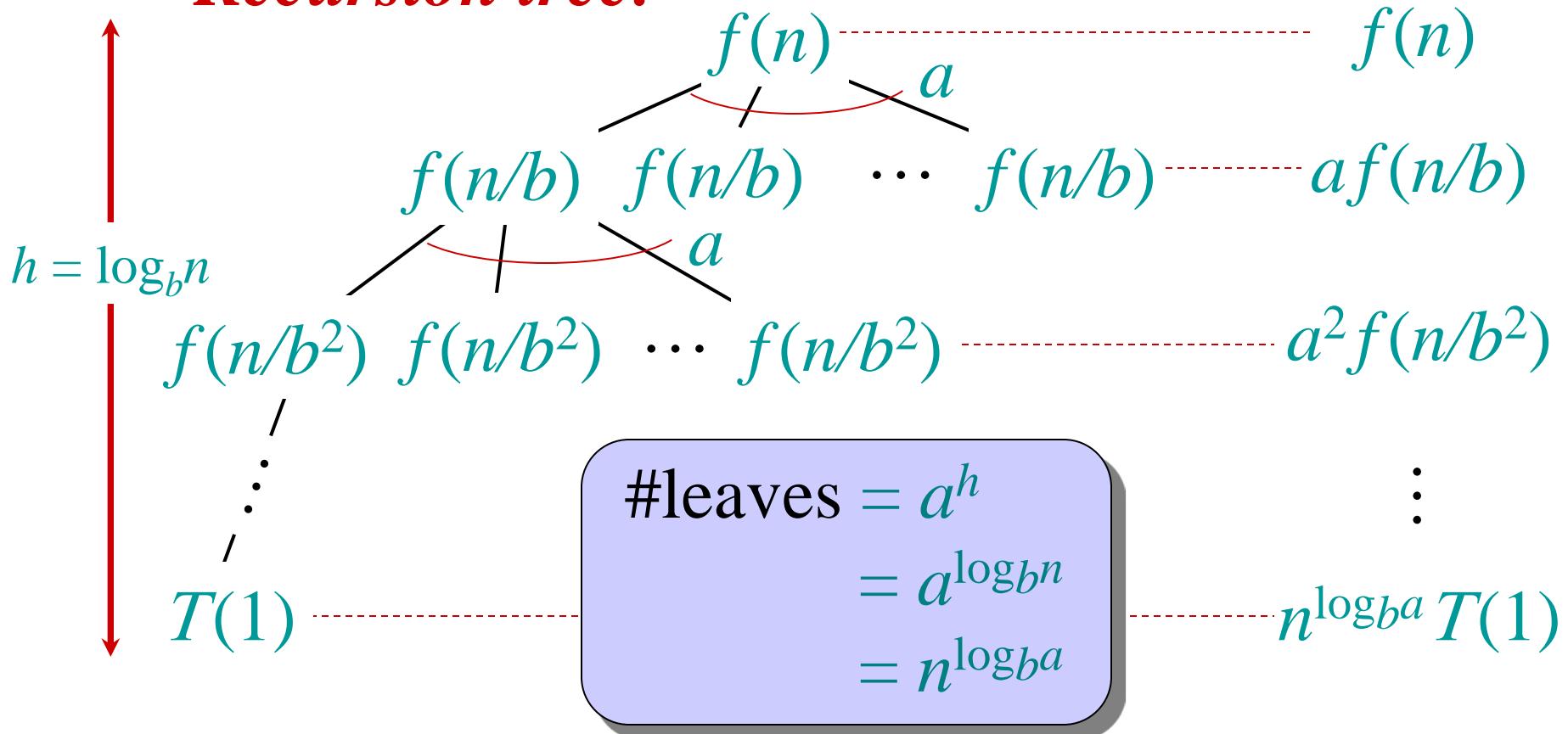
*Recursion tree:*

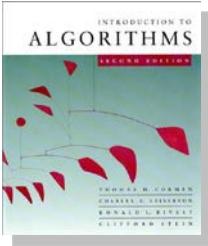




# Idea of master theorem

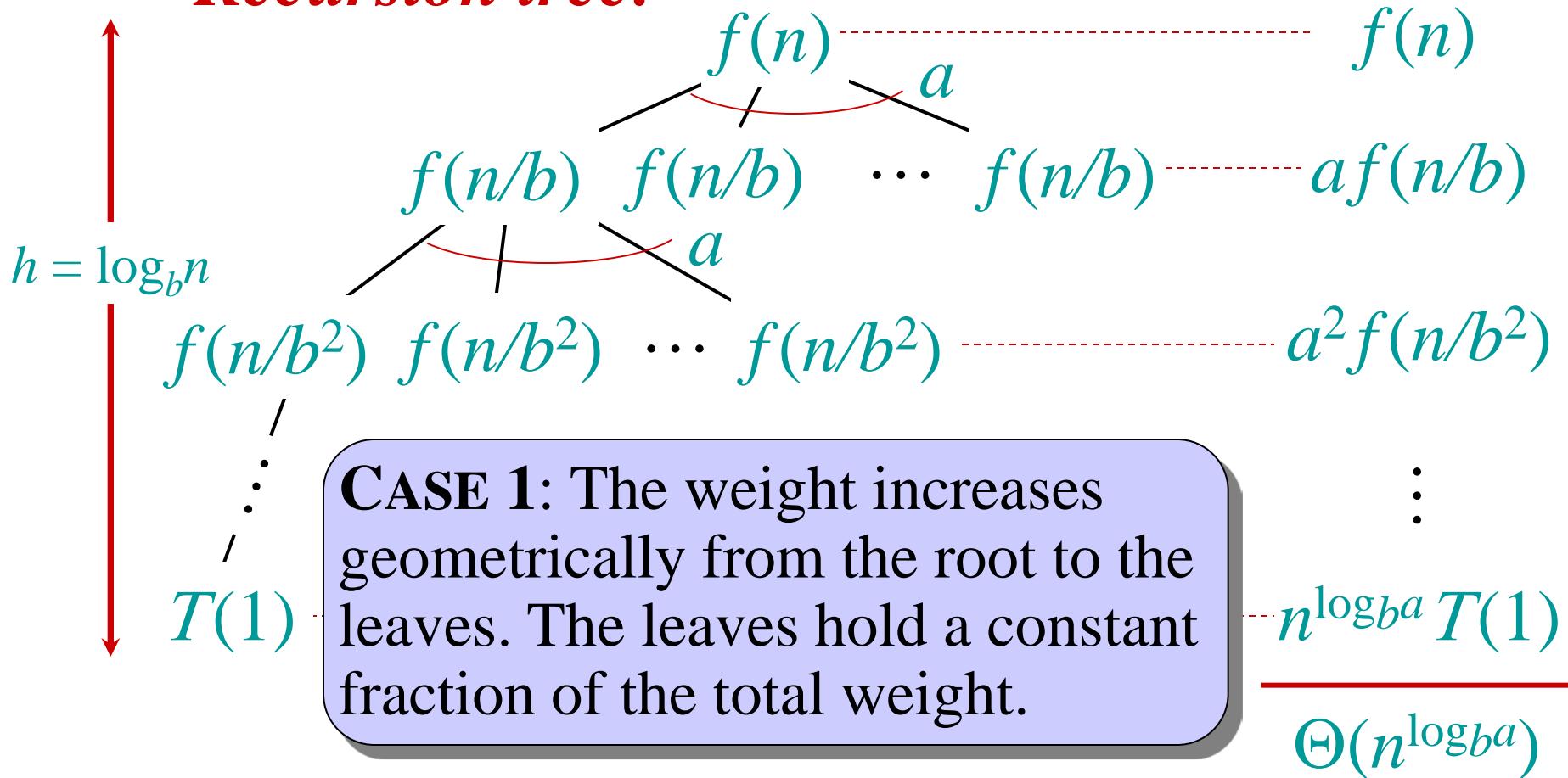
*Recursion tree:*

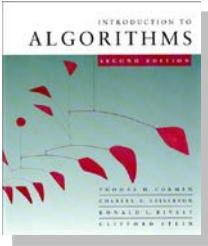




# Idea of master theorem

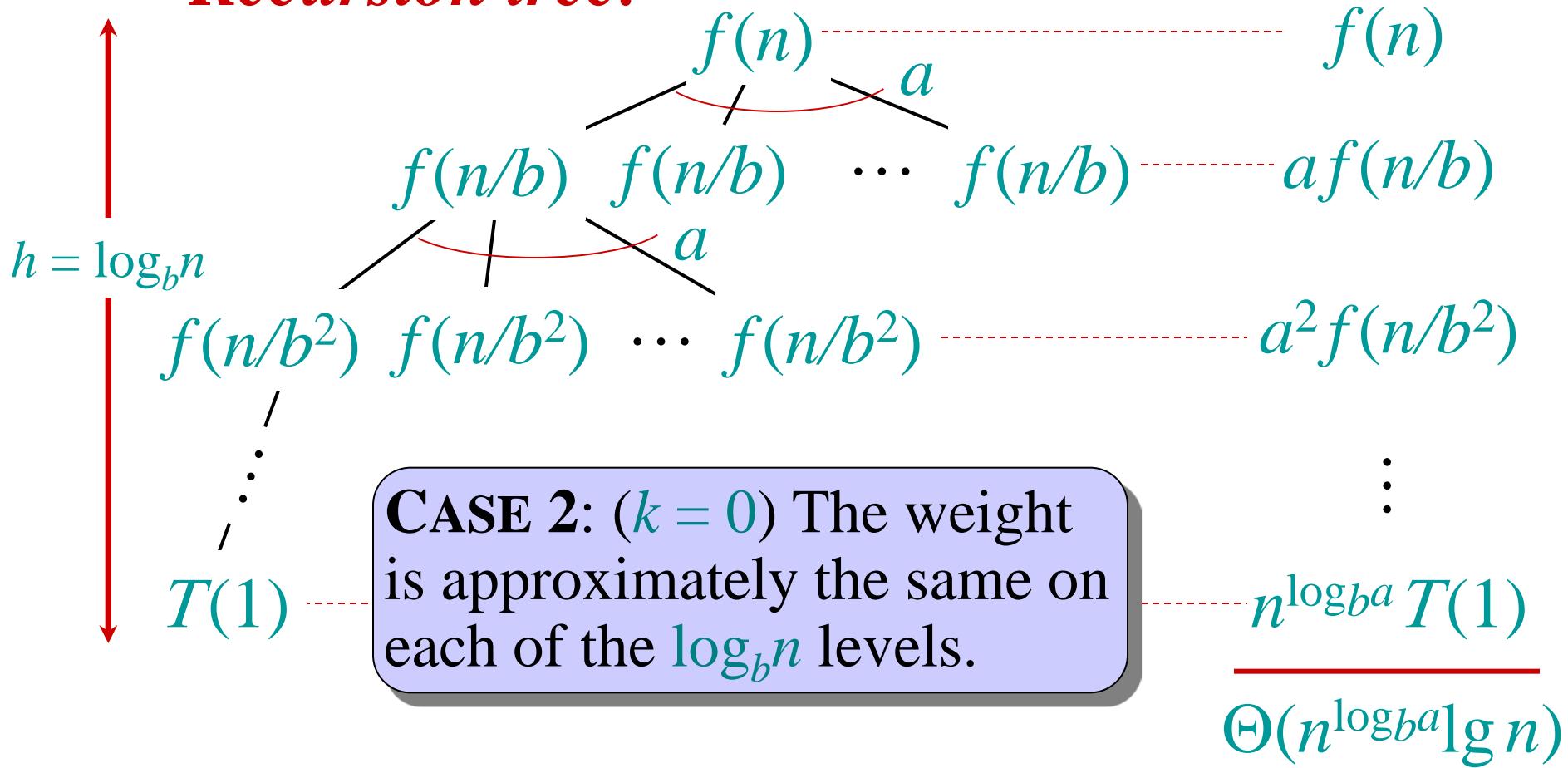
*Recursion tree:*

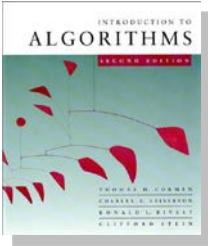




# Idea of master theorem

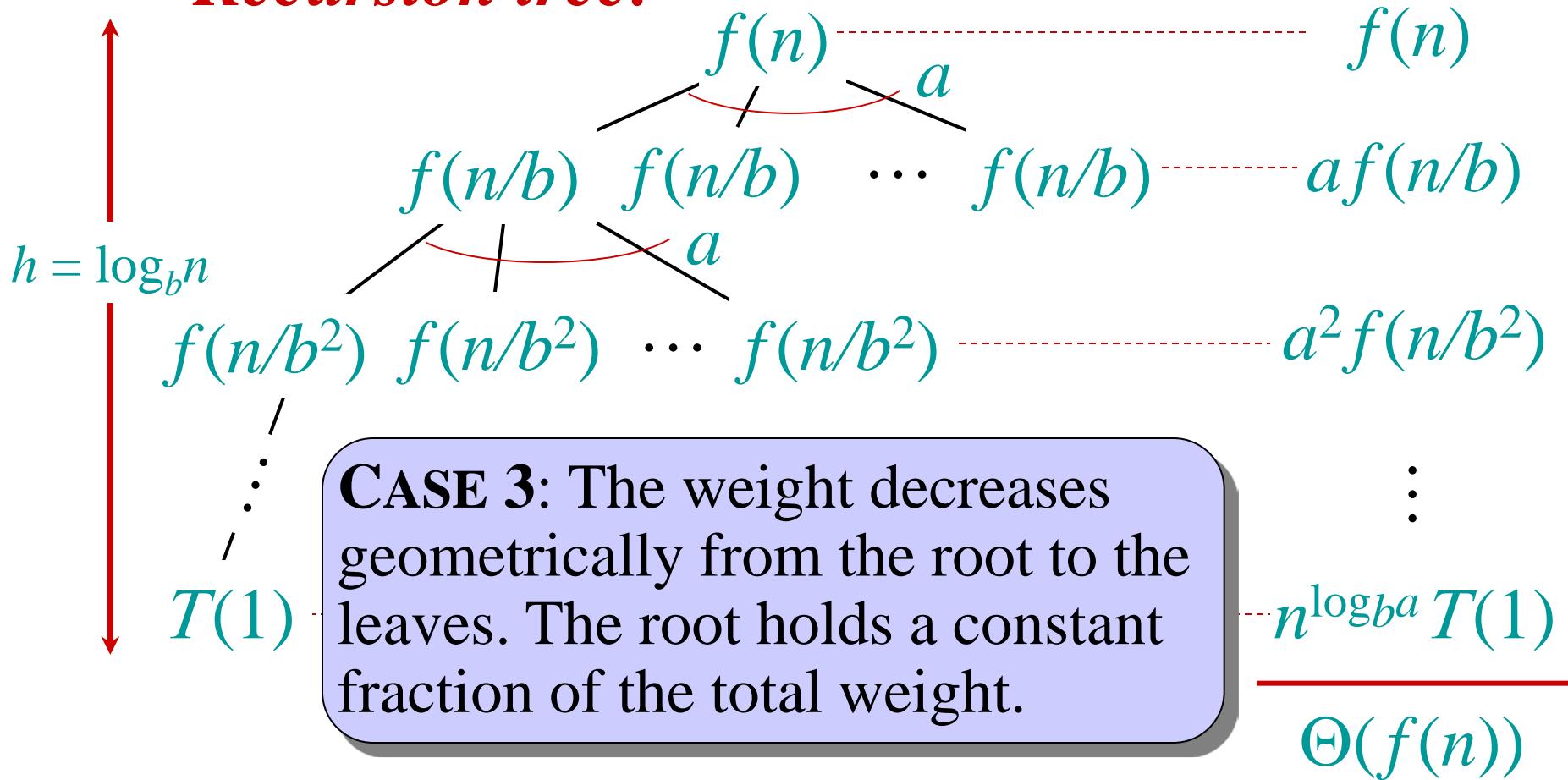
*Recursion tree:*





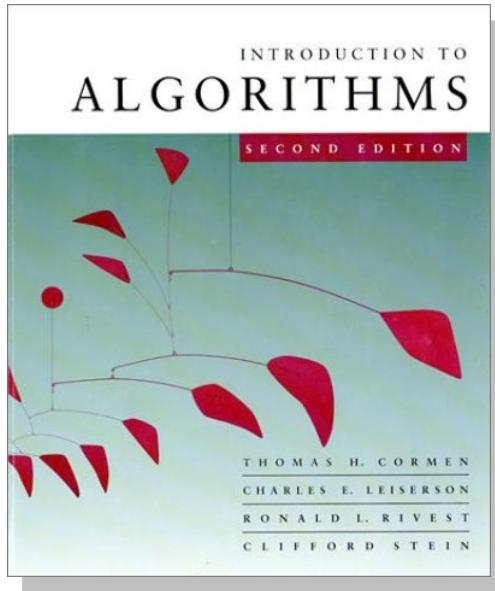
# Idea of master theorem

*Recursion tree:*



# *Introduction to Algorithms*

**6.046J/18.401J**

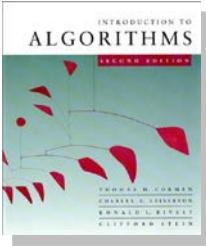


## **LECTURE 3**

### **Divide and Conquer**

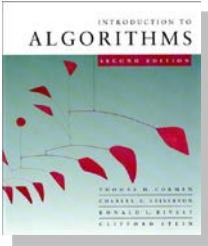
- Binary search
- Powering a number
- Fibonacci numbers
- Matrix multiplication
- Strassen's algorithm
- VLSI tree layout

**Prof. Erik D. Demaine**



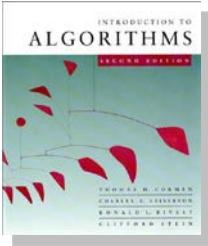
# The divide-and-conquer design paradigm

1. *Divide* the problem (instance) into subproblems.
2. *Conquer* the subproblems by solving them recursively.
3. *Combine* subproblem solutions.



# Merge sort

1. *Divide*: Trivial.
2. *Conquer*: Recursively sort 2 subarrays.
3. *Combine*: Linear-time merge.



# Merge sort

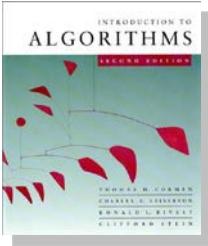
1. **Divide:** Trivial.
  2. **Conquer:** Recursively sort 2 subarrays.
  3. **Combine:** Linear-time merge.

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

# subproblems

subproblem size

work dividing and combining



# Master theorem (reprise)

$$T(n) = a T(n/b) + f(n)$$

**CASE 1:**  $f(n) = O(n^{\log_b a - \varepsilon})$ , constant  $\varepsilon > 0$

$$\Rightarrow T(n) = \Theta(n^{\log_b a}) .$$

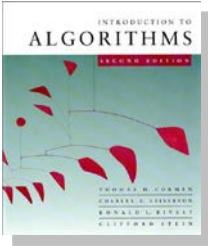
**CASE 2:**  $f(n) = \Theta(n^{\log_b a} \lg^k n)$ , constant  $k \geq 0$

$$\Rightarrow T(n) = \Theta(n^{\log_b a} \lg^{k+1} n) .$$

**CASE 3:**  $f(n) = \Omega(n^{\log_b a + \varepsilon})$ , constant  $\varepsilon > 0$ ,

and regularity condition

$$\Rightarrow T(n) = \Theta(f(n)) .$$



# Master theorem (reprise)

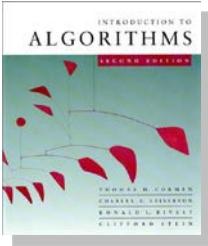
$$T(n) = a T(n/b) + f(n)$$

**CASE 1:**  $f(n) = O(n^{\log_b a - \varepsilon})$ , constant  $\varepsilon > 0$   
 $\Rightarrow T(n) = \Theta(n^{\log_b a})$ .

**CASE 2:**  $f(n) = \Theta(n^{\log_b a} \lg^k n)$ , constant  $k \geq 0$   
 $\Rightarrow T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$ .

**CASE 3:**  $f(n) = \Omega(n^{\log_b a + \varepsilon})$ , constant  $\varepsilon > 0$ ,  
and regularity condition  
 $\Rightarrow T(n) = \Theta(f(n))$ .

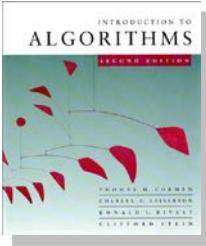
**Merge sort:**  $a = 2, b = 2 \Rightarrow n^{\log_b a} = n^{\log_2 2} = n$   
 $\Rightarrow$  CASE 2 ( $k = 0$ )  $\Rightarrow T(n) = \Theta(n \lg n)$ .



# Binary search

Find an element in a sorted array:

1. ***Divide:*** Check middle element.
2. ***Conquer:*** Recursively search 1 subarray.
3. ***Combine:*** Trivial.



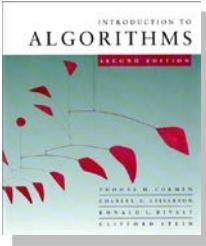
# Binary search

Find an element in a sorted array:

1. ***Divide***: Check middle element.
2. ***Conquer***: Recursively search 1 subarray.
3. ***Combine***: Trivial.

*Example*: Find 9

3    5    7    8    9    12    15



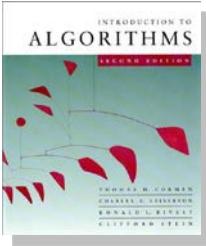
# Binary search

Find an element in a sorted array:

1. ***Divide***: Check middle element.
2. ***Conquer***: Recursively search 1 subarray.
3. ***Combine***: Trivial.

*Example*: Find 9

3    5    7    8    9    12    15



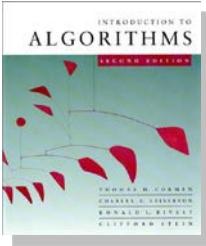
# Binary search

Find an element in a sorted array:

1. ***Divide***: Check middle element.
2. ***Conquer***: Recursively search 1 subarray.
3. ***Combine***: Trivial.

*Example*: Find 9

3    5    7    8    9    12    15



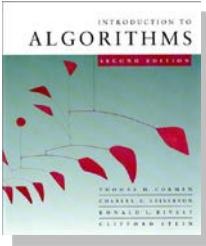
# Binary search

Find an element in a sorted array:

1. ***Divide***: Check middle element.
2. ***Conquer***: Recursively search 1 subarray.
3. ***Combine***: Trivial.

*Example*: Find 9

3    5    7    8    9    12    15



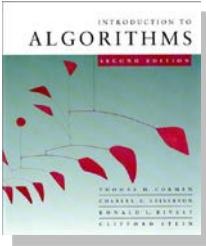
# Binary search

Find an element in a sorted array:

1. ***Divide***: Check middle element.
2. ***Conquer***: Recursively search 1 subarray.
3. ***Combine***: Trivial.

*Example*: Find 9

3    5    7    8    9    12    15



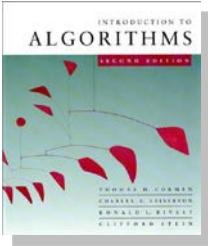
# Binary search

Find an element in a sorted array:

1. ***Divide***: Check middle element.
2. ***Conquer***: Recursively search 1 subarray.
3. ***Combine***: Trivial.

*Example*: Find 9





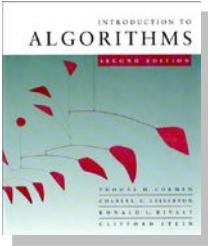
# Recurrence for binary search

$$T(n) = 1T(n/2) + \Theta(1)$$

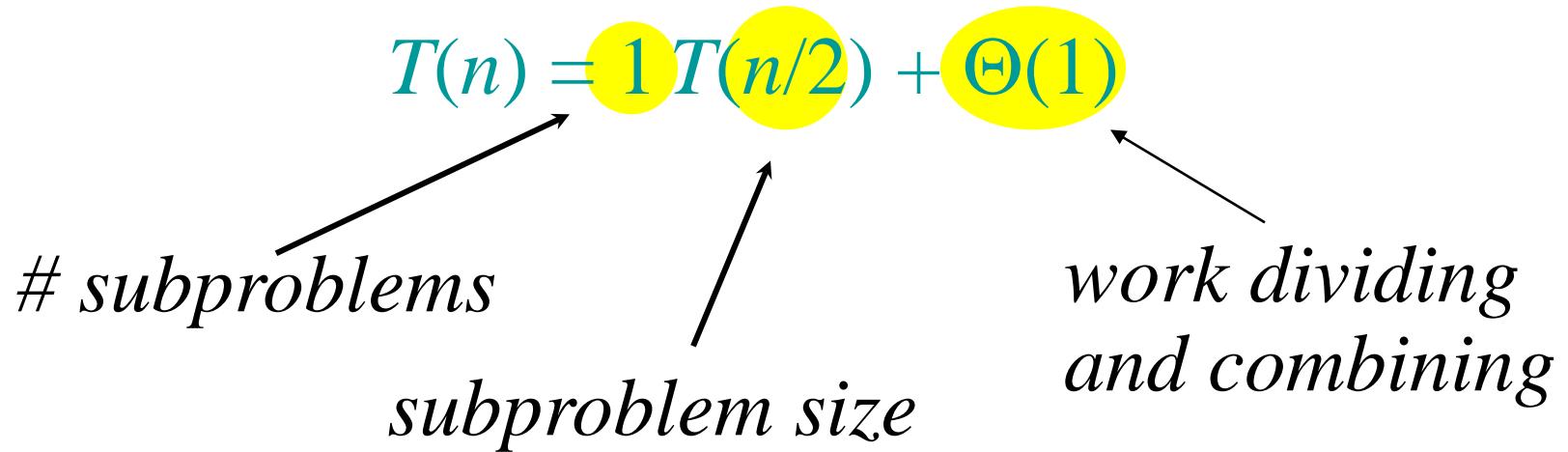
# subproblems                      subproblem size

work dividing  
and combining

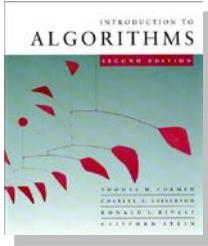
The diagram illustrates the components of the recurrence relation  $T(n) = 1T(n/2) + \Theta(1)$ . The equation is centered at the top. Below it, three arrows point to different parts of the formula: one arrow from the left points to the first  $T(n/2)$  term, labeled "# subproblems"; another arrow from the bottom points to the  $\Theta(1)$  term, labeled "subproblem size"; and a third arrow from the right points to the second  $T(n/2)$  term, labeled "work dividing and combining".



# Recurrence for binary search



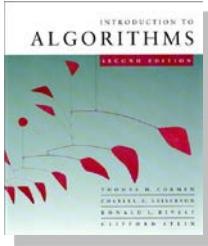
$$\begin{aligned} n^{\log_b a} &= n^{\log_2 1} = n^0 = 1 \Rightarrow \text{CASE 2 } (k = 0) \\ \Rightarrow T(n) &= \Theta(\lg n). \end{aligned}$$



# Powering a number

**Problem:** Compute  $a^n$ , where  $n \in \mathbb{N}$ .

**Naive algorithm:**  $\Theta(n)$ .



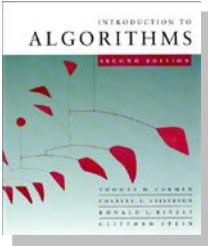
# Powering a number

**Problem:** Compute  $a^n$ , where  $n \in \mathbb{N}$ .

**Naive algorithm:**  $\Theta(n)$ .

**Divide-and-conquer algorithm:**

$$a^n = \begin{cases} a^{n/2} \cdot a^{n/2} & \text{if } n \text{ is even;} \\ a^{(n-1)/2} \cdot a^{(n-1)/2} \cdot a & \text{if } n \text{ is odd.} \end{cases}$$



# Powering a number

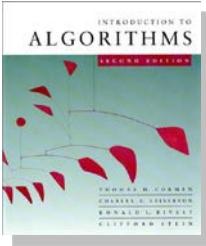
**Problem:** Compute  $a^n$ , where  $n \in \mathbb{N}$ .

**Naive algorithm:**  $\Theta(n)$ .

**Divide-and-conquer algorithm:**

$$a^n = \begin{cases} a^{n/2} \cdot a^{n/2} & \text{if } n \text{ is even;} \\ a^{(n-1)/2} \cdot a^{(n-1)/2} \cdot a & \text{if } n \text{ is odd.} \end{cases}$$

$$T(n) = T(n/2) + \Theta(1) \Rightarrow T(n) = \Theta(\lg n) .$$

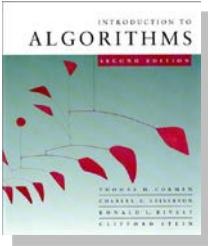


# Fibonacci numbers

Recursive definition:

$$F_n = \begin{cases} 0 & \text{if } n = 0; \\ 1 & \text{if } n = 1; \\ F_{n-1} + F_{n-2} & \text{if } n \geq 2. \end{cases}$$

0    1    1    2    3    5    8    13    21    34    ...



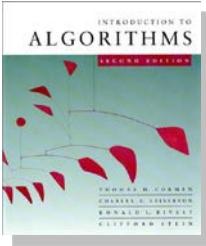
# Fibonacci numbers

Recursive definition:

$$F_n = \begin{cases} 0 & \text{if } n = 0; \\ 1 & \text{if } n = 1; \\ F_{n-1} + F_{n-2} & \text{if } n \geq 2. \end{cases}$$

0    1    1    2    3    5    8    13    21    34    ...

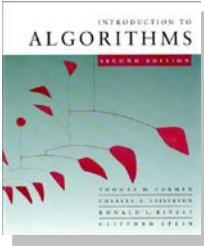
**Naive recursive algorithm:**  $\Omega(\phi^n)$   
(exponential time), where  $\phi = (1 + \sqrt{5})/2$   
is the *golden ratio*.



# Computing Fibonacci numbers

## Bottom-up:

- Compute  $F_0, F_1, F_2, \dots, F_n$  in order, forming each number by summing the two previous.
- Running time:  $\Theta(n)$ .



# Computing Fibonacci numbers

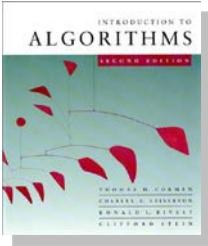
## Bottom-up:

- Compute  $F_0, F_1, F_2, \dots, F_n$  in order, forming each number by summing the two previous.
- Running time:  $\Theta(n)$ .

## Naive recursive squaring:

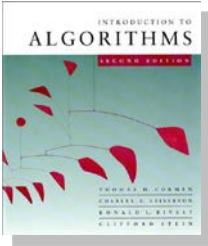
$$F_n = \phi^n / \sqrt{5} \text{ rounded to the nearest integer.}$$

- Recursive squaring:  $\Theta(\lg n)$  time.
- This method is unreliable, since floating-point arithmetic is prone to round-off errors.



# Recursive squaring

**Theorem:**  $\begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n.$

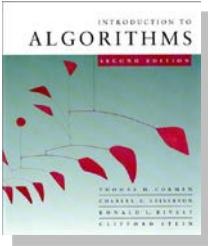


# Recursive squaring

**Theorem:** 
$$\begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n.$$

**Algorithm:** Recursive squaring.

Time =  $\Theta(\lg n)$ .



# Recursive squaring

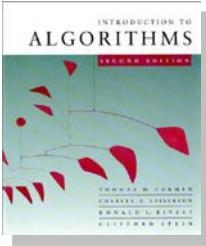
**Theorem:**  $\begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n.$

**Algorithm:** Recursive squaring.

Time =  $\Theta(\lg n)$ .

*Proof of theorem.* (Induction on  $n$ .)

Base ( $n = 1$ ):  $\begin{bmatrix} F_2 & F_1 \\ F_1 & F_0 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^1.$

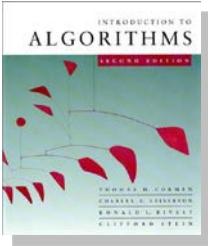


# Recursive squaring

Inductive step ( $n \geq 2$ ):

$$\begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} = \begin{bmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$
$$= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} \cdot \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$
$$= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n$$

■

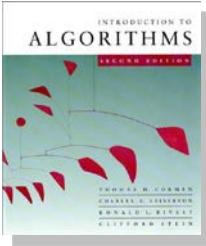


# Matrix multiplication

**Input:**  $A = [a_{ij}], B = [b_{ij}]$ . }  
**Output:**  $C = [c_{ij}] = A \cdot B.$  }  $i, j = 1, 2, \dots, n.$

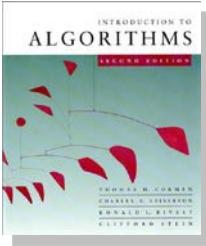
$$\begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \cdot \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{bmatrix}$$

$$c_{ij} = \sum_{k=1}^n a_{ik} \cdot b_{kj}$$



# Standard algorithm

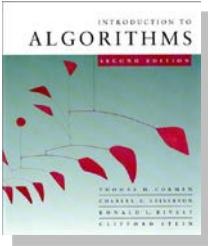
```
for  $i \leftarrow 1$  to  $n$ 
    do for  $j \leftarrow 1$  to  $n$ 
        do  $c_{ij} \leftarrow 0$ 
            for  $k \leftarrow 1$  to  $n$ 
                do  $c_{ij} \leftarrow c_{ij} + a_{ik} \cdot b_{kj}$ 
```



# Standard algorithm

```
for  $i \leftarrow 1$  to  $n$ 
    do for  $j \leftarrow 1$  to  $n$ 
        do  $c_{ij} \leftarrow 0$ 
            for  $k \leftarrow 1$  to  $n$ 
                do  $c_{ij} \leftarrow c_{ij} + a_{ik} \cdot b_{kj}$ 
```

Running time =  $\Theta(n^3)$



# Divide-and-conquer algorithm

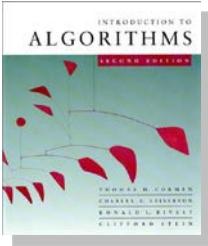
**IDEA:**

$n \times n$  matrix =  $2 \times 2$  matrix of  $(n/2) \times (n/2)$  submatrices:

$$\begin{bmatrix} r & s \\ t & u \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} e & f \\ g & h \end{bmatrix}$$

$$C = A \cdot B$$

$$\left. \begin{array}{l} r = ae + bg \\ s = af + bh \\ t = ce + dg \\ u = cf + dh \end{array} \right\} \begin{array}{l} 8 \text{ mults of } (n/2) \times (n/2) \text{ submatrices} \\ 4 \text{ adds of } (n/2) \times (n/2) \text{ submatrices} \end{array}$$



# Divide-and-conquer algorithm

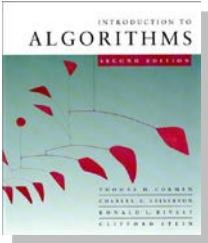
**IDEA:**

$n \times n$  matrix =  $2 \times 2$  matrix of  $(n/2) \times (n/2)$  submatrices:

$$\begin{bmatrix} r & s \\ t & u \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} e & f \\ g & h \end{bmatrix}$$

$$C = A \cdot B$$

$$\left. \begin{array}{l} r = ae + bg \\ s = af + bh \\ t = ce + dh \\ u = cf + dg \end{array} \right\} \begin{array}{l} \text{recursive} \\ 8 \text{ mults of } (n/2) \times (n/2) \text{ submatrices} \\ 4 \text{ adds of } (n/2) \times (n/2) \text{ submatrices} \end{array}$$



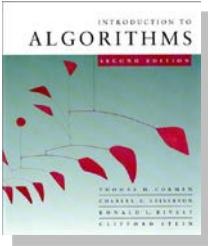
# Analysis of D&C algorithm

$$T(n) = 8T(n/2) + \Theta(n^2)$$

# submatrices      ↗  
submatrix size

work adding  
submatrices ↙

The equation  $T(n) = 8T(n/2) + \Theta(n^2)$  is displayed in teal. Three arrows point from labels to specific parts of the equation: one arrow from "# submatrices" points to the first  $T(n/2)$ ; another arrow from "submatrix size" points to the  $n^2$  term; and a third arrow from "work adding submatrices" points to the  $\Theta(n^2)$  term.



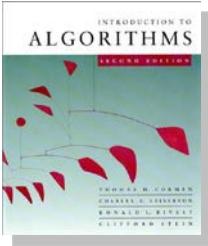
# Analysis of D&C algorithm

$$T(n) = 8T(n/2) + \Theta(n^2)$$

# submatrices      ↗  
                        ↓  
                        submatrix size  
                        ↗  
                        work adding  
                        submatrices

The equation  $T(n) = 8T(n/2) + \Theta(n^2)$  is displayed in teal. Three arrows point from descriptive text below to specific parts of the equation: one arrow points to the term  $8T(n/2)$  from the text "# submatrices"; another arrow points to the term  $\Theta(n^2)$  from the text "submatrix size"; a third arrow points to the plus sign from the text "work adding submatrices".

$$n^{\log_b a} = n^{\log_2 8} = n^3 \Rightarrow \text{CASE 1} \Rightarrow T(n) = \Theta(n^3).$$



# Analysis of D&C algorithm

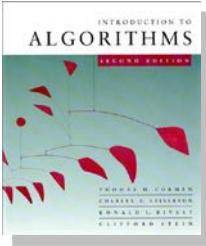
$$T(n) = 8T(n/2) + \Theta(n^2)$$

# submatrices      ↗  
                        ↓  
                        submatrix size  
                        ↗  
                        work adding  
                        submatrices

The equation  $T(n) = 8T(n/2) + \Theta(n^2)$  is displayed in teal. Three arrows point from descriptive text below to specific parts of the equation: one arrow from "# submatrices" points to the first  $T(n/2)$ ; another arrow from "submatrix size" points to the  $n^2$ ; and a third arrow from "work adding submatrices" points to the second  $T(n/2)$ .

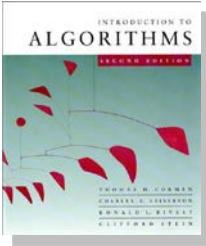
$$n^{\log_b a} = n^{\log_2 8} = n^3 \Rightarrow \text{CASE 1} \Rightarrow T(n) = \Theta(n^3).$$

*No better than the ordinary algorithm.*



# Strassen's idea

- Multiply  $2 \times 2$  matrices with only 7 recursive mults.



# Strassen's idea

- Multiply  $2 \times 2$  matrices with only 7 recursive mults.

$$P_1 = a \cdot (f - h)$$

$$P_2 = (a + b) \cdot h$$

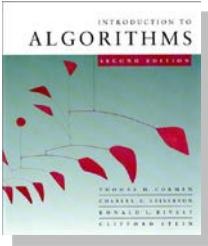
$$P_3 = (c + d) \cdot e$$

$$P_4 = d \cdot (g - e)$$

$$P_5 = (a + d) \cdot (e + h)$$

$$P_6 = (b - d) \cdot (g + h)$$

$$P_7 = (a - c) \cdot (e + f)$$



# Strassen's idea

- Multiply  $2 \times 2$  matrices with only 7 recursive mults.

$$P_1 = a \cdot (f - h)$$

$$P_2 = (a + b) \cdot h$$

$$P_3 = (c + d) \cdot e$$

$$P_4 = d \cdot (g - e)$$

$$P_5 = (a + d) \cdot (e + h)$$

$$P_6 = (b - d) \cdot (g + h)$$

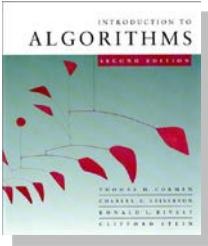
$$P_7 = (a - c) \cdot (e + f)$$

$$r = P_5 + P_4 - P_2 + P_6$$

$$s = P_1 + P_2$$

$$t = P_3 + P_4$$

$$u = P_5 + P_1 - P_3 - P_7$$



# Strassen's idea

- Multiply  $2 \times 2$  matrices with only 7 recursive mults.

$$P_1 = a \cdot (f - h)$$

$$P_2 = (a + b) \cdot h$$

$$P_3 = (c + d) \cdot e$$

$$P_4 = d \cdot (g - e)$$

$$P_5 = (a + d) \cdot (e + h)$$

$$P_6 = (b - d) \cdot (g + h)$$

$$P_7 = (a - c) \cdot (e + f)$$

$$r = P_5 + P_4 - P_2 + P_6$$

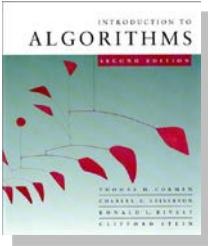
$$s = P_1 + P_2$$

$$t = P_3 + P_4$$

$$u = P_5 + P_1 - P_3 - P_7$$

7 mults, 18 adds/subs.

**Note:** No reliance on commutativity of mult!



# Strassen's idea

- Multiply  $2 \times 2$  matrices with only 7 recursive mults.

$$P_1 = a \cdot (f - h)$$

$$P_2 = (a + b) \cdot h$$

$$P_3 = (c + d) \cdot e$$

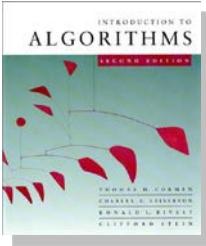
$$P_4 = d \cdot (g - e)$$

$$P_5 = (a + d) \cdot (e + h)$$

$$P_6 = (b - d) \cdot (g + h)$$

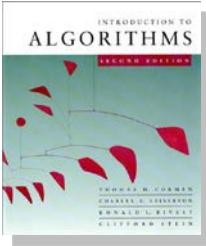
$$P_7 = (a - c) \cdot (e + f)$$

$$\begin{aligned} r &= P_5 + P_4 - P_2 + P_6 \\ &= (a + d)(e + h) \\ &\quad + d(g - e) - (a + b)h \\ &\quad + (b - d)(g + h) \\ &= ae + ah + de + dh \\ &\quad + dg - de - ah - bh \\ &\quad + bg + bh - dg - dh \\ &= ae + bg \end{aligned}$$



# Strassen's algorithm

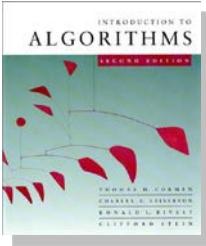
1. ***Divide:*** Partition  $A$  and  $B$  into  $(n/2) \times (n/2)$  submatrices. Form terms to be multiplied using  $+$  and  $-$ .
2. ***Conquer:*** Perform 7 multiplications of  $(n/2) \times (n/2)$  submatrices recursively.
3. ***Combine:*** Form  $C$  using  $+$  and  $-$  on  $(n/2) \times (n/2)$  submatrices.



# Strassen's algorithm

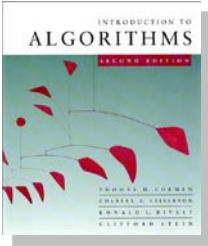
1. **Divide:** Partition  $A$  and  $B$  into  $(n/2) \times (n/2)$  submatrices. Form terms to be multiplied using  $+$  and  $-$ .
2. **Conquer:** Perform 7 multiplications of  $(n/2) \times (n/2)$  submatrices recursively.
3. **Combine:** Form  $C$  using  $+$  and  $-$  on  $(n/2) \times (n/2)$  submatrices.

$$T(n) = 7 T(n/2) + \Theta(n^2)$$



# Analysis of Strassen

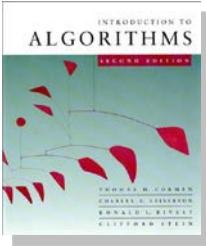
$$T(n) = 7 T(n/2) + \Theta(n^2)$$



# Analysis of Strassen

$$T(n) = 7 T(n/2) + \Theta(n^2)$$

$$n^{\log_b a} = n^{\log_2 7} \approx n^{2.81} \Rightarrow \text{CASE 1} \Rightarrow T(n) = \Theta(n^{\lg 7}).$$

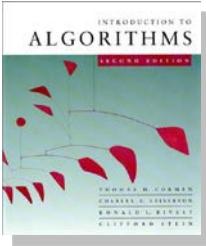


# Analysis of Strassen

$$T(n) = 7 T(n/2) + \Theta(n^2)$$

$$n^{\log_b a} = n^{\log_2 7} \approx n^{2.81} \Rightarrow \text{CASE 1} \Rightarrow T(n) = \Theta(n^{\lg 7}).$$

The number  $2.81$  may not seem much smaller than  $3$ , but because the difference is in the exponent, the impact on running time is significant. In fact, Strassen's algorithm beats the ordinary algorithm on today's machines for  $n \geq 32$  or so.



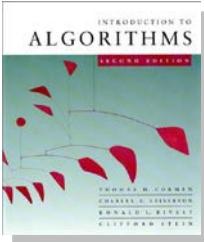
# Analysis of Strassen

$$T(n) = 7 T(n/2) + \Theta(n^2)$$

$$n^{\log_b a} = n^{\log_2 7} \approx n^{2.81} \Rightarrow \text{CASE 1} \Rightarrow T(n) = \Theta(n^{\lg 7}).$$

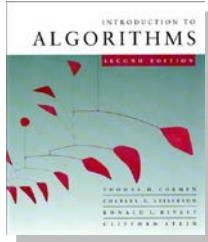
The number  $2.81$  may not seem much smaller than  $3$ , but because the difference is in the exponent, the impact on running time is significant. In fact, Strassen's algorithm beats the ordinary algorithm on today's machines for  $n \geq 32$  or so.

**Best to date** (of theoretical interest only):  $\Theta(n^{2.376\dots})$ .



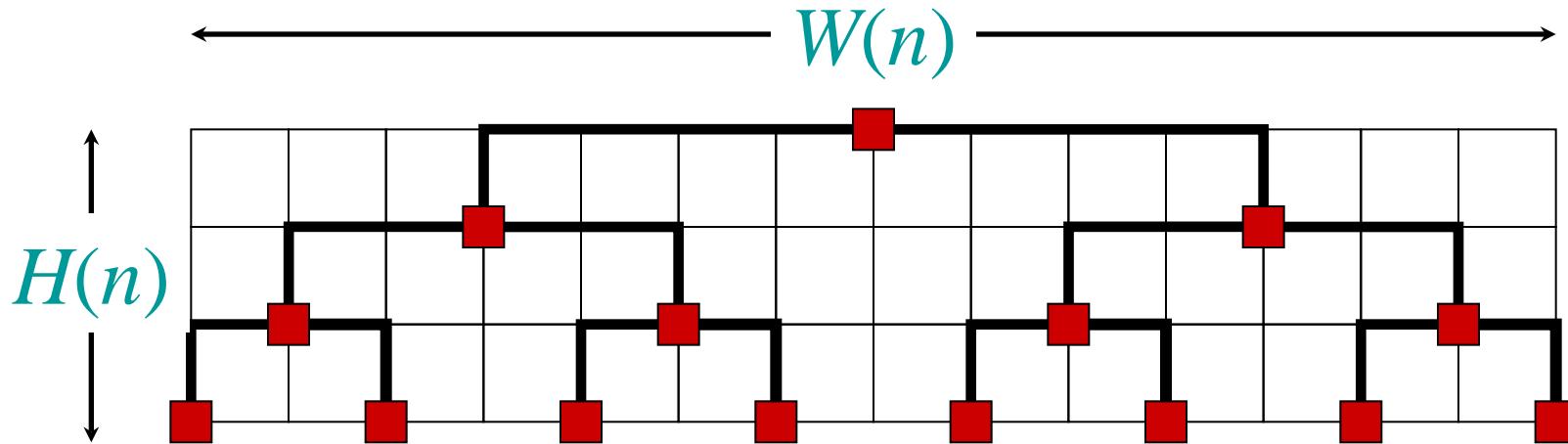
# VLSI layout

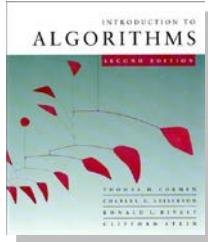
**Problem:** Embed a complete binary tree with  $n$  leaves in a grid using minimal area.



# VLSI layout

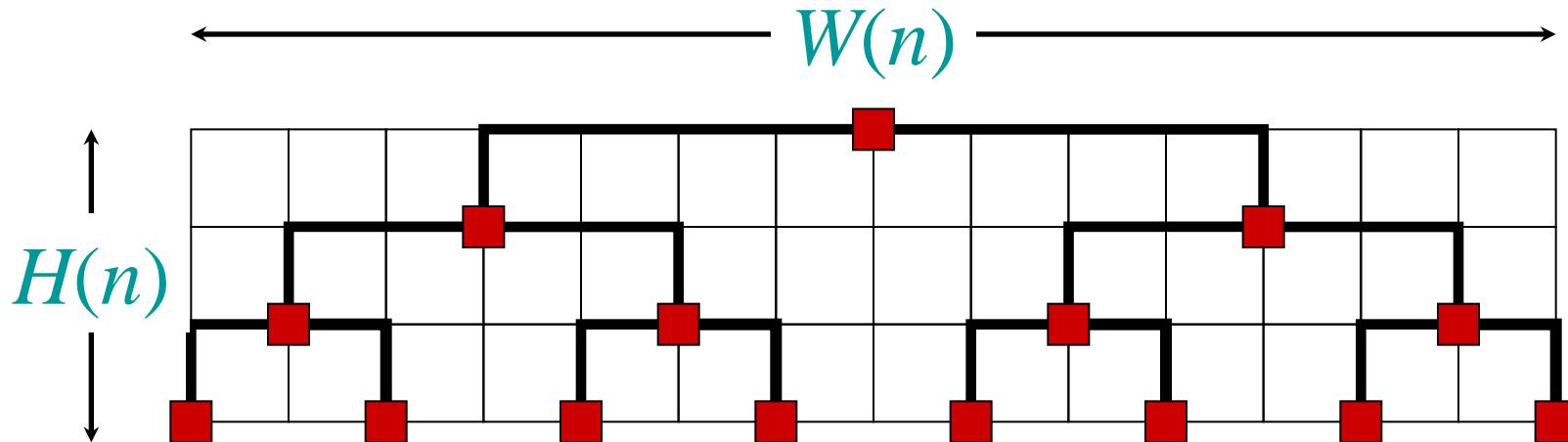
**Problem:** Embed a complete binary tree with  $n$  leaves in a grid using minimal area.



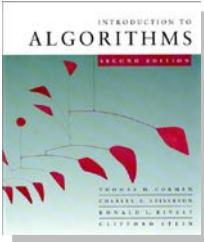


# VLSI layout

**Problem:** Embed a complete binary tree with  $n$  leaves in a grid using minimal area.

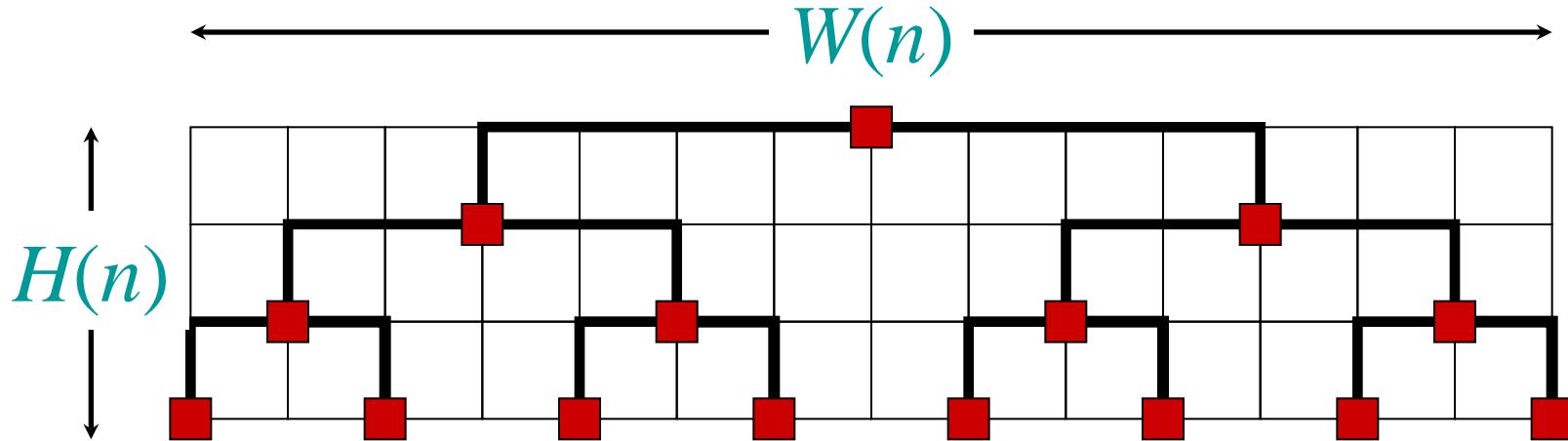


$$\begin{aligned} H(n) &= H(n/2) + \Theta(1) \\ &= \Theta(\lg n) \end{aligned}$$



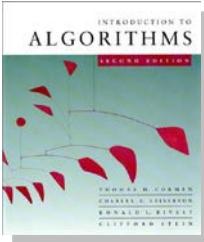
# VLSI layout

**Problem:** Embed a complete binary tree with  $n$  leaves in a grid using minimal area.



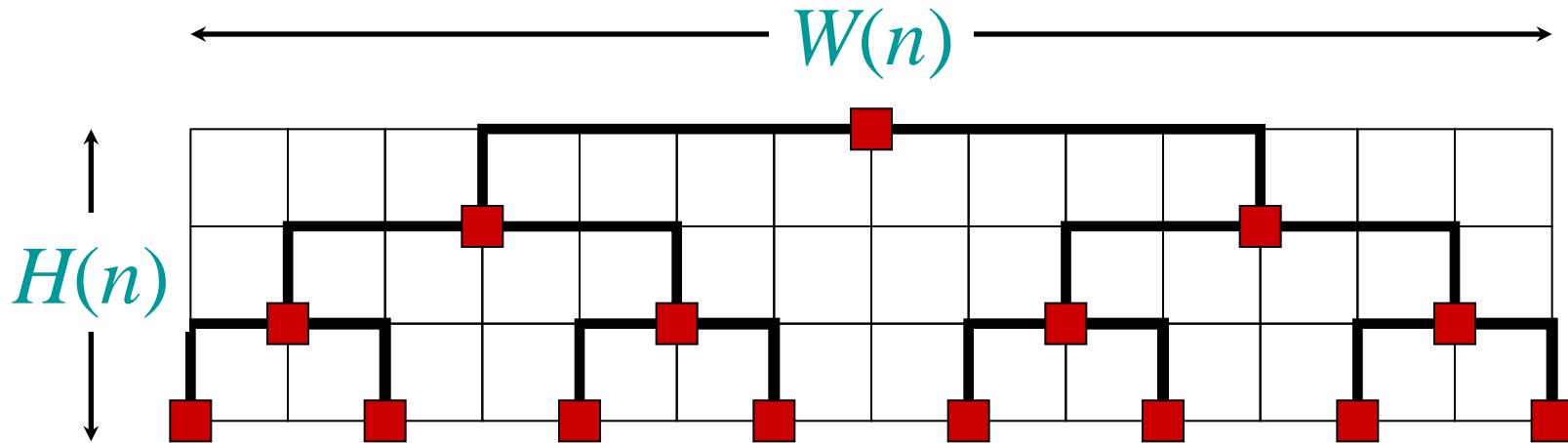
$$\begin{aligned} H(n) &= H(n/2) + \Theta(1) \\ &= \Theta(\lg n) \end{aligned}$$

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



# VLSI layout

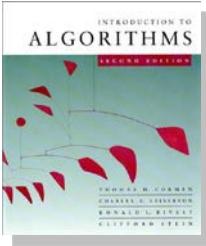
**Problem:** Embed a complete binary tree with  $n$  leaves in a grid using minimal area.



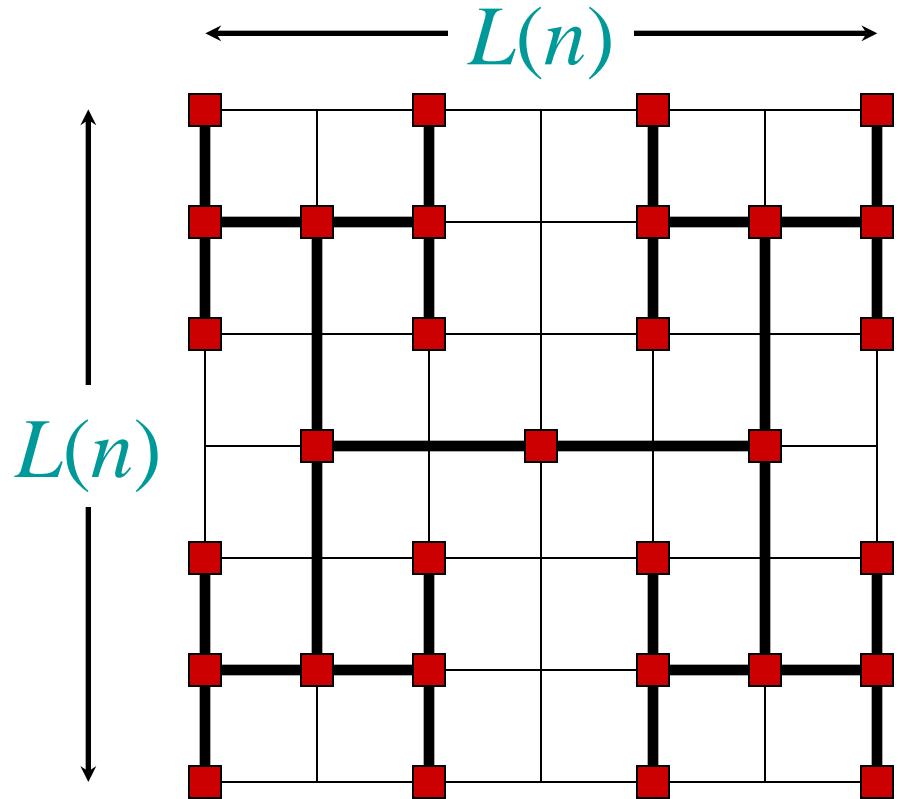
$$\begin{aligned} H(n) &= H(n/2) + \Theta(1) \\ &= \Theta(\lg n) \end{aligned}$$

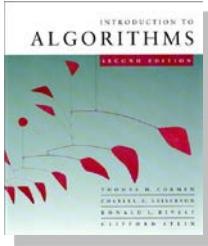
$$\begin{aligned} W(n) &= 2W(n/2) + \Theta(1) \\ &= \Theta(n) \end{aligned}$$

Area =  $\Theta(n \lg n)$

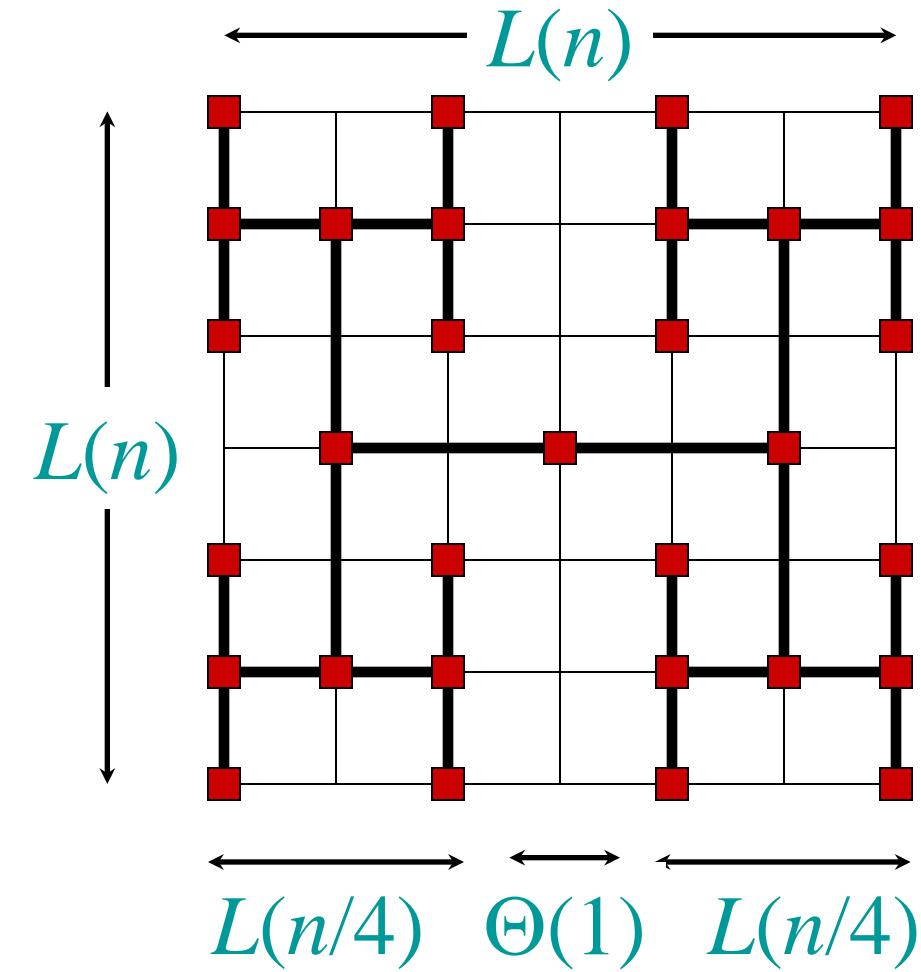


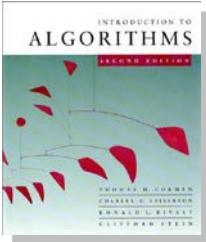
# H-tree embedding



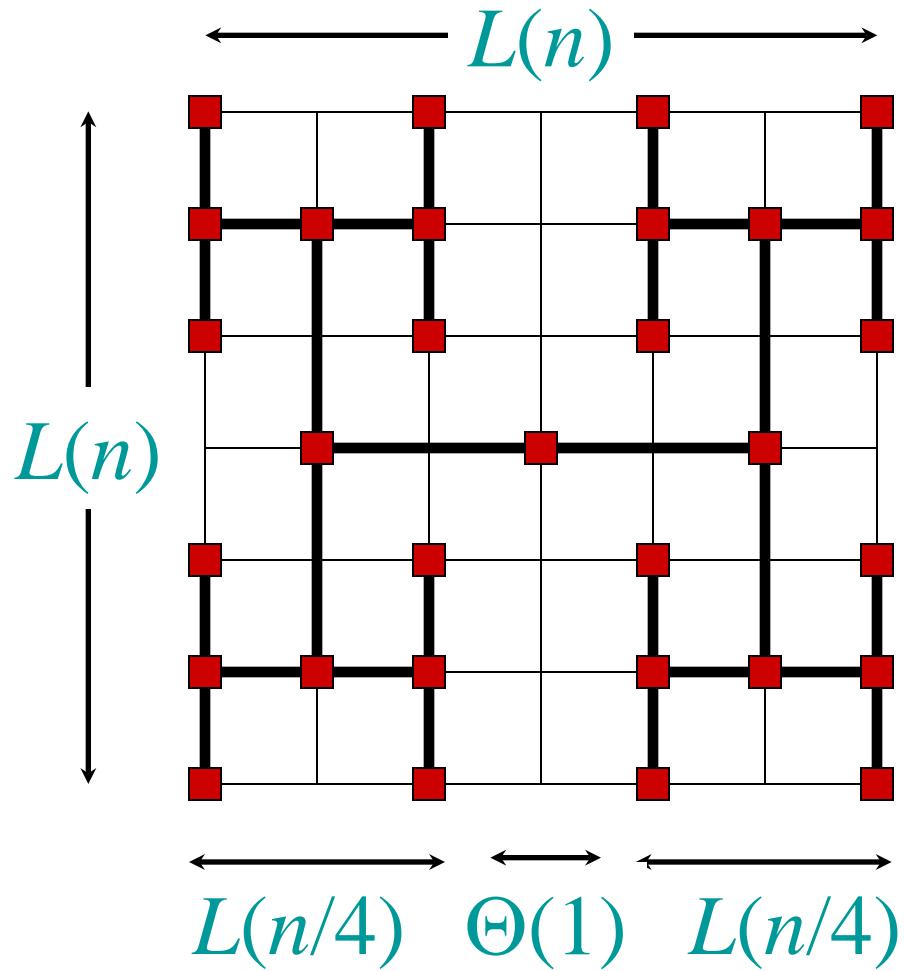


# H-tree embedding



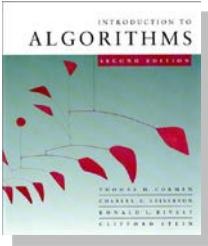


# H-tree embedding



$$\begin{aligned}L(n) &= 2L(n/4) + \Theta(1) \\&= \Theta(\sqrt{n})\end{aligned}$$

Area =  $\Theta(n)$

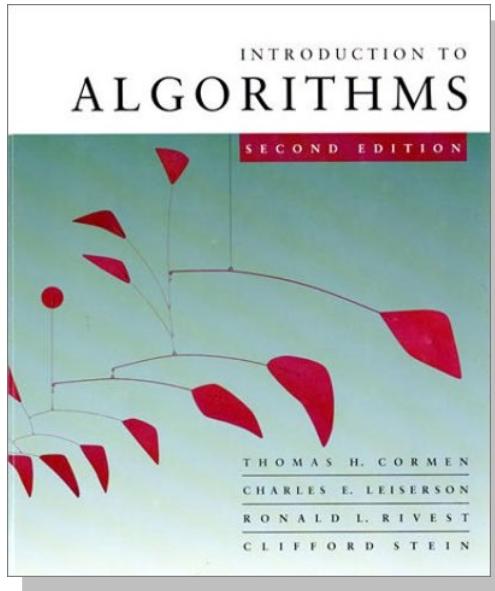


# Conclusion

- Divide and conquer is just one of several powerful techniques for algorithm design.
- Divide-and-conquer algorithms can be analyzed using recurrences and the master method (so practice this math).
- The divide-and-conquer strategy often leads to efficient algorithms.

# *Introduction to Algorithms*

**6.046J/18.401J**

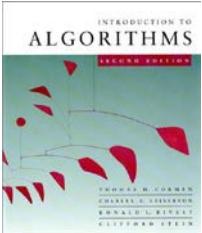


## **LECTURE 4**

### **Quicksort**

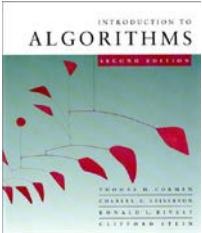
- Divide and conquer
- Partitioning
- Worst-case analysis
- Intuition
- Randomized quicksort
- Analysis

**Prof. Charles E. Leiserson**



# 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

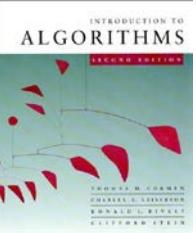
Quicksort an  $n$ -element array:

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( $A, p, q$ )  $\triangleright A[p \dots q]$

$x \leftarrow A[p]$   $\triangleright \text{pivot} = A[p]$

$i \leftarrow p$

**for**  $j \leftarrow p + 1$  **to**  $q$

**do if**  $A[j] \leq x$

**then**  $i \leftarrow i + 1$

exchange  $A[i] \leftrightarrow A[j]$

exchange  $A[p] \leftrightarrow A[i]$

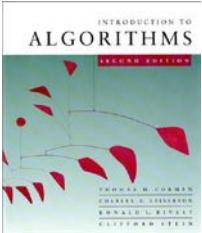
**return**  $i$

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

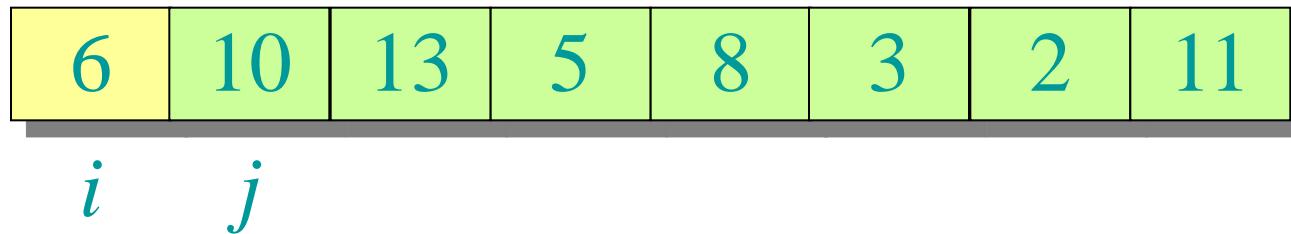
**Invariant:**

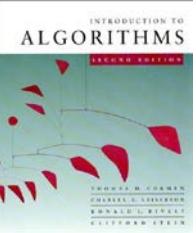


$p$   $i$   $j$   $q$

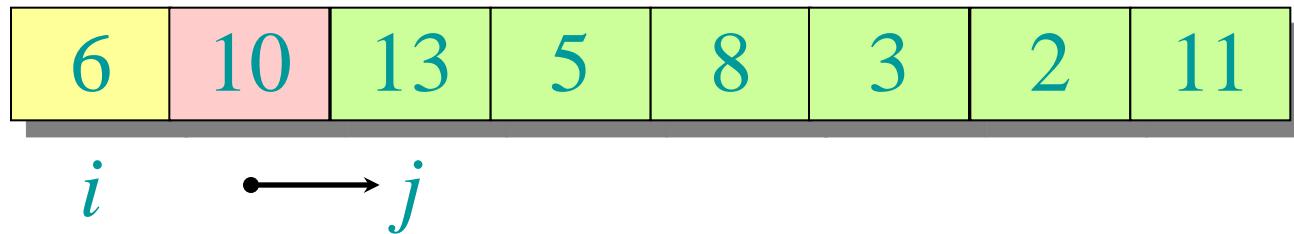


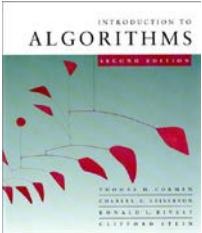
# Example of partitioning



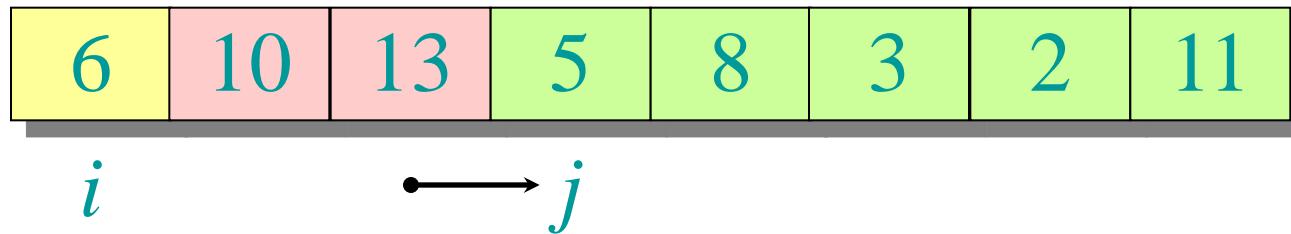


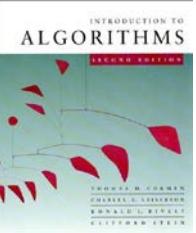
# Example of partitioning



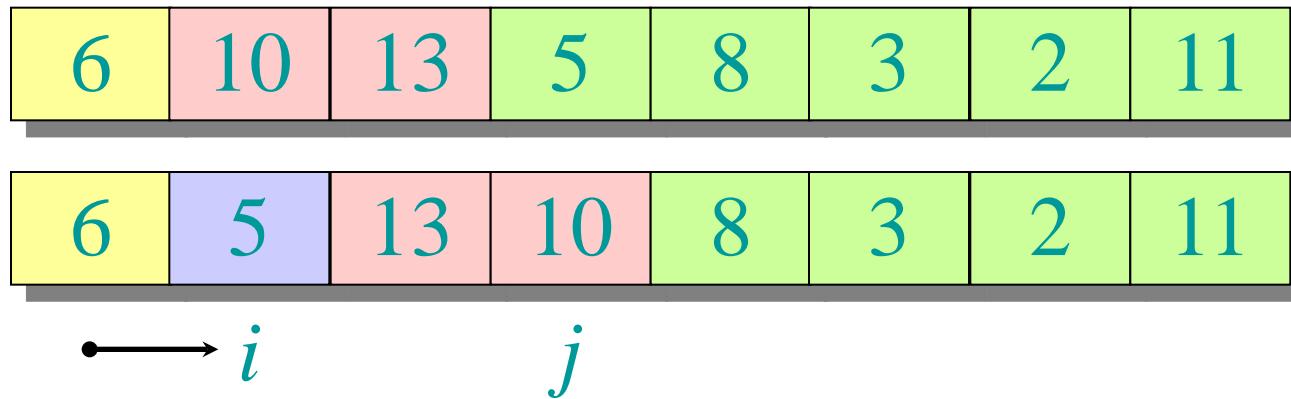


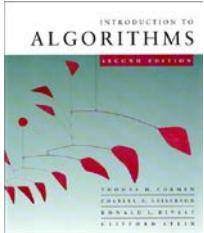
# Example of partitioning



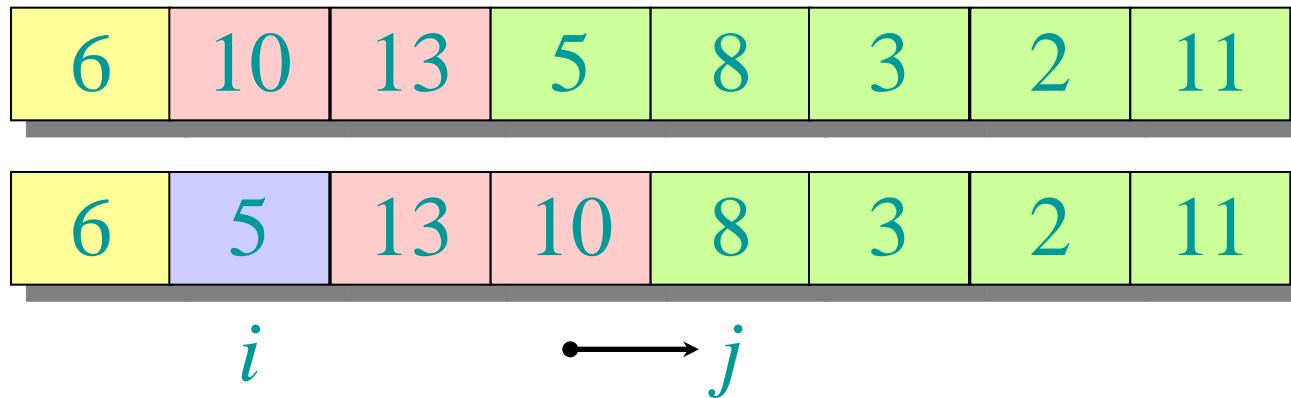


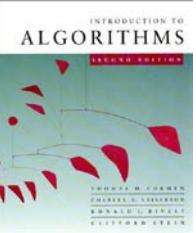
# Example of partitioning



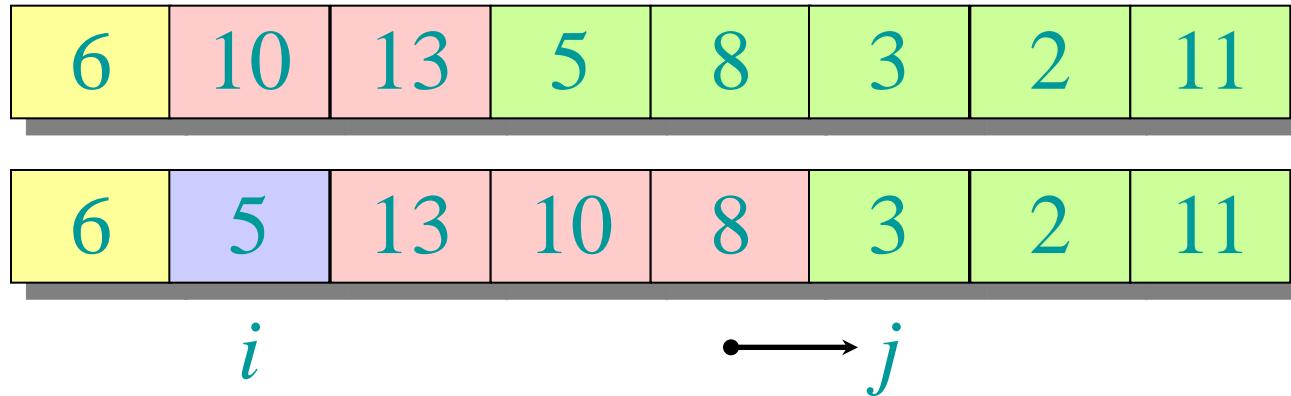


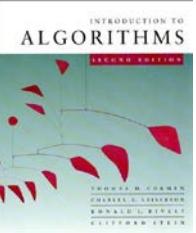
# Example of partitioning



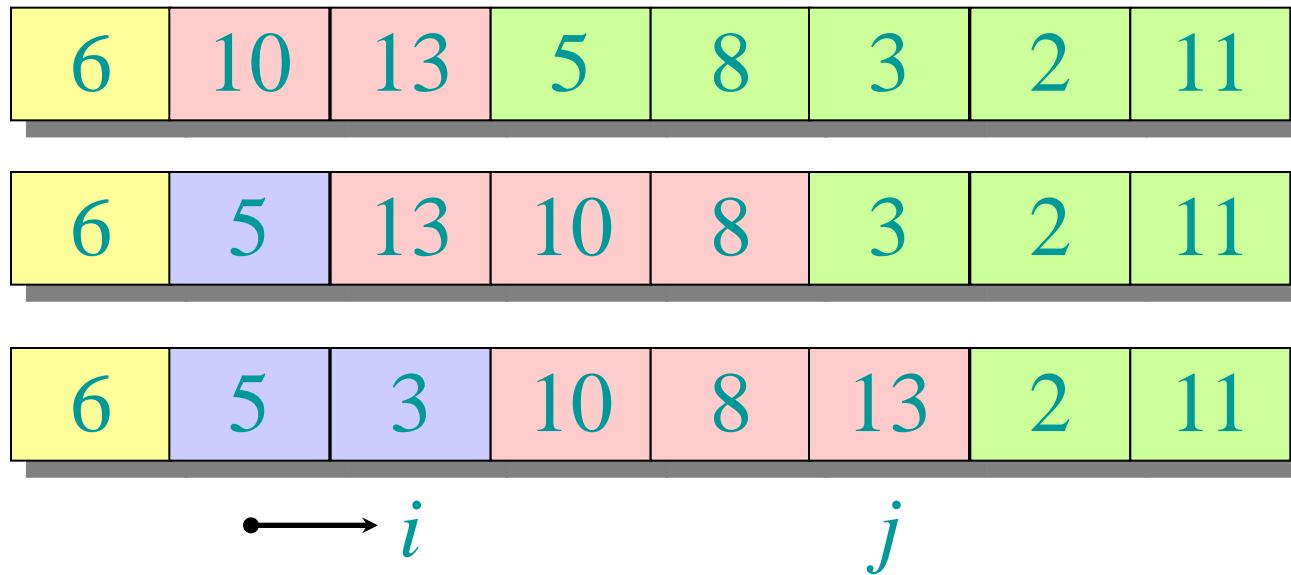


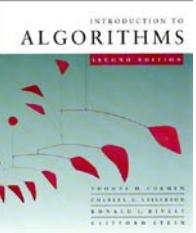
# Example of partitioning



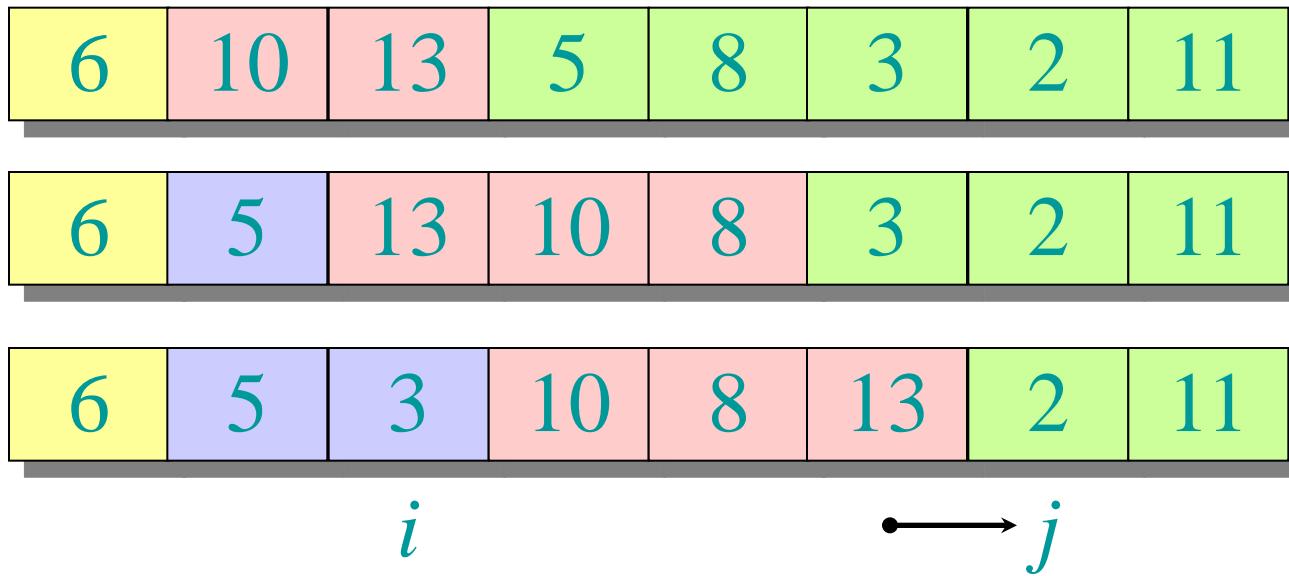


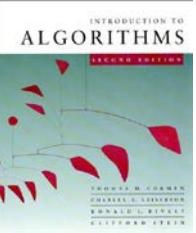
# Example of partitioning



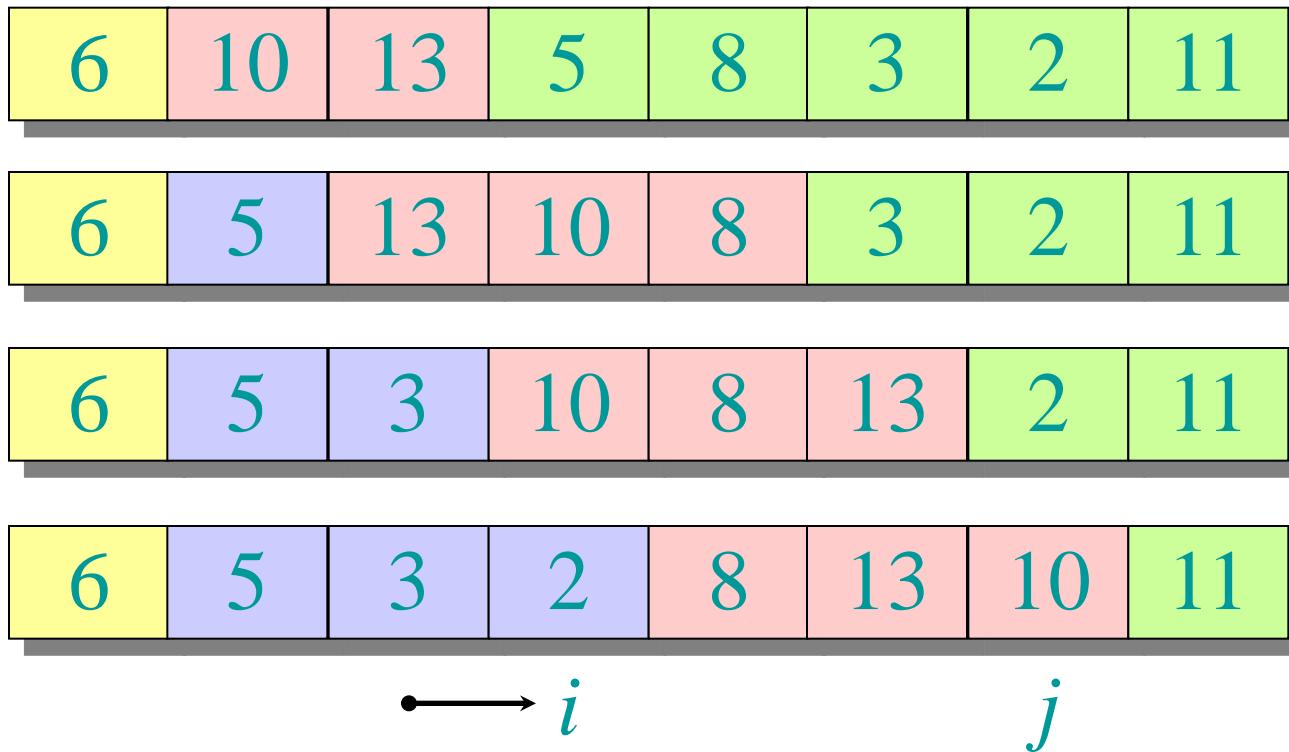


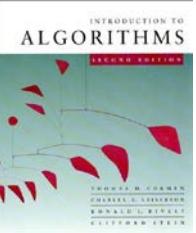
# Example of partitioning



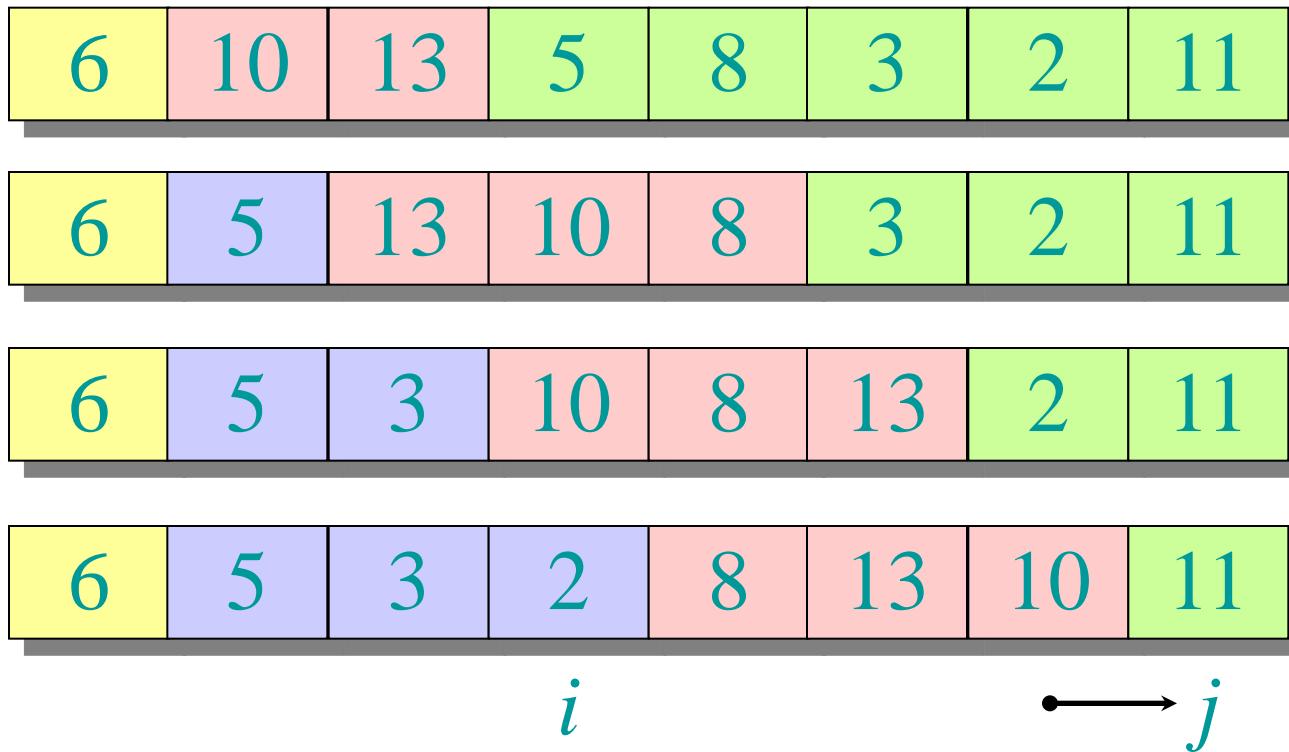


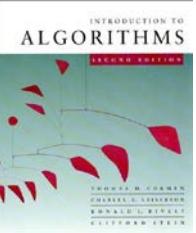
# Example of partitioning



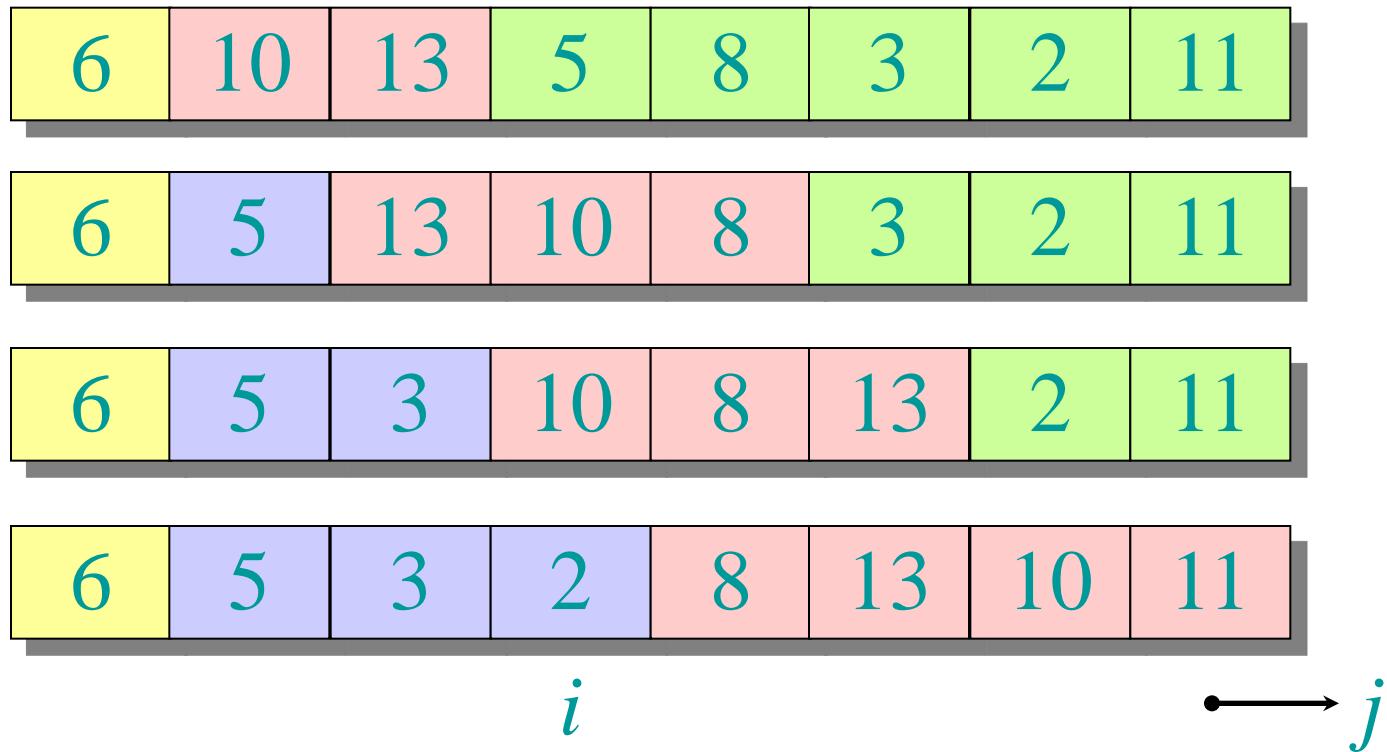


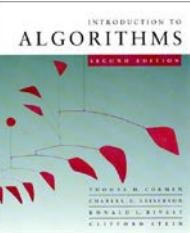
# Example of partitioning



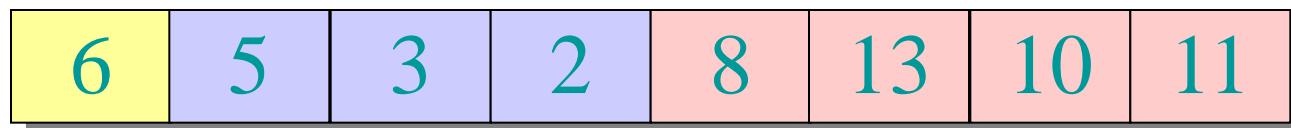
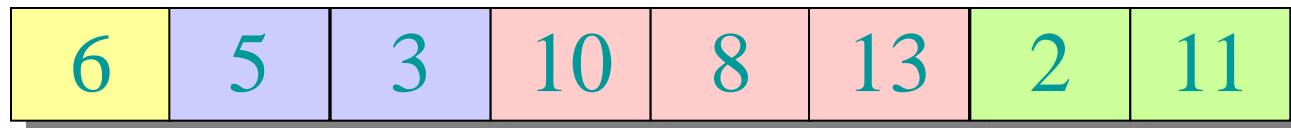
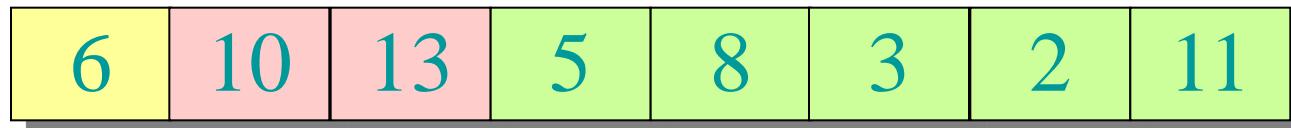


# Example of partitioning

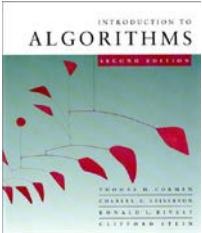




# Example of partitioning



$i$



# Pseudocode for quicksort

QUICKSORT( $A, p, r$ )

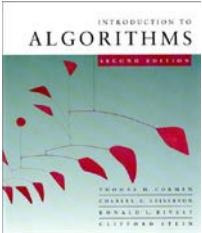
**if**  $p < r$

**then**  $q \leftarrow \text{PARTITION}(A, p, r)$

    QUICKSORT( $A, p, q-1$ )

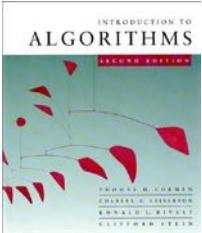
    QUICKSORT( $A, q+1, r$ )

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



# Analysis of quicksort

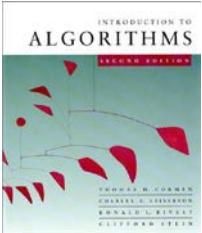
- Assume all input elements are distinct.
- In practice, there are better partitioning algorithms for when duplicate input elements may exist.
- Let  $T(n)$  = worst-case running time on an array of  $n$  elements.



# Worst-case of quicksort

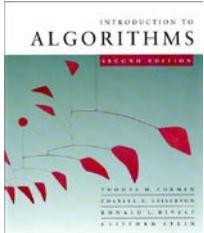
- Input sorted or reverse sorted.
- Partition around min or max element.
- One side of partition always has no elements.

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



# Worst-case recursion tree

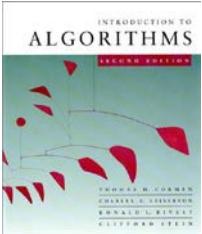
$$T(n) = T(0) + T(n-1) + cn$$



# Worst-case recursion tree

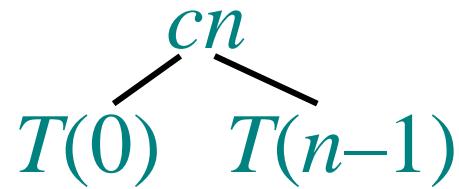
$$T(n) = T(0) + T(n-1) + cn$$

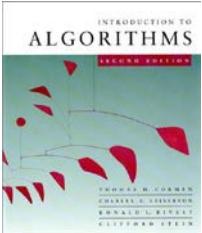
$$T(n)$$



# Worst-case recursion tree

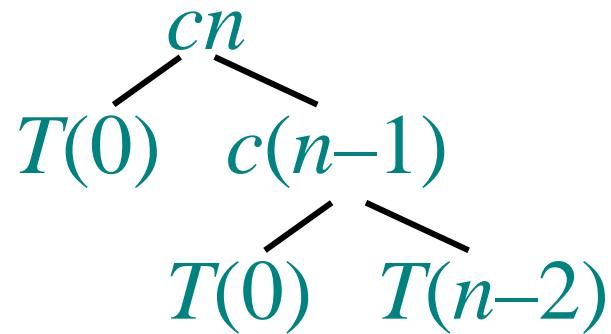
$$T(n) = T(0) + T(n-1) + cn$$

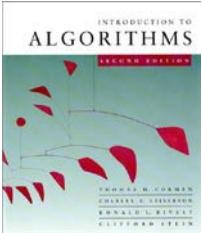




# Worst-case recursion tree

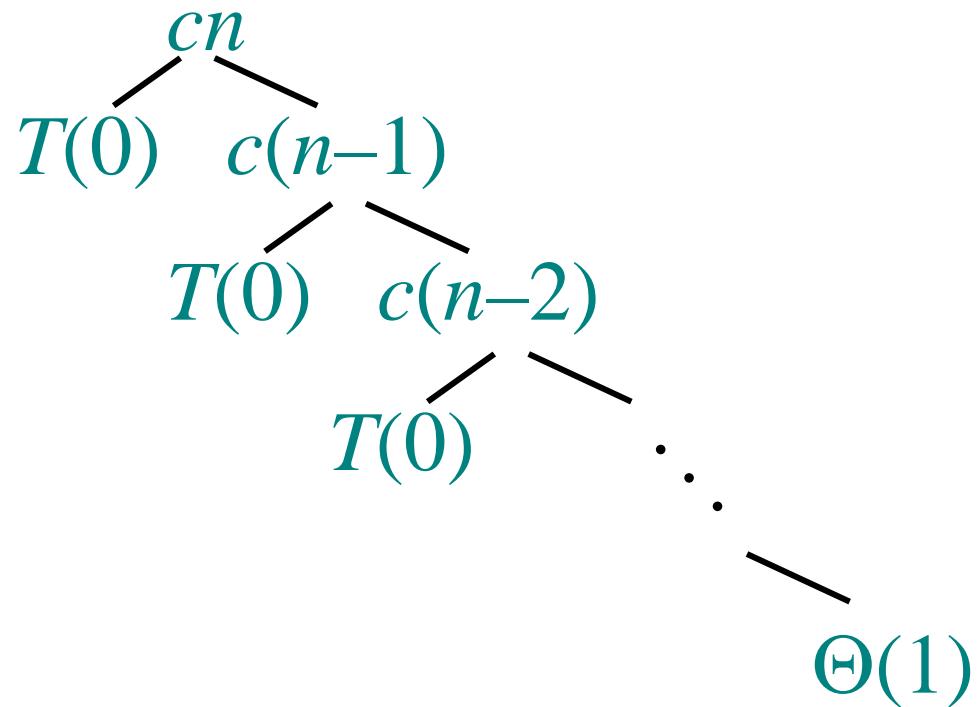
$$T(n) = T(0) + T(n-1) + cn$$

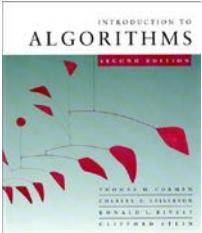




# Worst-case recursion tree

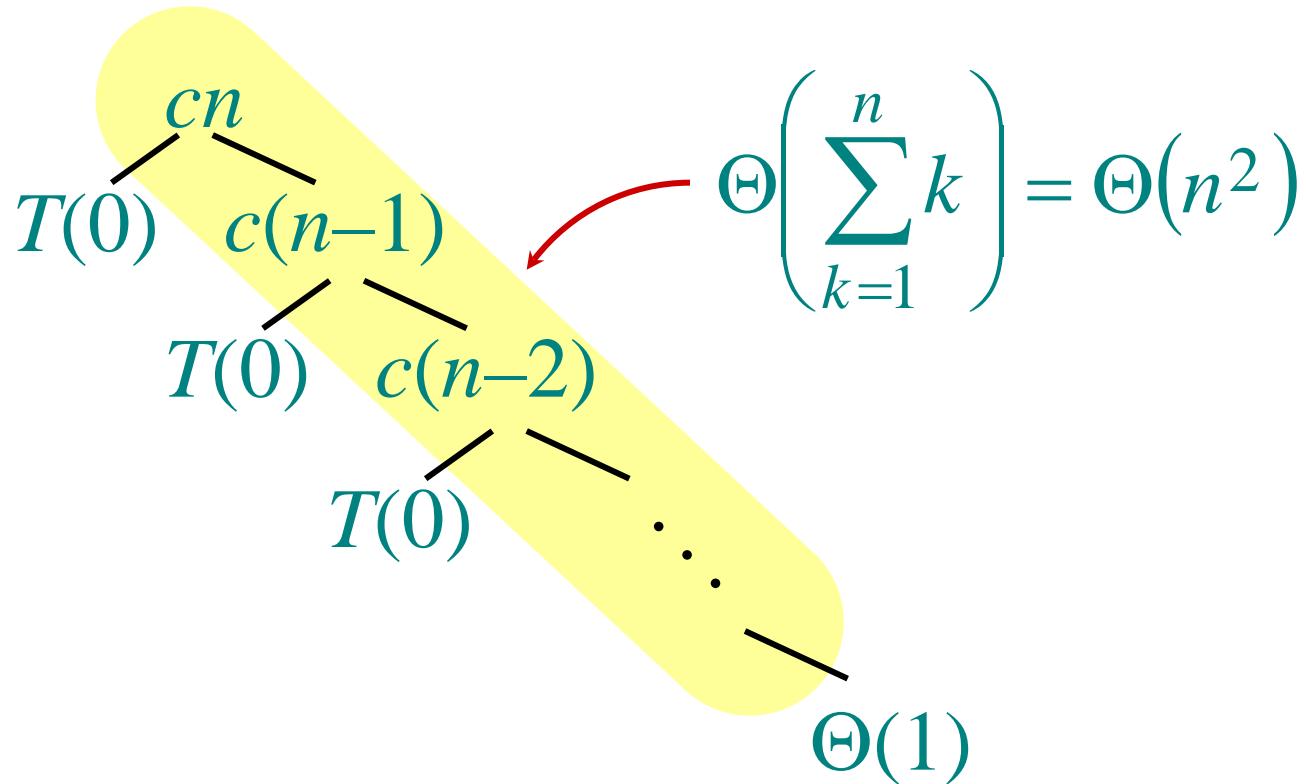
$$T(n) = T(0) + T(n-1) + cn$$

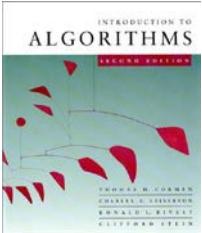




# Worst-case recursion tree

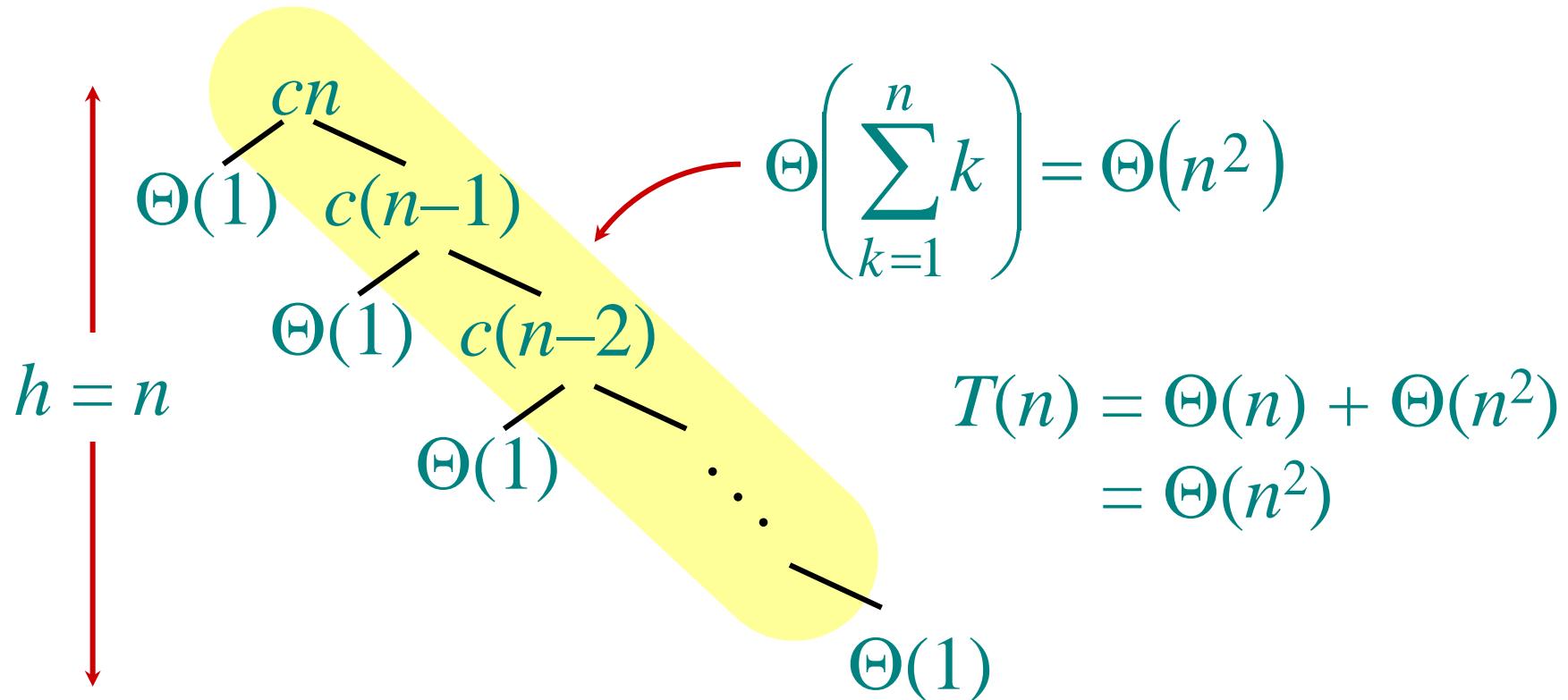
$$T(n) = T(0) + T(n-1) + cn$$

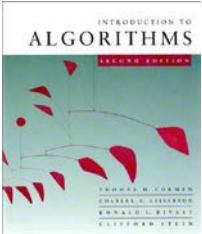




# Worst-case recursion tree

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





# Best-case analysis

*(For intuition only!)*

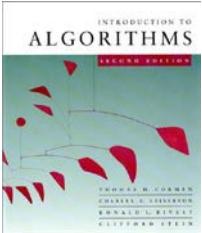
If we're lucky, PARTITION splits the array evenly:

$$\begin{aligned} T(n) &= 2T(n/2) + \Theta(n) \\ &= \Theta(n \lg n) \quad (\text{same as merge sort}) \end{aligned}$$

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

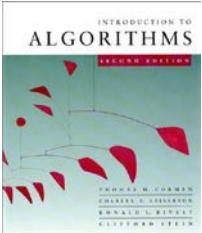
$$T(n) = T\left(\frac{1}{10}n\right) + T\left(\frac{9}{10}n\right) + \Theta(n)$$

What is the solution to this recurrence?



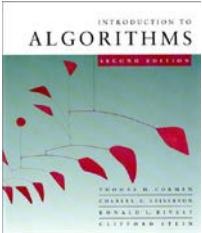
# Analysis of “almost-best” case

$$T(n)$$

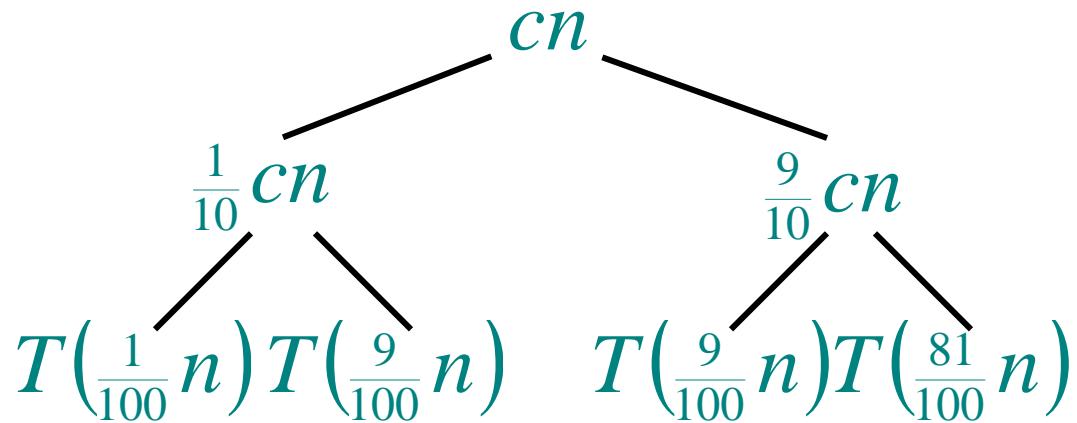


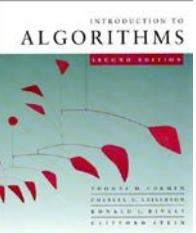
# Analysis of “almost-best” case

$$\begin{array}{ccc} & cn & \\ T\left(\frac{1}{10}n\right) & & T\left(\frac{9}{10}n\right) \end{array}$$

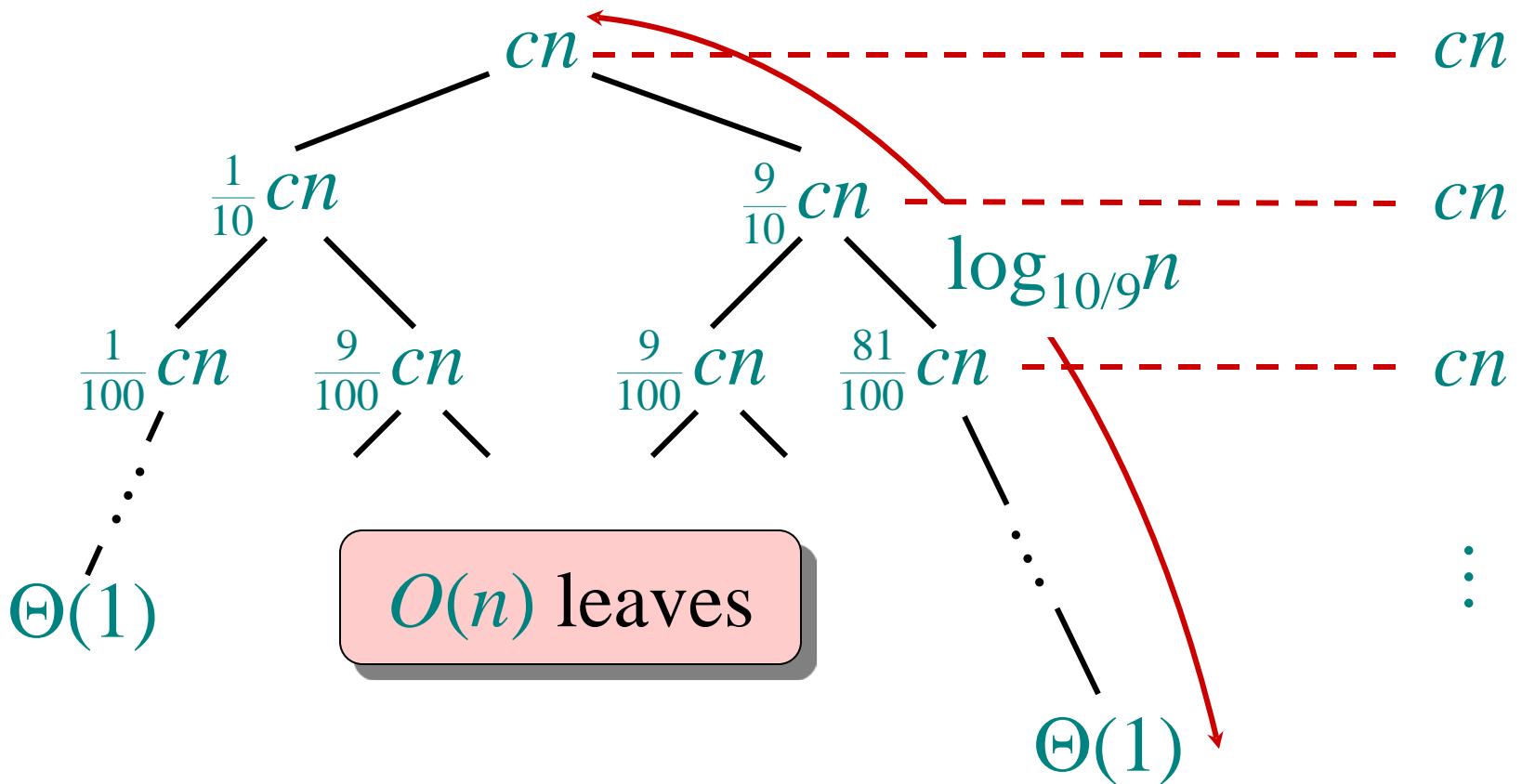


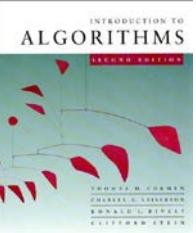
# Analysis of “almost-best” case



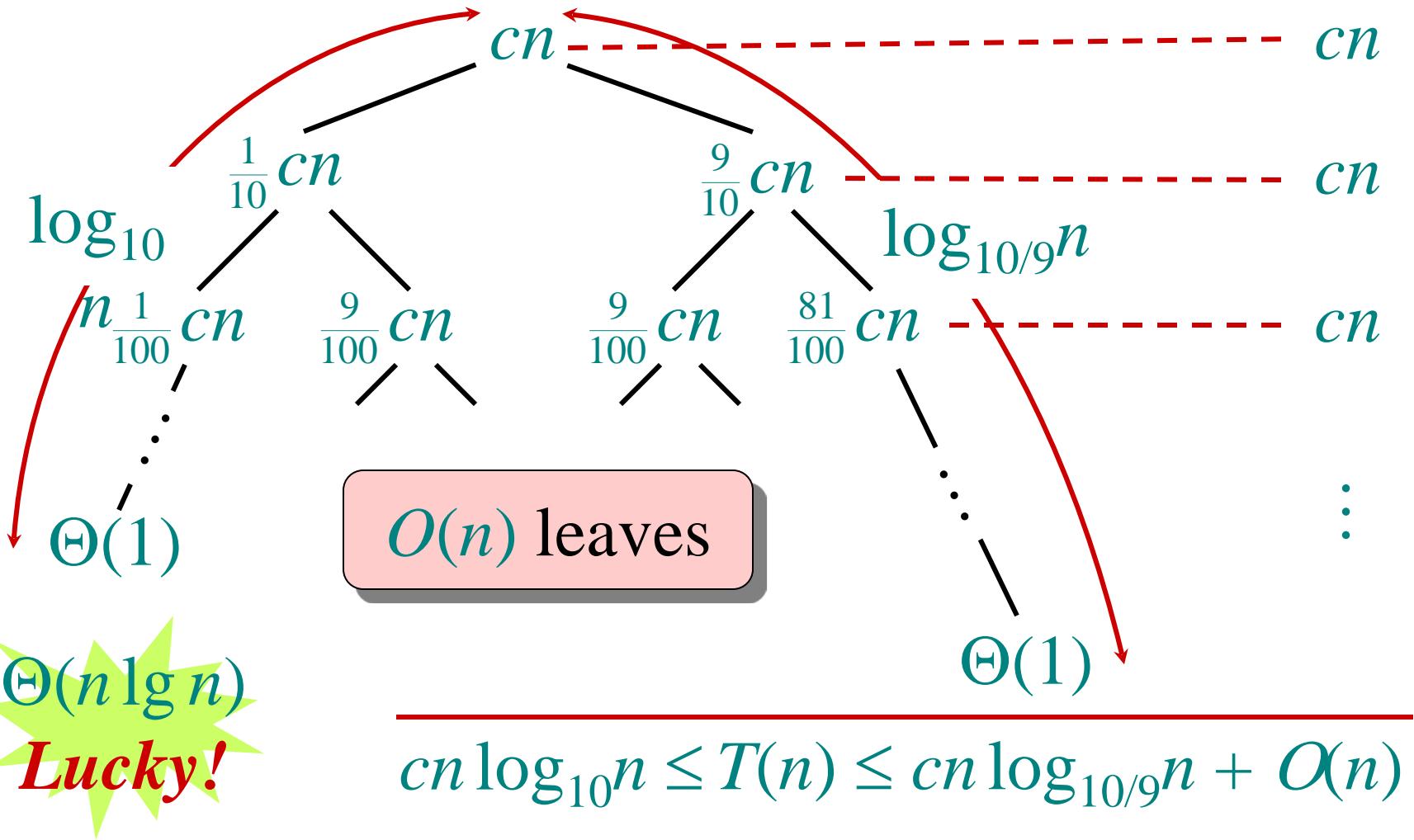


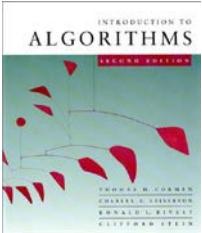
# Analysis of “almost-best” case





# Analysis of “almost-best” case





# More intuition

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

$$L(n) = 2U(n/2) + \Theta(n) \quad \textcolor{red}{lucky}$$

$$U(n) = L(n - 1) + \Theta(n) \quad \textcolor{red}{unlucky}$$

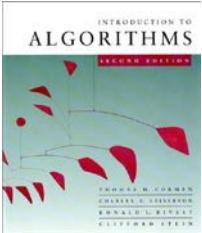
Solving:

$$L(n) = 2(L(n/2 - 1) + \Theta(n/2)) + \Theta(n)$$

$$= 2L(n/2 - 1) + \Theta(n)$$

$$= \Theta(n \lg n) \quad \textcolor{red}{Lucky!}$$

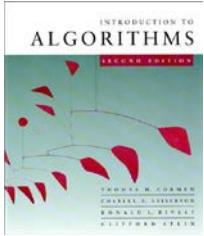
How can we make sure we are usually lucky?



# 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.



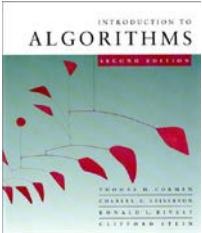
# Randomized quicksort analysis

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

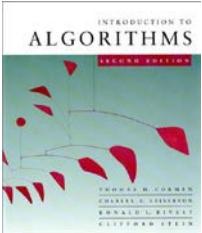
$$X_k = \begin{cases} 1 & \text{if PARTITION generates a } k : n-k-1 \text{ split,} \\ 0 & \text{otherwise.} \end{cases}$$

$E[X_k] = \Pr\{X_k = 1\} = 1/n$ , since all splits are equally likely, assuming elements are distinct.



# Analysis (continued)

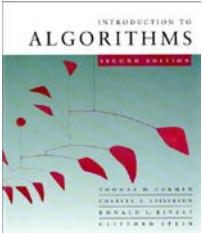
$$T(n) = \begin{cases} T(0) + T(n-1) + \Theta(n) & \text{if } 0:n-1 \text{ split,} \\ T(1) + T(n-2) + \Theta(n) & \text{if } 1:n-2 \text{ split,} \\ \vdots \\ T(n-1) + T(0) + \Theta(n) & \text{if } n-1:0 \text{ split,} \end{cases}$$
$$= \sum_{k=0}^{n-1} X_k (T(k) + T(n-k-1) + \Theta(n))$$



# Calculating expectation

$$E[T(n)] = E\left[ \sum_{k=0}^{n-1} X_k (T(k) + T(n-k-1) + \Theta(n)) \right]$$

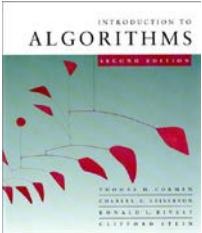
Take expectations of both sides.



# Calculating expectation

$$\begin{aligned}E[T(n)] &= E\left[\sum_{k=0}^{n-1} X_k (T(k) + T(n-k-1) + \Theta(n))\right] \\&= \sum_{k=0}^{n-1} E[X_k (T(k) + T(n-k-1) + \Theta(n))]\end{aligned}$$

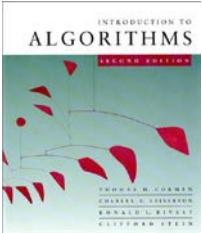
Linearity of expectation.



# Calculating expectation

$$\begin{aligned}E[T(n)] &= E\left[\sum_{k=0}^{n-1} X_k (T(k) + T(n-k-1) + \Theta(n))\right] \\&= \sum_{k=0}^{n-1} E[X_k (T(k) + T(n-k-1) + \Theta(n))] \\&= \sum_{k=0}^{n-1} E[X_k] \cdot E[T(k) + T(n-k-1) + \Theta(n)]\end{aligned}$$

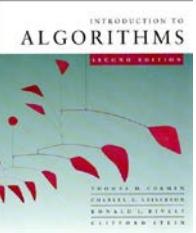
Independence of  $X_k$  from other random choices.



# Calculating expectation

$$\begin{aligned}E[T(n)] &= E\left[\sum_{k=0}^{n-1} X_k (T(k) + T(n-k-1) + \Theta(n))\right] \\&= \sum_{k=0}^{n-1} E[X_k (T(k) + T(n-k-1) + \Theta(n))] \\&= \sum_{k=0}^{n-1} E[X_k] \cdot E[T(k) + T(n-k-1) + \Theta(n)] \\&= \frac{1}{n} \sum_{k=0}^{n-1} E[T(k)] + \frac{1}{n} \sum_{k=0}^{n-1} E[T(n-k-1)] + \frac{1}{n} \sum_{k=0}^{n-1} \Theta(n)\end{aligned}$$

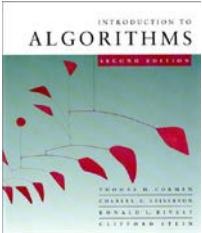
Linearity of expectation;  $E[X_k] = 1/n$ .



# Calculating expectation

$$\begin{aligned}E[T(n)] &= E\left[\sum_{k=0}^{n-1} X_k (T(k) + T(n-k-1) + \Theta(n))\right] \\&= \sum_{k=0}^{n-1} E[X_k (T(k) + T(n-k-1) + \Theta(n))] \\&= \sum_{k=0}^{n-1} E[X_k] \cdot E[T(k) + T(n-k-1) + \Theta(n)] \\&= \frac{1}{n} \sum_{k=0}^{n-1} E[T(k)] + \frac{1}{n} \sum_{k=0}^{n-1} E[T(n-k-1)] + \frac{1}{n} \sum_{k=0}^{n-1} \Theta(n) \\&= \frac{2}{n} \sum_{k=1}^{n-1} E[T(k)] + \Theta(n)\end{aligned}$$

Summations have identical terms.



# Hairy recurrence

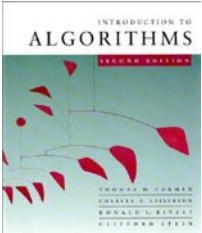
$$E[T(n)] = \frac{2}{n} \sum_{k=2}^{n-1} E[T(k)] + \Theta(n)$$

(The  $k = 0, 1$  terms can be absorbed in the  $\Theta(n)$ .)

**Prove:**  $E[T(n)] \leq an \lg n$  for constant  $a > 0$ .

- Choose  $a$  large enough so that  $an \lg n$  dominates  $E[T(n)]$  for sufficiently small  $n \geq 2$ .

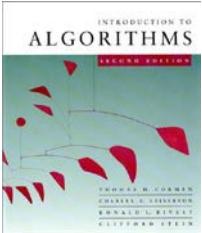
**Use fact:**  $\sum_{k=2}^{n-1} k \lg k \leq \frac{1}{2} n^2 \lg n - \frac{1}{8} n^2$  (exercise).



# Substitution method

$$E[T(n)] \leq \frac{2}{n} \sum_{k=2}^{n-1} ak \lg k + \Theta(n)$$

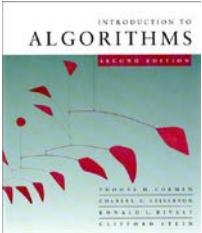
Substitute inductive hypothesis.



# Substitution method

$$\begin{aligned}E[T(n)] &\leq \frac{2}{n} \sum_{k=2}^{n-1} ak \lg k + \Theta(n) \\&\leq \frac{2a}{n} \left( \frac{1}{2} n^2 \lg n - \frac{1}{8} n^2 \right) + \Theta(n)\end{aligned}$$

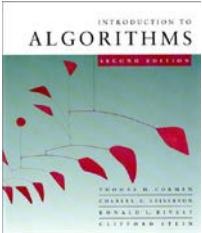
Use fact.



# Substitution method

$$\begin{aligned}E[T(n)] &\leq \frac{2}{n} \sum_{k=2}^{n-1} ak \lg k + \Theta(n) \\&\leq \frac{2a}{n} \left( \frac{1}{2} n^2 \lg n - \frac{1}{8} n^2 \right) + \Theta(n) \\&= an \lg n - \left( \frac{an}{4} - \Theta(n) \right)\end{aligned}$$

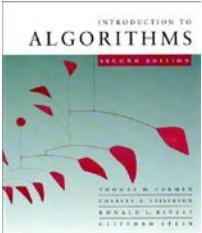
Express as *desired – residual*.



# Substitution method

$$\begin{aligned}E[T(n)] &\leq \frac{2}{n} \sum_{k=2}^{n-1} ak \lg k + \Theta(n) \\&= \frac{2a}{n} \left( \frac{1}{2} n^2 \lg n - \frac{1}{8} n^2 \right) + \Theta(n) \\&= an \lg n - \left( \frac{an}{4} - \Theta(n) \right) \\&\leq an \lg n,\end{aligned}$$

if  $a$  is chosen large enough so that  $an/4$  dominates the  $\Theta(n)$ .

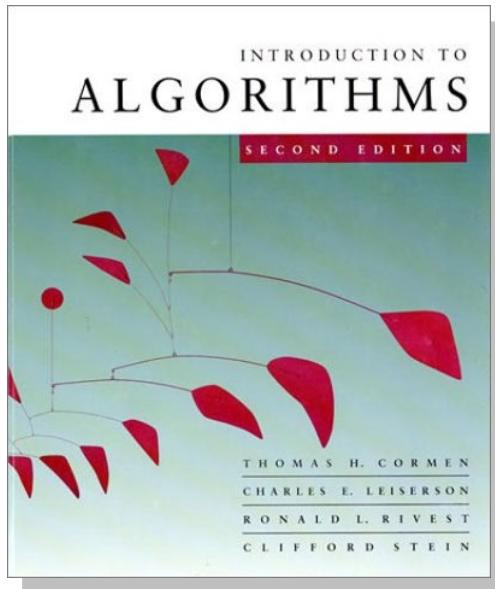


# Quicksort in practice

- 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.

# *Introduction to Algorithms*

**6.046J/18.401J**



## **LECTURE 5**

### **Sorting Lower Bounds**

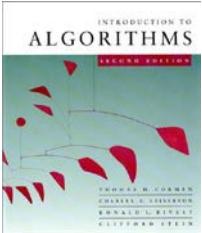
- Decision trees

### **Linear-Time Sorting**

- Counting sort
- Radix sort

### **Appendix: Punched cards**

**Prof. Erik Demaine**



# How fast can we sort?

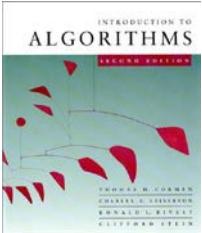
All the sorting algorithms we have seen so far are ***comparison sorts***: only use comparisons to determine the relative order of elements.

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

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

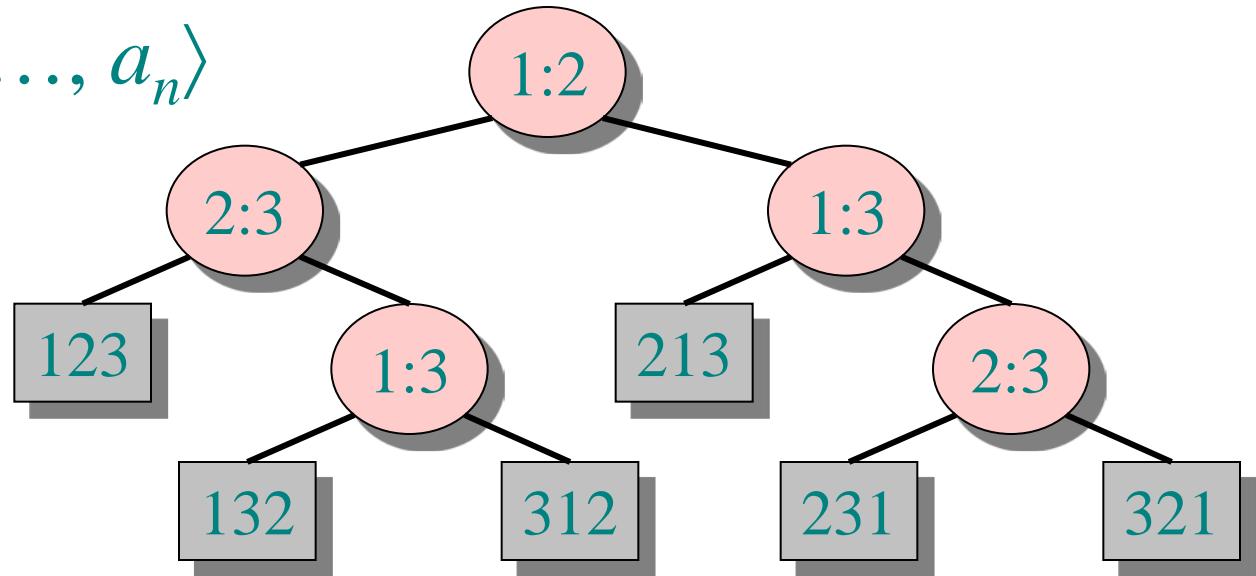
*Is  $O(n \lg n)$  the best we can do?*

***Decision trees*** can help us answer this question.



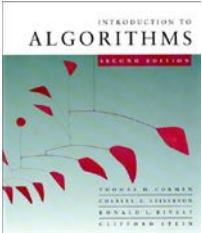
# Decision-tree example

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



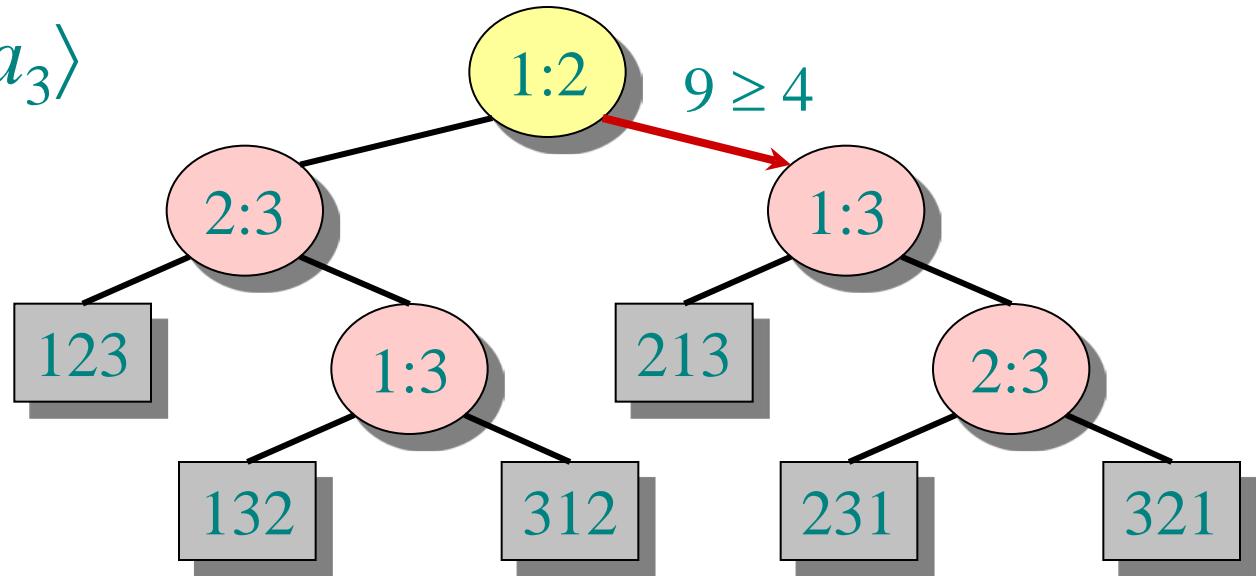
Each internal node is labeled  $i:j$  for  $i, j \in \{1, 2, \dots, n\}$ .

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



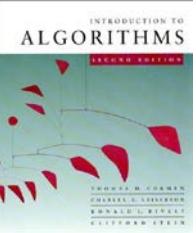
# Decision-tree example

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



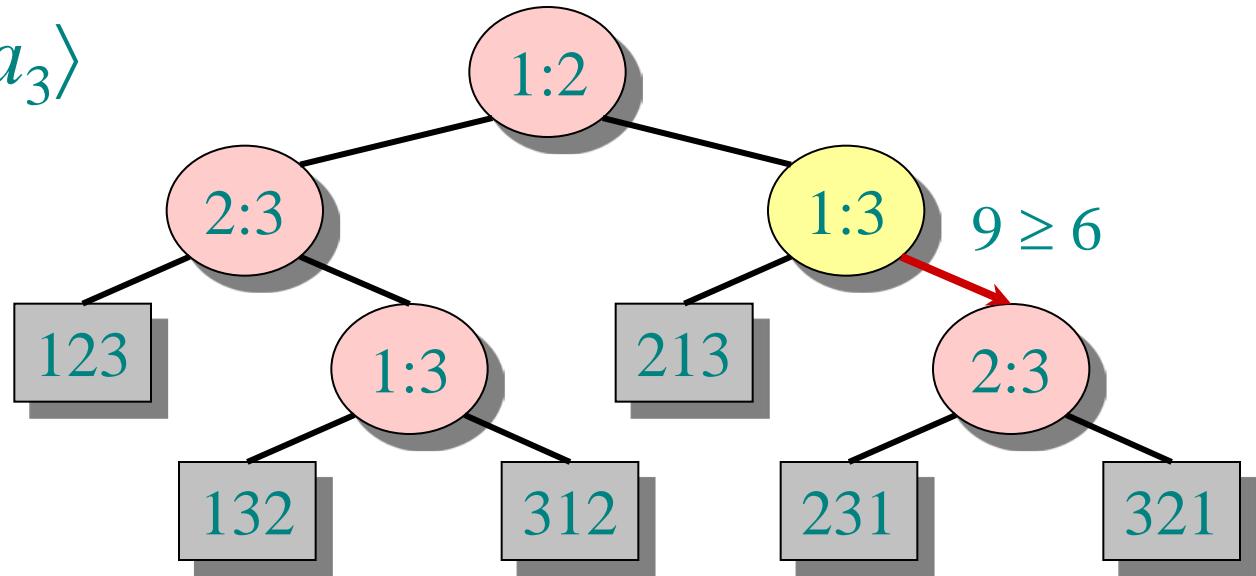
Each internal node is labeled  $i:j$  for  $i, j \in \{1, 2, \dots, n\}$ .

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



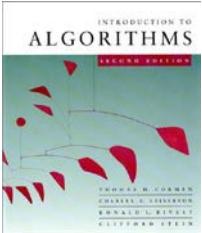
# Decision-tree example

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



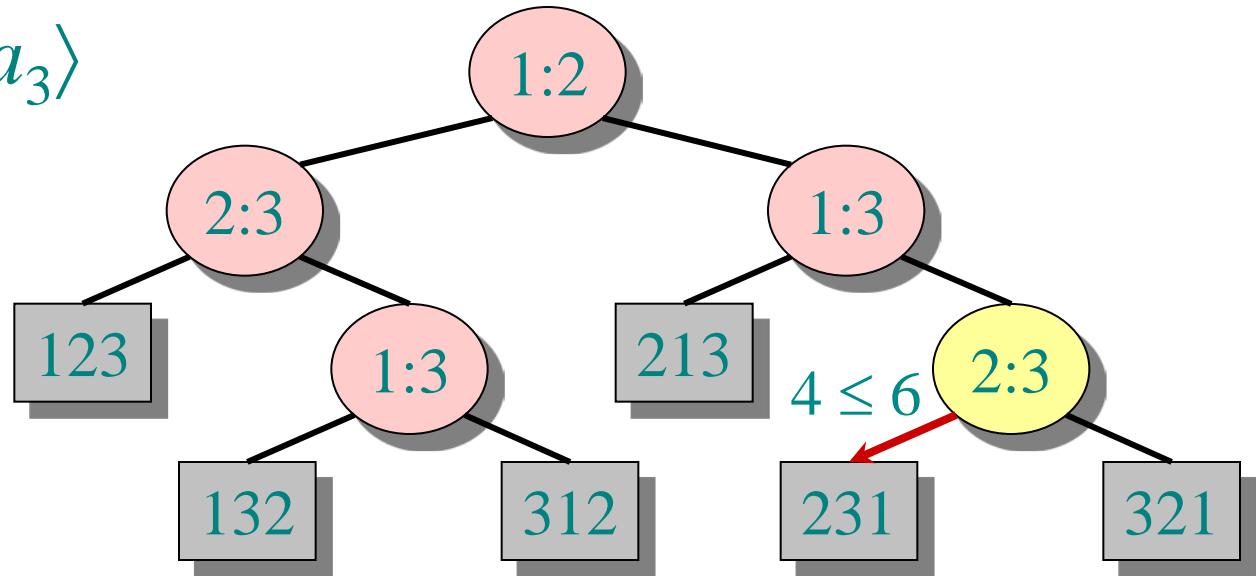
Each internal node is labeled  $i:j$  for  $i, j \in \{1, 2, \dots, n\}$ .

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



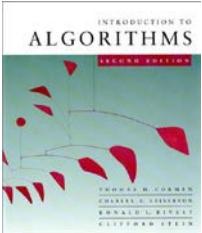
# Decision-tree example

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



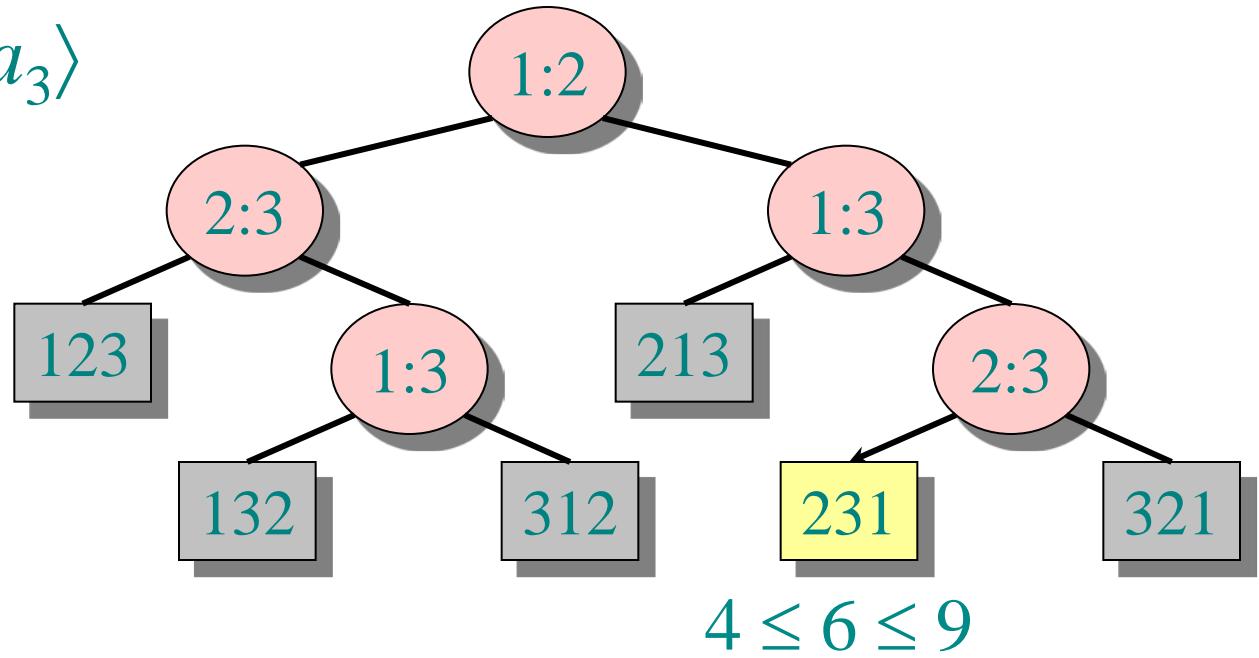
Each internal node is labeled  $i:j$  for  $i, j \in \{1, 2, \dots, n\}$ .

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

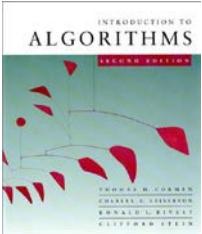


# Decision-tree example

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



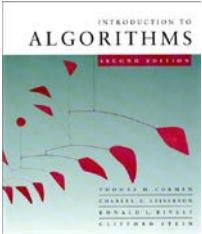
Each leaf contains a permutation  $\langle \pi(1), \pi(2), \dots, \pi(n) \rangle$  to indicate that the ordering  $a_{\pi(1)} \leq a_{\pi(2)} \leq \dots \leq a_{\pi(n)}$  has been established.



# 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.

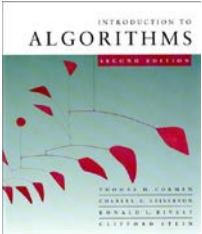


# Lower bound for decision-tree sorting

**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,  $n! \leq 2^h$ .

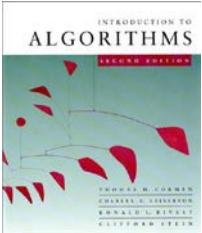
$$\begin{aligned} \therefore h &\geq \lg(n!) && (\lg \text{ is mono. increasing}) \\ &\geq \lg ((n/e)^n) && (\text{Stirling's formula}) \\ &= n \lg n - n \lg e \\ &= \Omega(n \lg n). \quad \square \end{aligned}$$



# Lower bound for comparison sorting

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

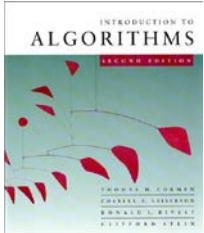




# Sorting in linear time

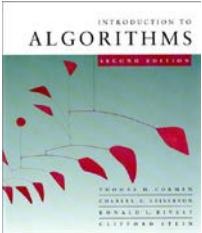
**Counting sort:** No comparisons between elements.

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

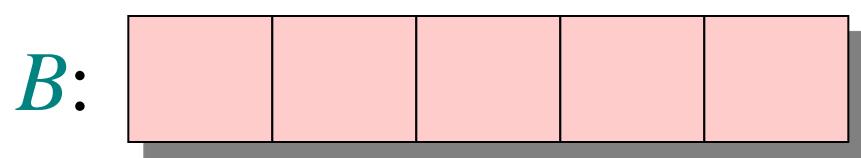
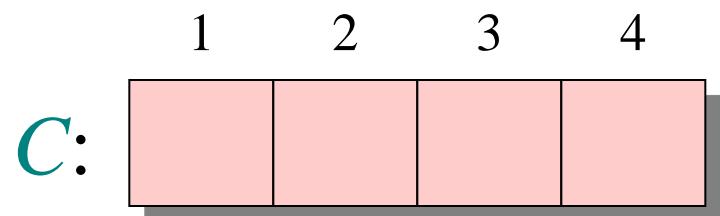
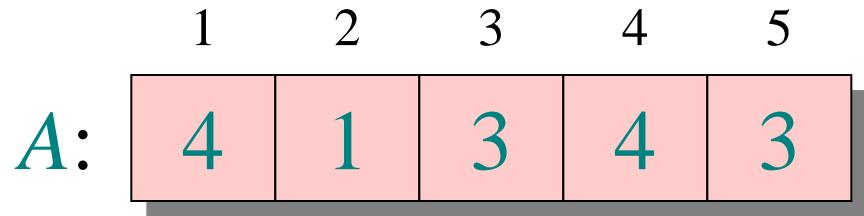


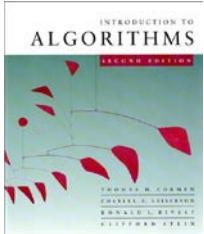
# Counting sort

```
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  $C[i] \leftarrow C[i] + C[i-1]$ 
for  $j \leftarrow n$  downto 1
    do  $B[C[A[j]]] \leftarrow A[j]$ 
         $C[A[j]] \leftarrow C[A[j]] - 1$       ▷  $C[i] = |\{ \text{key} \leq i \}|$ 
```



# Counting-sort example





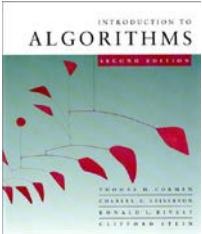
# Loop 1

|      | 1 | 2 | 3 | 4 | 5 |  |
|------|---|---|---|---|---|--|
| $A:$ | 4 | 1 | 3 | 4 | 3 |  |

|      | 1 | 2 | 3 | 4 |  |
|------|---|---|---|---|--|
| $C:$ | 0 | 0 | 0 | 0 |  |

|      |  |  |  |  |  |  |
|------|--|--|--|--|--|--|
| $B:$ |  |  |  |  |  |  |
|------|--|--|--|--|--|--|

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



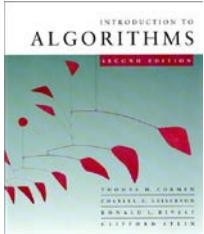
# Loop 2

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



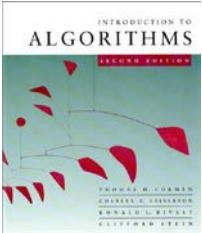
# Loop 2

|      | 1 | 2 | 3 | 4 | 5 |  |
|------|---|---|---|---|---|--|
| $A:$ | 4 | 1 | 3 | 4 | 3 |  |

|      | 1 | 2 | 3 | 4 |  |
|------|---|---|---|---|--|
| $C:$ | 1 | 0 | 0 | 1 |  |

|      |  |  |  |  |  |  |
|------|--|--|--|--|--|--|
| $B:$ |  |  |  |  |  |  |
|------|--|--|--|--|--|--|

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



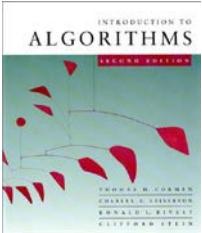
# Loop 2

|      | 1 | 2 | 3 | 4 | 5 |  |
|------|---|---|---|---|---|--|
| $A:$ | 4 | 1 | 3 | 4 | 3 |  |

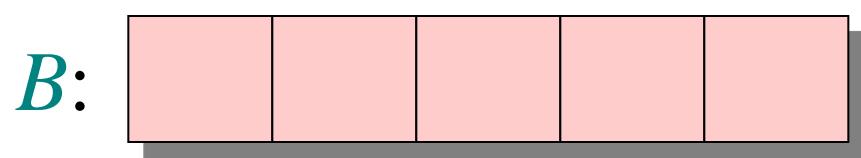
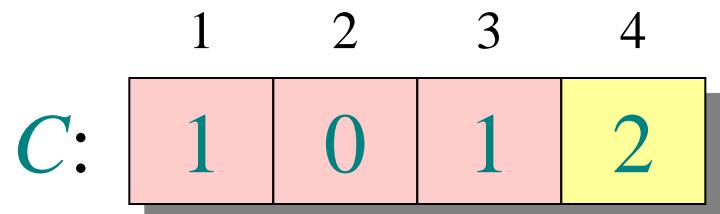
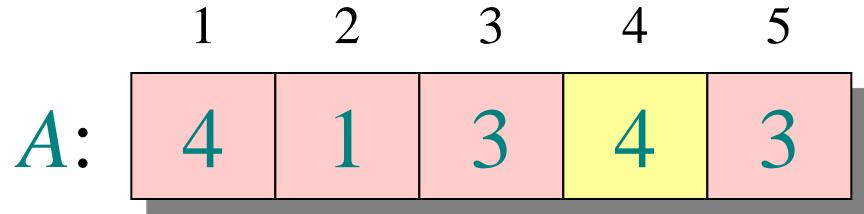
|      | 1 | 2 | 3 | 4 |  |
|------|---|---|---|---|--|
| $C:$ | 1 | 0 | 1 | 1 |  |

|      |  |  |  |  |  |  |
|------|--|--|--|--|--|--|
| $B:$ |  |  |  |  |  |  |
|------|--|--|--|--|--|--|

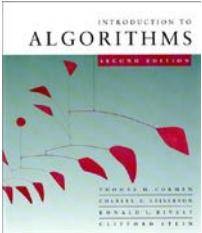
```
for  $j \leftarrow 1$  to  $n$ 
    do  $C[A[j]] \leftarrow C[A[j]] + 1$      $\triangleright C[i] = |\{ \text{key} = i \}|$ 
```



# Loop 2



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



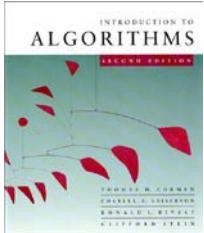
# Loop 2

|      | 1 | 2 | 3 | 4 | 5 |
|------|---|---|---|---|---|
| $A:$ | 4 | 1 | 3 | 4 | 3 |

|      | 1 | 2 | 3 | 4 |
|------|---|---|---|---|
| $C:$ | 1 | 0 | 2 | 2 |

|      |  |  |  |  |  |
|------|--|--|--|--|--|
| $B:$ |  |  |  |  |  |
|      |  |  |  |  |  |

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



# Loop 3

|      | 1 | 2 | 3 | 4 | 5 |  |
|------|---|---|---|---|---|--|
| $A:$ | 4 | 1 | 3 | 4 | 3 |  |

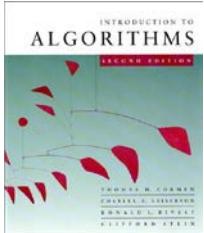
|      |  |  |  |  |  |  |
|------|--|--|--|--|--|--|
| $B:$ |  |  |  |  |  |  |
|------|--|--|--|--|--|--|

|      | 1 | 2 | 3 | 4 |  |
|------|---|---|---|---|--|
| $C:$ | 1 | 0 | 2 | 2 |  |

|       |   |   |   |   |  |
|-------|---|---|---|---|--|
| $C':$ | 1 | 1 | 2 | 2 |  |
|-------|---|---|---|---|--|

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

►  $C[i] = |\{\text{key} \leq i\}|$



# Loop 3

|      | 1 | 2 | 3 | 4 | 5 |  |
|------|---|---|---|---|---|--|
| $A:$ | 4 | 1 | 3 | 4 | 3 |  |

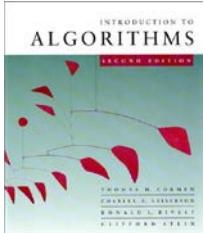
| $B:$ |  |  |  |  |  |  |
|------|--|--|--|--|--|--|
|      |  |  |  |  |  |  |

|      | 1 | 2 | 3 | 4 |  |
|------|---|---|---|---|--|
| $C:$ | 1 | 0 | 2 | 2 |  |

| $C':$ | 1 | 1 | 3 | 2 |  |
|-------|---|---|---|---|--|
|       |   |   |   |   |  |

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

►  $C[i] = |\{\text{key} \leq i\}|$



# Loop 3

|      | 1 | 2 | 3 | 4 | 5 |
|------|---|---|---|---|---|
| $A:$ | 4 | 1 | 3 | 4 | 3 |

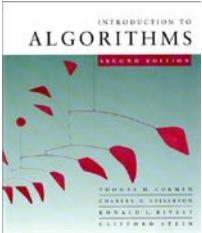
|      |  |  |  |  |  |
|------|--|--|--|--|--|
| $B:$ |  |  |  |  |  |
|------|--|--|--|--|--|

|      | 1 | 2 | 3 | 4 |
|------|---|---|---|---|
| $C:$ | 1 | 0 | 2 | 2 |

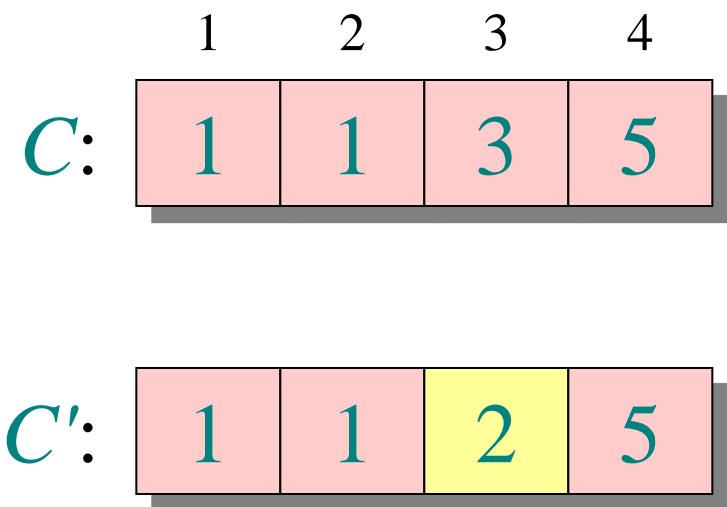
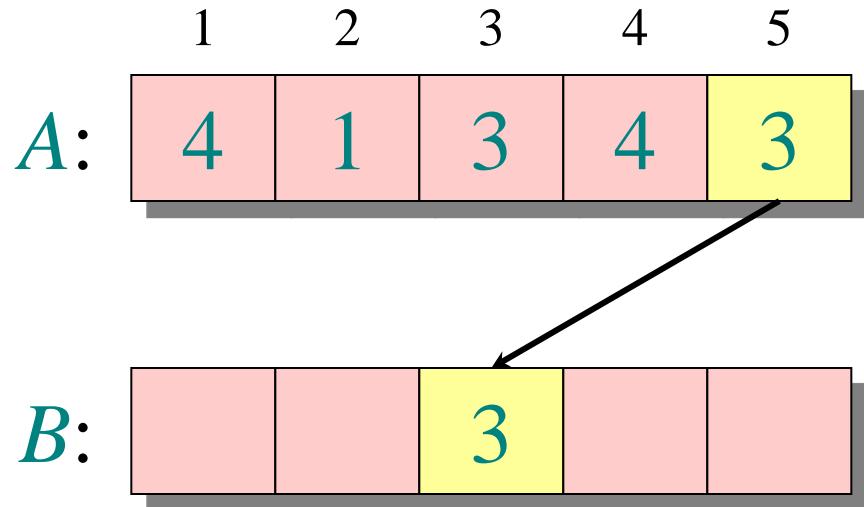
|       |   |   |   |   |
|-------|---|---|---|---|
| $C':$ | 1 | 1 | 3 | 5 |
|-------|---|---|---|---|

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

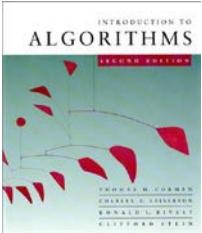
►  $C[i] = |\{\text{key} \leq i\}|$



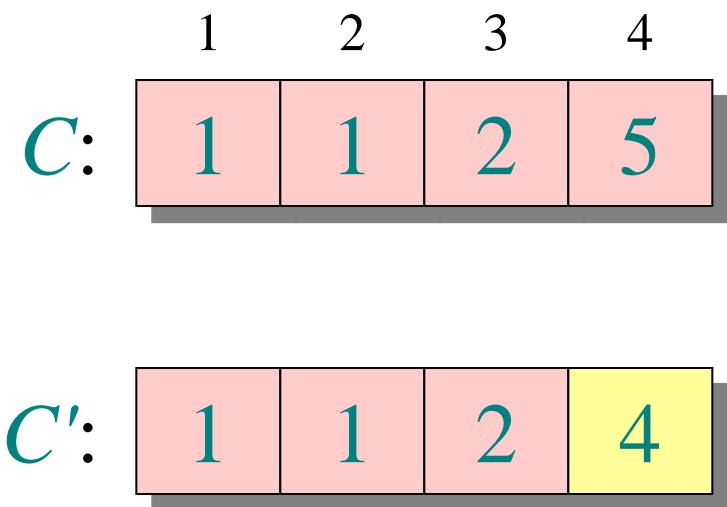
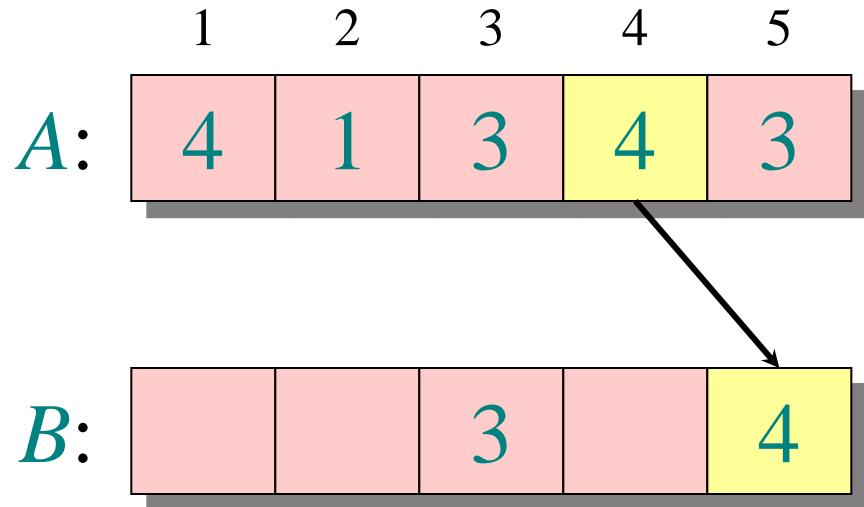
# Loop 4



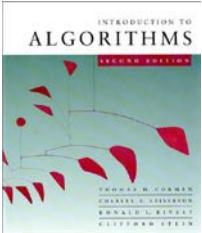
```
for  $j \leftarrow n$  downto 1  
do  $B[C[A[j]]] \leftarrow A[j]$   
    $C[A[j]] \leftarrow C[A[j]] - 1$ 
```



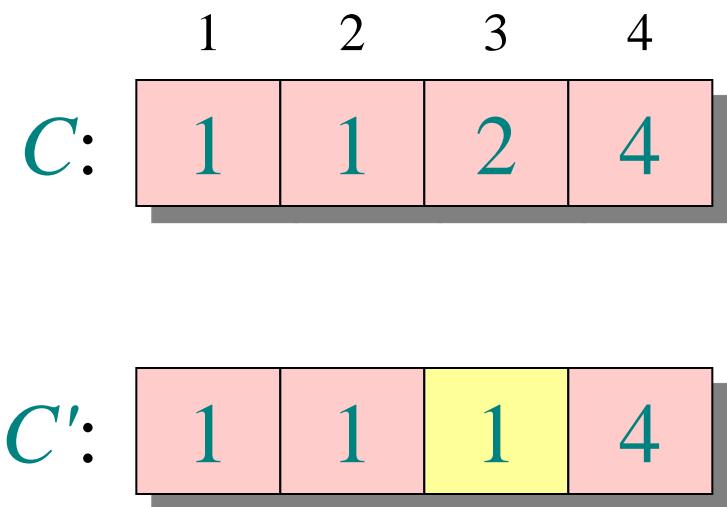
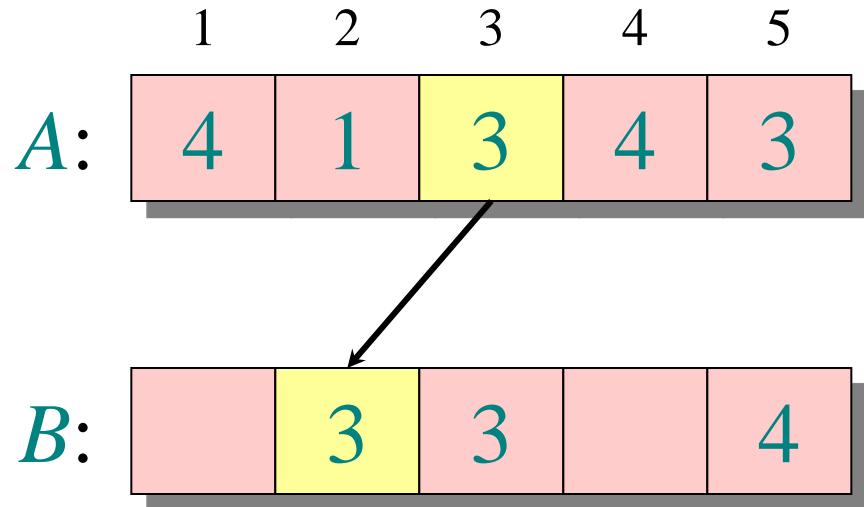
# Loop 4



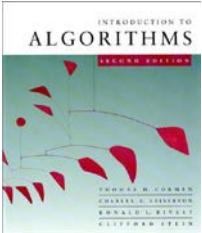
```
for  $j \leftarrow n$  downto 1  
do  $B[C[A[j]]] \leftarrow A[j]$   
     $C[A[j]] \leftarrow C[A[j]] - 1$ 
```



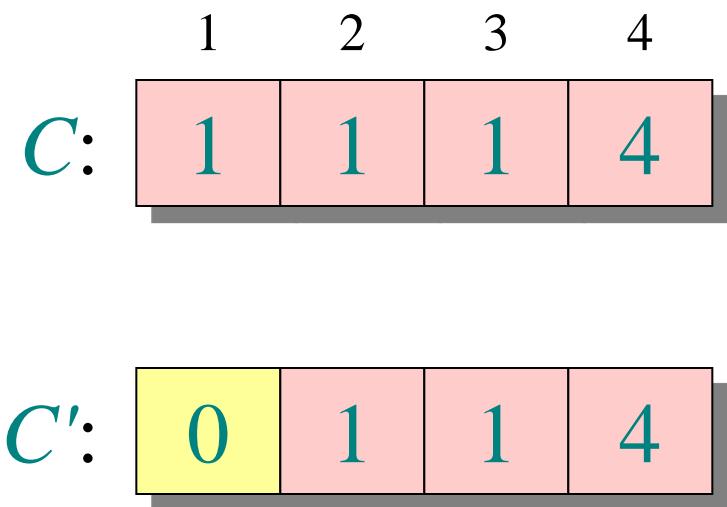
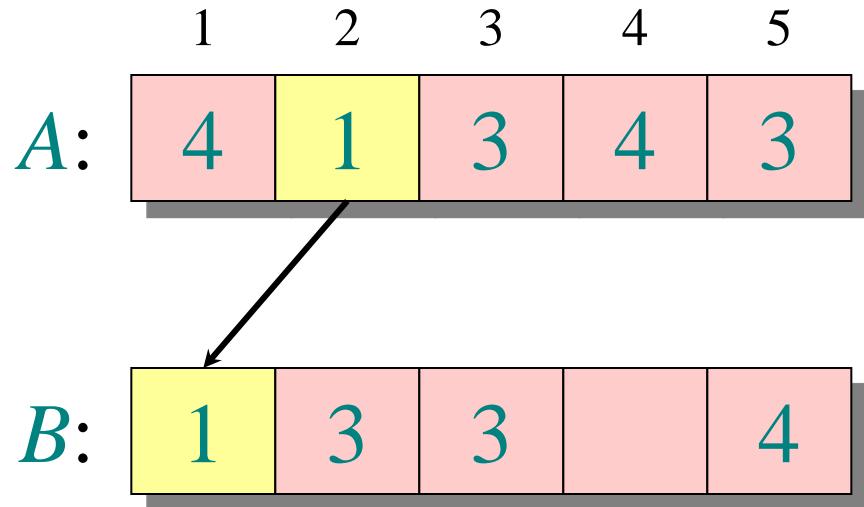
# Loop 4



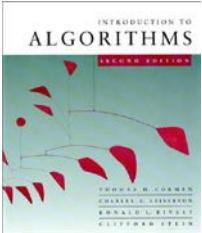
```
for  $j \leftarrow n$  downto 1  
do  $B[C[A[j]]] \leftarrow A[j]$   
     $C[A[j]] \leftarrow C[A[j]] - 1$ 
```



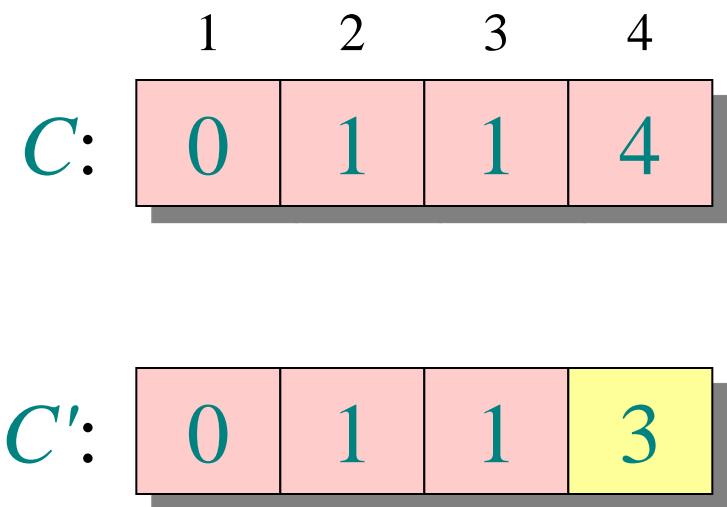
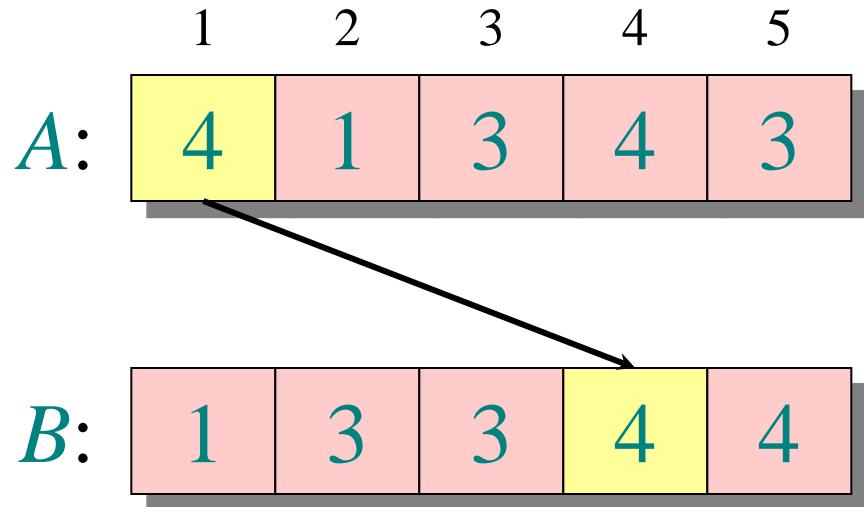
# Loop 4



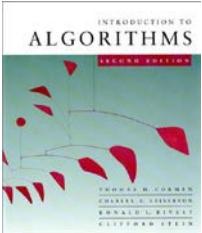
```
for  $j \leftarrow n$  downto 1  
do  $B[C[A[j]]] \leftarrow A[j]$   
     $C[A[j]] \leftarrow C[A[j]] - 1$ 
```



# Loop 4



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



# Analysis

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

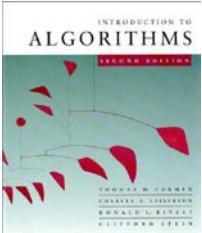
$\Theta(n)$     {    **for**  $j \leftarrow 1$  **to**  $n$   
            **do**  $C[A[j]] \leftarrow C[A[j]] + 1$

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

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

---

$\Theta(n + k)$



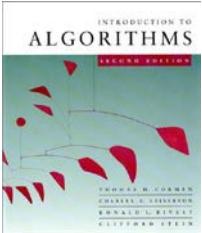
# Running time

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

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

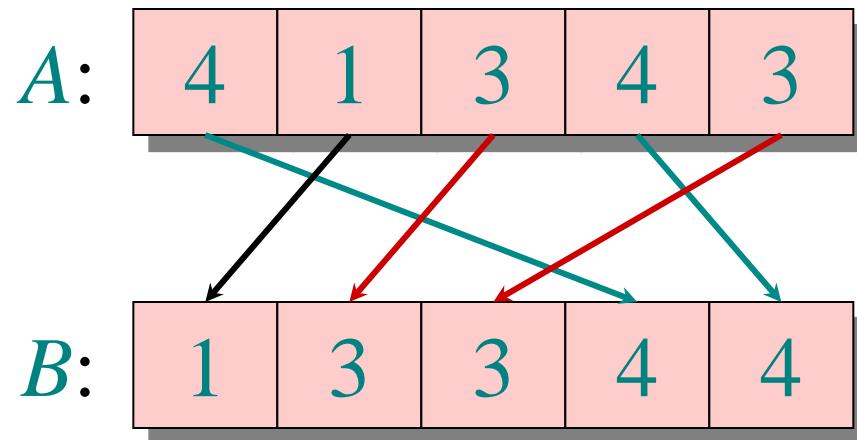
**Answer:**

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

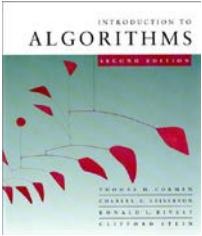


# Stable sorting

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

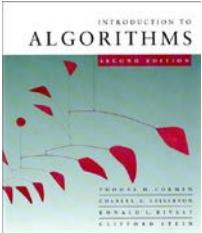


**Exercise:** What other sorts have this property?



# Radix sort

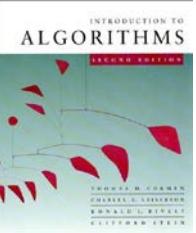
- *Origin*: Herman Hollerith's card-sorting machine for the 1890 U.S. Census. (See Appendix .)
- Digit-by-digit sort.
- Hollerith's original (bad) idea: sort on most-significant digit first.
- Good idea: Sort on *least-significant digit first* with auxiliary *stable* sort.



# Operation of radix sort

|       |       |       |       |
|-------|-------|-------|-------|
| 3 2 9 | 7 2 0 | 7 2 0 | 3 2 9 |
| 4 5 7 | 3 5 5 | 3 2 9 | 3 5 5 |
| 6 5 7 | 4 3 6 | 4 3 6 | 4 3 6 |
| 8 3 9 | 4 5 7 | 8 3 9 | 4 5 7 |
| 4 3 6 | 6 5 7 | 3 5 5 | 6 5 7 |
| 7 2 0 | 3 2 9 | 4 5 7 | 7 2 0 |
| 3 5 5 | 8 3 9 | 6 5 7 | 8 3 9 |

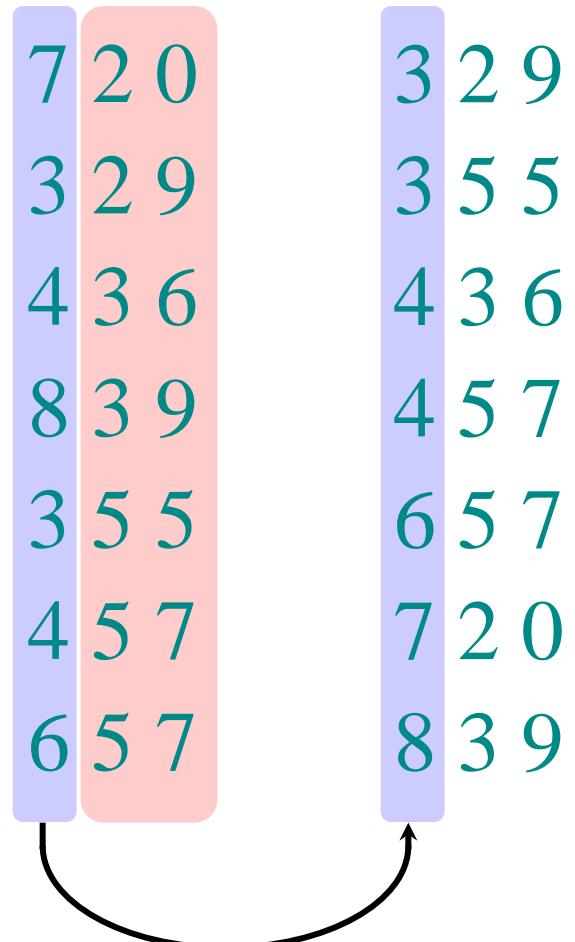


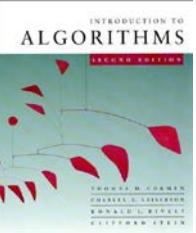


# Correctness of radix sort

*Induction on digit position*

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

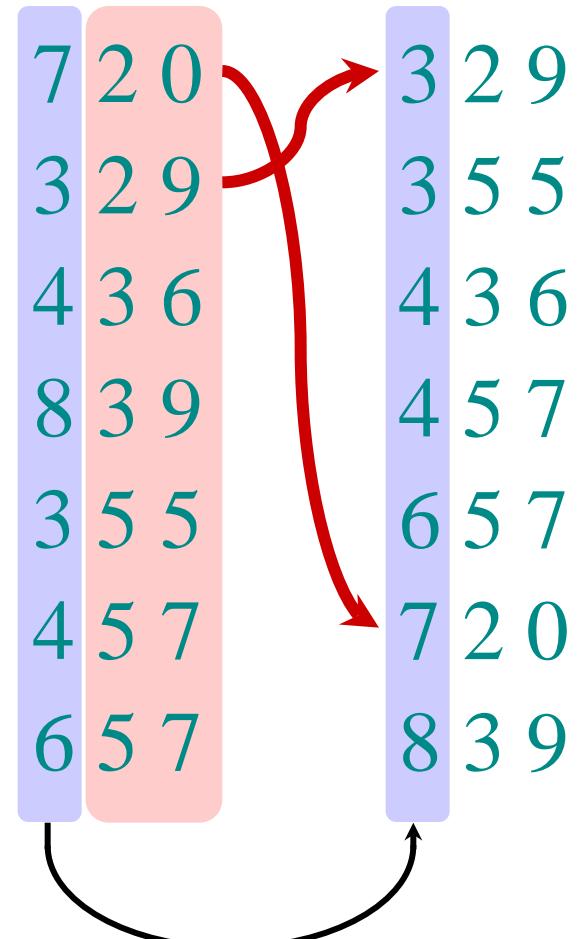


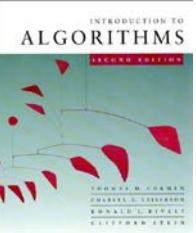


# Correctness of radix sort

*Induction on digit position*

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

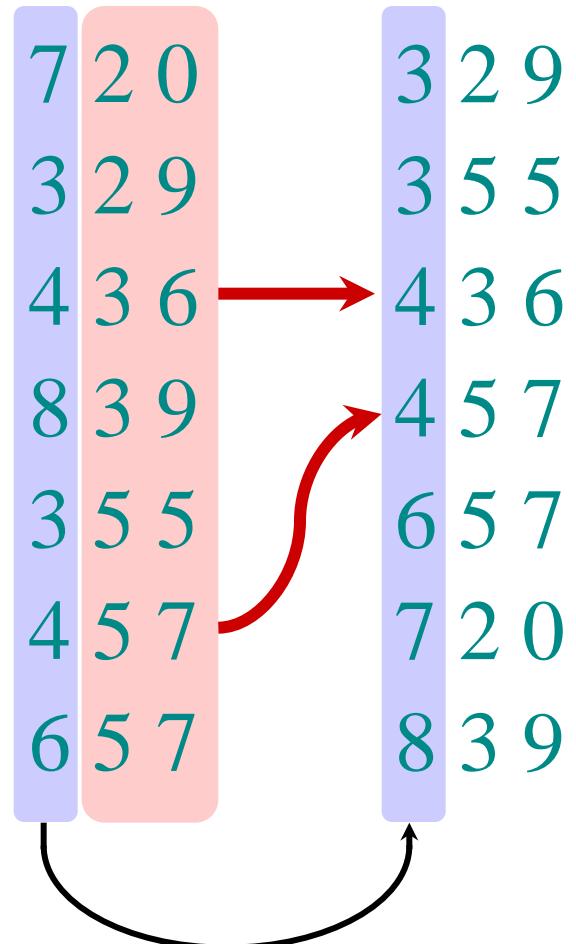


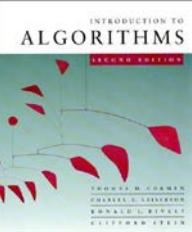


# Correctness of radix sort

*Induction on digit position*

- Assume that the numbers are sorted by their low-order  $t - 1$  digits.
- Sort on digit  $t$ 
  - Two numbers that differ in digit  $t$  are correctly sorted.
  - Two numbers equal in digit  $t$  are put in the same order as the input  $\Rightarrow$  correct order.

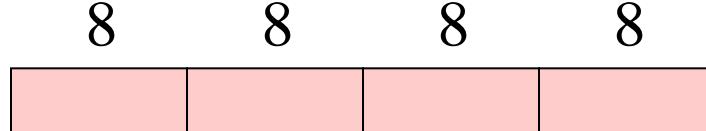




# Analysis of radix sort

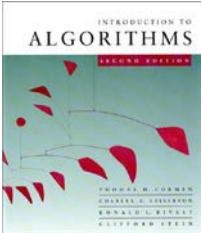
- Assume counting sort is the auxiliary stable sort.
- Sort  $n$  computer words of  $b$  bits each.
- Each word can be viewed as having  $b/r$  base- $2^r$  digits.

**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?***



# Analysis (continued)

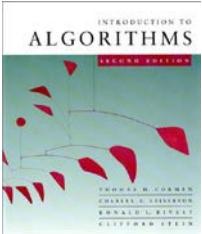
**Recall:** Counting sort takes  $\Theta(n + k)$  time to sort  $n$  numbers in the range from  $0$  to  $k - 1$ .

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

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

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

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



# Choosing $r$

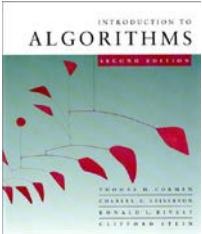
$$T(n, b) = \Theta\left(\frac{b}{r} (n + 2^r)\right)$$

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.



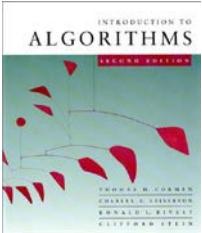
# Conclusions

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  $\geq 2000$  numbers.
- Merge sort and quicksort do at least  $\lceil \lg 2000 \rceil = 11$  passes.

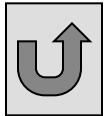
**Downside:** Unlike quicksort, radix sort displays little locality of reference, and thus a well-tuned quicksort fares better on modern processors, which feature steep memory hierarchies.

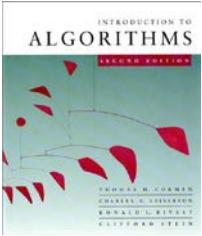


# Appendix: Punched-card technology

- 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.

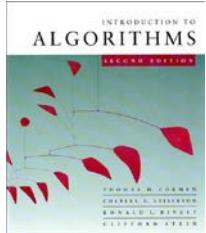




# Herman Hollerith (1860-1929)

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



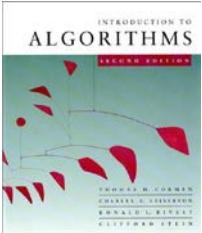


# Punched cards

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

|   |   |   |   |    |    |    |    |     |    |    |    |    |     |   |    |     |           |           |           |           |           |           |           |
|---|---|---|---|----|----|----|----|-----|----|----|----|----|-----|---|----|-----|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
| 1 | 2 | 3 | 4 | W  | M  | 0  | 1  | 5   | 9  | Un | 0  | 6  | 12  | 0 | 6  | 12  | Mr        | Ni        | vt        | ch        | wh        | ia        | sd        |
| 5 | 6 | 7 | 8 | F  | 10 | 15 | 18 | I   | R  | S  | I  | 7  | 13  | 1 | 7  | 13  | MAS<br>DR | RI<br>IR  | CT<br>SC  | IND<br>AS | WIS<br>CU | MO<br>HO  | NBR<br>SP |
| 1 | 2 | 3 | 4 | Ch | 20 | 21 | 25 | 30  | ?  | MD | 2  | 8  | 14  | 2 | 8  | N   | NY<br>SW  | NJ<br>CE  | PA<br>PA  | ILL<br>AI | MIN<br>EP | ND<br>IN  | KAN<br>SA |
| 5 | 6 | 7 | 8 | Ap | 35 | 40 | 45 | 50  | ?  | MI | 3  | 9  | 15  | 3 | 9  | F   | MD<br>WV  | VA<br>CF  | WH<br>HU  | SC<br>AT  | TEN<br>PK | ALA<br>JP | CLF<br>71 |
| 1 | 2 | 3 | 4 | In | 55 | 60 | 65 | 70  | ?  | Wd | 4  | 10 | 16  | 4 | 10 | QSL | MD<br>DH  | SC<br>PR  | MS<br>IT  | LA<br>BI  | TEX<br>GS | DRE<br>UX | WBN<br>W  |
| 5 | 6 | 7 | 8 | 75 | 80 | 85 | 90 | 95+ | Jn | D  | 5  | 11 | 17+ | 5 | 11 | PS  | DE<br>RI  | FLA<br>EO | OKL<br>SE | IT<br>CA  | ARK<br>GC | IDM<br>MX | NEV<br>CT |
| 1 | 2 | 3 | 4 | Er | OK | 0  | 9  | 4   | 17 | II | 5  | Un | 15  | 2 | 0  | US  | Un        | En        | US        | Un        | En        | VIA<br>PI | AHI<br>AP |
| 5 | 6 | 7 | 8 | Ot | NR | 1  | b  | 5   | 01 | 12 | 6  | NG | 20+ | 3 | 1  | Gr  | Ir        | Sc        | Gr        | Ir        | Sc        | NM<br>FHP | COL<br>GP |
| 1 | 2 | 3 | 4 | 2  | NW | 4  | c  | 6   | 0  | 13 | 7  | I  | No  | 4 | Au | Sw  | CE        | Wa        | Sw        | OE        | Wo        | WYO<br>PR | MNT<br>RP |
| 5 | 6 | 7 | 8 | 4  | 0  | 7  | d  | 7   | 1  | 14 | 8  | 2  | Pa  | 5 | Sz | Nw  | CF        | Hu        | Nw        | CH        | HJ        | ALK<br>PT | AB<br>UP  |
| 1 | 2 | 3 | 4 | 6  | 12 | 10 | e  | 8   | 2  | 15 | 9  | 3  | Al  | 6 | Po | DK  | Fr        | It        | Dk        | Fy        | Ir        | Au        | SEA       |
| 5 | 6 | 7 | 8 | B+ | Un | g  | f  | 9   | 3  | 16 | 10 | 4  | Un  | 0 | 01 | Ru  | Bu        | 01        | Ru        | Bu        | Sz        | Po        | NS        |

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



# Hollerith's tabulating system

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

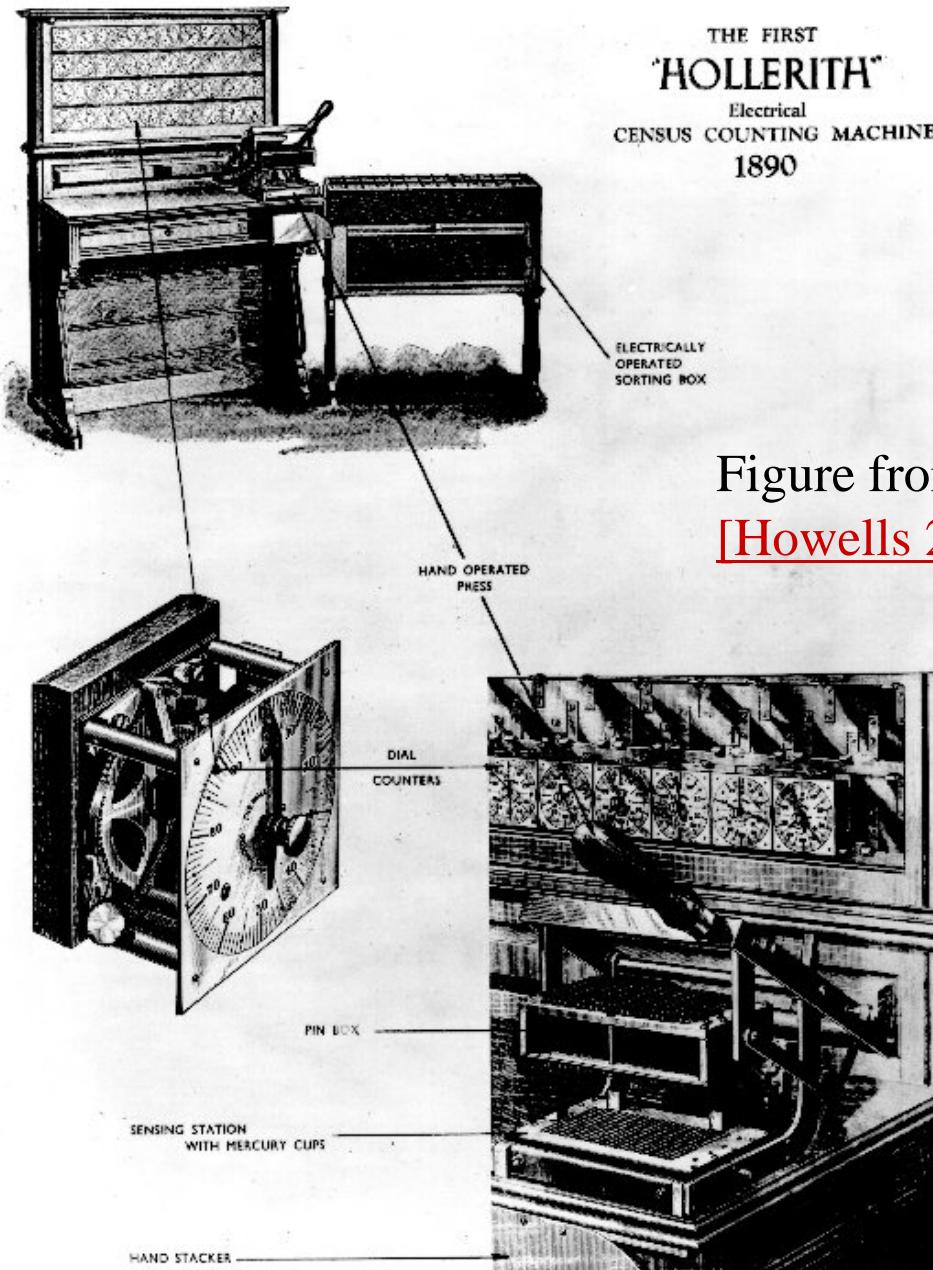
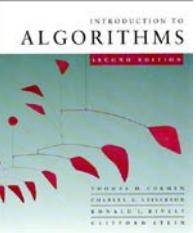
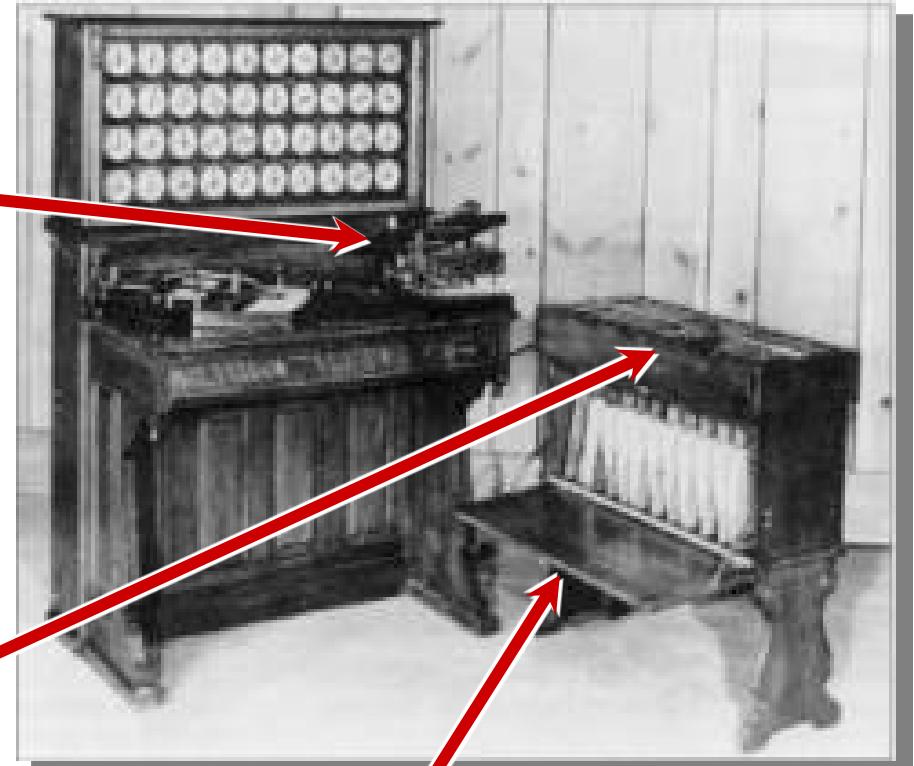


Figure from  
[\[Howells 2000\]](#).

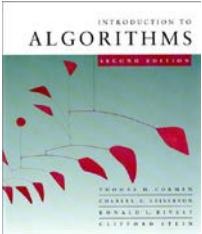


# Operation of the sorter

- 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.
- 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.



Hollerith Tabulator, Pantograph, Press, and Sorter

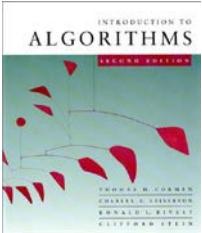


# Origin of radix sort

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

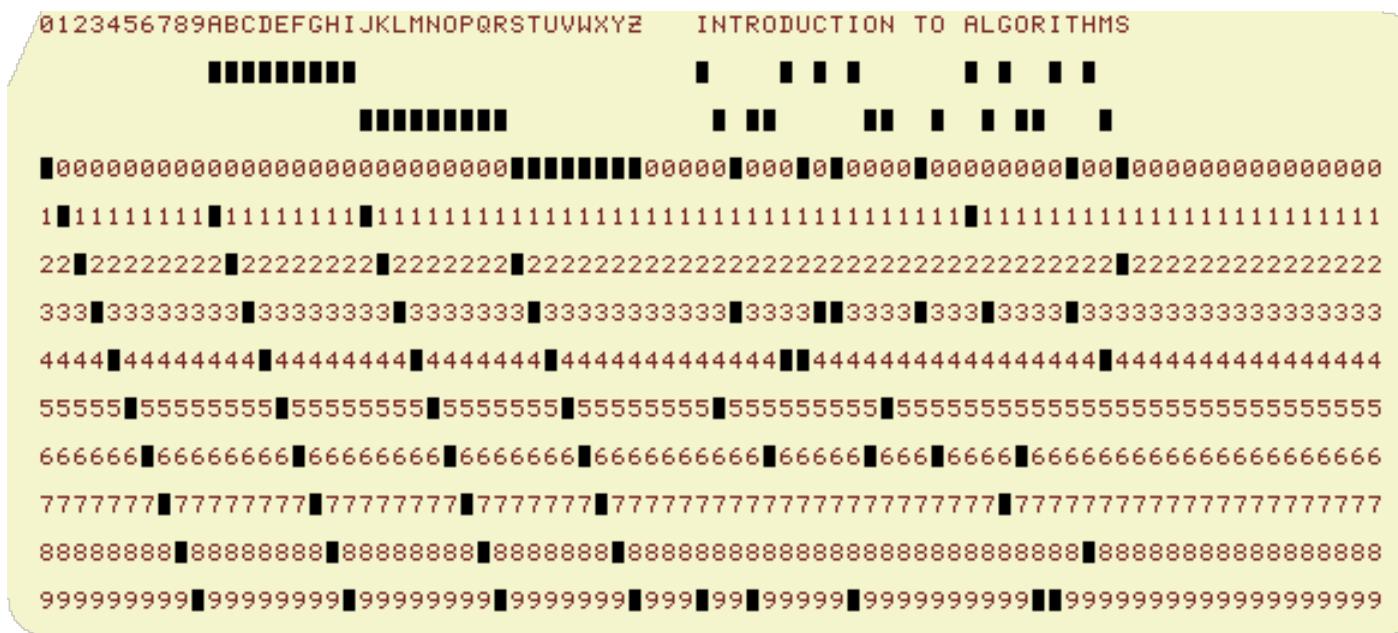
*“The most complicated combinations can readily be counted with comparatively few counters or relays by first assorting the cards according to the first items entering into the combinations, then reassorting each group according to the second item entering into the combination, and so on, and finally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



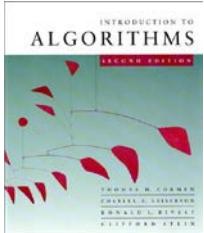
# “Modern” IBM card

- One character per column.



Produced by  
the WWW  
Virtual Punch-  
Card Server.

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

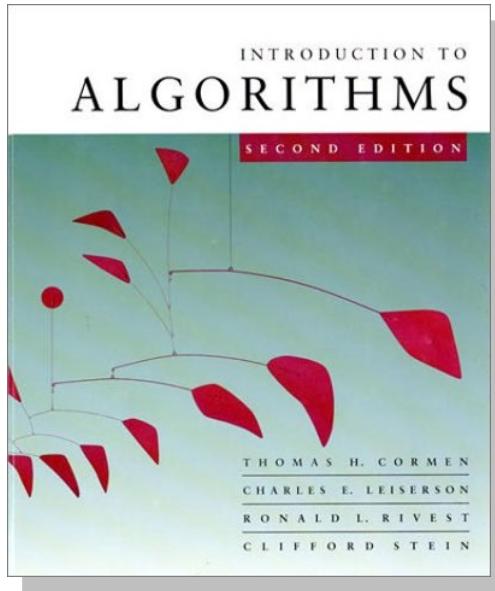


# Web resources on punched-card technology

- 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

# *Introduction to Algorithms*

**6.046J/18.401J**

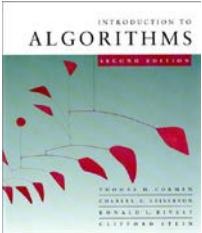


## **LECTURE 6**

### **Order Statistics**

- Randomized divide and conquer
- Analysis of expected time
- Worst-case linear-time order statistics
- Analysis

**Prof. Erik Demaine**



# Order statistics

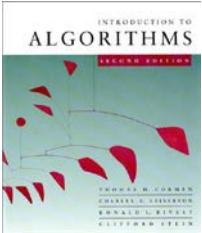
Select the  $i$ th smallest of  $n$  elements (the element with **rank  $i$** ).

- $i = 1$ : **minimum**;
- $i = n$ : **maximum**;
- $i = \lfloor (n+1)/2 \rfloor$  or  $\lceil (n+1)/2 \rceil$ : **median**.

*Naive algorithm*: Sort and index  $i$ th element.

$$\begin{aligned}\text{Worst-case running time} &= \Theta(n \lg n) + \Theta(1) \\ &= \Theta(n \lg n),\end{aligned}$$

using merge sort or heapsort (*not* quicksort).



# Randomized divide-and-conquer algorithm

RAND-SELECT( $A, p, q, i$ )  $\triangleright i$ th smallest of  $A[p \dots q]$

**if**  $p = q$  **then return**  $A[p]$

$r \leftarrow$  RAND-PARTITION( $A, p, q$ )

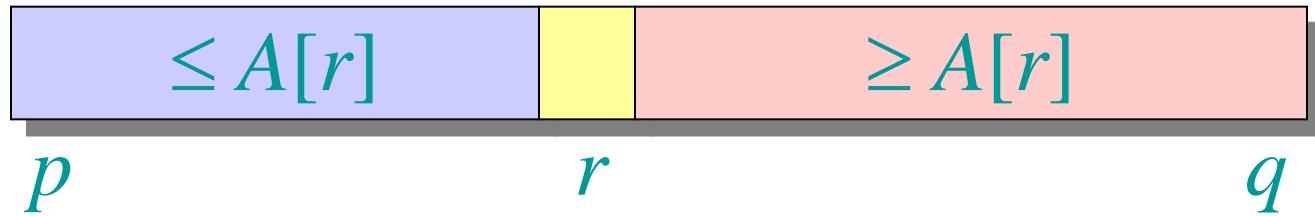
$k \leftarrow r - p + 1$   $\triangleright k = \text{rank}(A[r])$

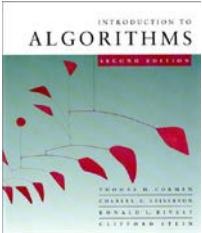
**if**  $i = k$  **then return**  $A[r]$

**if**  $i < k$

**then return** RAND-SELECT( $A, p, r - 1, i$ )

**else return** RAND-SELECT( $A, r + 1, q, i - k$ )



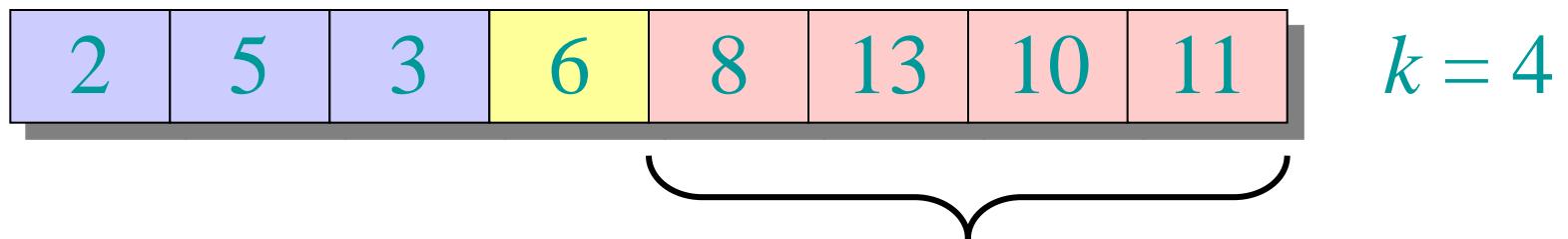


# Example

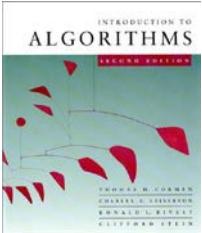
Select the  $i = 7$ th smallest:



Partition:



Select the  $7 - 4 = 3$ rd smallest recursively.



# Intuition for analysis

(All our analyses today assume that all elements are distinct.)

**Lucky:**

$$\begin{aligned} T(n) &= T(9n/10) + \Theta(n) \\ &= \Theta(n) \end{aligned}$$

**Unlucky:**

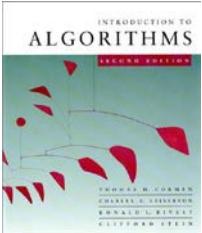
$$\begin{aligned} T(n) &= T(n - 1) + \Theta(n) \\ &= \Theta(n^2) \end{aligned}$$

*Worse than sorting!*

$$n^{\log_{10/9} 1} = n^0 = 1$$

CASE 3

arithmetic series



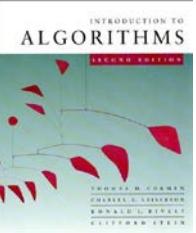
# Analysis of expected time

The analysis follows that of randomized quicksort, but it's a little different.

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

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

$$X_k = \begin{cases} 1 & \text{if PARTITION generates a } k : n-k-1 \text{ split,} \\ 0 & \text{otherwise.} \end{cases}$$

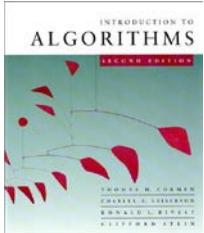


# Analysis (continued)

To obtain an upper bound, assume that the  $i$ th element always falls in the larger side of the partition:

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

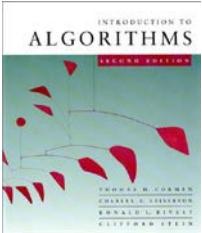
$$= \sum_{k=0}^{n-1} X_k (T(\max\{k, n-k-1\}) + \Theta(n)).$$



# Calculating expectation

$$E[T(n)] = E\left[ \sum_{k=0}^{n-1} X_k (T(\max\{k, n-k-1\}) + \Theta(n)) \right]$$

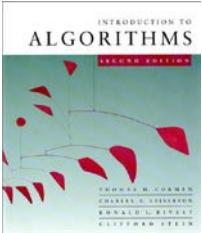
Take expectations of both sides.



# Calculating expectation

$$\begin{aligned}E[T(n)] &= E\left[\sum_{k=0}^{n-1} X_k \left(T(\max\{k, n-k-1\}) + \Theta(n)\right)\right] \\&= \sum_{k=0}^{n-1} E[X_k \left(T(\max\{k, n-k-1\}) + \Theta(n)\right)]\end{aligned}$$

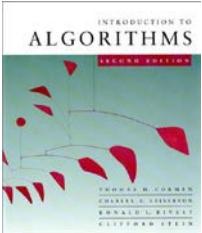
Linearity of expectation.



# Calculating expectation

$$\begin{aligned}E[T(n)] &= E\left[\sum_{k=0}^{n-1} X_k \left(T(\max\{k, n-k-1\}) + \Theta(n)\right)\right] \\&= \sum_{k=0}^{n-1} E[X_k \left(T(\max\{k, n-k-1\}) + \Theta(n)\right)] \\&= \sum_{k=0}^{n-1} E[X_k] \cdot E[T(\max\{k, n-k-1\}) + \Theta(n)]\end{aligned}$$

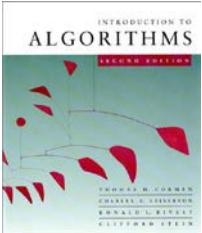
Independence of  $X_k$  from other random choices.



# Calculating expectation

$$\begin{aligned}E[T(n)] &= E\left[\sum_{k=0}^{n-1} X_k \left(T(\max\{k, n-k-1\}) + \Theta(n)\right)\right] \\&= \sum_{k=0}^{n-1} E[X_k \left(T(\max\{k, n-k-1\}) + \Theta(n)\right)] \\&= \sum_{k=0}^{n-1} E[X_k] \cdot E[T(\max\{k, n-k-1\}) + \Theta(n)] \\&= \frac{1}{n} \sum_{k=0}^{n-1} E[T(\max\{k, n-k-1\})] + \frac{1}{n} \sum_{k=0}^{n-1} \Theta(n)\end{aligned}$$

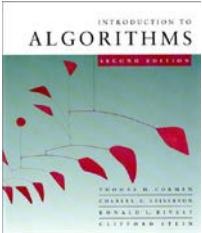
Linearity of expectation;  $E[X_k] = 1/n$ .



# Calculating expectation

$$\begin{aligned}E[T(n)] &= E\left[\sum_{k=0}^{n-1} X_k \left(T(\max\{k, n-k-1\}) + \Theta(n)\right)\right] \\&= \sum_{k=0}^{n-1} E[X_k \left(T(\max\{k, n-k-1\}) + \Theta(n)\right)] \\&= \sum_{k=0}^{n-1} E[X_k] \cdot E[T(\max\{k, n-k-1\}) + \Theta(n)] \\&= \frac{1}{n} \sum_{k=0}^{n-1} E[T(\max\{k, n-k-1\})] + \frac{1}{n} \sum_{k=0}^{n-1} \Theta(n) \\&\leq \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} E[T(k)] + \Theta(n)\end{aligned}$$

Upper terms  
appear twice.



# Hairy recurrence

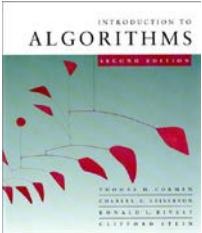
(But not quite as hairy as the quicksort one.)

$$E[T(n)] = \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} E[T(k)] + \Theta(n)$$

**Prove:**  $E[T(n)] \leq cn$  for constant  $c > 0$ .

- The constant  $c$  can be chosen large enough so that  $E[T(n)] \leq cn$  for the base cases.

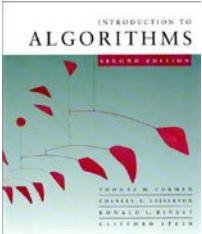
**Use fact:**  $\sum_{k=\lfloor n/2 \rfloor}^{n-1} k \leq \frac{3}{8}n^2$  (exercise).



# Substitution method

$$E[T(n)] \leq \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} ck + \Theta(n)$$

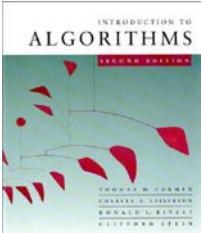
Substitute inductive hypothesis.



# Substitution method

$$\begin{aligned}E[T(n)] &\leq \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} ck + \Theta(n) \\&\leq \frac{2c}{n} \left( \frac{3}{8} n^2 \right) + \Theta(n)\end{aligned}$$

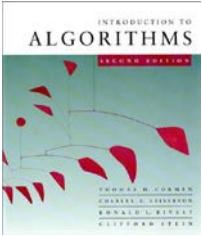
Use fact.



# Substitution method

$$\begin{aligned}E[T(n)] &\leq \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} ck + \Theta(n) \\&\leq \frac{2c}{n} \left( \frac{3}{8} n^2 \right) + \Theta(n) \\&= cn - \left( \frac{cn}{4} - \Theta(n) \right)\end{aligned}$$

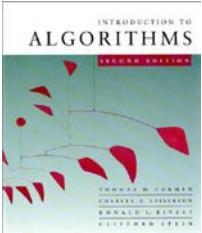
Express as *desired – residual*.



# Substitution method

$$\begin{aligned}E[T(n)] &\leq \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} ck + \Theta(n) \\&\leq \frac{2c}{n} \left( \frac{3}{8} n^2 \right) + \Theta(n) \\&= cn - \left( \frac{cn}{4} - \Theta(n) \right) \\&\leq cn,\end{aligned}$$

if  $c$  is chosen large enough so that  $cn/4$  dominates the  $\Theta(n)$ .



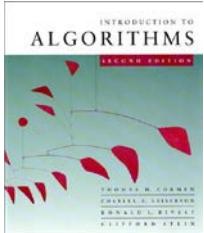
# Summary of randomized order-statistic selection

- Works fast: linear expected time.
- Excellent algorithm in practice.
- But, the worst case is **very** bad:  $\Theta(n^2)$ .

**Q.** Is there an algorithm that runs in linear time in the worst case?

**A.** Yes, due to Blum, Floyd, Pratt, Rivest, and Tarjan [1973].

**IDEA:** Generate a good pivot recursively.

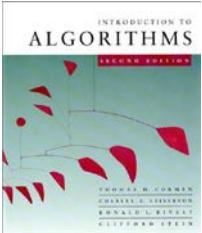


# Worst-case linear-time order statistics

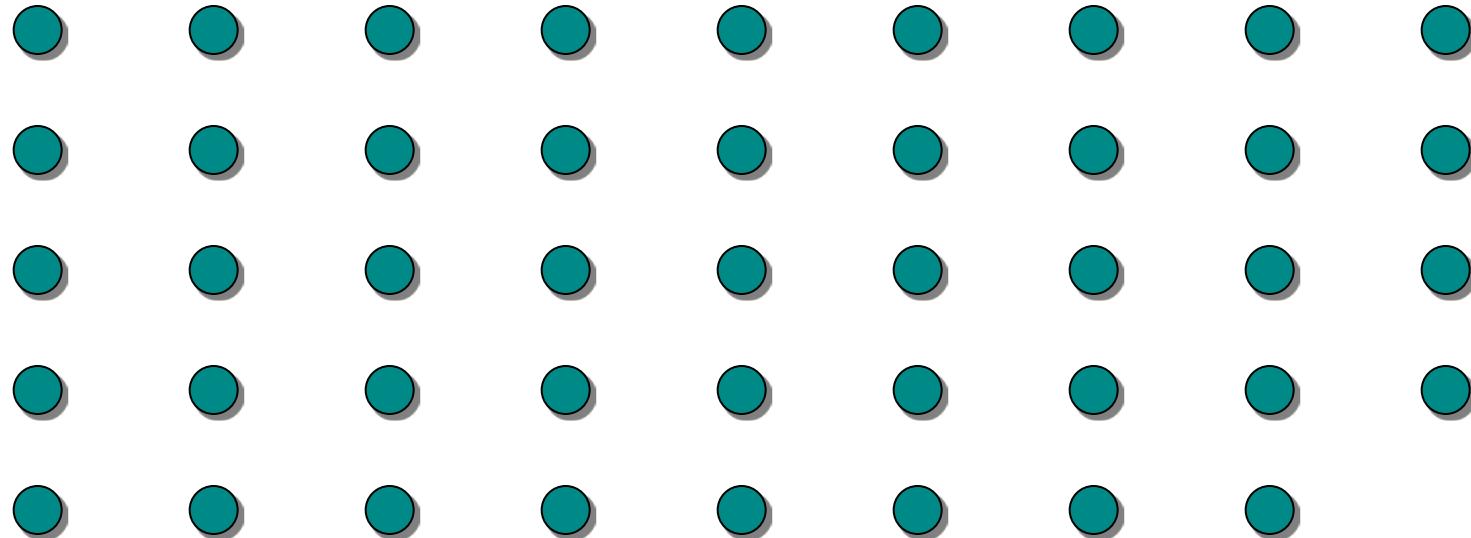
**SELECT( $i, n$ )**

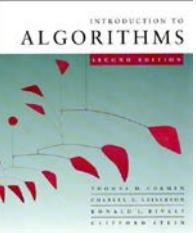
1. Divide the  $n$  elements into groups of 5. Find the median of each 5-element group by rote.
2. Recursively SELECT the median  $x$  of the  $\lfloor n/5 \rfloor$  group medians to be the pivot.
3. Partition around the pivot  $x$ . Let  $k = \text{rank}(x)$ .
4. if  $i = k$  then return  $x$   
elseif  $i < k$   
    then recursively SELECT the  $i$ th  
        smallest element in the lower part  
    else recursively SELECT the  $(i-k)$ th  
        smallest element in the upper part

Same as  
RAND-  
SELECT

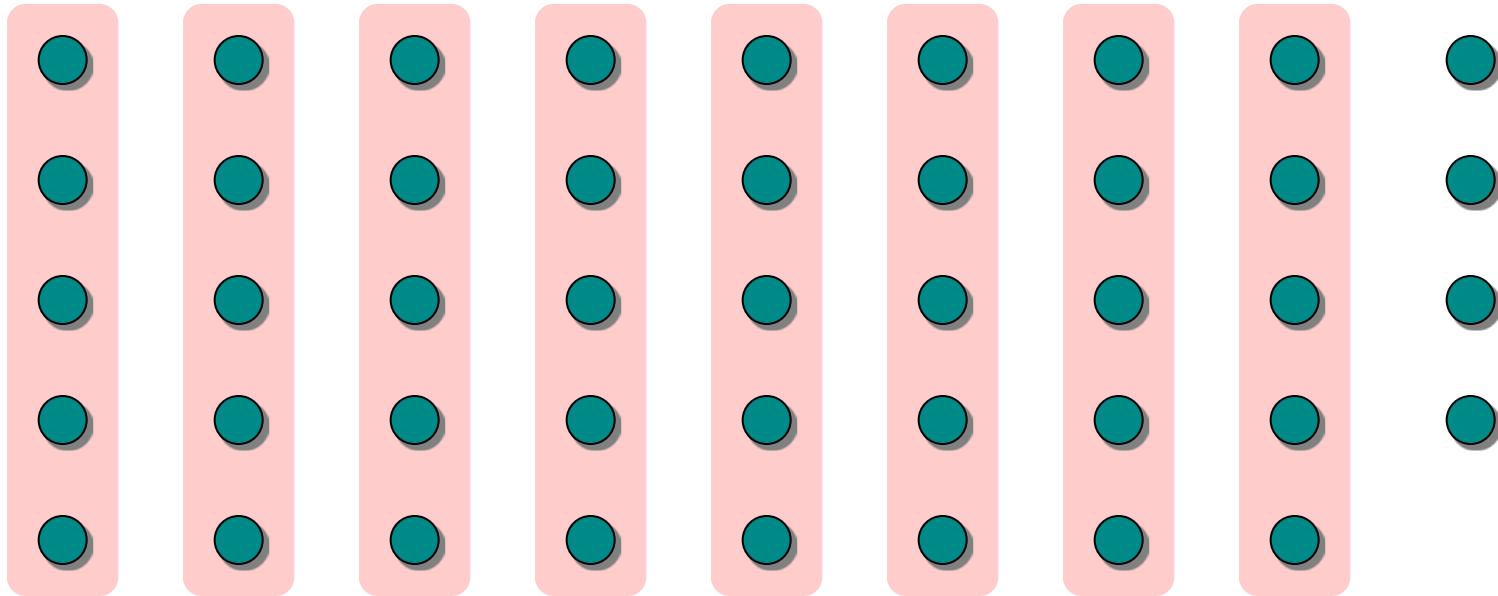


# Choosing the pivot

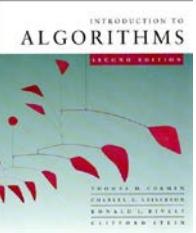




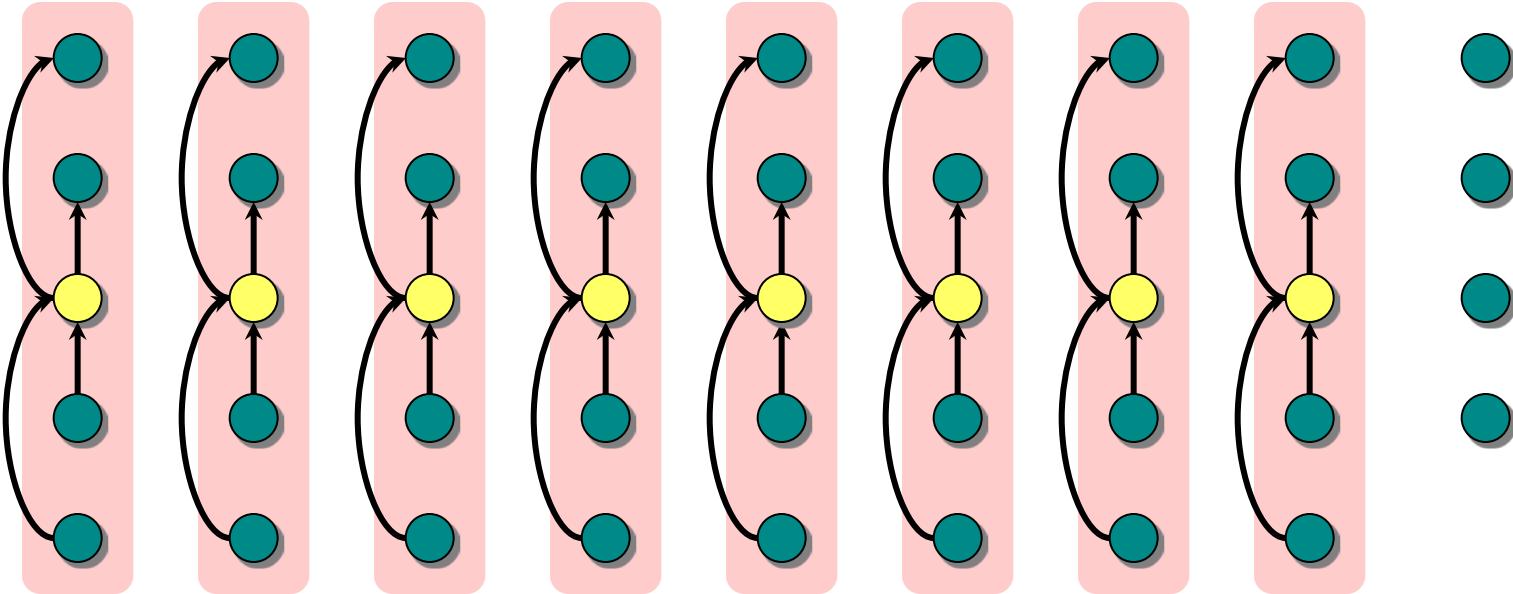
# Choosing the pivot



1. Divide the  $n$  elements into groups of 5.



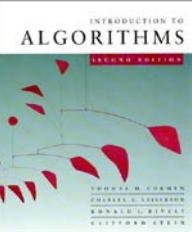
# Choosing the pivot



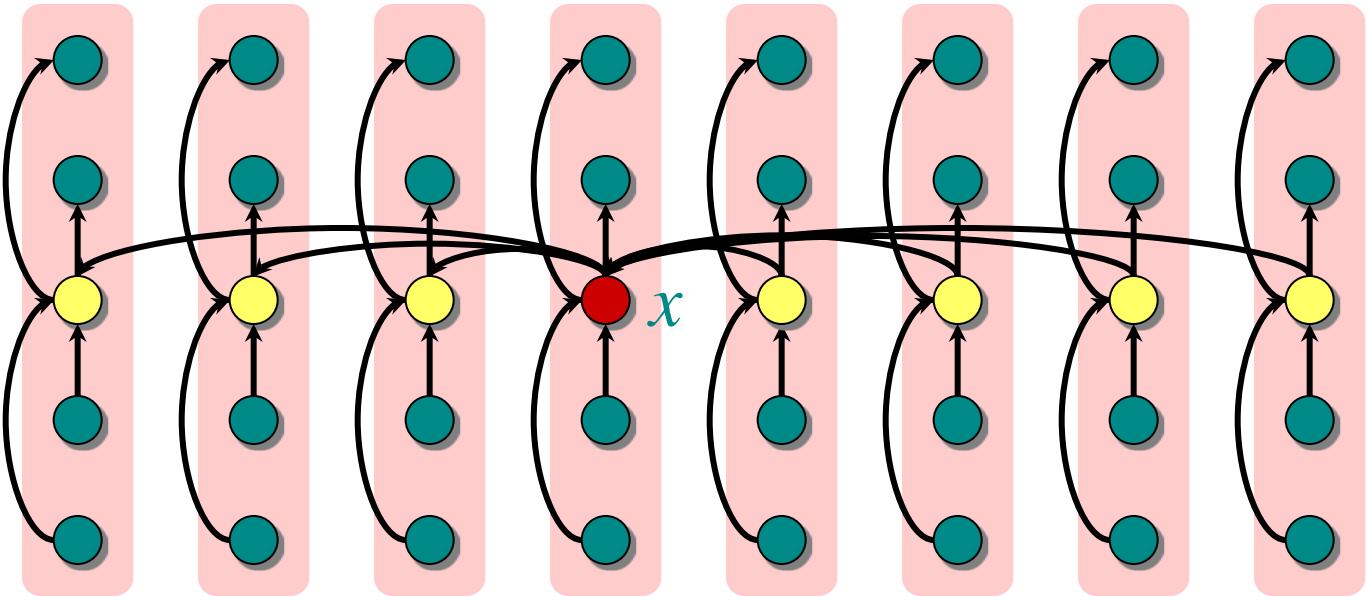
1. Divide the  $n$  elements into groups of 5. Find the median of each 5-element group by rote.

*lesser*  
  
*greater*

The diagram shows a vertical column of five teal circles. The top three circles are connected by a horizontal line with arrows pointing upwards, labeled "lesser". The bottom two circles are connected by a horizontal line with arrows pointing downwards, labeled "greater".



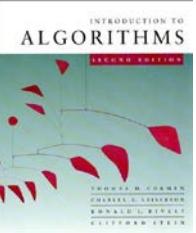
# Choosing the pivot



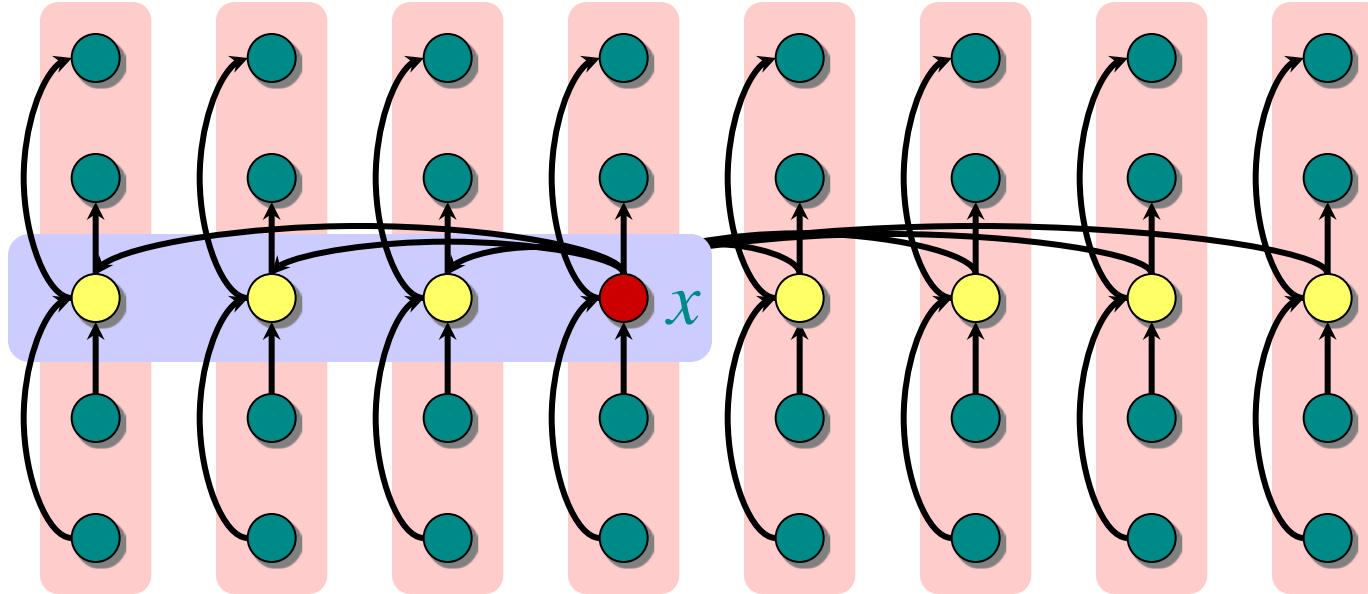
1. Divide the  $n$  elements into groups of 5. Find the median of each 5-element group by rote.
2. Recursively SELECT the median  $x$  of the  $\lfloor n/5 \rfloor$  group medians to be the pivot.

*lesser*  
  
*greater*

The text "lesser" is aligned with the top teal circle in the diagram above. The text "greater" is aligned with the bottom teal circle in the diagram below. Between the two lines of text is a small diagram consisting of a teal circle with a horizontal line extending from its right side, pointing upwards towards the "lesser" text.

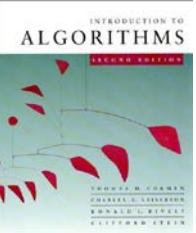


# Analysis



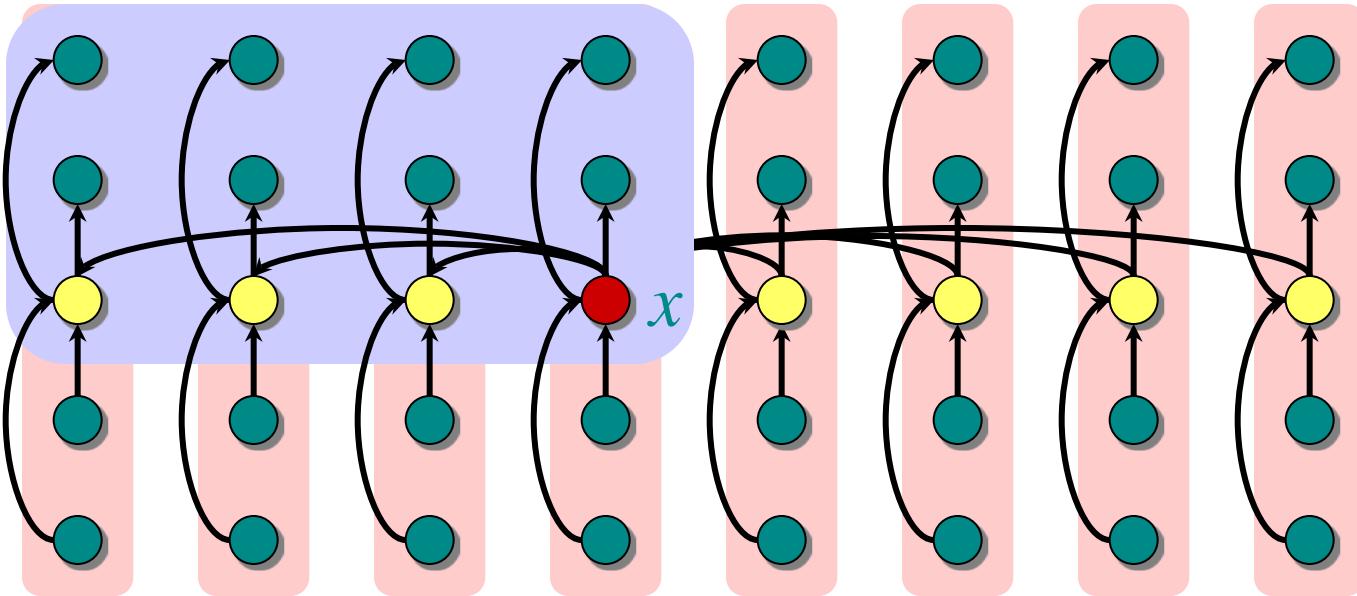
At least half the group medians are  $\leq x$ , which is at least  $\lfloor \lfloor n/5 \rfloor / 2 \rfloor = \lfloor n/10 \rfloor$  group medians.

*lesser*  
  
*greater*



# Analysis

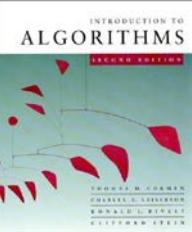
(Assume all elements are distinct.)



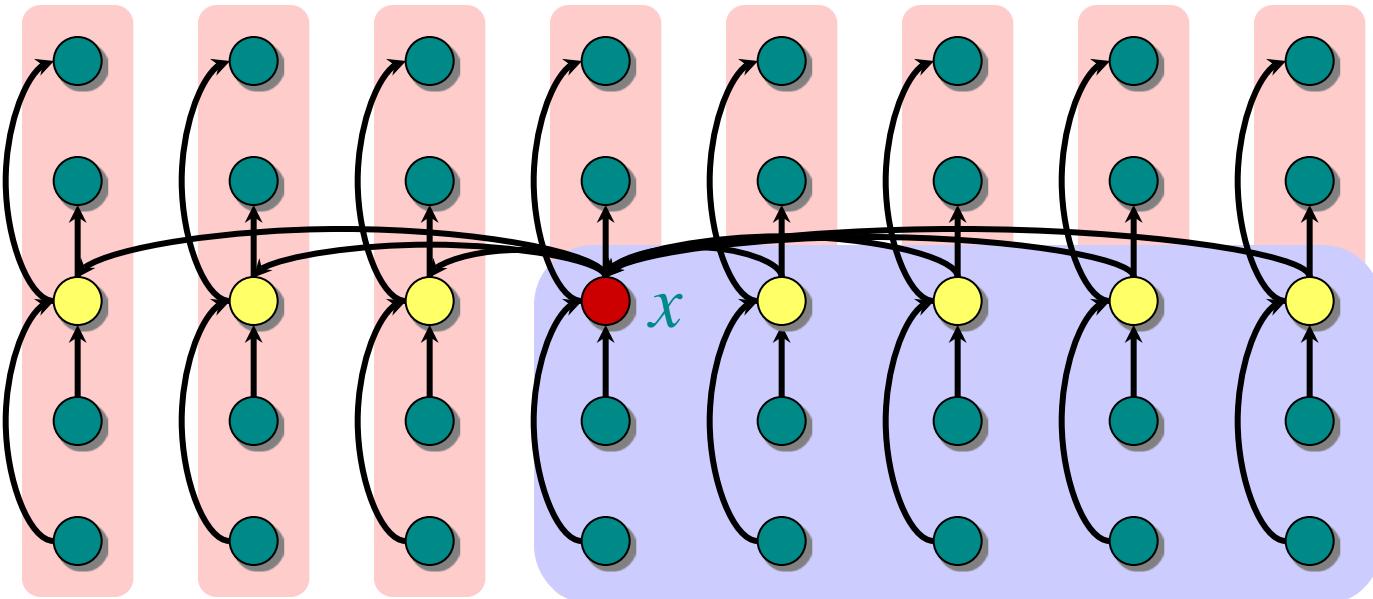
At least half the group medians are  $\leq x$ , which is at least  $\lfloor \lfloor n/5 \rfloor / 2 \rfloor = \lfloor n/10 \rfloor$  group medians.

- Therefore, at least  $3\lfloor n/10 \rfloor$  elements are  $\leq x$ .

*lesser*  
  
*greater*



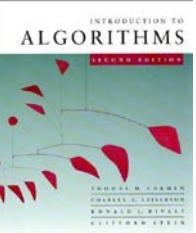
# Analysis

 (Assume all elements are distinct.)

At least half the group medians are  $\leq x$ , which is at least  $\lfloor \lfloor n/5 \rfloor / 2 \rfloor = \lfloor n/10 \rfloor$  group medians.

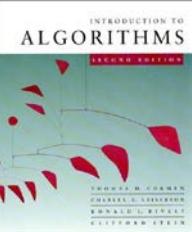
- Therefore, at least  $3\lfloor n/10 \rfloor$  elements are  $\leq x$ .
- Similarly, at least  $3\lfloor n/10 \rfloor$  elements are  $\geq x$ .

*lesser*  
  
*greater*



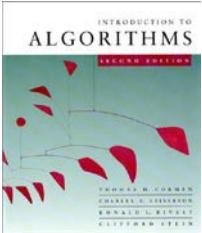
# Minor simplification

- For  $n \geq 50$ , we have  $3\lfloor n/10 \rfloor \geq n/4$ .
- Therefore, for  $n \geq 50$  the recursive call to SELECT in Step 4 is executed recursively on  $\leq 3n/4$  elements.
- Thus, the recurrence for running time can assume that Step 4 takes time  $T(3n/4)$  in the worst case.
- For  $n < 50$ , we know that the worst-case time is  $T(n) = \Theta(1)$ .



# Developing the recurrence

|             |   |
|-------------|---|
| $T(n)$      | SELECT( $i, n$ )  |
| $\Theta(n)$ | { 1. Divide the $n$ elements into groups of 5. Find the median of each 5-element group by rote.   |
| $T(n/5)$    | { 2. Recursively SELECT the median $x$ of the $\lfloor n/5 \rfloor$ group medians to be the pivot.  |
| $\Theta(n)$ | 3. Partition around the pivot $x$ . Let $k = \text{rank}(x)$ .  |
| $T(3n/4)$   | { 4. if $i = k$ then return $x$<br>elseif $i < k$<br>then recursively SELECT the $i$ th<br>smallest element in the lower part<br>else recursively SELECT the $(i-k)$ th<br>smallest element in the upper part |



# Solving the recurrence

$$T(n) = T\left(\frac{1}{5}n\right) + T\left(\frac{3}{4}n\right) + \Theta(n)$$

---

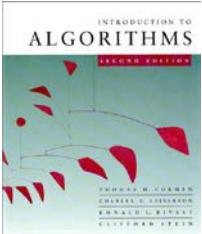
---

**Substitution:**

$$T(n) \leq cn$$

$$\begin{aligned} T(n) &\leq \frac{1}{5}cn + \frac{3}{4}cn + \Theta(n) \\ &= \frac{19}{20}cn + \Theta(n) \\ &= cn - \left( \frac{1}{20}cn - \Theta(n) \right) \\ &\leq cn \quad , \end{aligned}$$

if  $c$  is chosen large enough to handle both the  $\Theta(n)$  and the initial conditions.



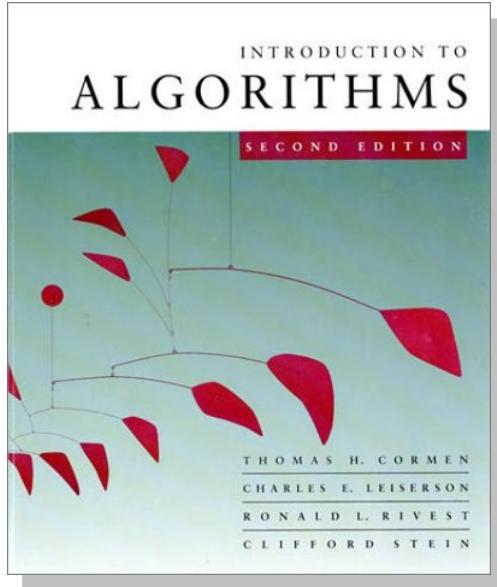
# Conclusions

- Since the work at each level of recursion is a constant fraction ( $\frac{19}{20}$ ) smaller, the work per level is a geometric series dominated by the linear work at the root.
- In practice, this algorithm runs slowly, because the constant in front of  $n$  is large.
- The randomized algorithm is far more practical.

**Exercise:** *Why not divide into groups of 3?*

# *Introduction to Algorithms*

## 6.046J/18.401J

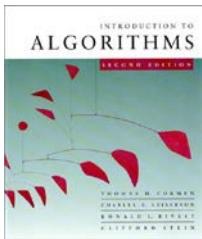


### LECTURE 7

#### Hashing I

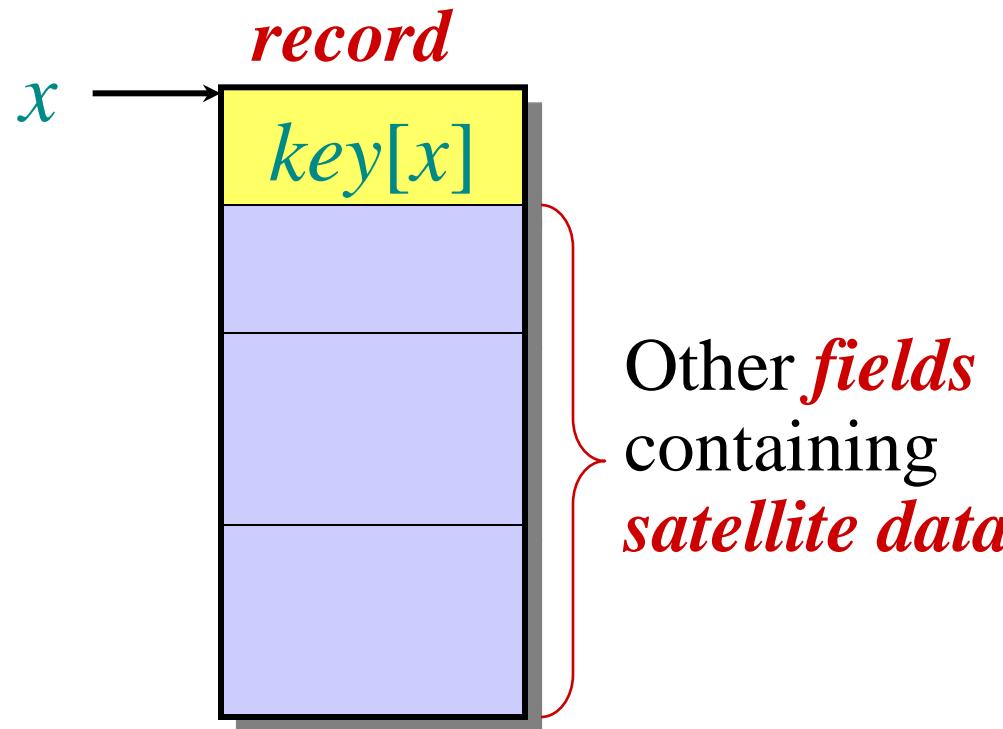
- Direct-access tables
- Resolving collisions by chaining
- Choosing hash functions
- Open addressing

Prof. Charles E. Leiserson



# Symbol-table problem

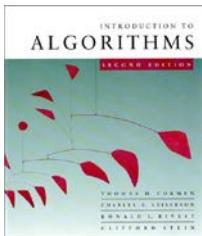
Symbol table  $S$  holding  $n$  records:



Operations on  $S$ :

- INSERT( $S, x$ )
- DELETE( $S, x$ )
- SEARCH( $S, k$ )

How should the data structure  $S$  be organized?



# Direct-access table

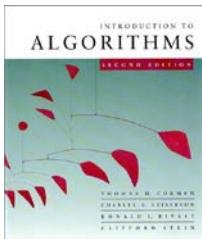
**IDEA:** Suppose that the keys are drawn from the set  $U \subseteq \{0, 1, \dots, m-1\}$ , and keys are distinct. Set up an array  $T[0 \dots m-1]$ :

$$T[k] = \begin{cases} x & \text{if } x \in K \text{ and } \text{key}[x] = k, \\ \text{NIL} & \text{otherwise.} \end{cases}$$

Then, operations take  $\Theta(1)$  time.

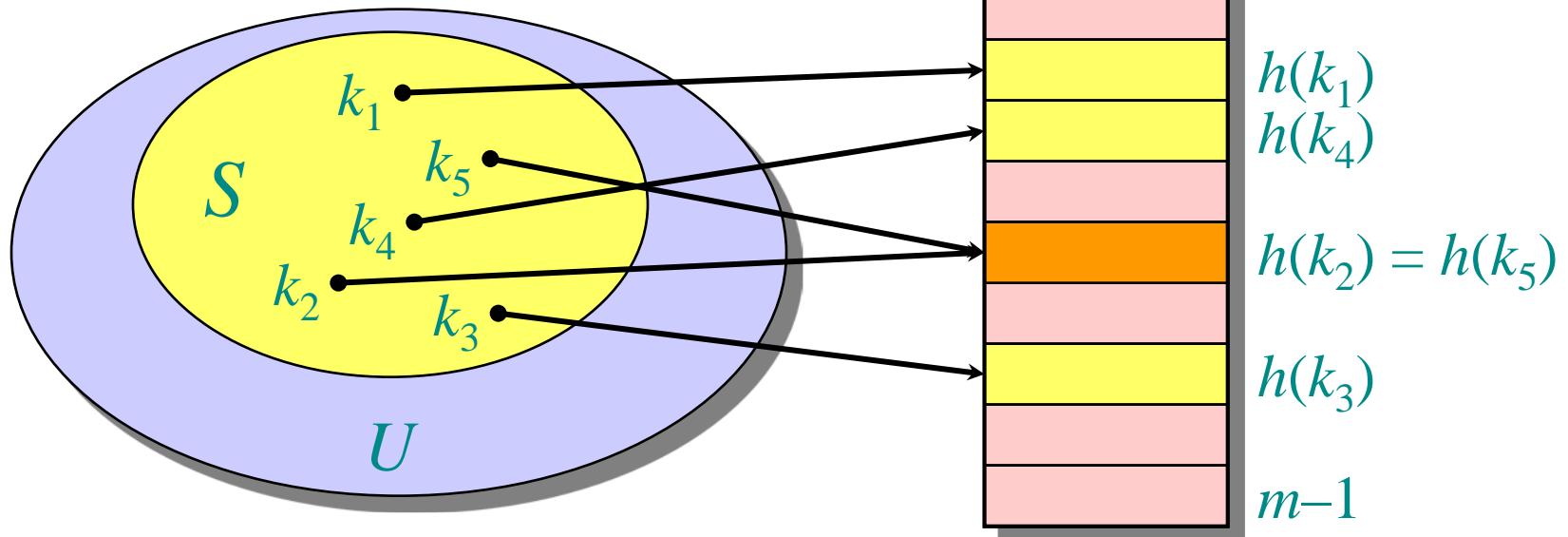
**Problem:** The range of keys can be large:

- 64-bit numbers (which represent  $18,446,744,073,709,551,616$  different keys),
- character strings (even larger!).

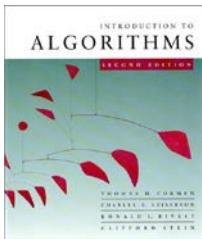


# Hash functions

**Solution:** Use a *hash function*  $h$  to map the universe  $U$  of all keys into  $\{0, 1, \dots, m-1\}$ :

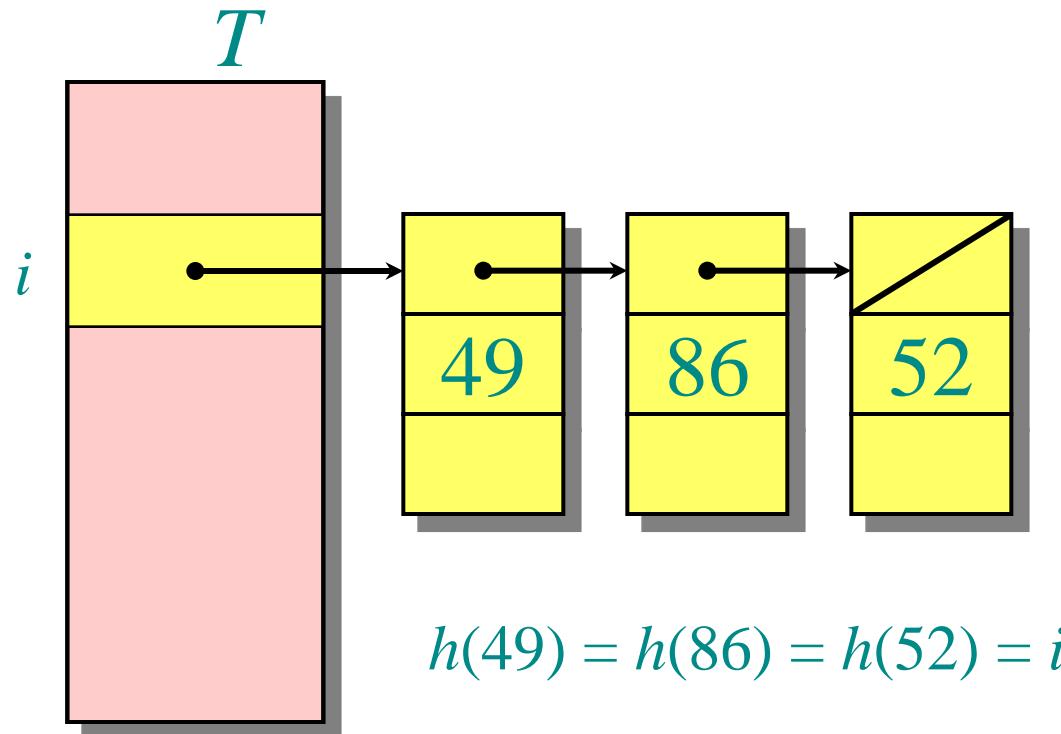


When a record to be inserted maps to an already occupied slot in  $T$ , a *collision* occurs.



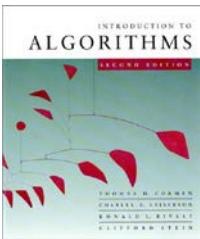
# Resolving collisions by chaining

- Link records in the same slot into a list.



*Worst case:*

- Every key hashes to the same slot.
- Access time =  $\Theta(n)$  if  $|S| = n$



# Average-case analysis of chaining

We make the assumption of *simple uniform hashing*:

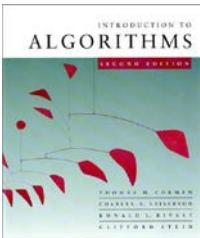
- Each key  $k \in S$  is equally likely to be hashed to any slot of table  $T$ , independent of where other keys are hashed.

Let  $n$  be the number of keys in the table, and let  $m$  be the number of slots.

Define the *load factor* of  $T$  to be

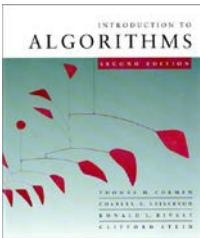
$$\alpha = n/m$$

= average number of keys per slot.



# Search cost

The expected time for an *unsuccessful* search for a record with a given key is  
=  $\Theta(1 + \alpha)$ .



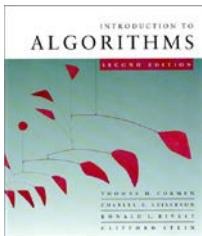
# Search cost

The expected time for an *unsuccessful* search for a record with a given key is

$$= \Theta(1 + \alpha).$$

*search  
the list*

*apply hash function  
and access slot*



# Search cost

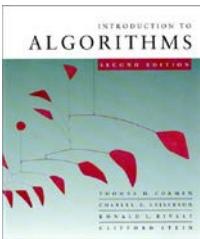
The expected time for an *unsuccessful* search for a record with a given key is

$$= \Theta(1 + \alpha).$$

*search  
the list*

*apply hash function  
and access slot*

Expected search time =  $\Theta(1)$  if  $\alpha = O(1)$ ,  
or equivalently, if  $n = O(m)$ .



# Search cost

The expected time for an *unsuccessful* search for a record with a given key is

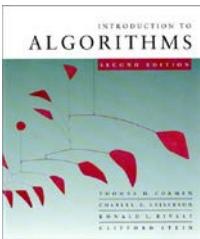
$$= \Theta(1 + \alpha)$$

*search  
the list*

*apply hash function  
and access slot*

Expected search time =  $\Theta(1)$  if  $\alpha = O(1)$ ,  
or equivalently, if  $n = O(m)$ .

A *successful* search has same asymptotic bound, but a rigorous argument is a little more complicated. (See textbook.)

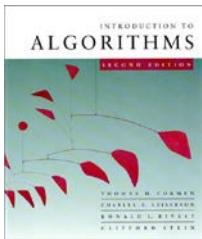


# Choosing a hash function

The assumption of simple uniform hashing is hard to guarantee, but several common techniques tend to work well in practice as long as their deficiencies can be avoided.

## Desirata:

- A good hash function should distribute the keys uniformly into the slots of the table.
- Regularity in the key distribution should not affect this uniformity.



# Division method

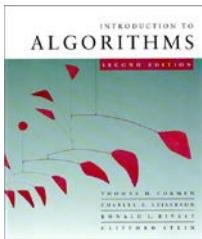
Assume all keys are integers, and define

$$h(k) = k \bmod m.$$

**Deficiency:** Don't pick an  $m$  that has a small divisor  $d$ . A preponderance of keys that are congruent modulo  $d$  can adversely affect uniformity.

**Extreme deficiency:** If  $m = 2^r$ , then the hash doesn't even depend on all the bits of  $k$ :

- If  $k = 10110001110\underset{r}{\underbrace{11010}}_2$  and  $r = 6$ , then  
$$h(k) = 011010_2 . \quad h(k)$$



# Division method (continued)

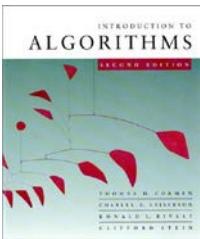
$$h(k) = k \bmod m.$$

Pick  $m$  to be a prime not too close to a power of 2 or 10 and not otherwise used prominently in the computing environment.

## Annoyance:

- Sometimes, making the table size a prime is inconvenient.

But, this method is popular, although the next method we'll see is usually superior.



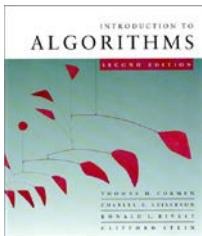
# Multiplication method

Assume that all keys are integers,  $m = 2^r$ , and our computer has  $w$ -bit words. Define

$$h(k) = (A \cdot k \bmod 2^w) \text{ rsh } (w - r),$$

where `rsh` is the “bitwise right-shift” operator and  $A$  is an odd integer in the range  $2^{w-1} < A < 2^w$ .

- Don’t pick  $A$  too close to  $2^{w-1}$  or  $2^w$ .
- Multiplication modulo  $2^w$  is fast compared to division.
- The `rsh` operator is fast.



# Multiplication method example

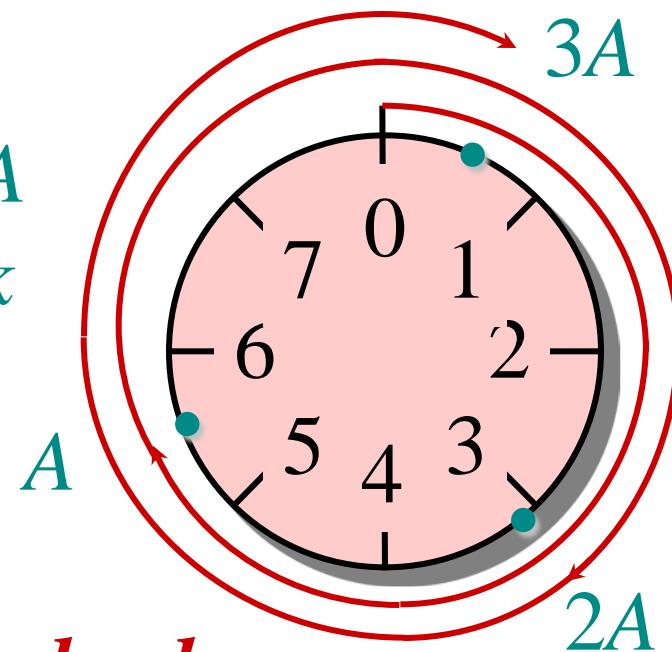
$$h(k) = (A \cdot k \bmod 2^w) \text{ rsh } (w - r)$$

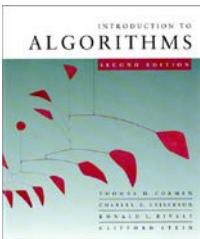
Suppose that  $m = 8 = 2^3$  and that our computer has  $w = 7$ -bit words:

$$\begin{array}{r} & 1011001 \\ \times & 1101011 \\ \hline & 1001010011 \end{array}$$

$\underbrace{\hspace{1cm}}_{h(k)}$

*Modular wheel*





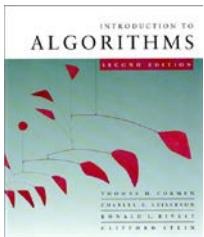
# Resolving collisions by open addressing

No storage is used outside of the hash table itself.

- Insertion systematically probes the table until an empty slot is found.
- The hash function depends on both the key and probe number:

$$h : U \times \{0, 1, \dots, m-1\} \rightarrow \{0, 1, \dots, m-1\}.$$

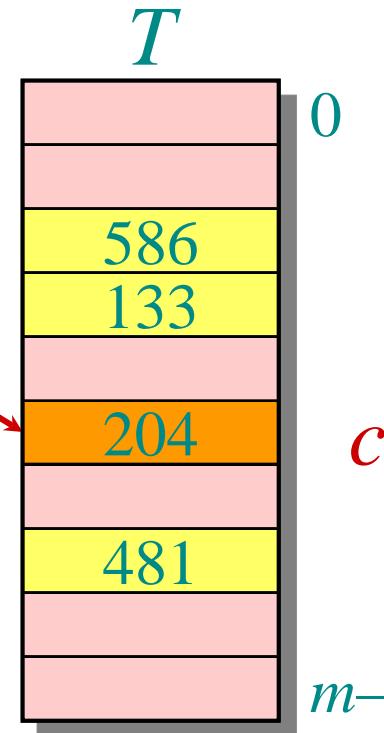
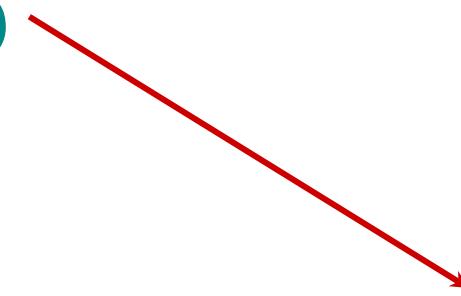
- The probe sequence  $\langle h(k,0), h(k,1), \dots, h(k,m-1) \rangle$  should be a permutation of  $\{0, 1, \dots, m-1\}$ .
- The table may fill up, and deletion is difficult (but not impossible).

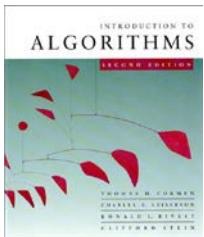


# Example of open addressing

Insert key  $k = 496$ :

0. Probe  $h(496, 0)$

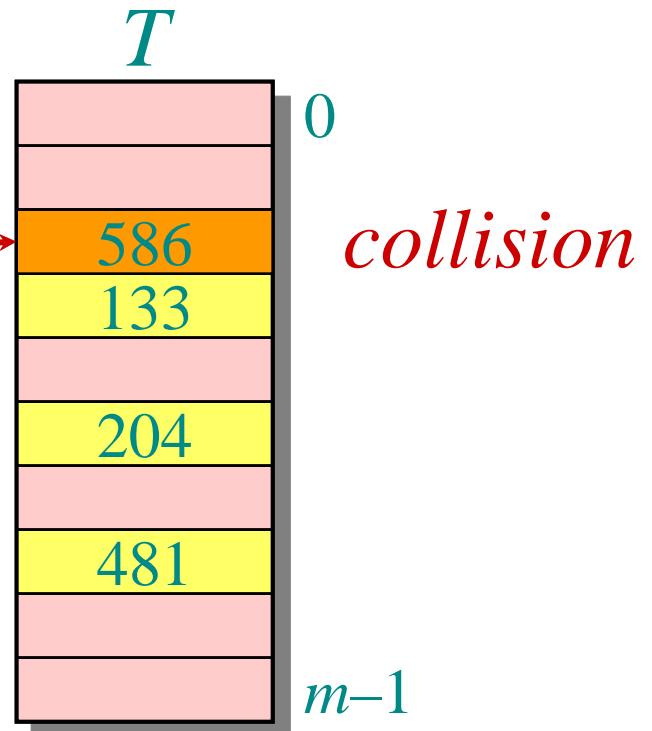


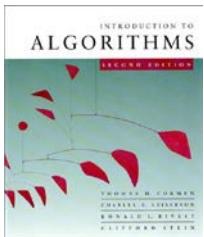


# Example of open addressing

Insert key  $k = 496$ :

0. Probe  $h(496,0)$
1. Probe  $h(496,1)$

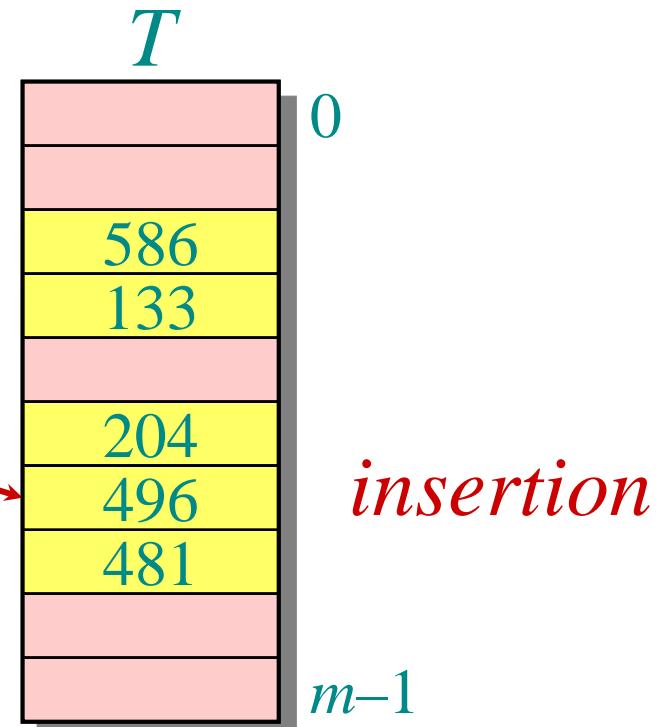
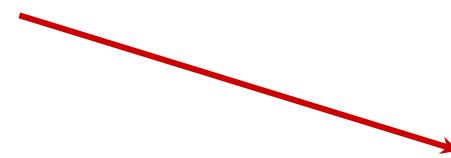


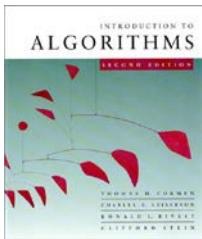


# Example of open addressing

Insert key  $k = 496$ :

0. Probe  $h(496,0)$
1. Probe  $h(496,1)$
2. Probe  $h(496,2)$

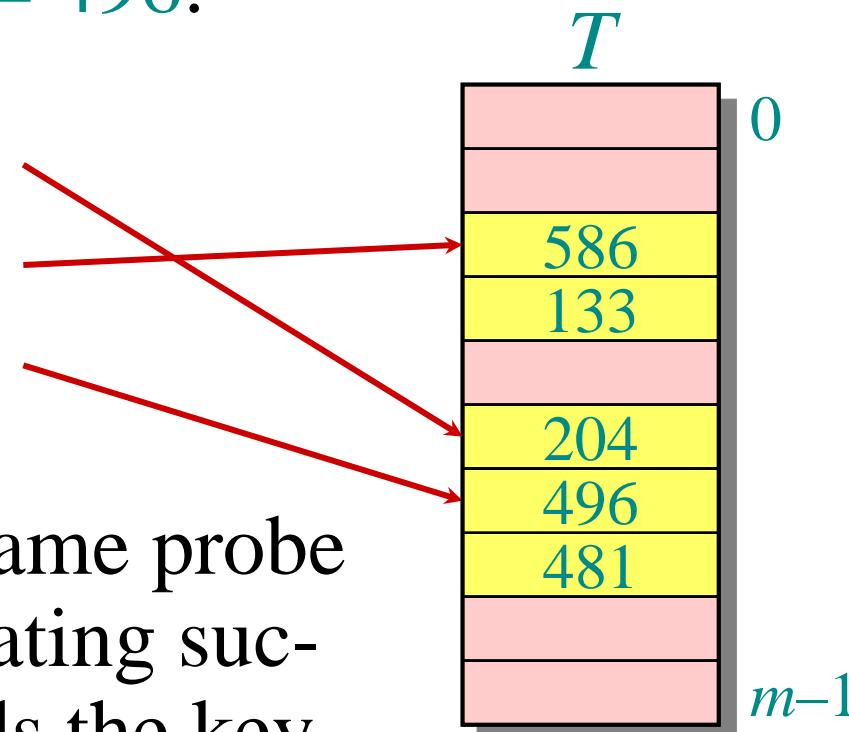




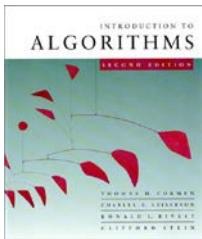
# Example of open addressing

Search for key  $k = 496$ :

0. Probe  $h(496,0)$
1. Probe  $h(496,1)$
2. Probe  $h(496,2)$



Search uses the same probe sequence, terminating successfully if it finds the key and unsuccessfully if it encounters an empty slot.



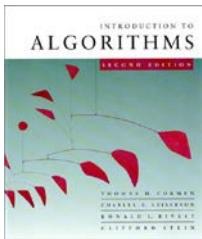
# Probing strategies

## Linear probing:

Given an ordinary hash function  $h'(k)$ , linear probing uses the hash function

$$h(k, i) = (h'(k) + i) \bmod m.$$

This method, though simple, suffers from ***primary clustering***, where long runs of occupied slots build up, increasing the average search time. Moreover, the long runs of occupied slots tend to get longer.



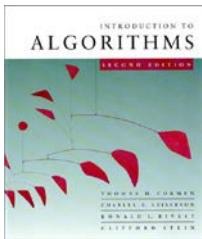
# Probing strategies

## Double hashing

Given two ordinary hash functions  $h_1(k)$  and  $h_2(k)$ , double hashing uses the hash function

$$h(k,i) = (h_1(k) + i \cdot h_2(k)) \bmod m.$$

This method generally produces excellent results, but  $h_2(k)$  must be relatively prime to  $m$ . One way is to make  $m$  a power of 2 and design  $h_2(k)$  to produce only odd numbers.

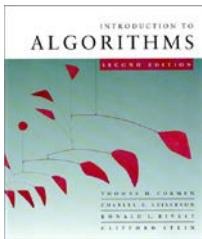


# Analysis of open addressing

We make the assumption of ***uniform hashing***:

- Each key is equally likely to have any one of the  $m!$  permutations as its probe sequence.

**Theorem.** Given an open-addressed hash table with load factor  $\alpha = n/m < 1$ , the expected number of probes in an unsuccessful search is at most  $1/(1-\alpha)$ .

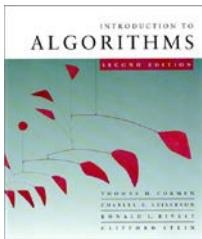


# Proof of the theorem

*Proof.*

- At least one probe is always necessary.
- With probability  $n/m$ , the first probe hits an occupied slot, and a second probe is necessary.
- With probability  $(n-1)/(m-1)$ , the second probe hits an occupied slot, and a third probe is necessary.
- With probability  $(n-2)/(m-2)$ , the third probe hits an occupied slot, etc.

Observe that  $\frac{n-i}{m-i} < \frac{n}{m} = \alpha$  for  $i = 1, 2, \dots, n$ .



# Proof (continued)

Therefore, the expected number of probes is

$$1 + \frac{n}{m} \left( 1 + \frac{n-1}{m-1} \left( 1 + \frac{n-2}{m-2} \left( \dots \left( 1 + \frac{1}{m-n+1} \right) \dots \right) \right) \right)$$

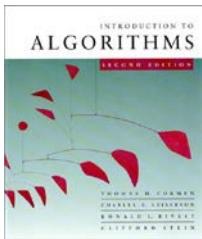
$$\leq 1 + \alpha (1 + \alpha (1 + \alpha (\dots (1 + \alpha) \dots)))$$

$$\leq 1 + \alpha + \alpha^2 + \alpha^3 + \dots$$

$$= \sum_{i=0}^{\infty} \alpha^i$$

$$= \frac{1}{1-\alpha}.$$
 □

*The textbook has a more rigorous proof and an analysis of successful searches.*

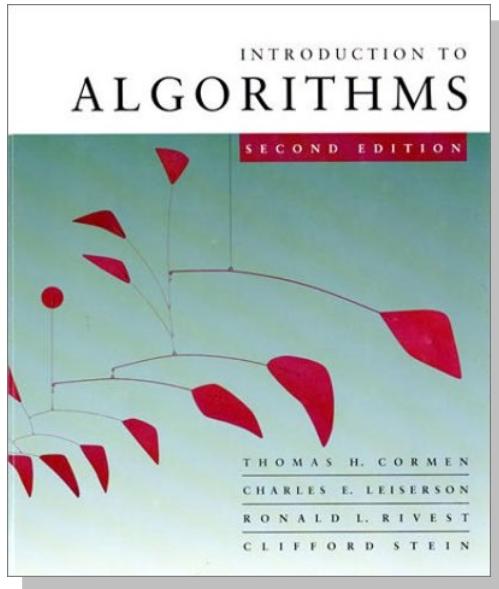


# Implications of the theorem

- If  $\alpha$  is constant, then accessing an open-addressed hash table takes constant time.
- If the table is half full, then the expected number of probes is  $1/(1-0.5) = 2$ .
- If the table is 90% full, then the expected number of probes is  $1/(1-0.9) = 10$ .

# *Introduction to Algorithms*

## 6.046J/18.401J

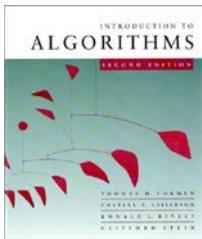


### LECTURE 8

#### Hashing II

- Universal hashing
- Universality theorem
- Constructing a set of universal hash functions
- Perfect hashing

Prof. Charles E. Leiserson



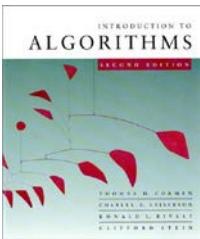
# A weakness of hashing

**Problem:** For any hash function  $h$ , a set of keys exists that can cause the average access time of a hash table to skyrocket.

- An adversary can pick all keys from  $\{k \in U : h(k) = i\}$  for some slot  $i$ .

**IDEA:** Choose the hash function at random, independently of the keys.

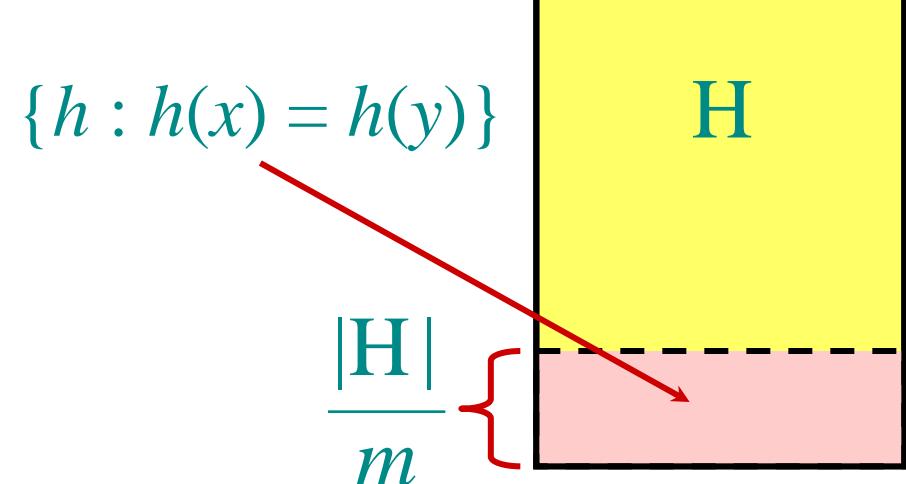
- Even if an adversary can see your code, he or she cannot find a bad set of keys, since he or she doesn't know exactly which hash function will be chosen.

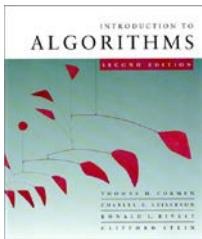


# Universal hashing

**Definition.** Let  $U$  be a universe of keys, and let  $H$  be a finite collection of hash functions, each mapping  $U$  to  $\{0, 1, \dots, m-1\}$ . We say  $H$  is *universal* if for all  $x, y \in U$ , where  $x \neq y$ , we have  $|\{h \in H : h(x) = h(y)\}| \leq |H|/m$ .

That is, the chance  
of a collision  
between  $x$  and  $y$  is  
 $\leq 1/m$  if we choose  $h$   
randomly from  $H$ .

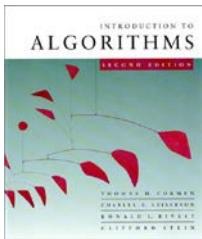




# Universality is good

**Theorem.** Let  $h$  be a hash function chosen (uniformly) at random from a universal set  $\mathbb{H}$  of hash functions. Suppose  $h$  is used to hash  $n$  arbitrary keys into the  $m$  slots of a table  $T$ . Then, for a given key  $x$ , we have

$$E[\#\text{collisions with } x] < n/m.$$

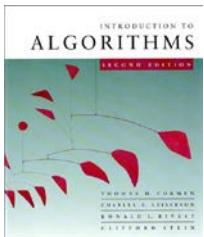


# Proof of theorem

*Proof.* Let  $C_x$  be the random variable denoting the total number of collisions of keys in  $T$  with  $x$ , and let

$$c_{xy} = \begin{cases} 1 & \text{if } h(x) = h(y), \\ 0 & \text{otherwise.} \end{cases}$$

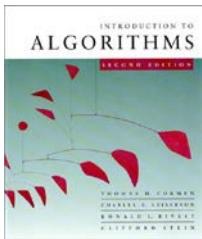
Note:  $E[c_{xy}] = 1/m$  and  $C_x = \sum_{y \in T - \{x\}} c_{xy}$ .



# Proof (continued)

$$E[C_x] = E\left[ \sum_{y \in T - \{x\}} c_{xy} \right]$$

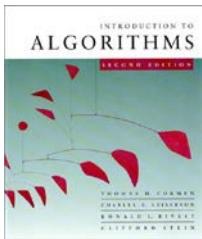
- Take expectation of both sides.



# Proof (continued)

$$\begin{aligned}E[C_x] &= E\left[\sum_{y \in T - \{x\}} c_{xy}\right] \\&= \sum_{y \in T - \{x\}} E[c_{xy}]\end{aligned}$$

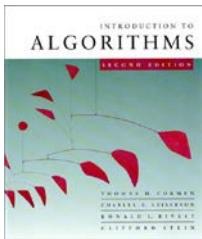
- Take expectation of both sides.
- Linearity of expectation.



# Proof (continued)

$$\begin{aligned}E[C_x] &= E\left[\sum_{y \in T - \{x\}} c_{xy}\right] \\&= \sum_{y \in T - \{x\}} E[c_{xy}] \\&= \sum_{y \in T - \{x\}} 1/m\end{aligned}$$

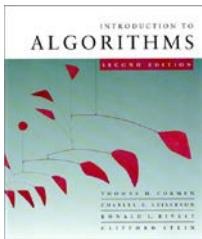
- Take expectation of both sides.
- Linearity of expectation.
- $E[c_{xy}] = 1/m$ .



# Proof (continued)

$$\begin{aligned}E[C_x] &= E\left[\sum_{y \in T - \{x\}} c_{xy}\right] \\&= \sum_{y \in T - \{x\}} E[c_{xy}] \\&= \sum_{y \in T - \{x\}} 1/m \\&= \frac{n-1}{m}.\end{aligned}$$

- Take expectation of both sides.
- Linearity of expectation.
- $E[c_{xy}] = 1/m$ .
- Algebra.



# Constructing a set of universal hash functions

Let  $m$  be prime. Decompose key  $k$  into  $r + 1$  digits, each with value in the set  $\{0, 1, \dots, m-1\}$ . That is, let  $k = \langle k_0, k_1, \dots, k_r \rangle$ , where  $0 \leq k_i < m$ .

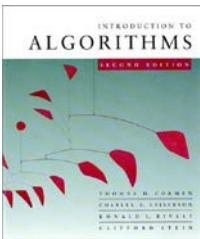
## Randomized strategy:

Pick  $a = \langle a_0, a_1, \dots, a_r \rangle$  where each  $a_i$  is chosen randomly from  $\{0, 1, \dots, m-1\}$ .

Define  $h_a(k) = \sum_{i=0}^r a_i k_i \bmod m$ .

*Dot product,  
modulo  $m$*

How big is  $H = \{h_a\}$ ?  $|H| = m^{r+1}$ . ← **REMEMBER THIS!**



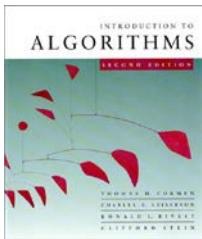
# Universality of dot-product hash functions

**Theorem.** The set  $H = \{h_a\}$  is universal.

*Proof.* Suppose that  $x = \langle x_0, x_1, \dots, x_r \rangle$  and  $y = \langle y_0, y_1, \dots, y_r \rangle$  be distinct keys. Thus, they differ in at least one digit position, wlog position 0. For how many  $h_a \in H$  do  $x$  and  $y$  collide?

We must have  $h_a(x) = h_a(y)$ , which implies that

$$\sum_{i=0}^r a_i x_i \equiv \sum_{i=0}^r a_i y_i \pmod{m}.$$



# Proof (continued)

Equivalently, we have

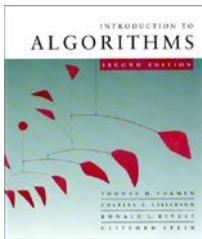
$$\sum_{i=0}^r a_i(x_i - y_i) \equiv 0 \pmod{m}$$

or

$$a_0(x_0 - y_0) + \sum_{i=1}^r a_i(x_i - y_i) \equiv 0 \pmod{m},$$

which implies that

$$a_0(x_0 - y_0) \equiv -\sum_{i=1}^r a_i(x_i - y_i) \pmod{m}.$$



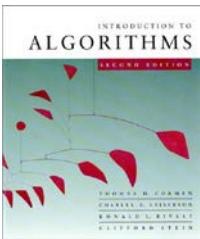
# Fact from number theory

**Theorem.** Let  $m$  be prime. For any  $z \in \mathbb{Z}_m$  such that  $z \neq 0$ , there exists a unique  $z^{-1} \in \mathbb{Z}_m$  such that

$$z \cdot z^{-1} \equiv 1 \pmod{m}.$$

**Example:**  $m = 7$ .

|          |   |   |   |   |   |   |
|----------|---|---|---|---|---|---|
| $z$      | 1 | 2 | 3 | 4 | 5 | 6 |
| $z^{-1}$ | 1 | 4 | 5 | 2 | 3 | 6 |



# Back to the proof

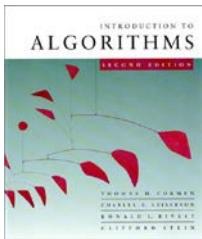
We have

$$a_0(x_0 - y_0) \equiv -\sum_{i=1}^r a_i(x_i - y_i) \pmod{m},$$

and since  $x_0 \neq y_0$ , an inverse  $(x_0 - y_0)^{-1}$  must exist, which implies that

$$a_0 \equiv \left( -\sum_{i=1}^r a_i(x_i - y_i) \right) \cdot (x_0 - y_0)^{-1} \pmod{m}.$$

Thus, for any choices of  $a_1, a_2, \dots, a_r$ , exactly one choice of  $a_0$  causes  $x$  and  $y$  to collide.



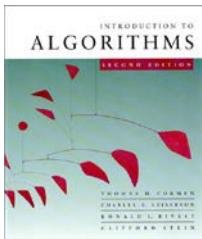
# Proof (completed)

**Q.** How many  $h_a$ 's cause  $x$  and  $y$  to collide?

**A.** There are  $m$  choices for each of  $a_1, a_2, \dots, a_r$ , but once these are chosen, exactly one choice for  $a_0$  causes  $x$  and  $y$  to collide, namely

$$a_0 = \left( \left( - \sum_{i=1}^r a_i (x_i - y_i) \right) \cdot (x_0 - y_0)^{-1} \right) \bmod m.$$

Thus, the number of  $h_a$ 's that cause  $x$  and  $y$  to collide is  $m^r \cdot 1 = m^r = |\mathbb{H}|/m$ . □

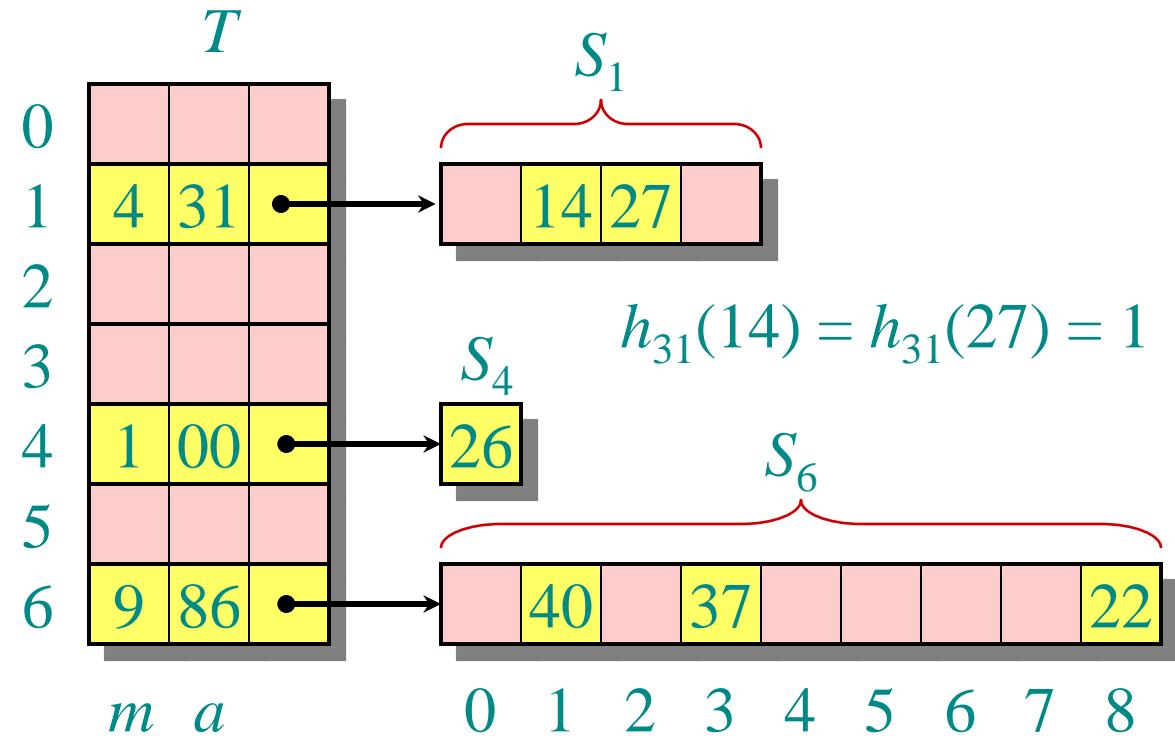


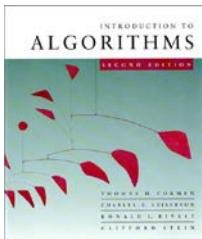
# Perfect hashing

Given a set of  $n$  keys, construct a static hash table of size  $m = O(n)$  such that **SEARCH** takes  $\Theta(1)$  time in the *worst case*.

**IDEA:** Two-level scheme with universal hashing at both levels.

*No collisions at level 2!*



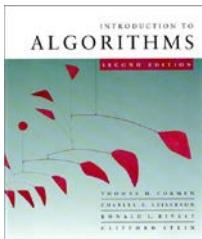


# Collisions at level 2

**Theorem.** Let  $H$  be a class of universal hash functions for a table of size  $m = n^2$ . Then, if we use a random  $h \in H$  to hash  $n$  keys into the table, the expected number of collisions is at most  $1/2$ .

*Proof.* By the definition of universality, the probability that 2 given keys in the table collide under  $h$  is  $1/m = 1/n^2$ . Since there are  $\binom{n}{2}$  pairs of keys that can possibly collide, the expected number of collisions is

$$\binom{n}{2} \cdot \frac{1}{n^2} = \frac{n(n-1)}{2} \cdot \frac{1}{n^2} < \frac{1}{2}. \quad \square$$



# No collisions at level 2

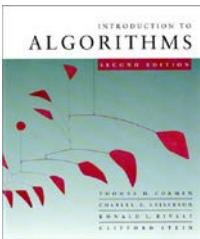
**Corollary.** The probability of no collisions is at least  $1/2$ .

*Proof.* **Markov's inequality** says that for any nonnegative random variable  $X$ , we have

$$\Pr\{X \geq t\} \leq E[X]/t.$$

Applying this inequality with  $t = 1$ , we find that the probability of 1 or more collisions is at most  $1/2$ . □

*Thus, just by testing random hash functions in  $H$ , we'll quickly find one that works.*



# Analysis of storage

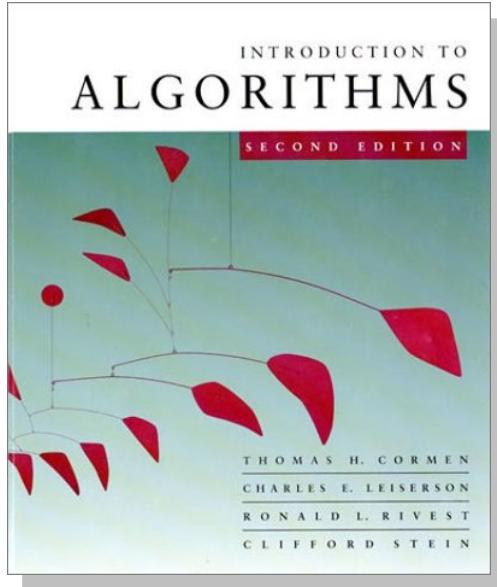
For the level-1 hash table  $T$ , choose  $m = n$ , and let  $n_i$  be random variable for the number of keys that hash to slot  $i$  in  $T$ . By using  $n_i^2$  slots for the level-2 hash table  $S_i$ , the expected total storage required for the two-level scheme is therefore

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

since the analysis is identical to the analysis from recitation of the expected running time of bucket sort. (For a probability bound, apply Markov.)

# *Introduction to Algorithms*

## 6.046J/18.401J

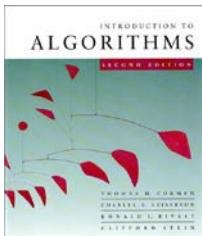


### LECTURE 9

#### Randomly built binary search trees

- Expected node depth
- Analyzing height
  - Convexity lemma
  - Jensen's inequality
  - Exponential height
- Post mortem

Prof. Erik Demaine



# Binary-search-tree sort

$T \leftarrow \emptyset$

▷ Create an empty BST

**for**  $i = 1$  to  $n$

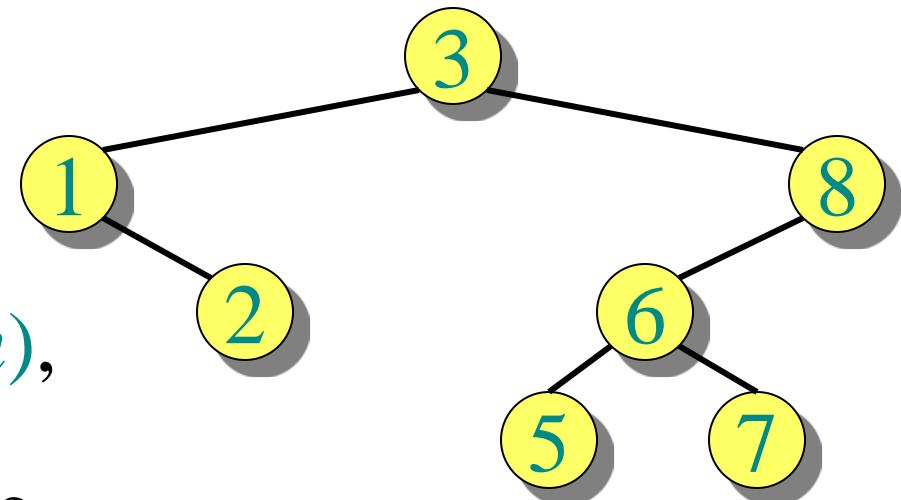
**do** TREE-INSERT( $T, A[i]$ )

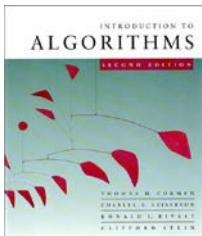
Perform an inorder tree walk of  $T$ .

## Example:

$A = [3 \ 1 \ 8 \ 2 \ 6 \ 7 \ 5]$

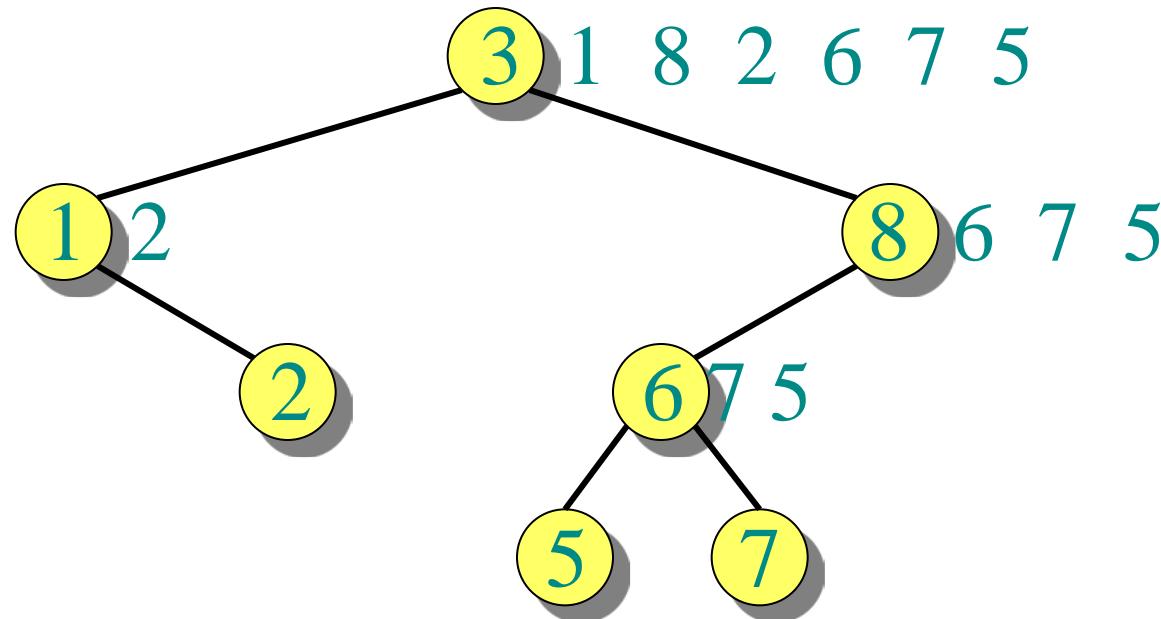
Tree-walk time =  $O(n)$ ,  
but how long does it  
take to build the BST?



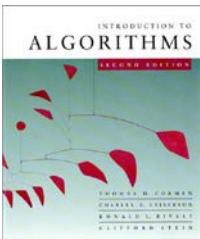


# Analysis of BST sort

BST sort performs the same comparisons as quicksort, but in a different order!



The expected time to build the tree is asymptotically the same as the running time of quicksort.



# Node depth

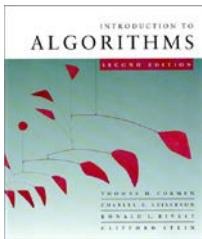
The depth of a node = the number of comparisons made during TREE-INSERT. Assuming all input permutations are equally likely, we have

Average node depth

$$= \frac{1}{n} E \left[ \sum_{i=1}^n (\# \text{comparisons to insert node } i) \right]$$

$$= \frac{1}{n} O(n \lg n) \quad (\text{quicksort analysis})$$

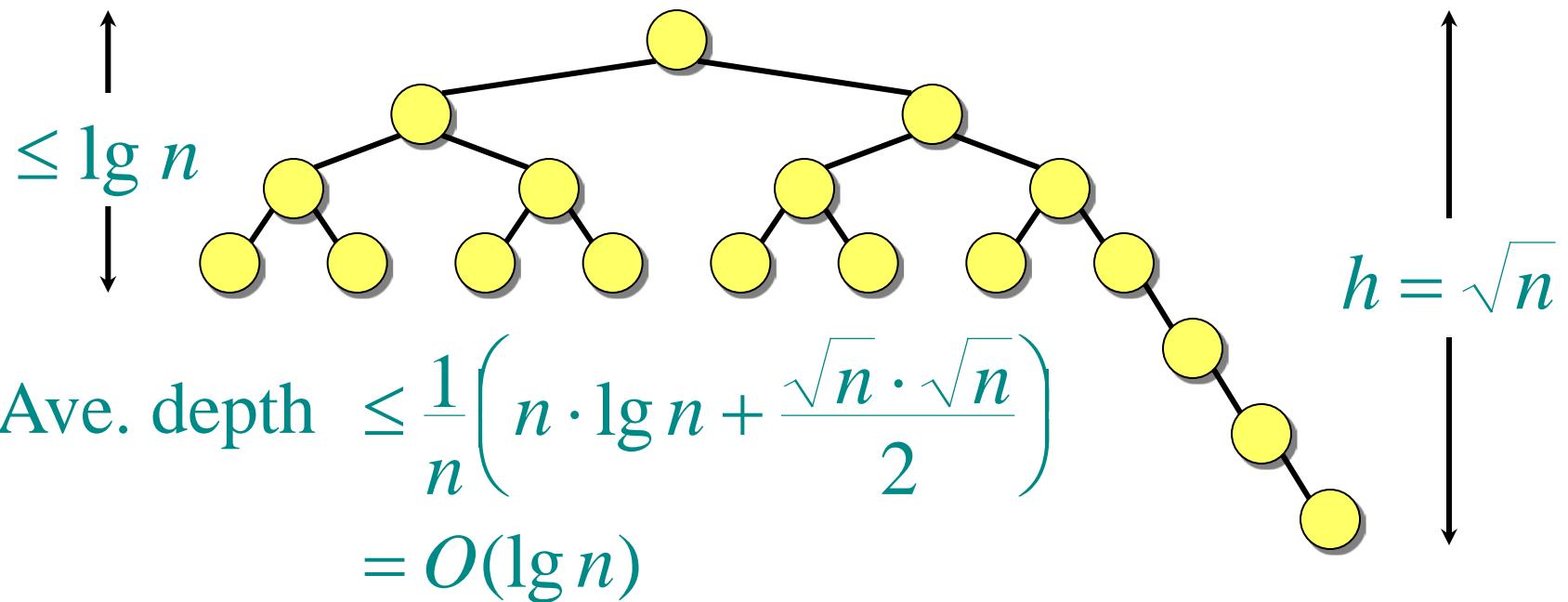
$$= O(\lg n) .$$

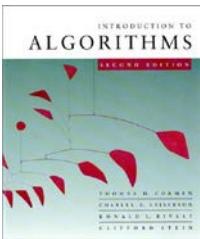


# Expected tree height

But, average node depth of a randomly built BST  $= O(\lg n)$  does not necessarily mean that its expected height is also  $O(\lg n)$  (although it is).

## Example.

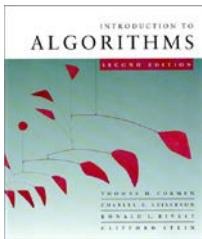




# Height of a randomly built binary search tree

## Outline of the analysis:

- Prove *Jensen's inequality*, which says that  $f(E[X]) \leq E[f(X)]$  for any convex function  $f$  and random variable  $X$ .
- Analyze the *exponential height* of a randomly built BST on  $n$  nodes, which is the random variable  $Y_n = 2^{X_n}$ , where  $X_n$  is the random variable denoting the height of the BST.
- Prove that  $2^{E[X_n]} \leq E[2^{X_n}] = E[Y_n] = O(n^3)$ , and hence that  $E[X_n] = O(\lg n)$ .

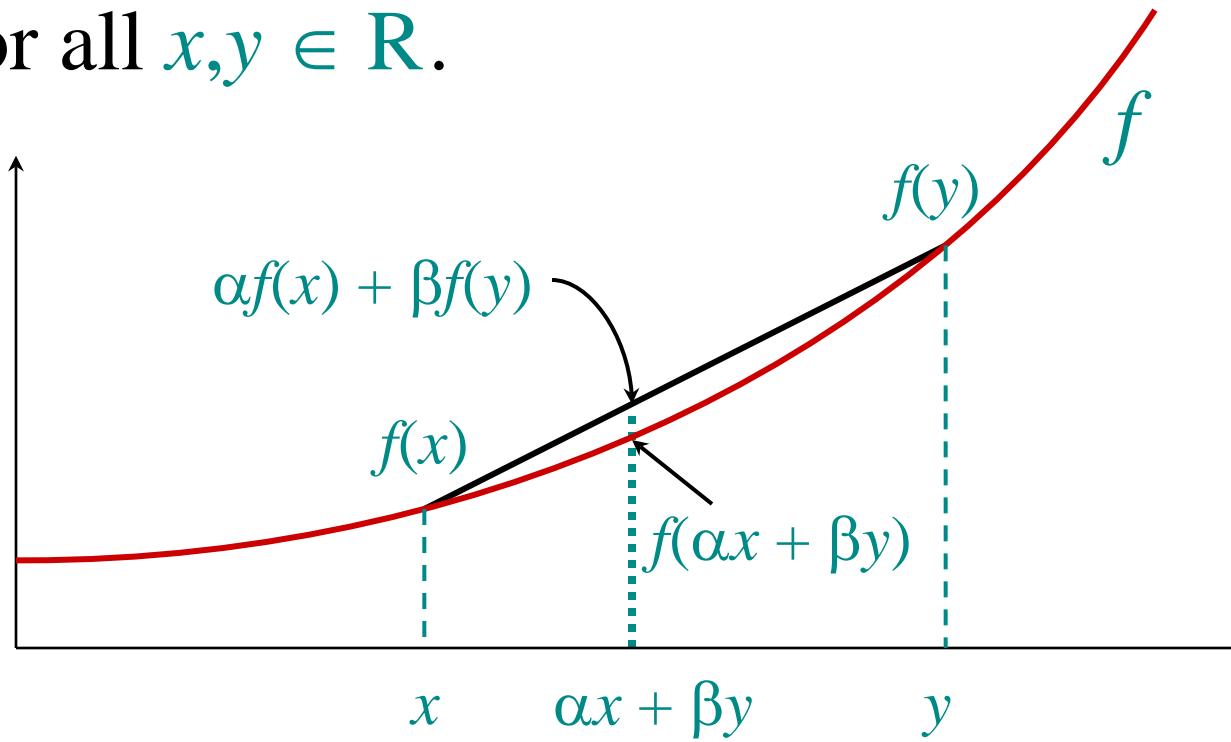


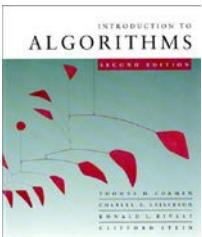
# Convex functions

A function  $f: \mathbb{R} \rightarrow \mathbb{R}$  is **convex** if for all  $\alpha, \beta \geq 0$  such that  $\alpha + \beta = 1$ , we have

$$f(\alpha x + \beta y) \leq \alpha f(x) + \beta f(y)$$

for all  $x, y \in \mathbb{R}$ .



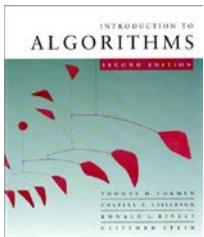


# Convexity lemma

**Lemma.** Let  $f: \mathbb{R} \rightarrow \mathbb{R}$  be a convex function, and let  $\alpha_1, \alpha_2, \dots, \alpha_n$  be nonnegative real numbers such that  $\sum_k \alpha_k = 1$ . Then, for any real numbers  $x_1, x_2, \dots, x_n$ , we have

$$f\left(\sum_{k=1}^n \alpha_k x_k\right) \leq \sum_{k=1}^n \alpha_k f(x_k).$$

**Proof.** By induction on  $n$ . For  $n = 1$ , we have  $\alpha_1 = 1$ , and hence  $f(\alpha_1 x_1) \leq \alpha_1 f(x_1)$  trivially.

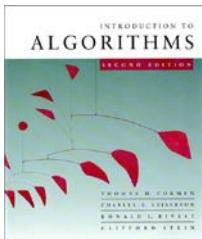


# Proof (continued)

Inductive step:

$$f\left(\sum_{k=1}^n \alpha_k x_k\right) = f\left(\alpha_n x_n + (1 - \alpha_n) \sum_{k=1}^{n-1} \frac{\alpha_k}{1 - \alpha_n} x_k\right)$$

Algebra.

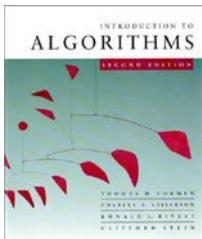


# Proof (continued)

Inductive step:

$$\begin{aligned} f\left(\sum_{k=1}^n \alpha_k x_k\right) &= f\left(\alpha_n x_n + (1 - \alpha_n) \sum_{k=1}^{n-1} \frac{\alpha_k}{1 - \alpha_n} x_k\right) \\ &\leq \alpha_n f(x_n) + (1 - \alpha_n) f\left(\sum_{k=1}^{n-1} \frac{\alpha_k}{1 - \alpha_n} x_k\right) \end{aligned}$$

Convexity.

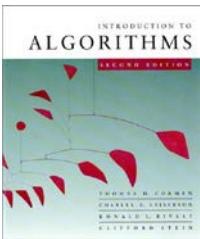


# Proof (continued)

Inductive step:

$$\begin{aligned} f\left(\sum_{k=1}^n \alpha_k x_k\right) &= f\left(\alpha_n x_n + (1 - \alpha_n) \sum_{k=1}^{n-1} \frac{\alpha_k}{1 - \alpha_n} x_k\right) \\ &\leq \alpha_n f(x_n) + (1 - \alpha_n) f\left(\sum_{k=1}^{n-1} \frac{\alpha_k}{1 - \alpha_n} x_k\right) \\ &\leq \alpha_n f(x_n) + (1 - \alpha_n) \sum_{k=1}^{n-1} \frac{\alpha_k}{1 - \alpha_n} f(x_k) \end{aligned}$$

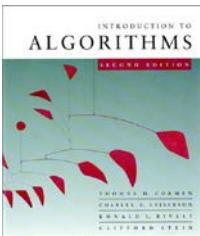
Induction.



# Proof (continued)

Inductive step:

$$\begin{aligned} f\left(\sum_{k=1}^n \alpha_k x_k\right) &= f\left(\alpha_n x_n + (1 - \alpha_n) \sum_{k=1}^{n-1} \frac{\alpha_k}{1 - \alpha_n} x_k\right) \\ &\leq \alpha_n f(x_n) + (1 - \alpha_n) f\left(\sum_{k=1}^{n-1} \frac{\alpha_k}{1 - \alpha_n} x_k\right) \\ &\leq \alpha_n f(x_n) + (1 - \alpha_n) \sum_{k=1}^{n-1} \frac{\alpha_k}{1 - \alpha_n} f(x_k) \\ &= \sum_{k=1}^n \alpha_k f(x_k). \quad \square \qquad \text{Algebra.} \end{aligned}$$

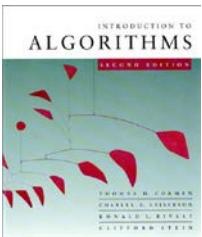


# Convexity lemma: infinite case

**Lemma.** Let  $f: \mathbb{R} \rightarrow \mathbb{R}$  be a convex function, and let  $\alpha_1, \alpha_2, \dots,$  be nonnegative real numbers such that  $\sum_k \alpha_k = 1.$  Then, for any real numbers  $x_1, x_2, \dots,$  we have

$$f\left(\sum_{k=1}^{\infty} \alpha_k x_k\right) \leq \sum_{k=1}^{\infty} \alpha_k f(x_k) ,$$

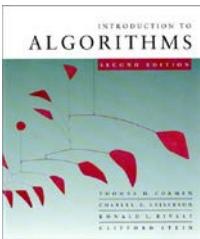
assuming that these summations exist.



# Convexity lemma: infinite case

*Proof.* By the convexity lemma, for any  $n \geq 1$ ,

$$f\left(\sum_{k=1}^n \frac{\alpha_k}{\sum_{i=1}^n \alpha_i} x_k\right) \leq \sum_{k=1}^n \frac{\alpha_k}{\sum_{i=1}^n \alpha_i} f(x_k).$$



# Convexity lemma: infinite case

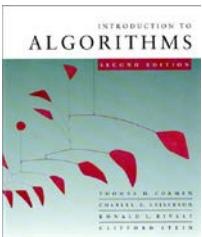
*Proof.* By the convexity lemma, for any  $n \geq 1$ ,

$$f\left(\sum_{k=1}^n \frac{\alpha_k}{\sum_{i=1}^n \alpha_i} x_k\right) \leq \sum_{k=1}^n \frac{\alpha_k}{\sum_{i=1}^n \alpha_i} f(x_k).$$

Taking the limit of both sides  
(and because the inequality is not strict):

$$\lim_{n \rightarrow \infty} f\left(\underbrace{\frac{1}{\sum_{i=1}^n \alpha_i} \sum_{k=1}^n \alpha_k x_k}_{\rightarrow 1 \quad \rightarrow \sum_{k=1}^{\infty} \alpha_k x_k}\right) \leq \lim_{n \rightarrow \infty} \underbrace{\frac{1}{\sum_{i=1}^n \alpha_i} \sum_{k=1}^n \alpha_k f(x_k)}_{\rightarrow 1 \quad \rightarrow \sum_{k=1}^{\infty} \alpha_k f(x_k)}$$





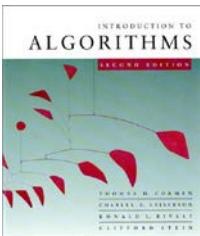
# Jensen's inequality

**Lemma.** Let  $f$  be a convex function, and let  $X$  be a random variable. Then,  $f(E[X]) \leq E[f(X)]$ .

*Proof.*

$$f(E[X]) = f\left( \sum_{k=-\infty}^{\infty} k \cdot \Pr\{X = k\} \right)$$

Definition of expectation.



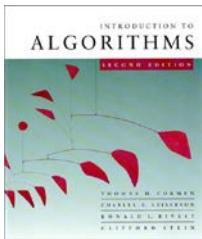
# Jensen's inequality

**Lemma.** Let  $f$  be a convex function, and let  $X$  be a random variable. Then,  $f(E[X]) \leq E[f(X)]$ .

*Proof.*

$$\begin{aligned} f(E[X]) &= f\left(\sum_{k=-\infty}^{\infty} k \cdot \Pr\{X = k\}\right) \\ &\leq \sum_{k=-\infty}^{\infty} f(k) \cdot \Pr\{X = k\} \end{aligned}$$

Convexity lemma (infinite case).



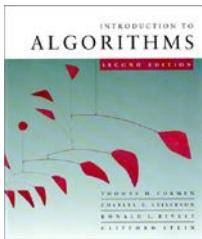
# Jensen's inequality

**Lemma.** Let  $f$  be a convex function, and let  $X$  be a random variable. Then,  $f(E[X]) \leq E[f(X)]$ .

*Proof.*

$$\begin{aligned} f(E[X]) &= f\left(\sum_{k=-\infty}^{\infty} k \cdot \Pr\{X = k\}\right) \\ &\leq \sum_{k=-\infty}^{\infty} f(k) \cdot \Pr\{X = k\} \\ &= E[f(X)]. \quad \square \end{aligned}$$

Tricky step, but true—think about it.



# Analysis of BST height

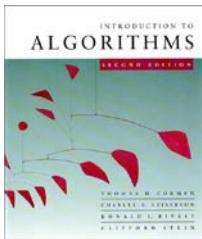
Let  $X_n$  be the random variable denoting the height of a randomly built binary search tree on  $n$  nodes, and let  $Y_n = 2^{X_n}$  be its exponential height.

If the root of the tree has rank  $k$ , then

$$X_n = 1 + \max\{X_{k-1}, X_{n-k}\} ,$$

since each of the left and right subtrees of the root are randomly built. Hence, we have

$$Y_n = 2 \cdot \max\{Y_{k-1}, Y_{n-k}\} .$$



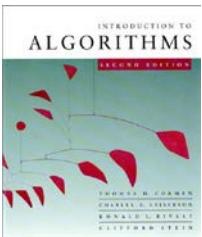
# Analysis (continued)

Define the indicator random variable  $Z_{nk}$  as

$$Z_{nk} = \begin{cases} 1 & \text{if the root has rank } k, \\ 0 & \text{otherwise.} \end{cases}$$

Thus,  $\Pr\{Z_{nk} = 1\} = \mathbb{E}[Z_{nk}] = 1/n$ , and

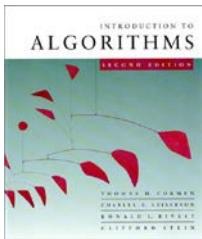
$$Y_n = \sum_{k=1}^n Z_{nk} (2 \cdot \max\{Y_{k-1}, Y_{n-k}\}) .$$



# Exponential height recurrence

$$E[Y_n] = E\left[\sum_{k=1}^n Z_{nk} (2 \cdot \max\{Y_{k-1}, Y_{n-k}\})\right]$$

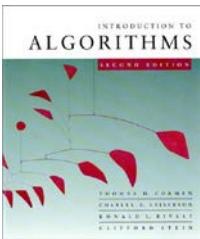
Take expectation of both sides.



# Exponential height recurrence

$$\begin{aligned}E[Y_n] &= E\left[\sum_{k=1}^n Z_{nk} (2 \cdot \max\{Y_{k-1}, Y_{n-k}\})\right] \\&= \sum_{k=1}^n E[Z_{nk} (2 \cdot \max\{Y_{k-1}, Y_{n-k}\})]\end{aligned}$$

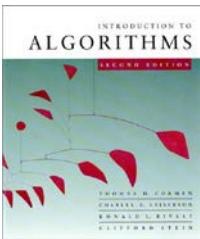
Linearity of expectation.



# Exponential height recurrence

$$\begin{aligned}E[Y_n] &= E\left[\sum_{k=1}^n Z_{nk} (2 \cdot \max\{Y_{k-1}, Y_{n-k}\})\right] \\&= \sum_{k=1}^n E[Z_{nk} (2 \cdot \max\{Y_{k-1}, Y_{n-k}\})] \\&= 2 \sum_{k=1}^n E[Z_{nk}] \cdot E[\max\{Y_{k-1}, Y_{n-k}\}]\end{aligned}$$

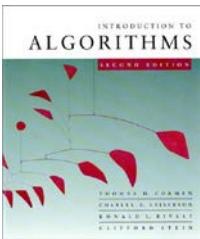
Independence of the rank of the root from the ranks of subtree roots.



# Exponential height recurrence

$$\begin{aligned}E[Y_n] &= E\left[\sum_{k=1}^n Z_{nk} (2 \cdot \max\{Y_{k-1}, Y_{n-k}\})\right] \\&= \sum_{k=1}^n E[Z_{nk} (2 \cdot \max\{Y_{k-1}, Y_{n-k}\})] \\&= 2 \sum_{k=1}^n E[Z_{nk}] \cdot E[\max\{Y_{k-1}, Y_{n-k}\}] \\&\leq \frac{2}{n} \sum_{k=1}^n E[Y_{k-1} + Y_{n-k}]\end{aligned}$$

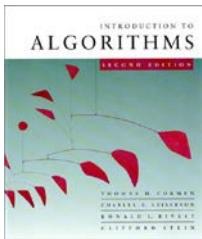
The max of two nonnegative numbers is at most their sum, and  $E[Z_{nk}] = 1/n$ .



# Exponential height recurrence

$$\begin{aligned}E[Y_n] &= E\left[\sum_{k=1}^n Z_{nk} (2 \cdot \max\{Y_{k-1}, Y_{n-k}\})\right] \\&= \sum_{k=1}^n E[Z_{nk} (2 \cdot \max\{Y_{k-1}, Y_{n-k}\})] \\&= 2 \sum_{k=1}^n E[Z_{nk}] \cdot E[\max\{Y_{k-1}, Y_{n-k}\}] \\&\leq \frac{2}{n} \sum_{k=1}^n E[Y_{k-1} + Y_{n-k}] \\&= \frac{4}{n} \sum_{k=0}^{n-1} E[Y_k]\end{aligned}$$

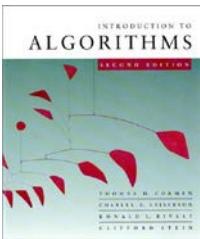
Each term appears twice, and reindex.



# Solving the recurrence

Use substitution to show that  $E[Y_n] \leq cn^3$  for some positive constant  $c$ , which we can pick sufficiently large to handle the initial conditions.

$$E[Y_n] = \frac{4}{n} \sum_{k=0}^{n-1} E[Y_k]$$

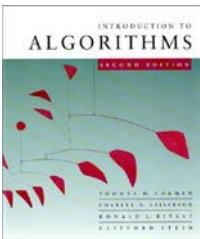


# Solving the recurrence

Use substitution to show that  $E[Y_n] \leq cn^3$  for some positive constant  $c$ , which we can pick sufficiently large to handle the initial conditions.

$$\begin{aligned}E[Y_n] &= \frac{4}{n} \sum_{k=0}^{n-1} E[Y_k] \\&\leq \frac{4}{n} \sum_{k=0}^{n-1} ck^3\end{aligned}$$

Substitution.

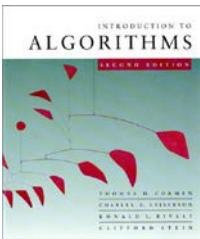


# Solving the recurrence

Use substitution to show that  $E[Y_n] \leq cn^3$  for some positive constant  $c$ , which we can pick sufficiently large to handle the initial conditions.

$$\begin{aligned}E[Y_n] &= \frac{4}{n} \sum_{k=0}^{n-1} E[Y_k] \\&\leq \frac{4}{n} \sum_{k=0}^{n-1} ck^3 \\&\leq \frac{4c}{n} \int_0^n x^3 dx\end{aligned}$$

Integral method.

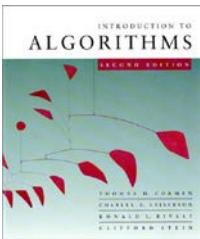


# Solving the recurrence

Use substitution to show that  $E[Y_n] \leq cn^3$  for some positive constant  $c$ , which we can pick sufficiently large to handle the initial conditions.

$$\begin{aligned}E[Y_n] &= \frac{4}{n} \sum_{k=0}^{n-1} E[Y_k] \\&\leq \frac{4}{n} \sum_{k=0}^{n-1} ck^3 \\&\leq \frac{4c}{n} \int_0^n x^3 dx \\&= \frac{4c}{n} \left( \frac{n^4}{4} \right)\end{aligned}$$

Solve the integral.

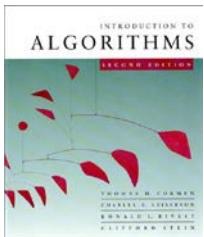


# Solving the recurrence

Use substitution to show that  $E[Y_n] \leq cn^3$  for some positive constant  $c$ , which we can pick sufficiently large to handle the initial conditions.

$$\begin{aligned}E[Y_n] &= \frac{4}{n} \sum_{k=0}^{n-1} E[Y_k] \\&\leq \frac{4}{n} \sum_{k=0}^{n-1} ck^3 \\&\leq \frac{4c}{n} \int_0^n x^3 dx \\&= \frac{4c}{n} \left( \frac{n^4}{4} \right) \\&= cn^3.\end{aligned}$$

Algebra.

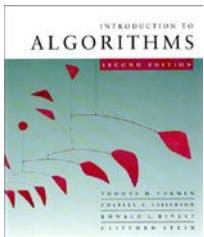


# The grand finale

Putting it all together, we have

$$2^{E[X_n]} \leq E[2^{X_n}]$$

Jensen's inequality, since  
 $f(x) = 2^x$  is convex.

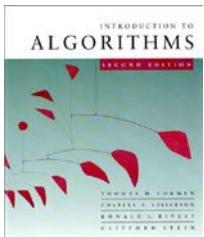


# The grand finale

Putting it all together, we have

$$\begin{aligned} 2^{E[X_n]} &\leq E[2^{X_n}] \\ &= E[Y_n] \end{aligned}$$

Definition.

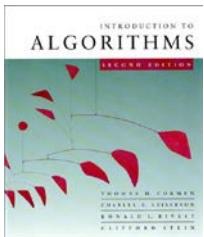


# The grand finale

Putting it all together, we have

$$\begin{aligned} 2^{E[X_n]} &\leq E[2^{X_n}] \\ &= E[Y_n] \\ &\leq cn^3. \end{aligned}$$

What we just showed.



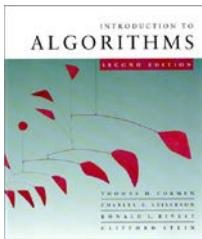
# The grand finale

Putting it all together, we have

$$\begin{aligned} 2^{E[X_n]} &\leq E[2^{X_n}] \\ &= E[Y_n] \\ &\leq cn^3. \end{aligned}$$

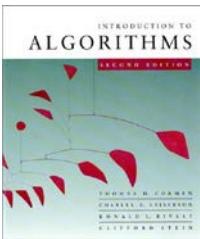
Taking the  $\lg$  of both sides yields

$$E[X_n] \leq 3 \lg n + O(1).$$



# Post mortem

- Q.** Does the analysis have to be this hard?
- Q.** Why bother with analyzing exponential height?
- Q.** Why not just develop the recurrence on
$$X_n = 1 + \max\{X_{k-1}, X_{n-k}\}$$
directly?



# Post mortem (continued)

## A. The inequality

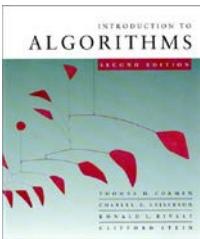
$$\max\{a, b\} \leq a + b .$$

provides a poor upper bound, since the RHS approaches the LHS slowly as  $|a - b|$  increases.

The bound

$$\max\{2^a, 2^b\} \leq 2^a + 2^b$$

allows the RHS to approach the LHS far more quickly as  $|a - b|$  increases. By using the convexity of  $f(x) = 2^x$  via Jensen's inequality, we can manipulate the sum of exponentials, resulting in a tight analysis.

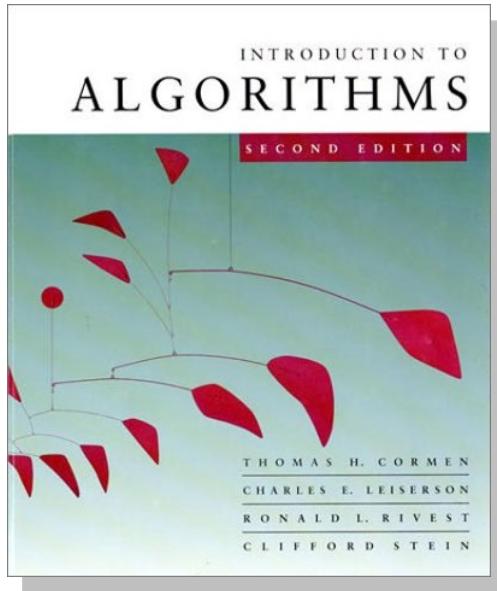


# Thought exercises

- See what happens when you try to do the analysis on  $X_n$  directly.
- Try to understand better why the proof uses an exponential. Will a quadratic do?
- See if you can find a simpler argument.  
(This argument is a little simpler than the one in the book—I hope it's correct!)

# *Introduction to Algorithms*

## 6.046J/18.401J

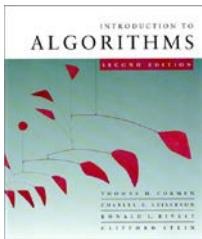


### LECTURE 10

#### Balanced Search Trees

- Red-black trees
- Height of a red-black tree
- Rotations
- Insertion

Prof. Erik Demaine

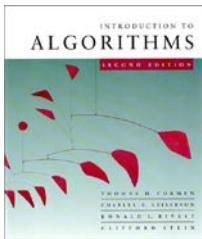


# Balanced search trees

***Balanced search tree:*** A search-tree data structure for which a height of  $O(\lg n)$  is guaranteed when implementing a dynamic set of  $n$  items.

**Examples:**

- AVL trees
- 2-3 trees
- 2-3-4 trees
- B-trees
- Red-black trees

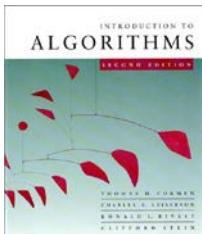


# Red-black trees

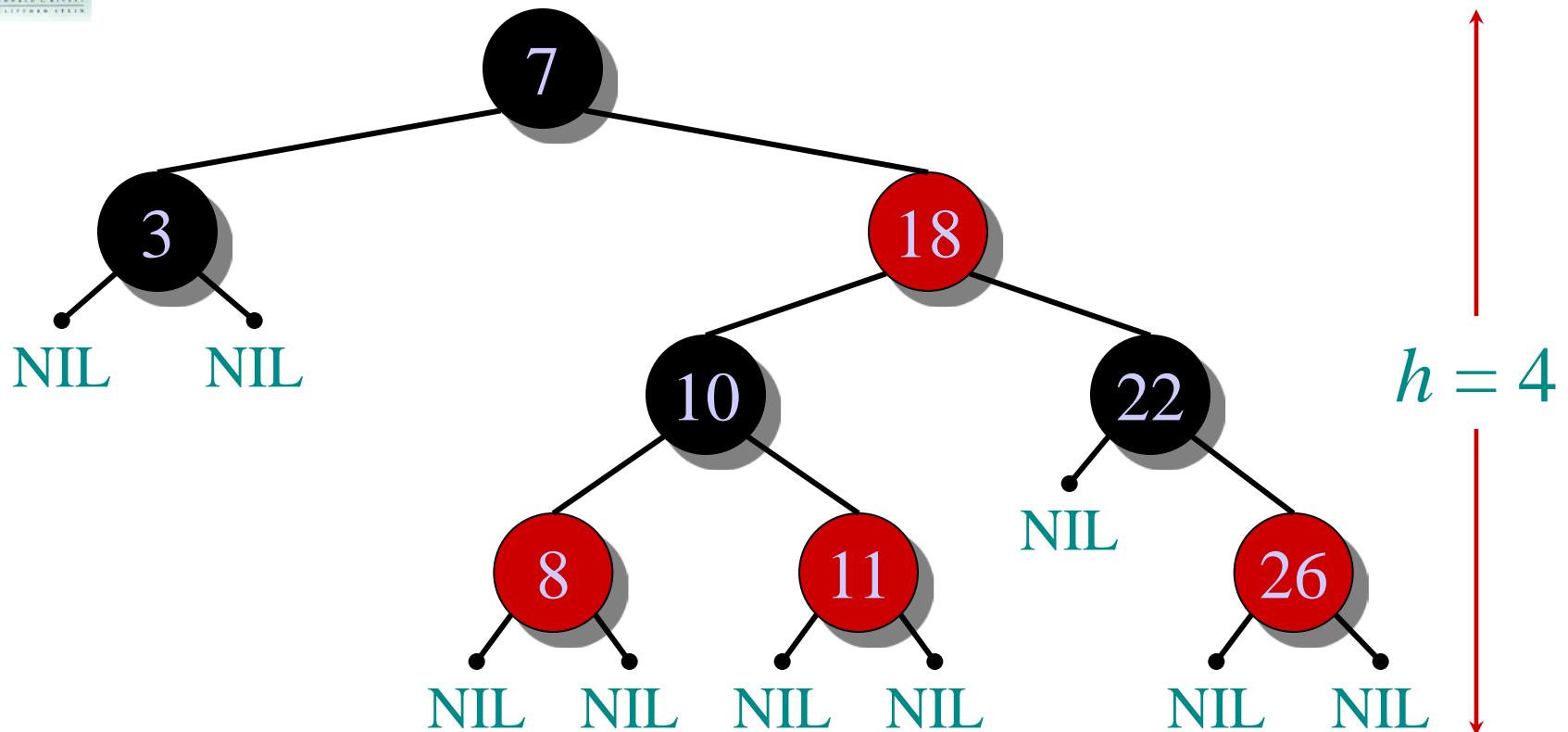
This data structure requires an extra one-bit **color** field in each node.

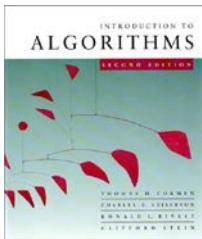
## *Red-black properties:*

1. Every node is either red or black.
2. The root and leaves (**NIL**'s) are black.
3. If a node is red, then its parent is black.
4. All simple paths from any node  $x$  to a descendant leaf have the same number of black nodes = **black-height**( $x$ ).

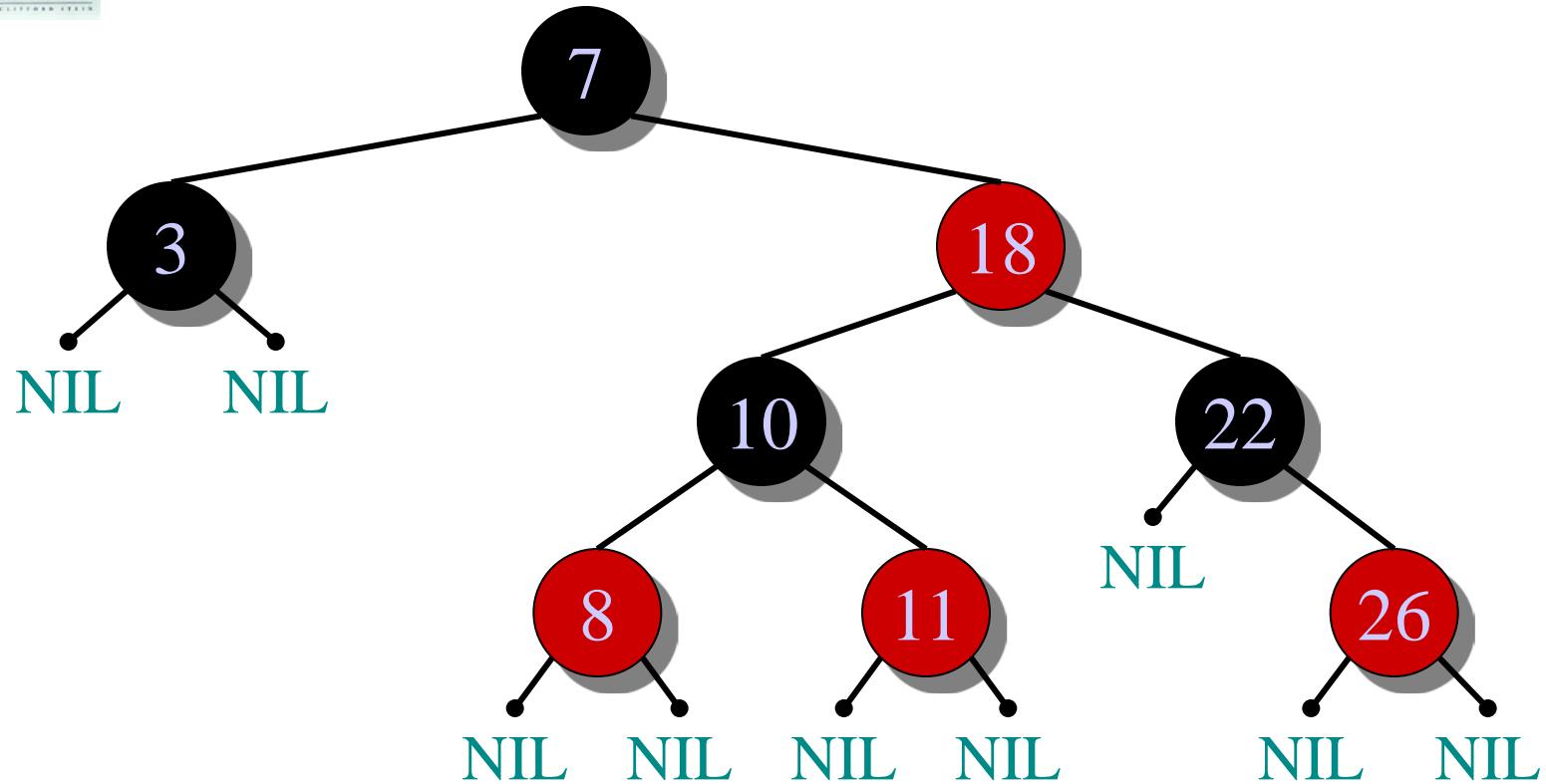


# Example of a red-black tree

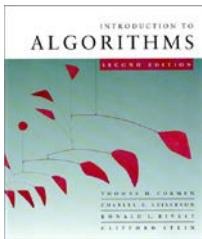




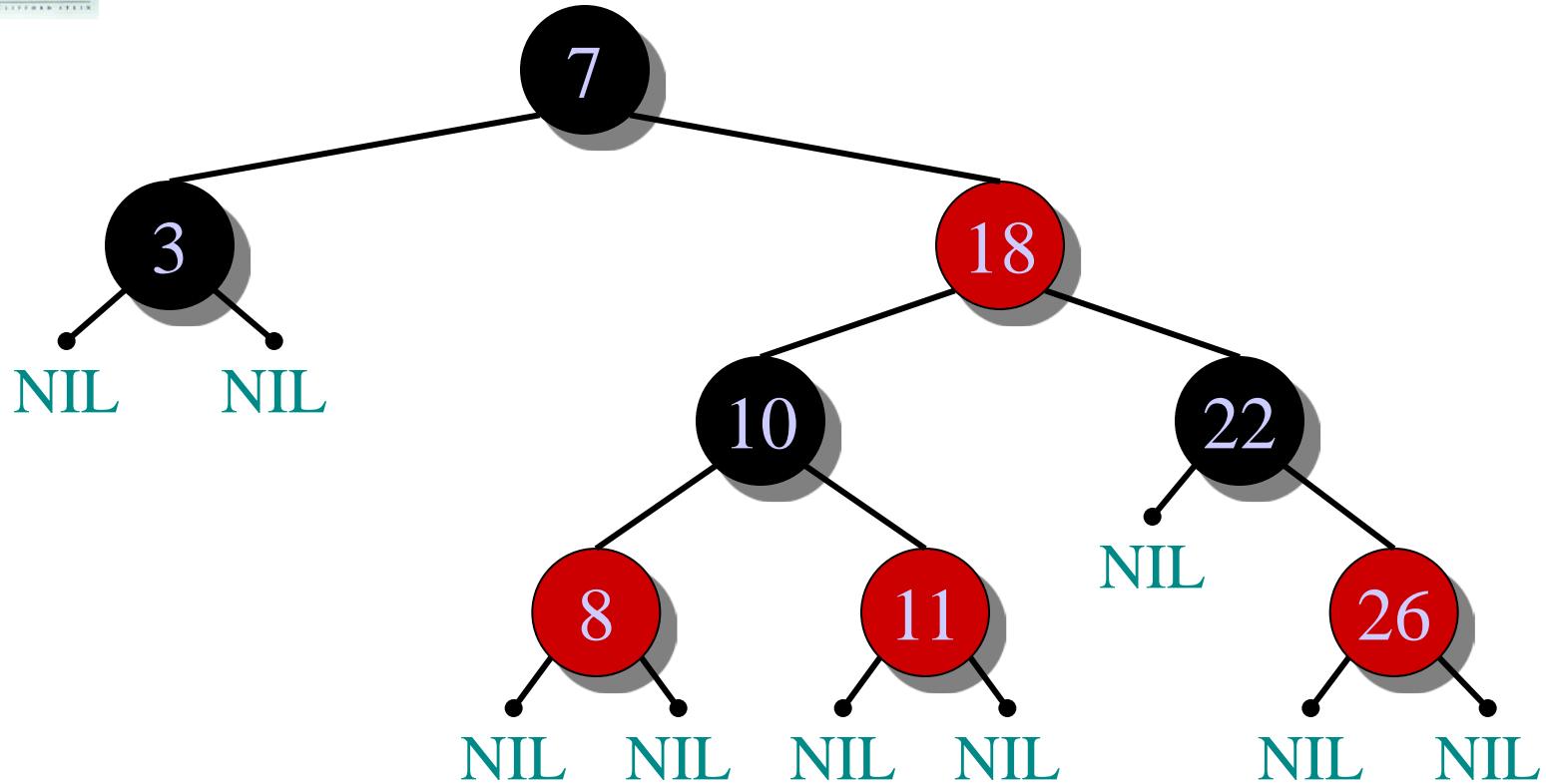
# Example of a red-black tree



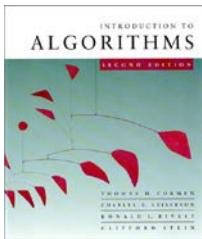
1. Every node is either red or black.



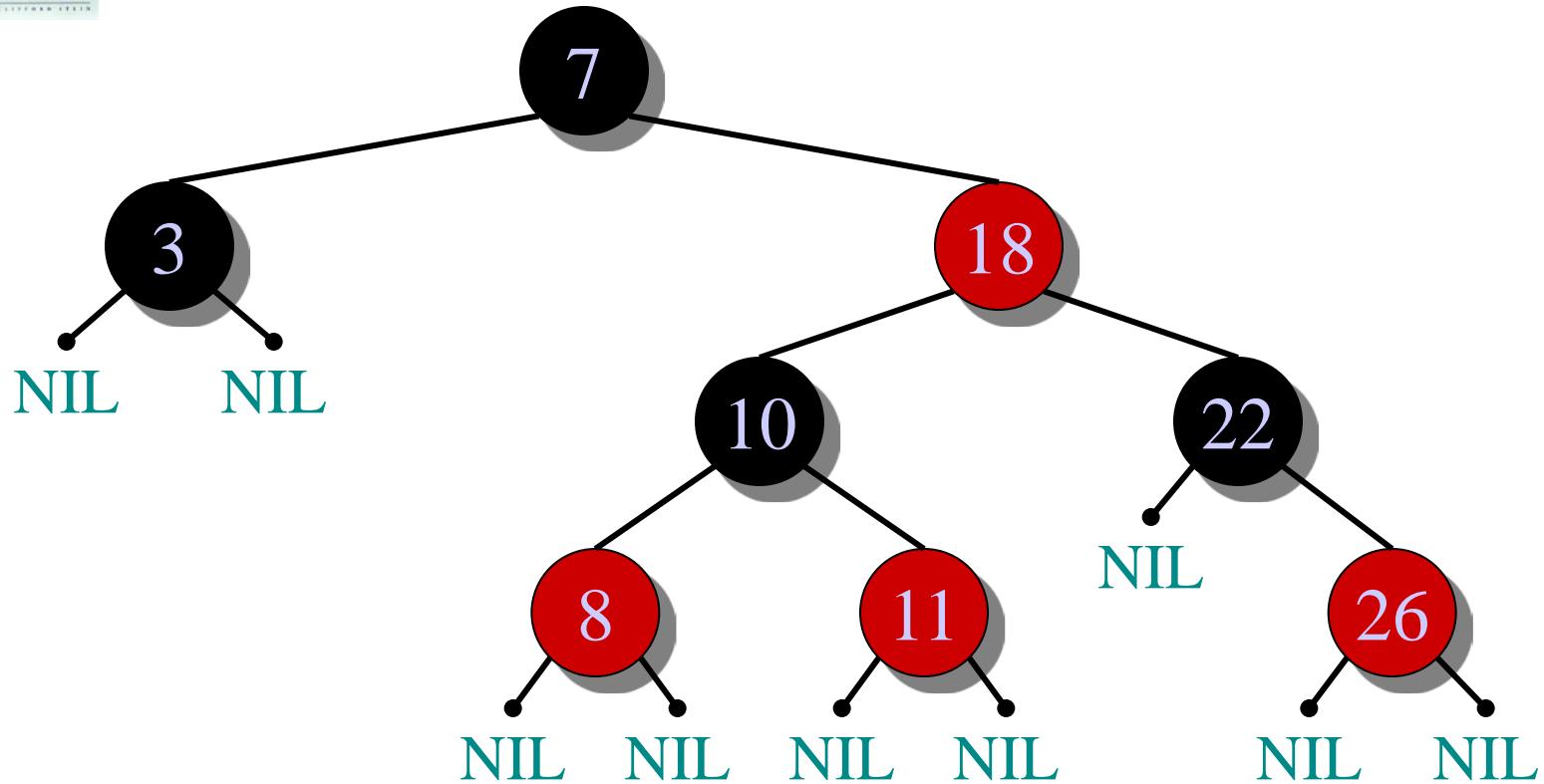
# Example of a red-black tree



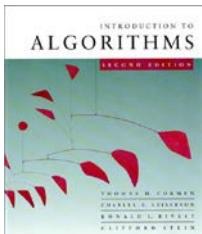
2. The root and leaves (NIL's) are black.



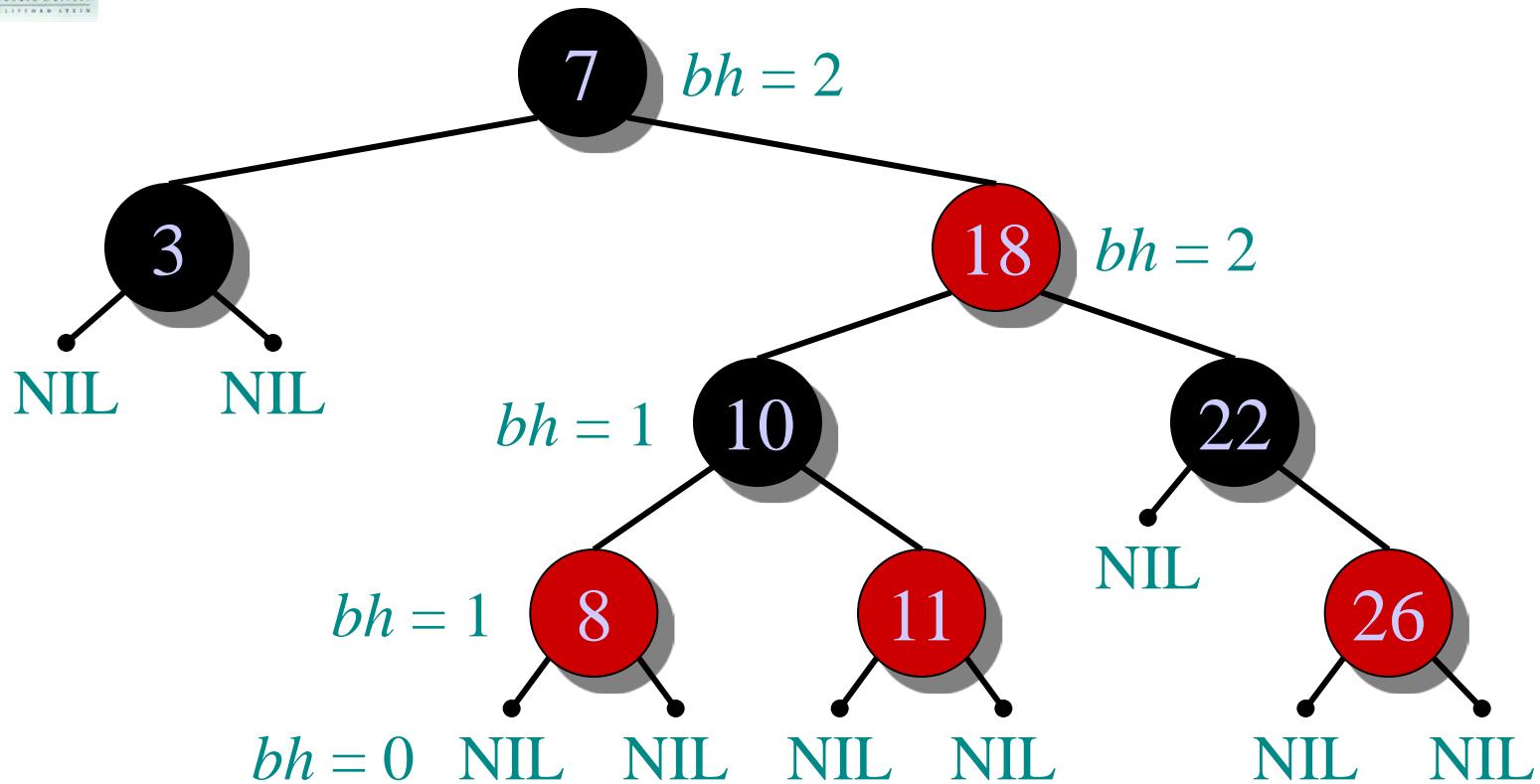
# Example of a red-black tree



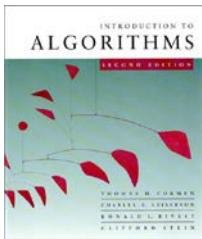
3. If a node is red, then its parent is black.



# Example of a red-black tree



4. All simple paths from any node  $x$  to a descendant leaf have the same number of black nodes =  $black\text{-}height(x)$ .



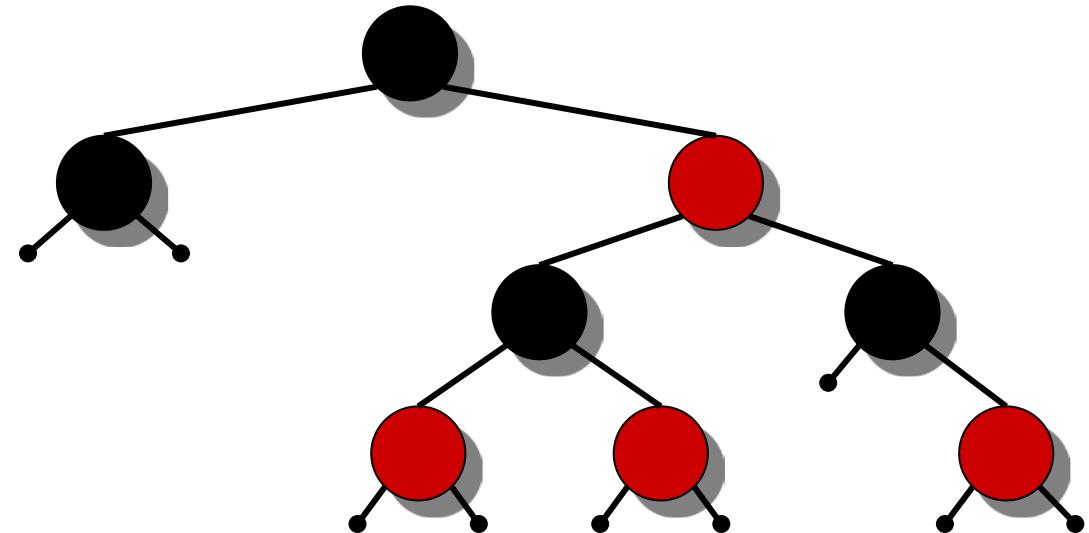
# Height of a red-black tree

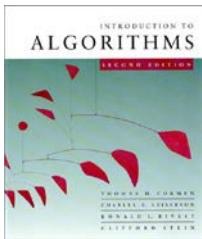
**Theorem.** A red-black tree with  $n$  keys has height  
$$h \leq 2 \lg(n + 1).$$

*Proof.* (The book uses induction. Read carefully.)

## INTUITION:

- Merge red nodes into their black parents.





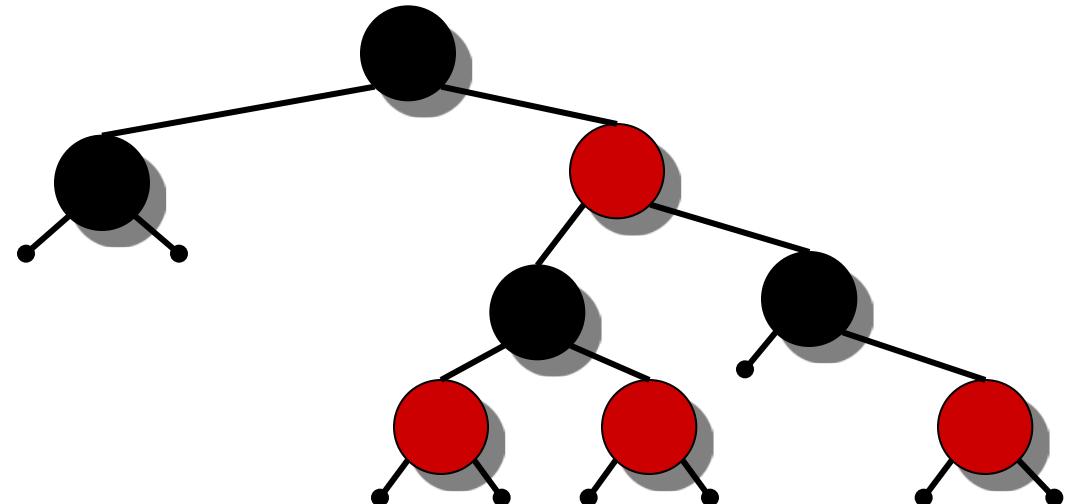
# Height of a red-black tree

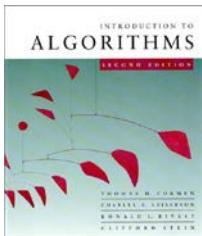
**Theorem.** A red-black tree with  $n$  keys has height  
$$h \leq 2 \lg(n + 1).$$

*Proof.* (The book uses induction. Read carefully.)

## INTUITION:

- Merge red nodes into their black parents.





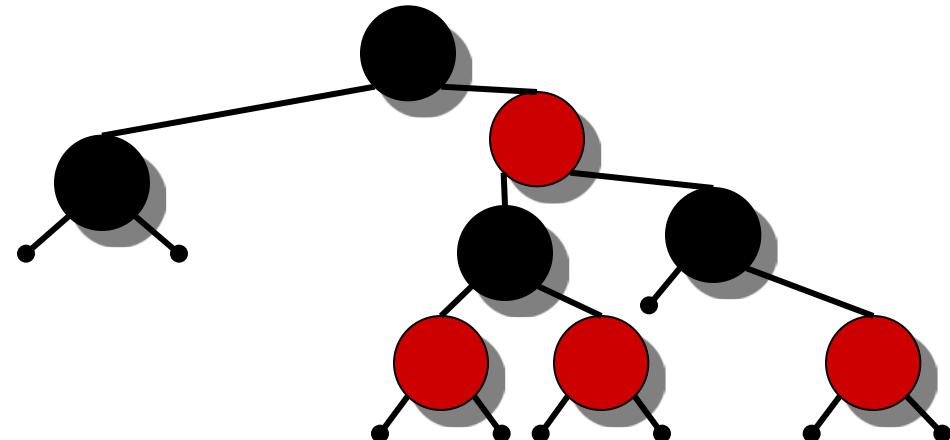
# Height of a red-black tree

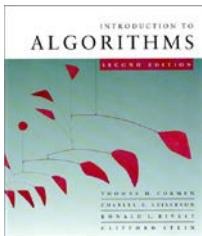
**Theorem.** A red-black tree with  $n$  keys has height  
$$h \leq 2 \lg(n + 1).$$

*Proof.* (The book uses induction. Read carefully.)

## INTUITION:

- Merge red nodes into their black parents.





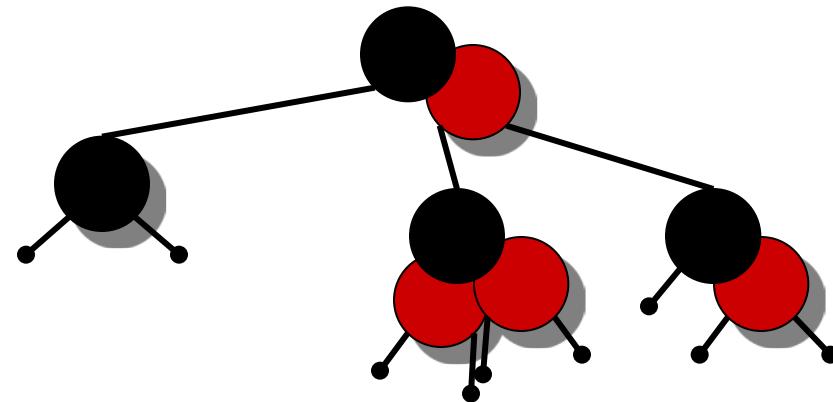
# Height of a red-black tree

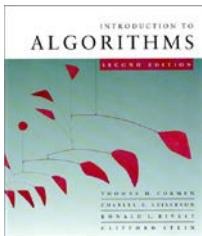
**Theorem.** A red-black tree with  $n$  keys has height  
$$h \leq 2 \lg(n + 1).$$

*Proof.* (The book uses induction. Read carefully.)

## INTUITION:

- Merge red nodes into their black parents.





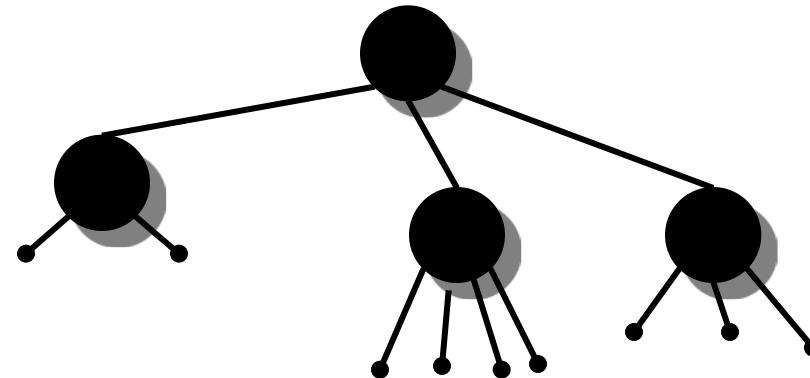
# Height of a red-black tree

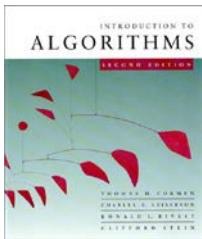
**Theorem.** A red-black tree with  $n$  keys has height  
$$h \leq 2 \lg(n + 1).$$

*Proof.* (The book uses induction. Read carefully.)

## INTUITION:

- Merge red nodes into their black parents.





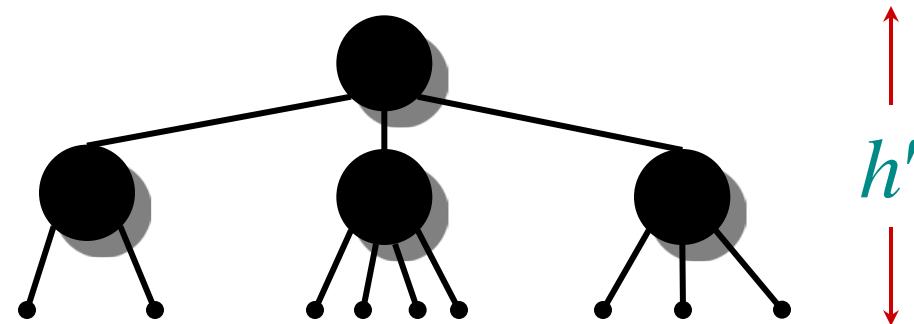
# Height of a red-black tree

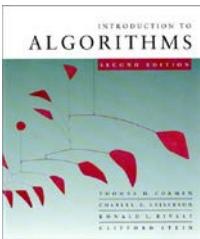
**Theorem.** A red-black tree with  $n$  keys has height  
$$h \leq 2 \lg(n + 1).$$

*Proof.* (The book uses induction. Read carefully.)

## INTUITION:

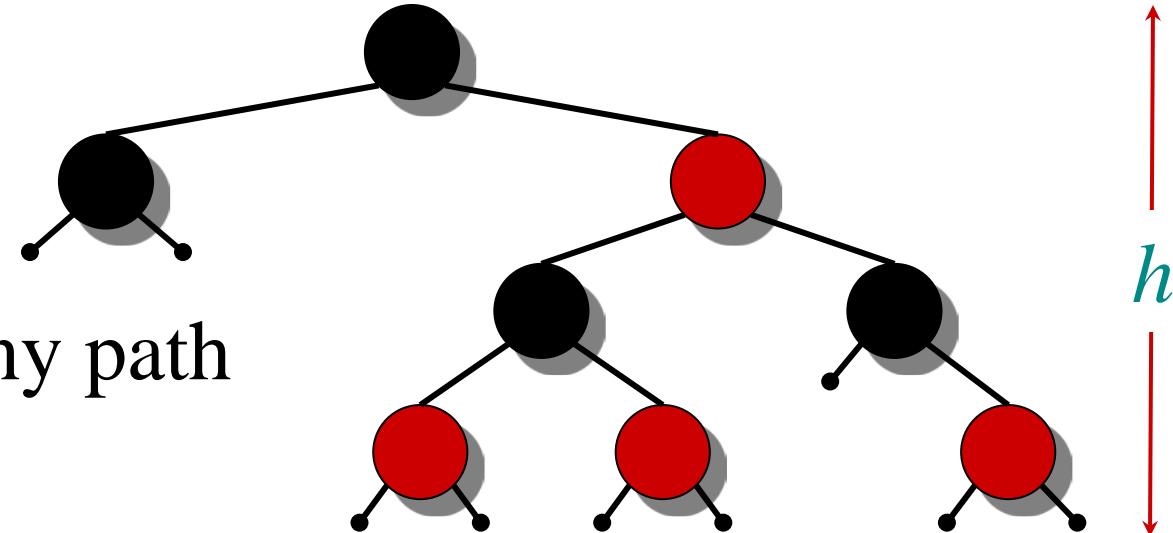
- Merge red nodes into their black parents.
- This process produces a tree in which each node has 2, 3, or 4 children.
- The 2-3-4 tree has uniform depth  $h'$  of leaves.





# Proof (continued)

- We have  $h' \geq h/2$ , since at most half the leaves on any path are red.

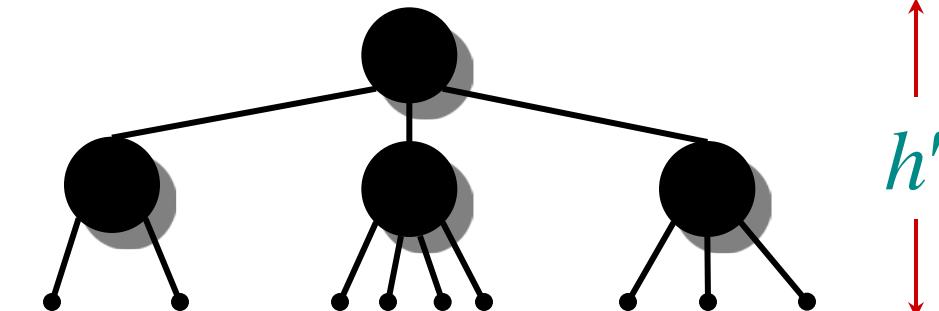


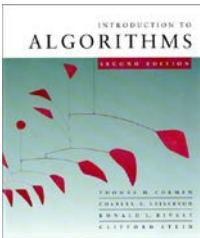
- The number of leaves in each tree is  $n + 1$

$$\Rightarrow n + 1 \geq 2^{h'}$$

$$\Rightarrow \lg(n + 1) \geq h' \geq h/2$$

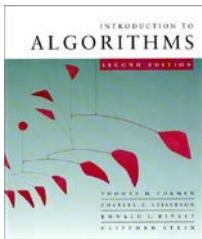
$$\Rightarrow h \leq 2 \lg(n + 1).$$





# Query operations

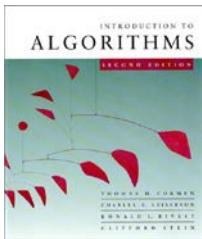
**Corollary.** The queries SEARCH, MIN, MAX, SUCCESSOR, and PREDECESSOR all run in  $O(\lg n)$  time on a red-black tree with  $n$  nodes.



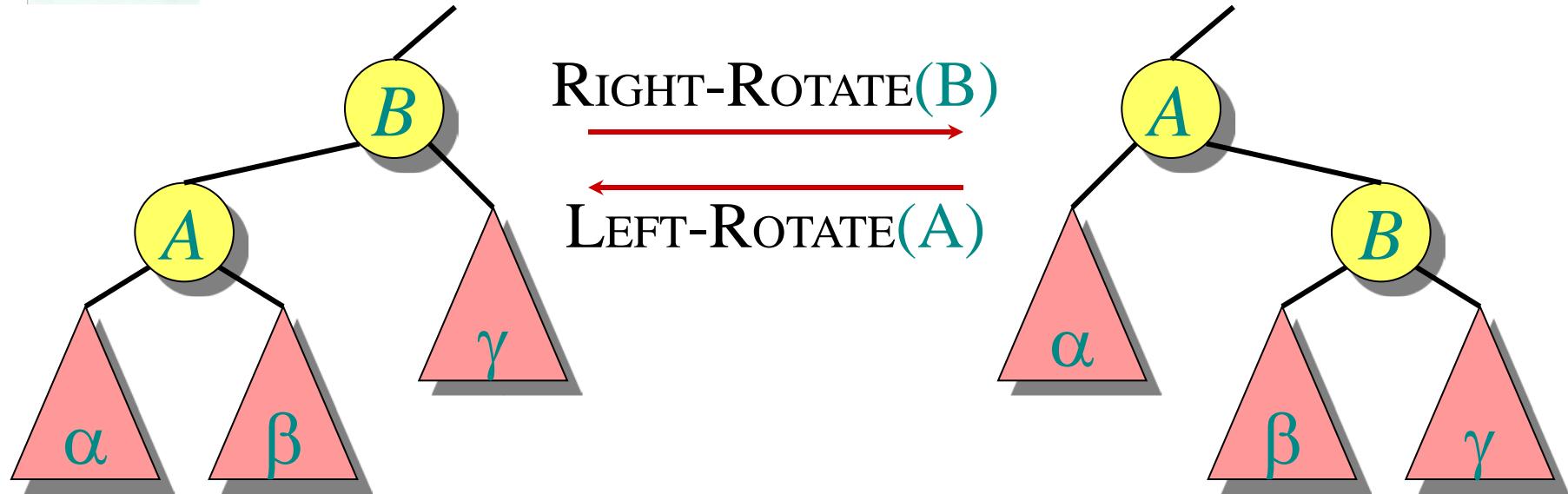
# Modifying operations

The operations INSERT and DELETE cause modifications to the red-black tree:

- the operation itself,
- color changes,
- restructuring the links of the tree via ***“rotations”***.



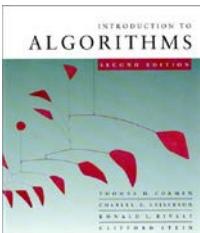
# Rotations



Rotations maintain the inorder ordering of keys:

- $a \in \alpha, b \in \beta, c \in \gamma \Rightarrow a \leq A \leq b \leq B \leq c$ .

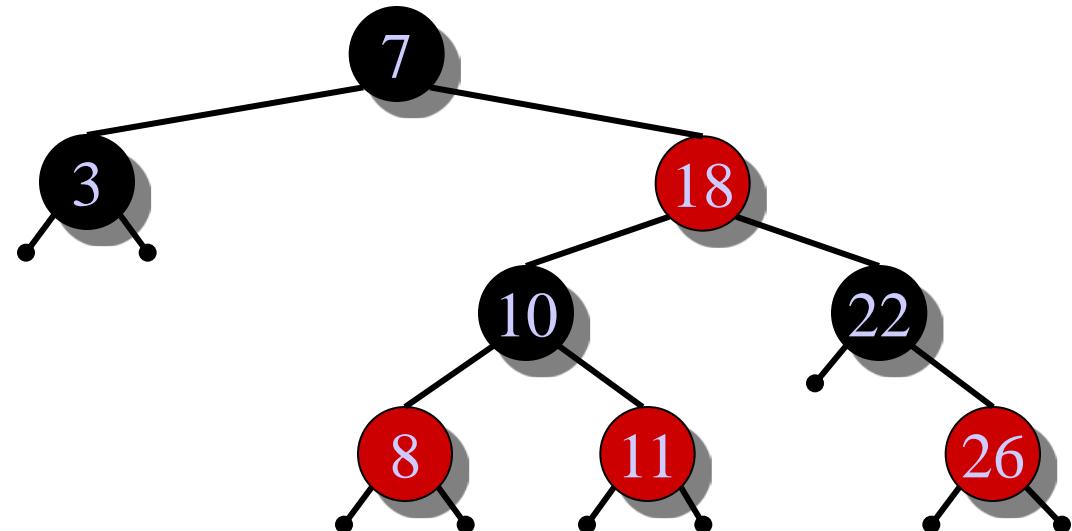
A rotation can be performed in  $O(1)$  time.

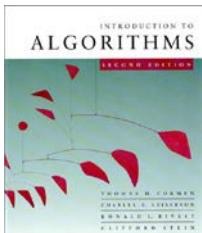


# Insertion into a red-black tree

**IDEA:** Insert  $x$  in tree. Color  $x$  red. Only red-black property 3 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring.

**Example:**



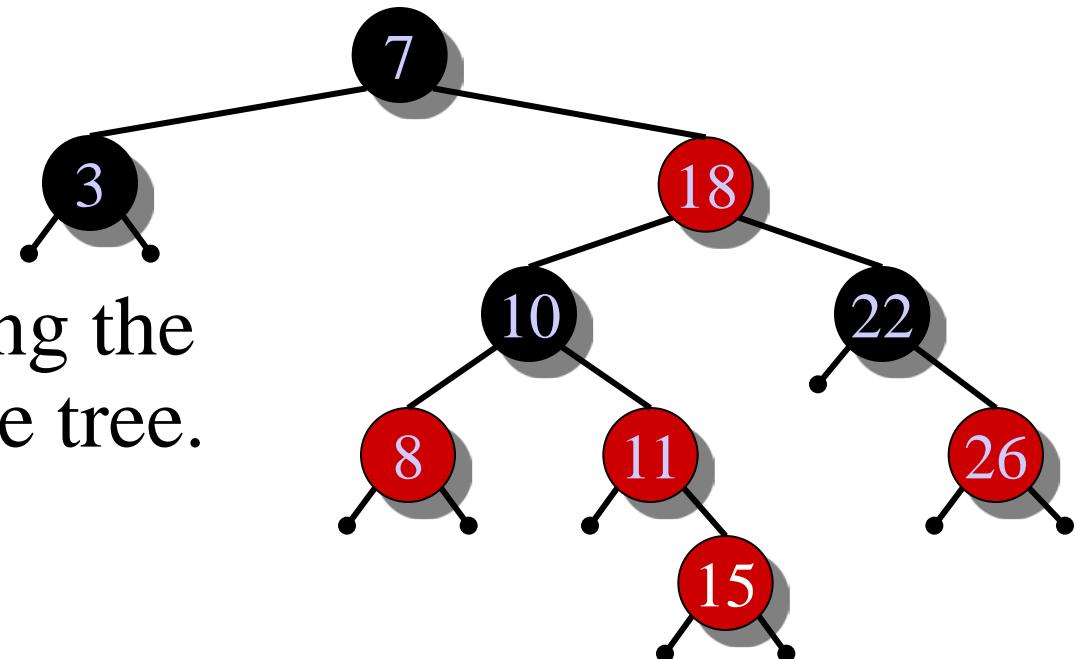


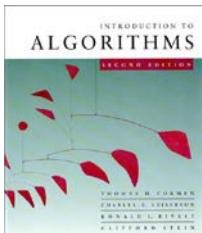
# Insertion into a red-black tree

**IDEA:** Insert  $x$  in tree. Color  $x$  red. Only red-black property 3 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring.

**Example:**

- Insert  $x = 15$ .
- Recolor, moving the violation up the tree.



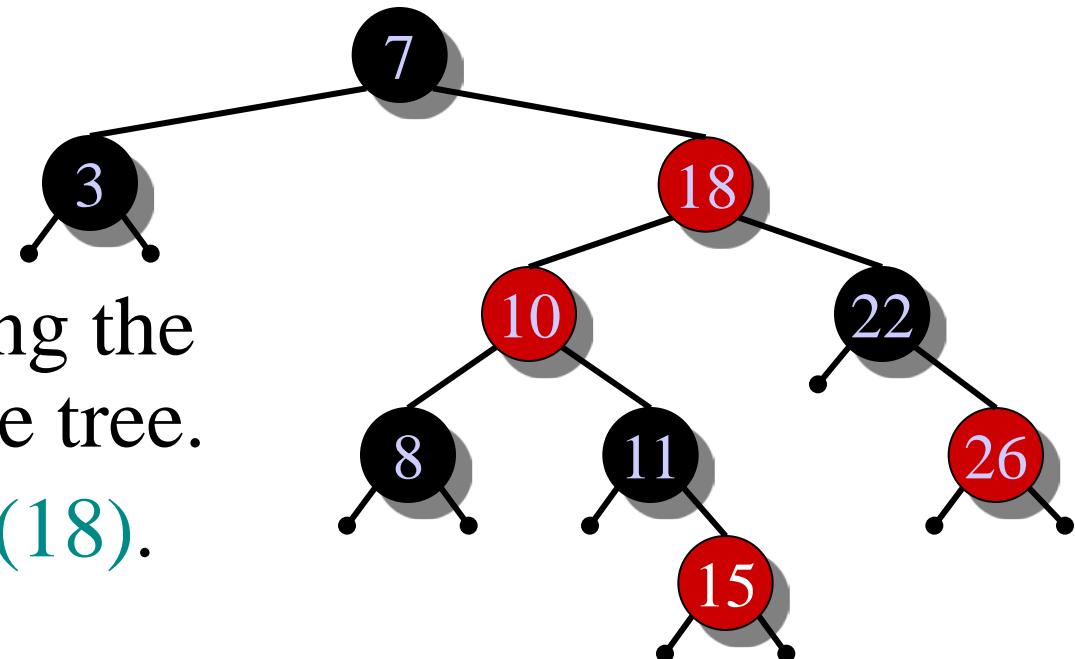


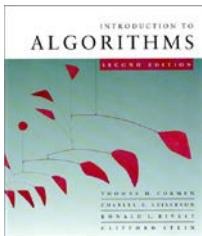
# Insertion into a red-black tree

**IDEA:** Insert  $x$  in tree. Color  $x$  red. Only red-black property 3 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring.

## Example:

- Insert  $x = 15$ .
- Recolor, moving the violation up the tree.
- RIGHT-ROTATE(18).



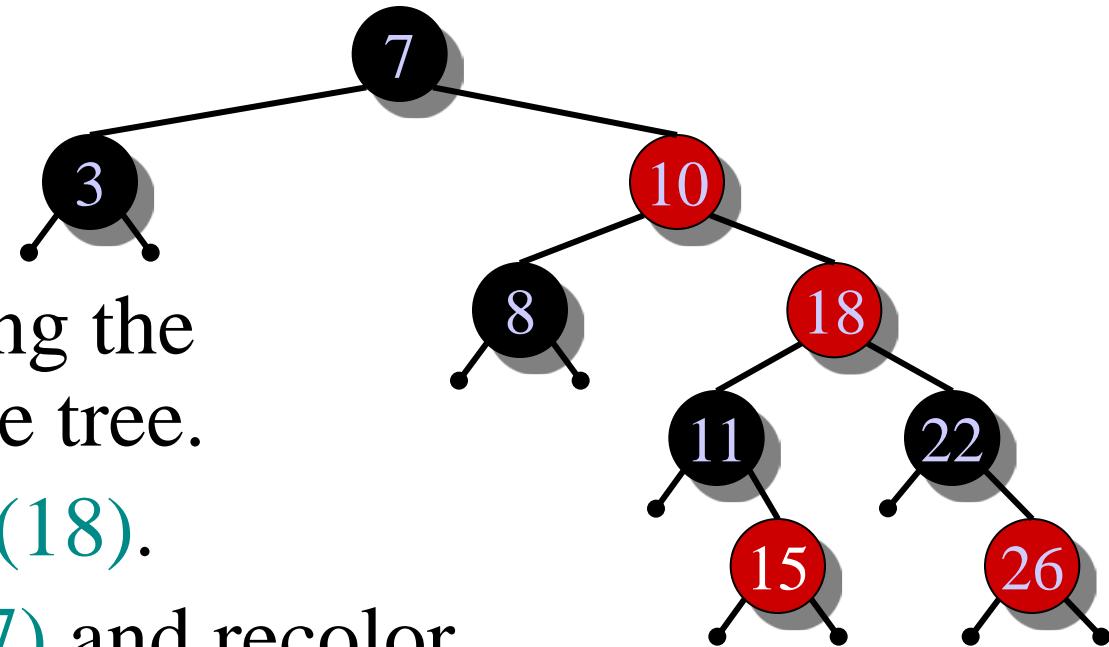


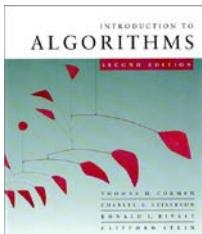
# Insertion into a red-black tree

**IDEA:** Insert  $x$  in tree. Color  $x$  red. Only red-black property 3 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring.

**Example:**

- Insert  $x = 15$ .
- Recolor, moving the violation up the tree.
- RIGHT-ROTATE(18).
- LEFT-ROTATE(7) and recolor.



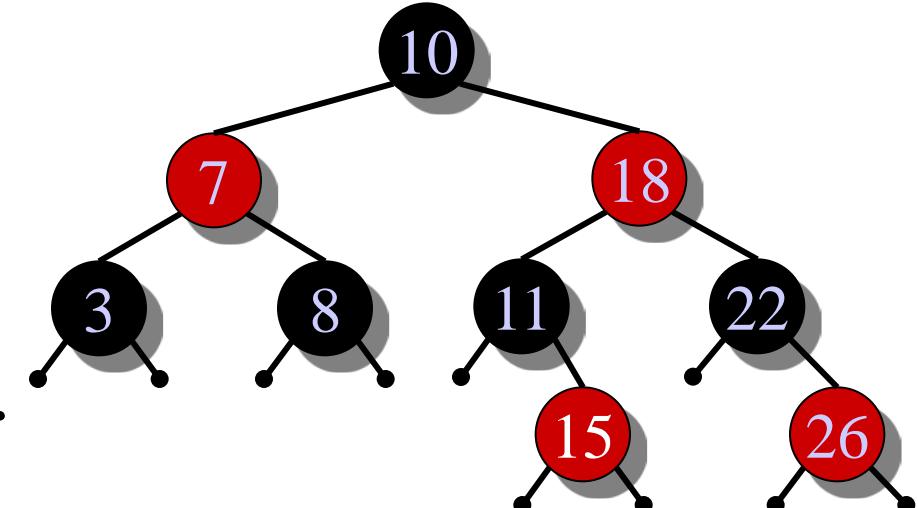


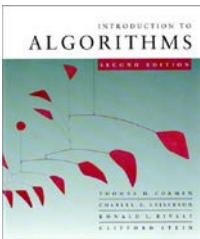
# Insertion into a red-black tree

**IDEA:** Insert  $x$  in tree. Color  $x$  red. Only red-black property 3 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring.

## Example:

- Insert  $x = 15$ .
- Recolor, moving the violation up the tree.
- RIGHT-ROTATE(18).
- LEFT-ROTATE(7) and recolor.





# Pseudocode

RB-INSERT( $T, x$ )

  TREE-INSERT( $T, x$ )

$\text{color}[x] \leftarrow \text{RED}$    ▷ only RB property 3 can be violated

**while**  $x \neq \text{root}[T]$  and  $\text{color}[p[x]] = \text{RED}$

**do if**  $p[x] = \text{left}[p[p[x]]]$

**then**  $y \leftarrow \text{right}[p[p[x]]]$                            ▷  $y = \text{aunt/uncle of } x$

**if**  $\text{color}[y] = \text{RED}$

**then** ⟨Case 1⟩

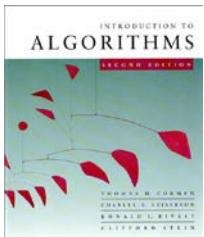
**else if**  $x = \text{right}[p[x]]$

**then** ⟨Case 2⟩   ▷ Case 2 falls into Case 3

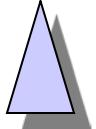
                ⟨Case 3⟩

**else** ⟨“then” clause with “left” and “right” swapped⟩

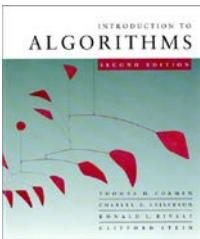
$\text{color}[\text{root}[T]] \leftarrow \text{BLACK}$



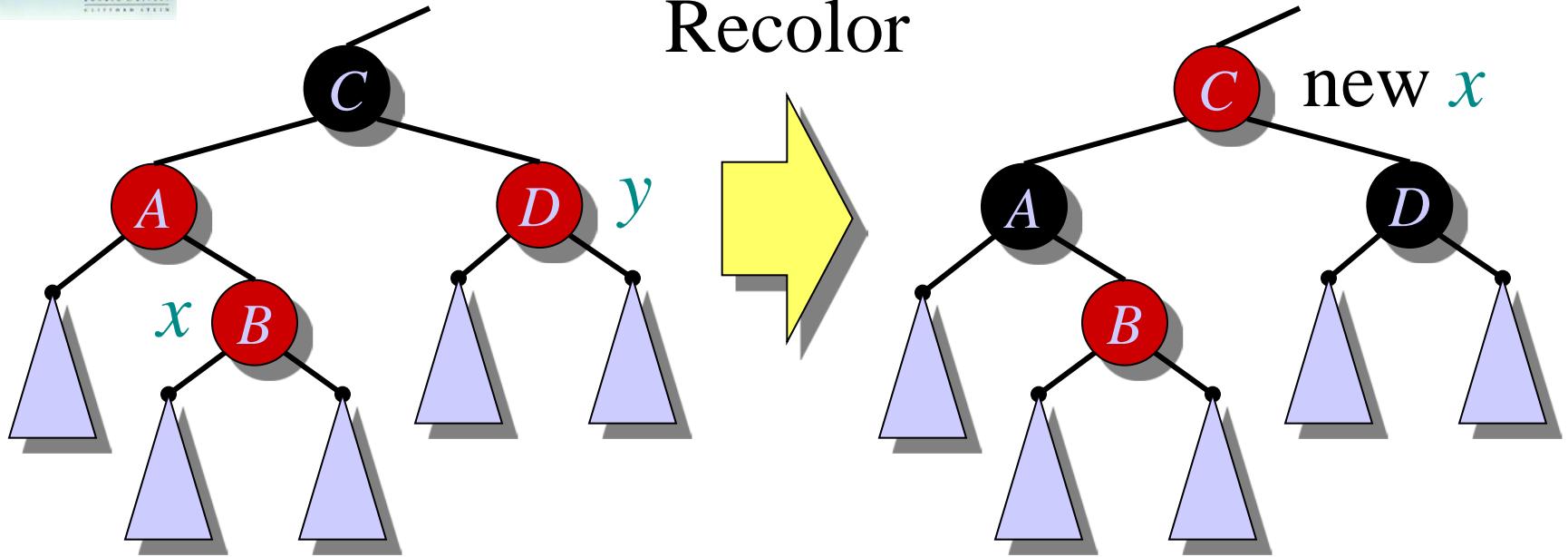
# Graphical notation

Let  denote a subtree with a black root.

All 's have the same black-height.

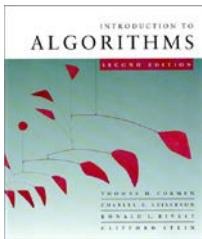


# Case 1

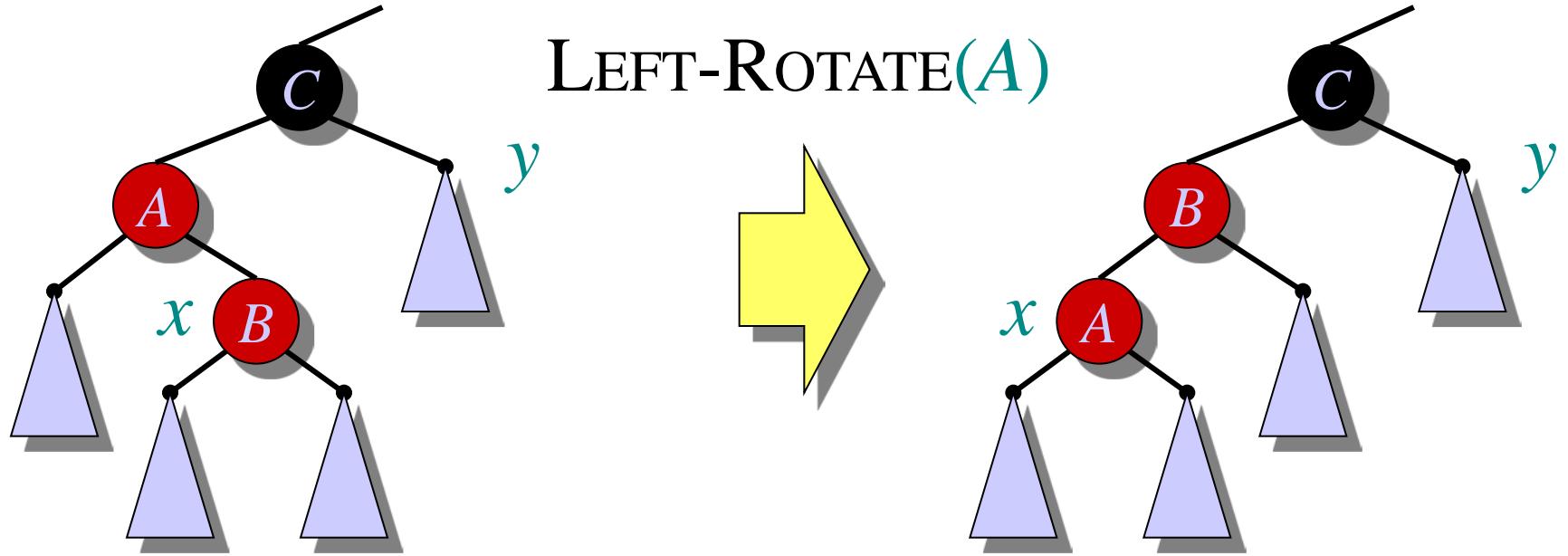


(Or, children of  $A$  are swapped.)

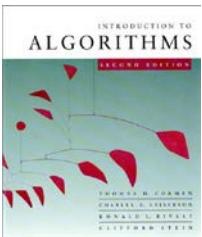
Push  $C$ 's black onto  $A$  and  $D$ , and recurse, since  $C$ 's parent may be red.



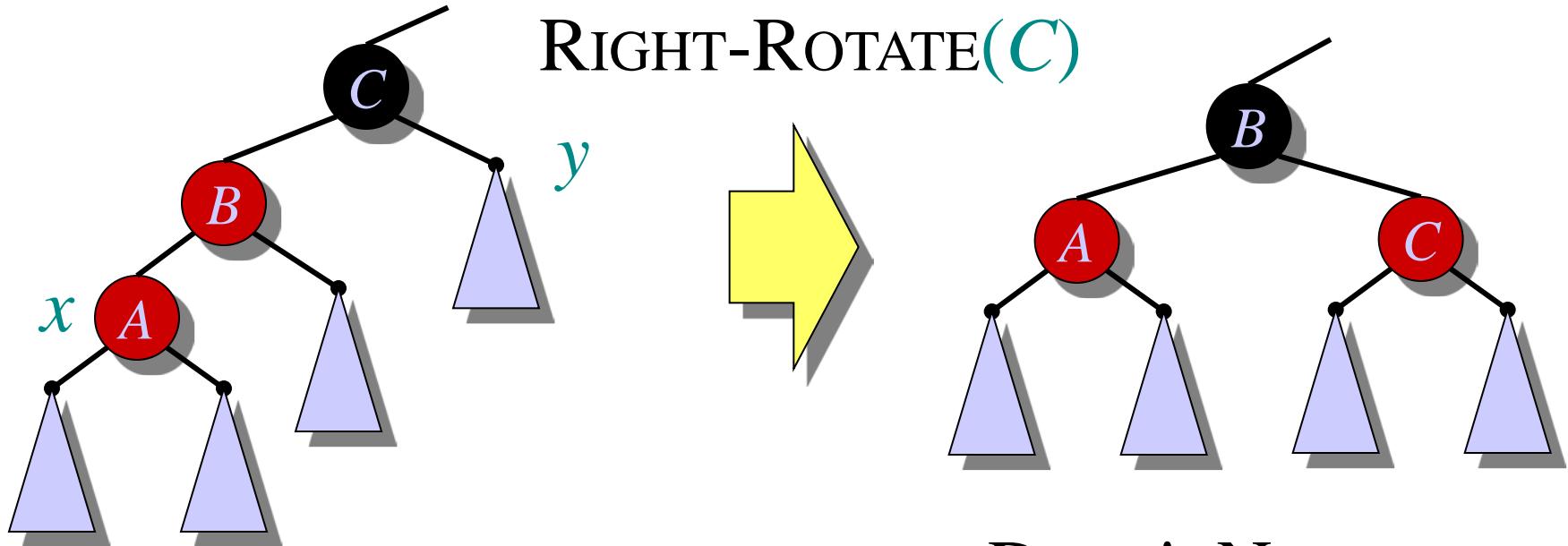
# Case 2



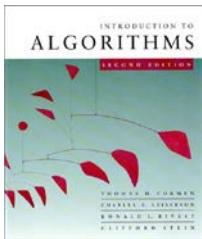
Transform to Case 3.



# Case 3



Done! No more violations of RB property 3 are possible.



# Analysis

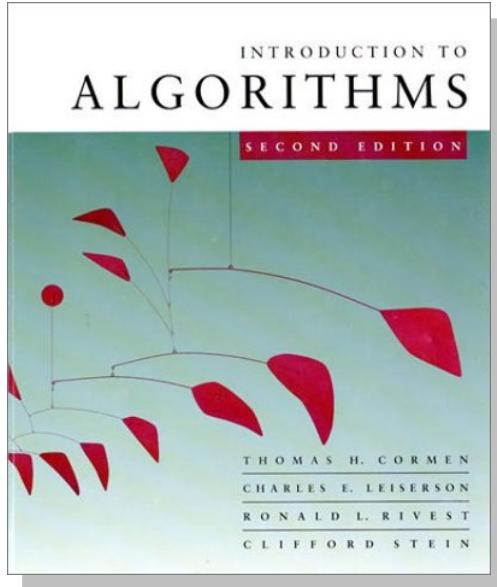
- Go up the tree performing Case 1, which only recolors nodes.
- If Case 2 or Case 3 occurs, perform 1 or 2 rotations, and terminate.

**Running time:**  $O(\lg n)$  with  $O(1)$  rotations.

RB-DELETE — same asymptotic running time and number of rotations as RB-INSERT (see textbook).

# *Introduction to Algorithms*

## 6.046J/18.401J

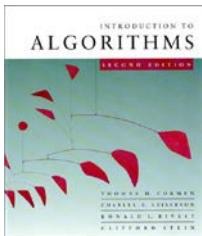


## LECTURE 11

### Augmenting Data Structures

- Dynamic order statistics
- Methodology
- Interval trees

Prof. Charles E. Leiserson



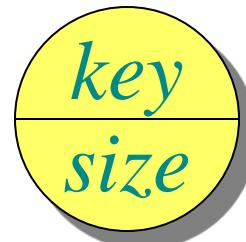
# Dynamic order statistics

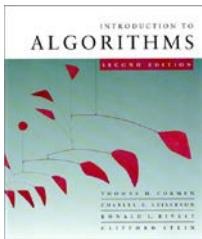
**OS-SELECT( $i, S$ ):** returns the  $i$ th smallest element in the dynamic set  $S$ .

**OS-RANK( $x, S$ ):** returns the rank of  $x \in S$  in the sorted order of  $S$ 's elements.

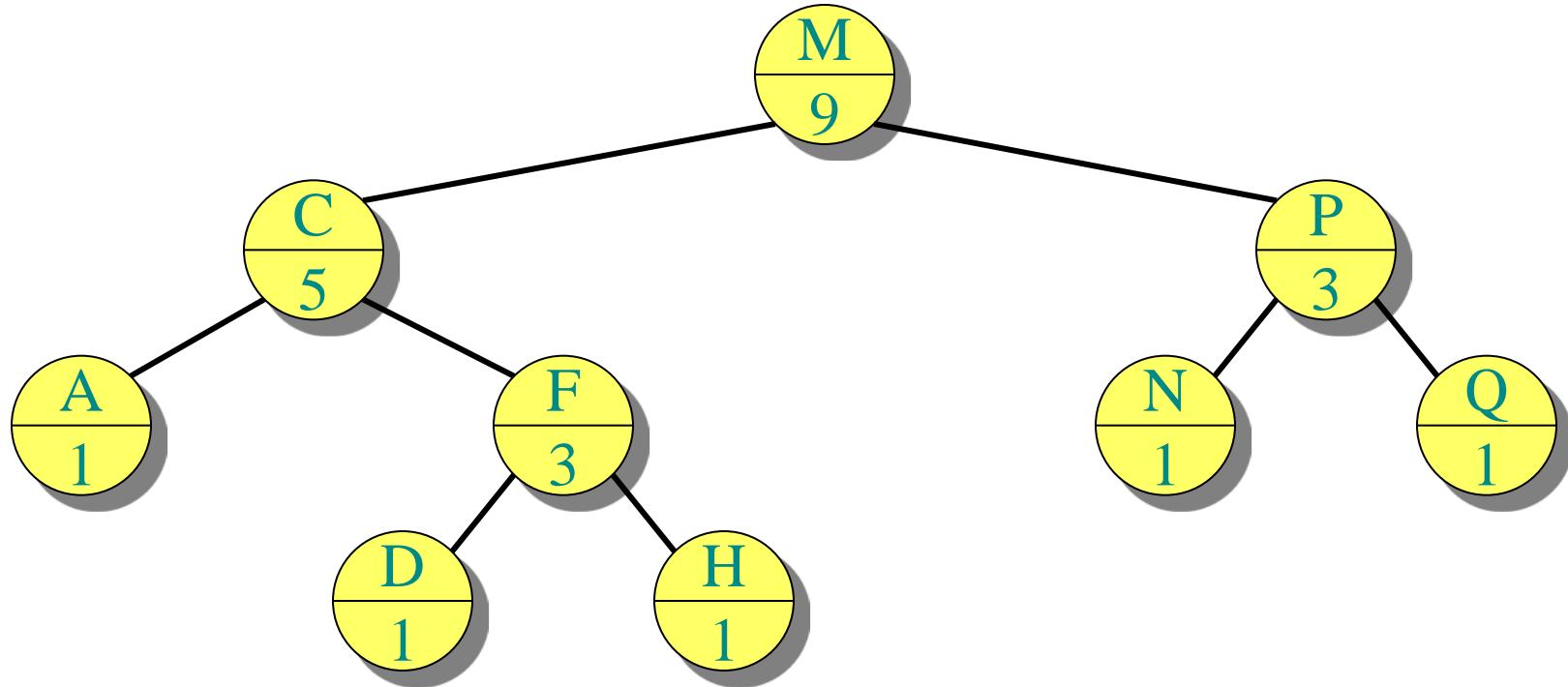
**IDEA:** Use a red-black tree for the set  $S$ , but keep subtree sizes in the nodes.

Notation for nodes:

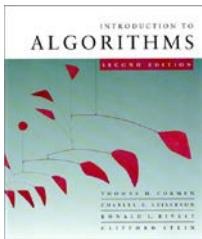




# Example of an OS-tree



$$\text{size}[x] = \text{size}[\text{left}[x]] + \text{size}[\text{right}[x]] + 1$$



# Selection

**Implementation trick:** Use a *sentinel* (dummy record) for `NIL` such that  $\text{size}[\text{NIL}] = 0$ .

**OS-SELECT( $x, i$ )**  $\triangleright$   $i$ th smallest element in the subtree rooted at  $x$

$k \leftarrow \text{size}[\text{left}[x]] + 1$   $\triangleright k = \text{rank}(x)$

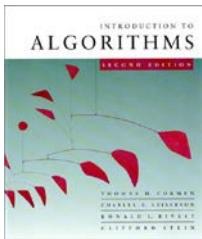
**if**  $i = k$  **then return**  $x$

**if**  $i < k$

**then return** OS-SELECT( $\text{left}[x], i$ )

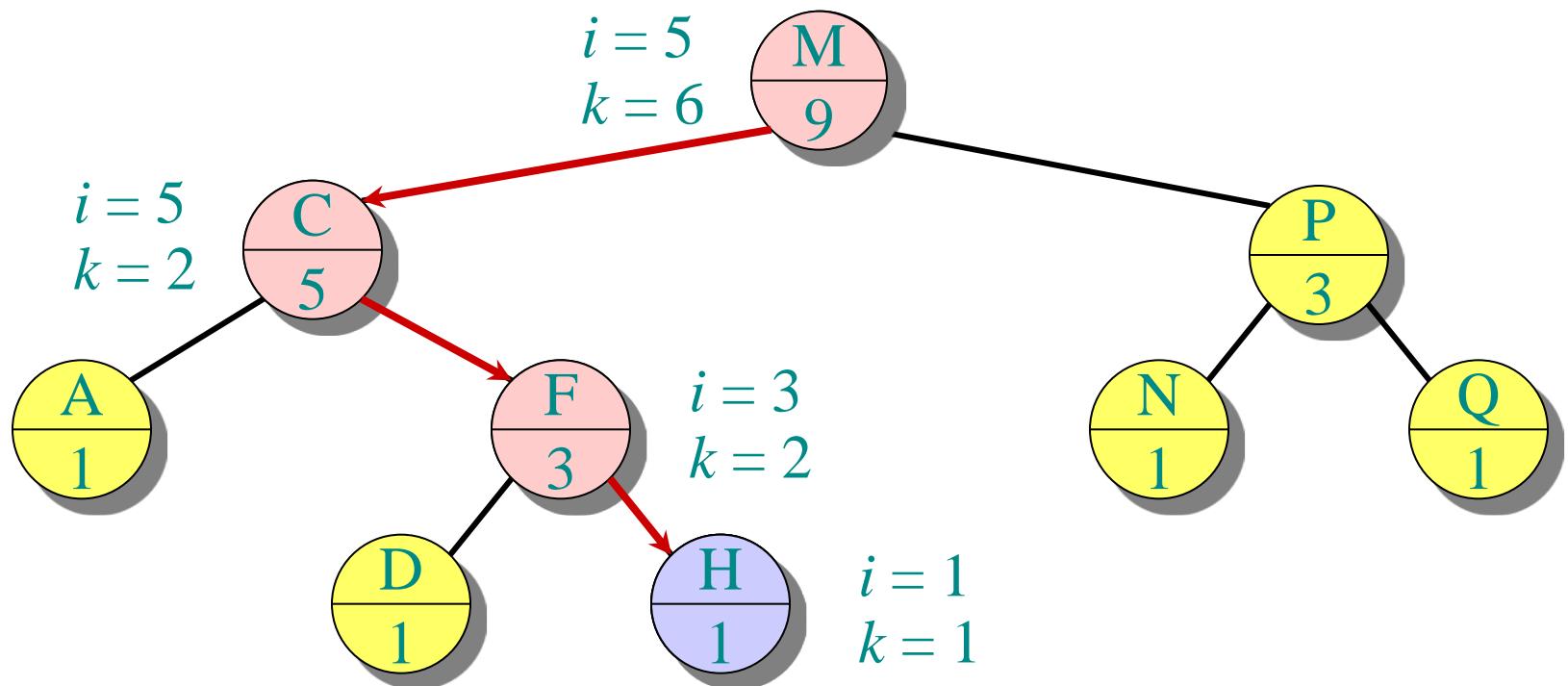
**else return** OS-SELECT( $\text{right}[x], i - k$ )

(OS-RANK is in the textbook.)

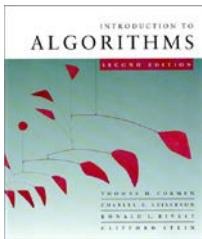


# Example

OS-SELECT(*root*, 5)



Running time =  $O(h) = O(\lg n)$  for red-black trees.

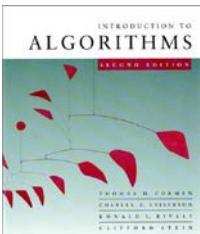


# Data structure maintenance

- Q.** Why not keep the ranks themselves in the nodes instead of subtree sizes?
- A.** They are hard to maintain when the red-black tree is modified.

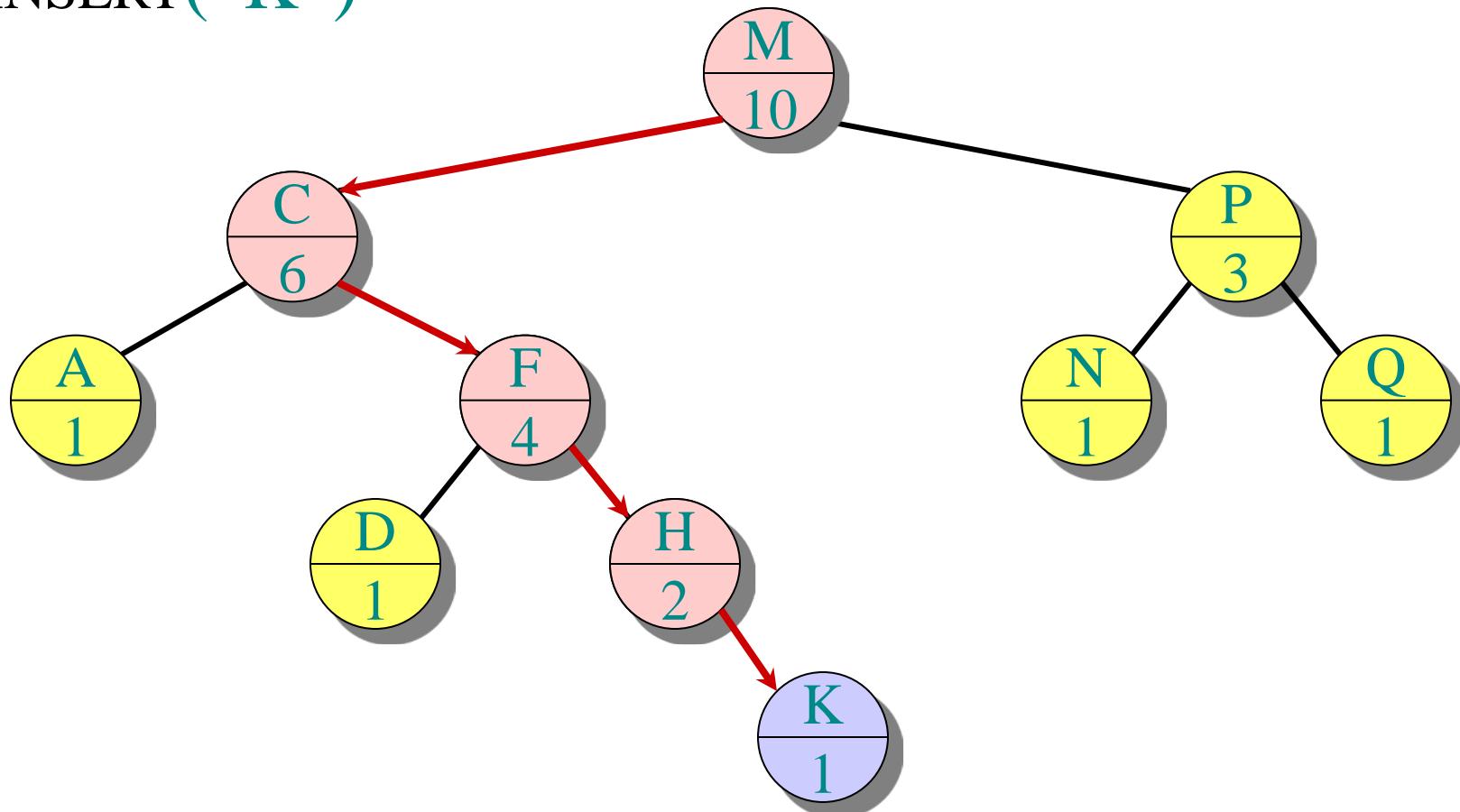
**Modifying operations:** INSERT and DELETE.

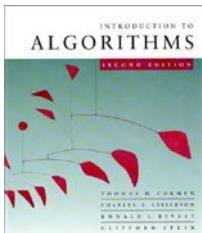
**Strategy:** Update subtree sizes when inserting or deleting.



# Example of insertion

INSERT("K")



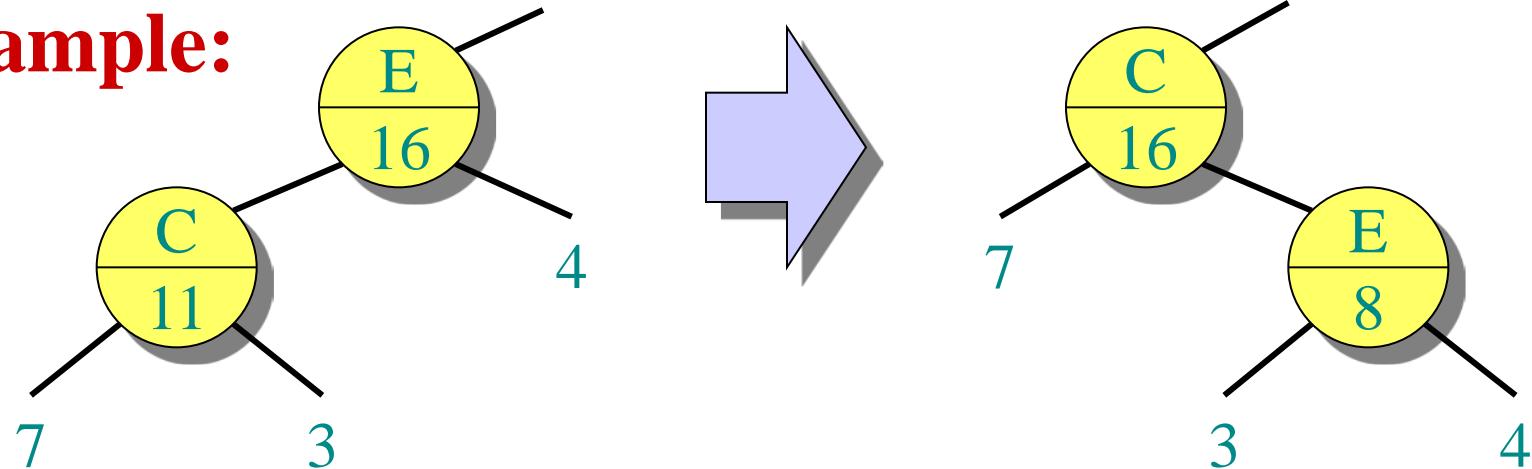


# Handling rebalancing

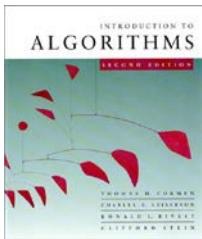
Don't forget that RB-INSERT and RB-DELETE may also need to modify the red-black tree in order to maintain balance.

- *Recolorings*: no effect on subtree sizes.
- *Rotations*: fix up subtree sizes in  $O(1)$  time.

**Example:**



∴ RB-INSERT and RB-DELETE still run in  $O(\lg n)$  time.

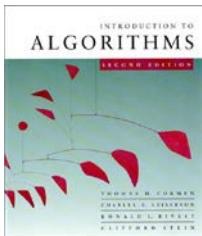


# Data-structure augmentation

**Methodology:** (*e.g., order-statistics trees*)

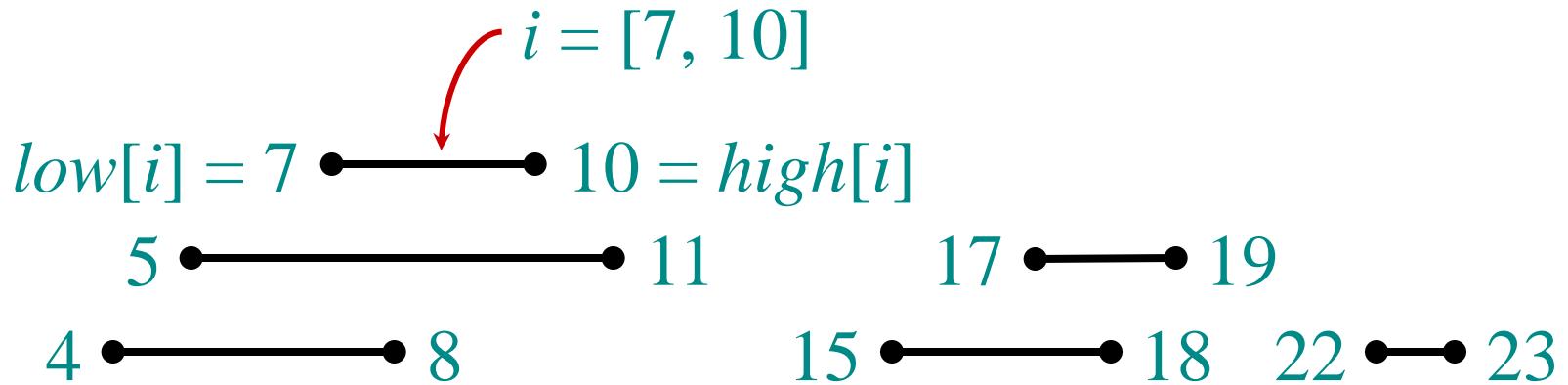
1. Choose an underlying data structure (*red-black trees*).
2. Determine additional information to be stored in the data structure (*subtree sizes*).
3. Verify that this information can be maintained for modifying operations (*RB-INSERT, RB-DELETE — don't forget rotations*).
4. Develop new dynamic-set operations that use the information (*OS-SELECT and OS-RANK*).

These steps are guidelines, not rigid rules.

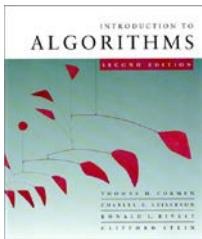


# Interval trees

**Goal:** To maintain a dynamic set of intervals, such as time intervals.

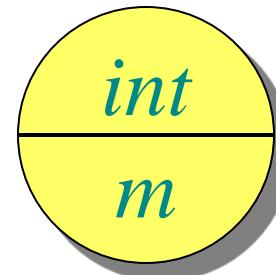


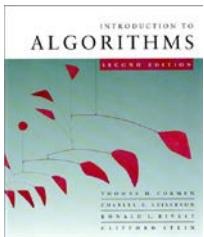
**Query:** For a given query interval  $i$ , find an interval in the set that overlaps  $i$ .



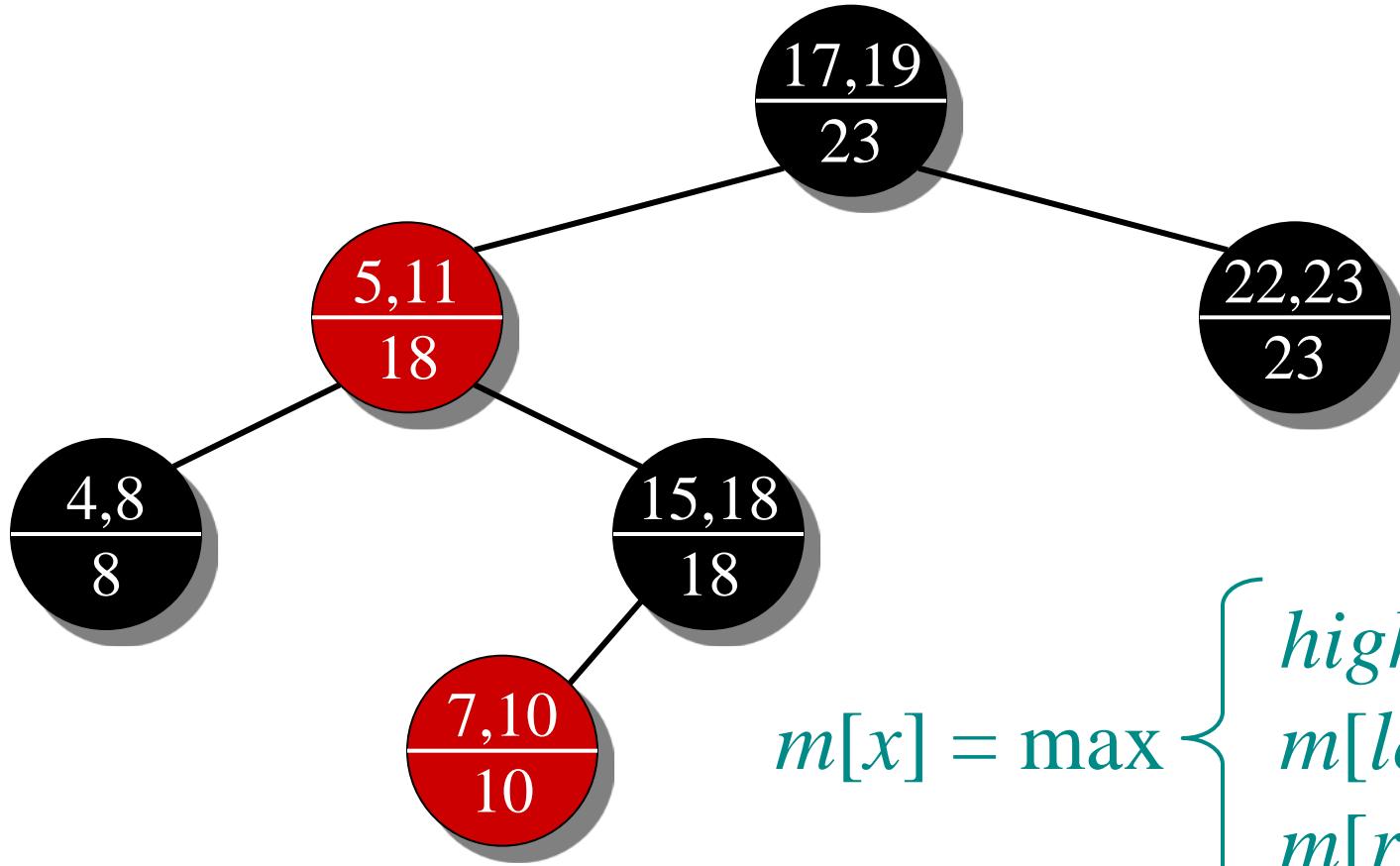
# Following the methodology

1. *Choose an underlying data structure.*
  - Red-black tree keyed on low (left) endpoint.
2. *Determine additional information to be stored in the data structure.*
  - Store in each node  $x$  the largest value  $m[x]$  in the subtree rooted at  $x$ , as well as the interval  $int[x]$  corresponding to the key.

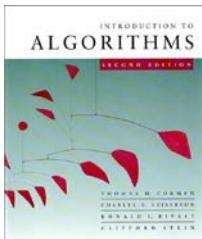




# Example interval tree



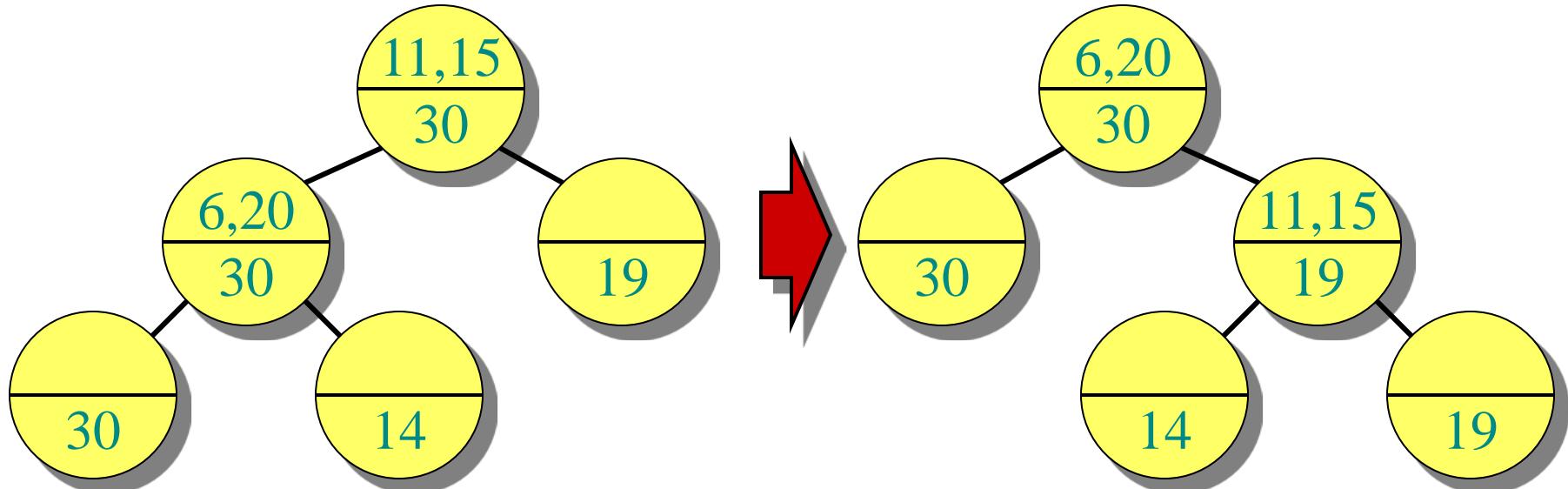
$$m[x] = \max \begin{cases} \text{high[int}[x]] \\ m[\text{left}[x]] \\ m[\text{right}[x]] \end{cases}$$



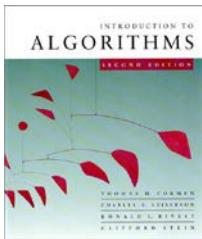
# Modifying operations

3. Verify that this information can be maintained for modifying operations.

- INSERT: Fix  $m$ 's on the way down.
- Rotations — Fixup =  $O(1)$  time per rotation:



Total INSERT time =  $O(\lg n)$ ; DELETE similar.



# New operations

4. Develop new dynamic-set operations that use the information.

INTERVAL-SEARCH( $i$ )

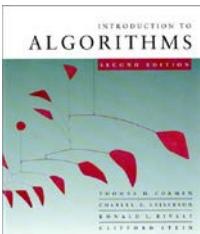
$x \leftarrow root$

**while**  $x \neq \text{NIL}$  and ( $low[i] > high[int[x]]$   
or  $low[int[x]] > high[i]$ )

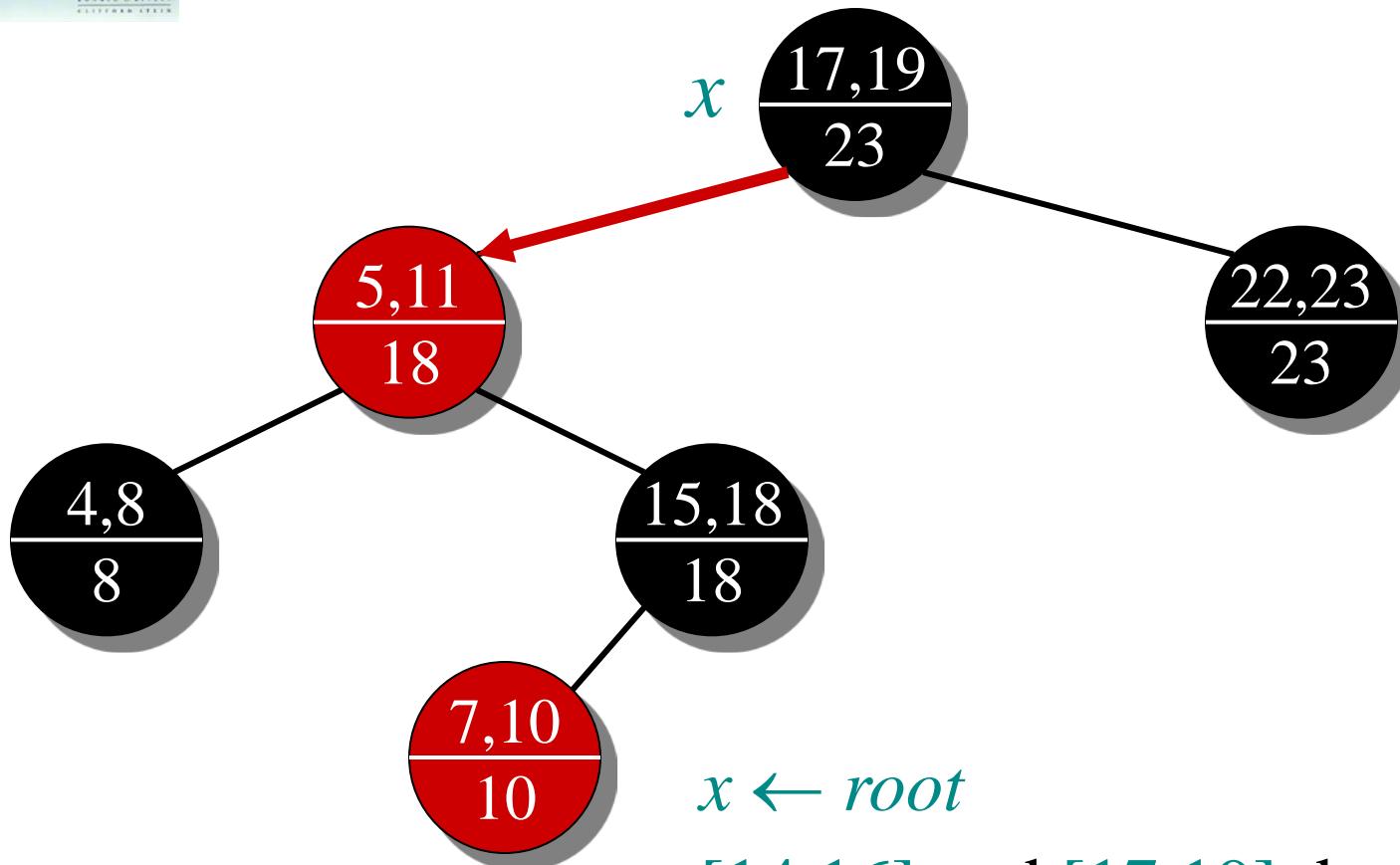
**do** ▷  $i$  and  $int[x]$  don't overlap

**if**  $left[x] \neq \text{NIL}$  and  $low[i] \leq m[left[x]]$   
**then**  $x \leftarrow left[x]$   
**else**  $x \leftarrow right[x]$

**return**  $x$



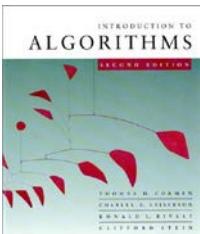
# Example 1: INTERVAL-SEARCH([14,16])



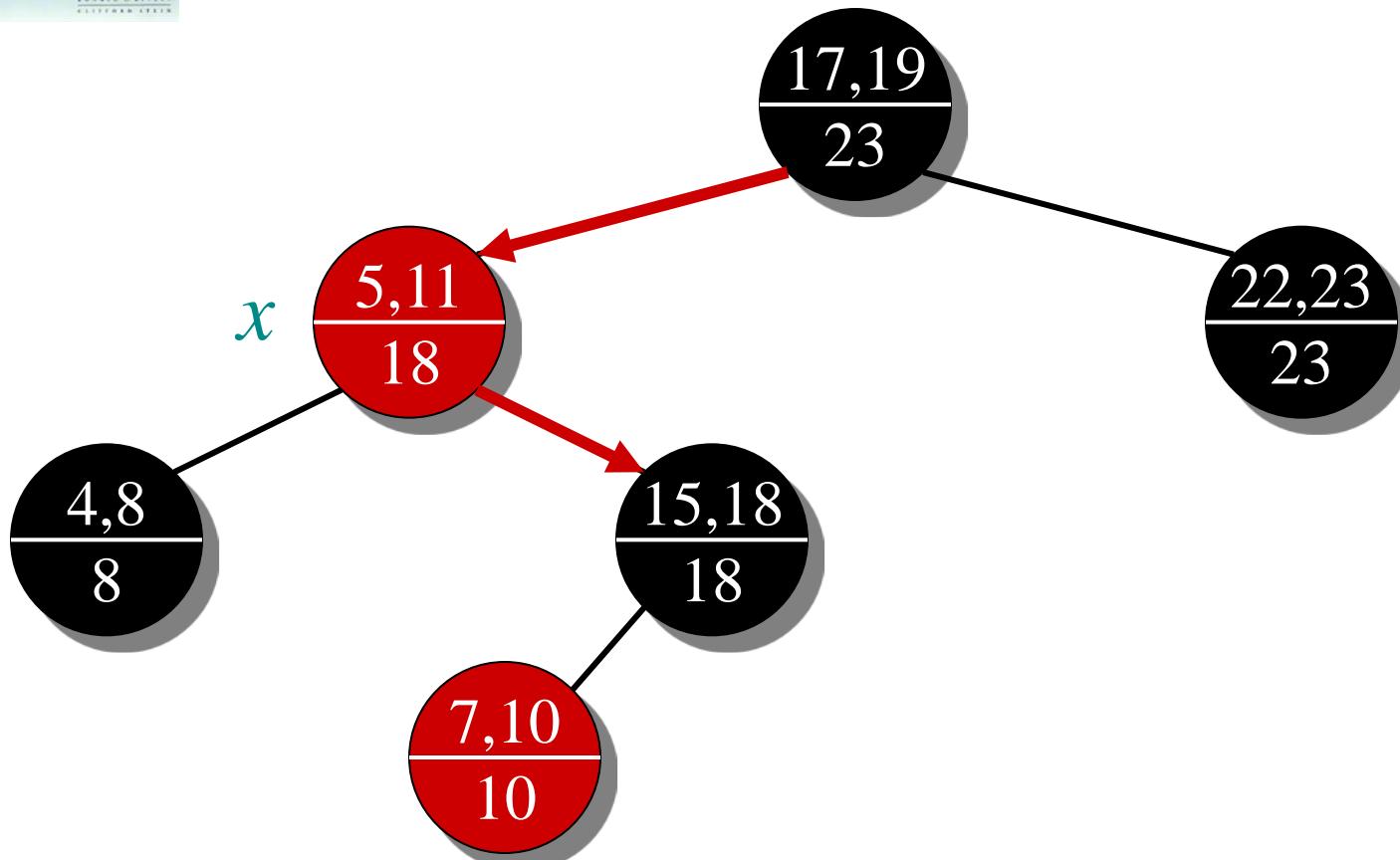
$x \leftarrow root$

[14,16] and [17,19] don't overlap

$14 \leq 18 \Rightarrow x \leftarrow left[x]$

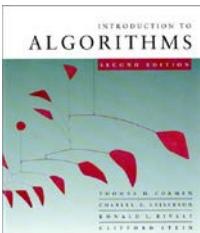


# Example 1: INTERVAL-SEARCH([14,16])

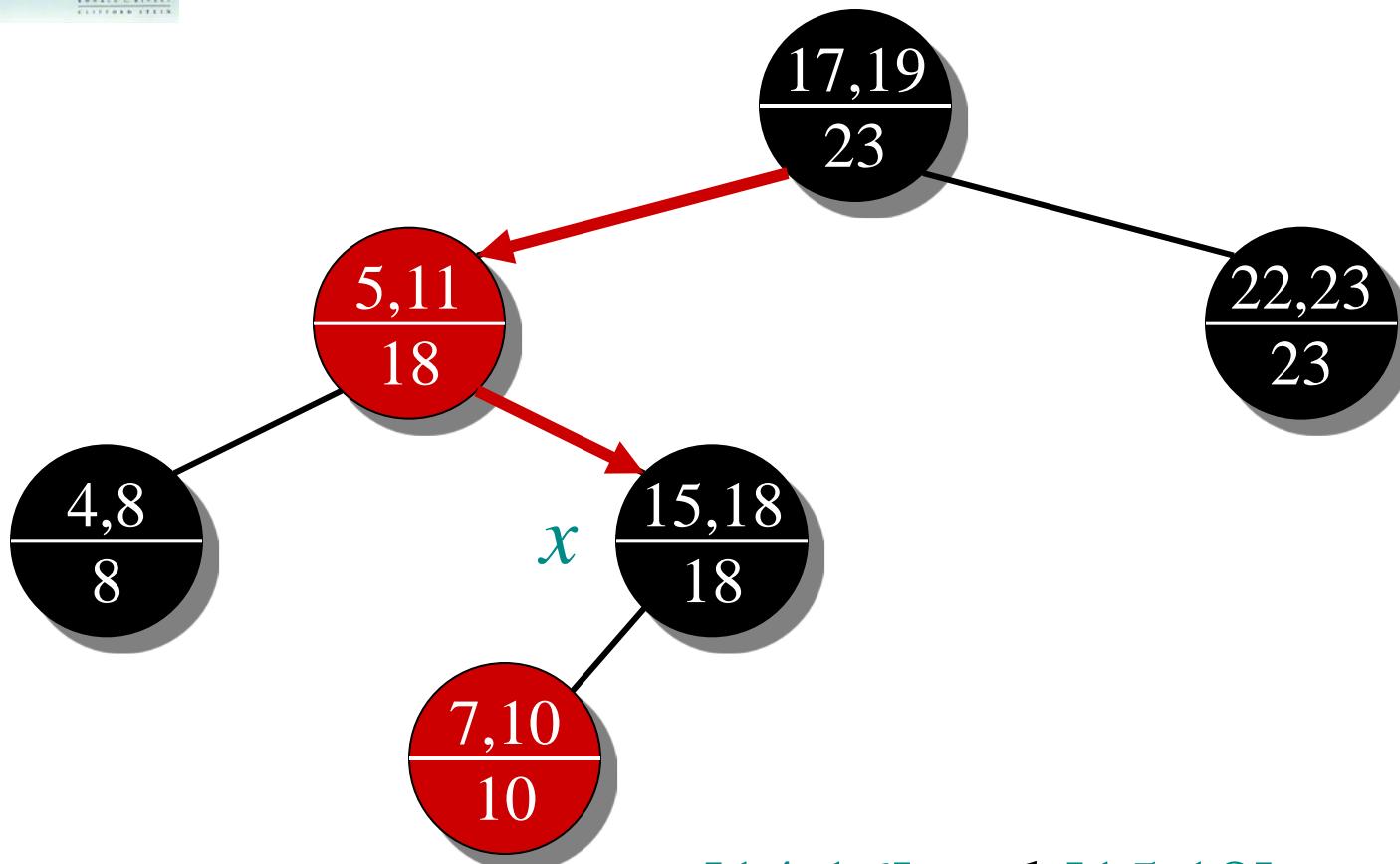


[14,16] and [5,11] don't overlap

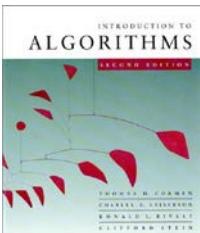
$14 > 8 \Rightarrow x \leftarrow \text{right}[x]$



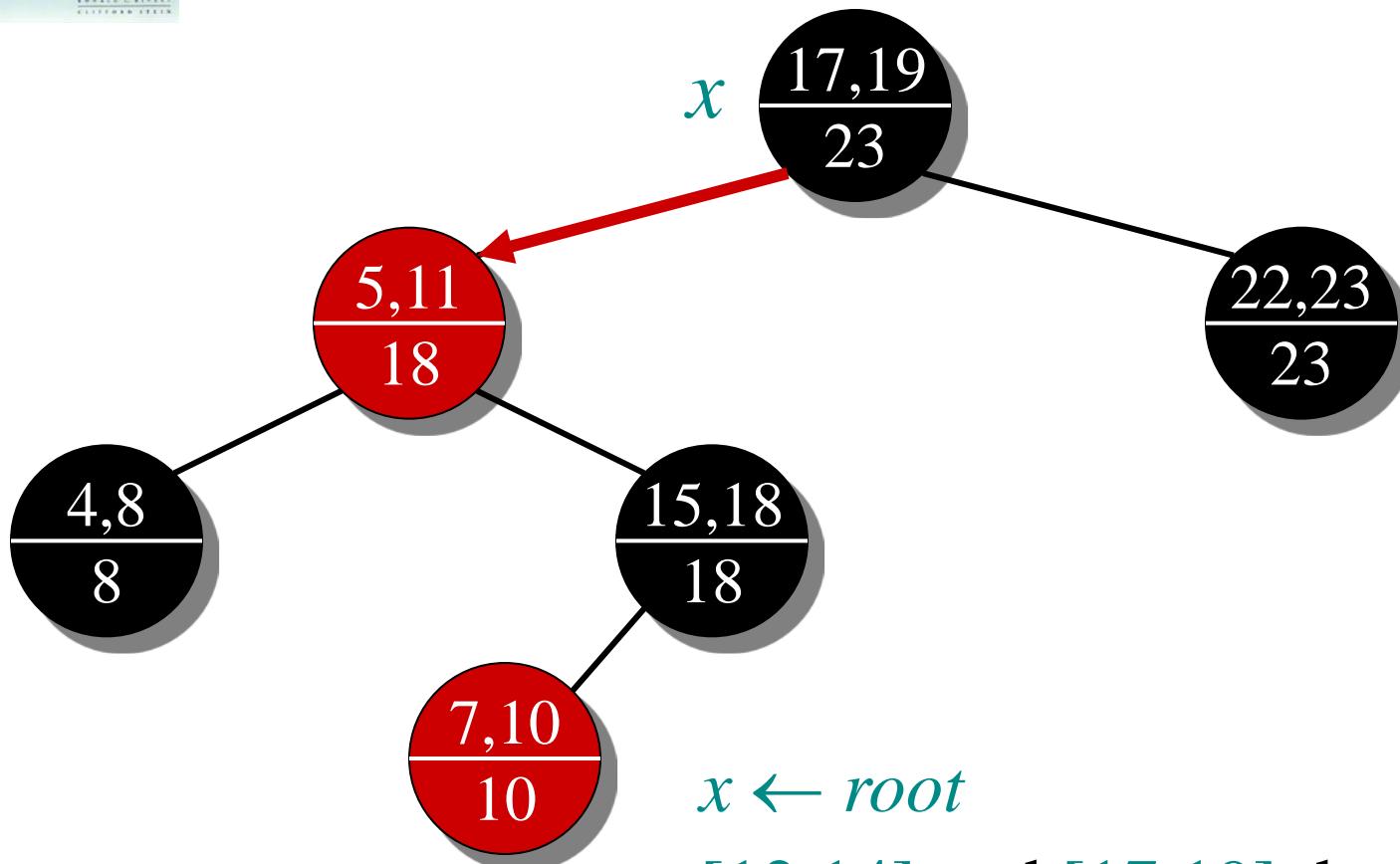
# Example 1: INTERVAL-SEARCH([14,16])



[14,16] and [15,18] overlap  
**return [15,18]**



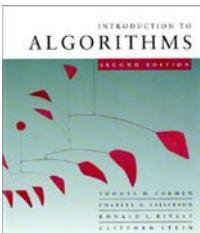
# Example 2: INTERVAL-SEARCH([12,14])



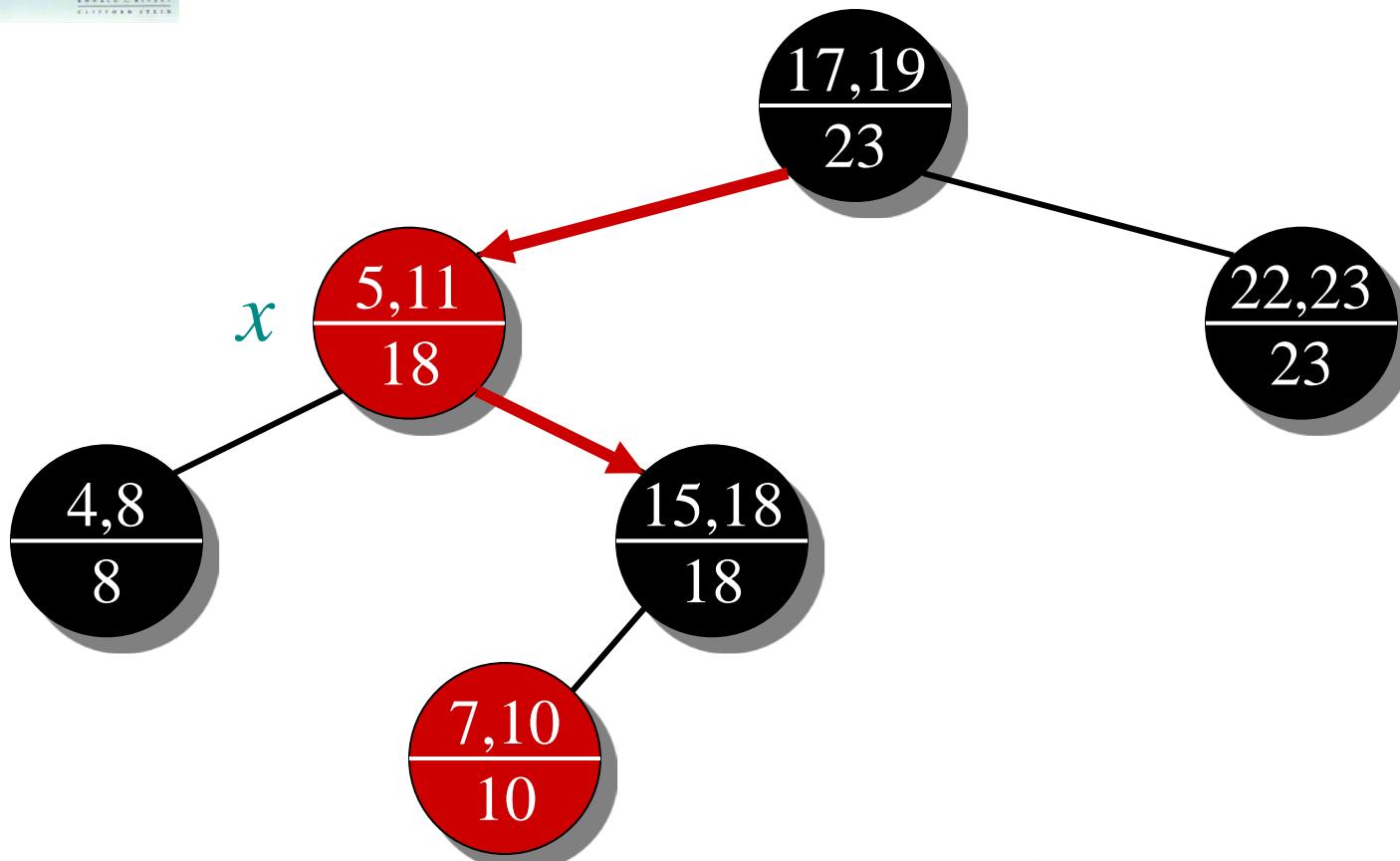
$x \leftarrow root$

[12,14] and [17,19] don't overlap

$12 \leq 18 \Rightarrow x \leftarrow left[x]$

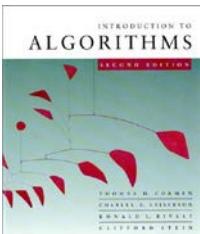


# Example 2: INTERVAL-SEARCH([12,14])

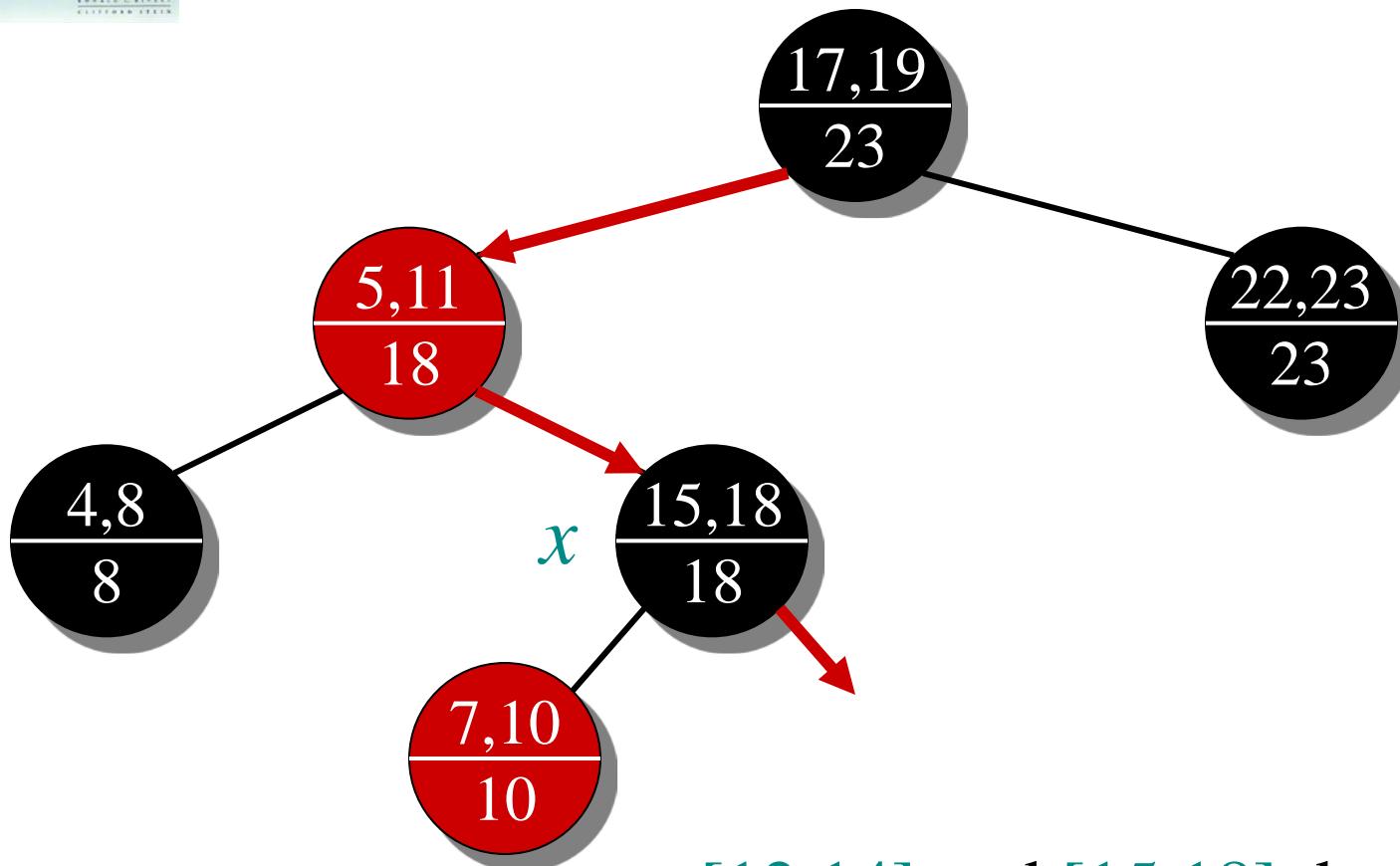


[12,14] and [5,11] don't overlap

$12 > 8 \Rightarrow x \leftarrow right[x]$

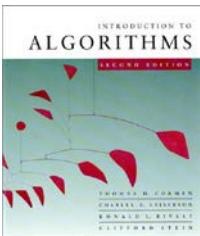


# Example 2: INTERVAL-SEARCH([12,14])

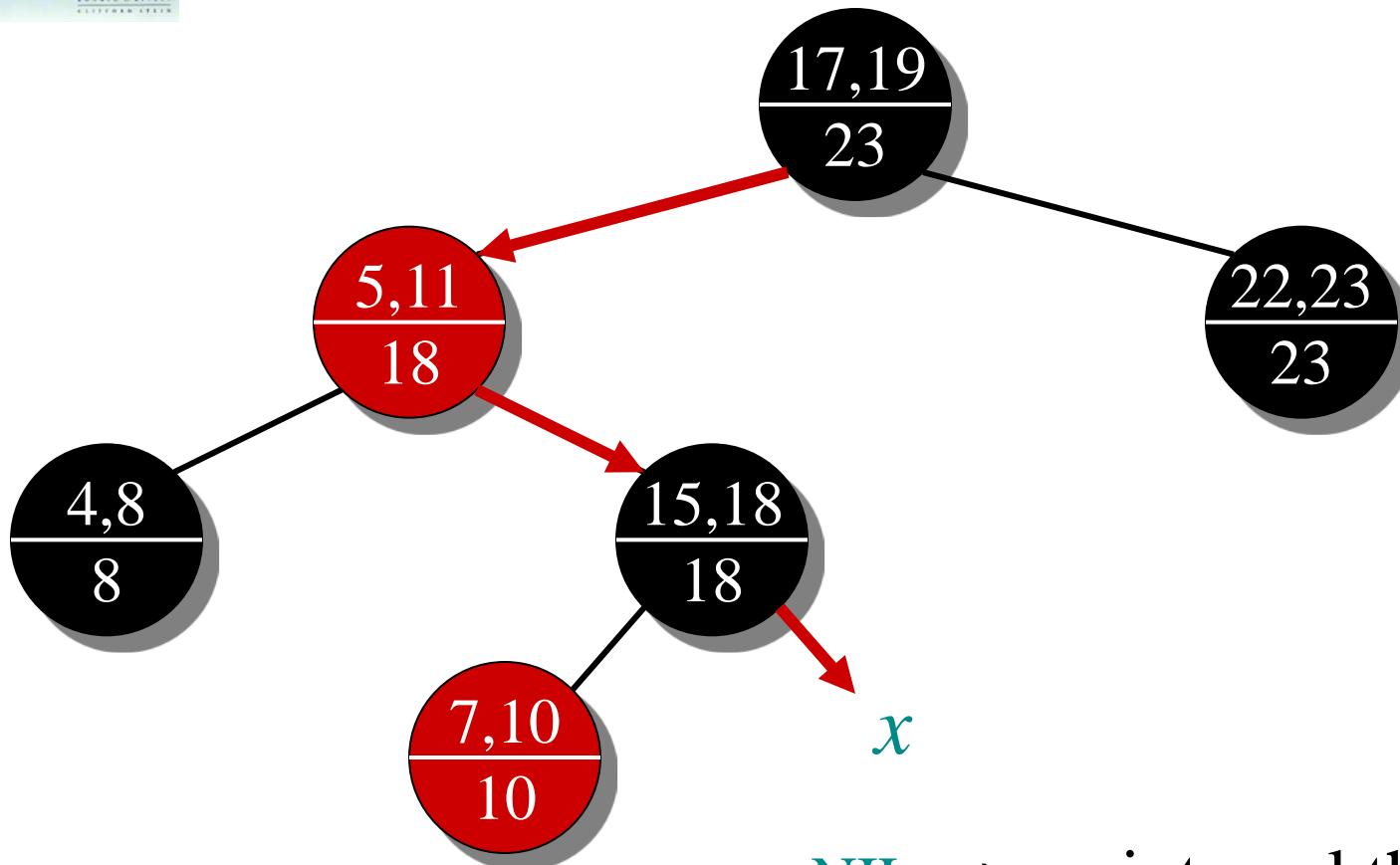


[12,14] and [15,18] don't overlap

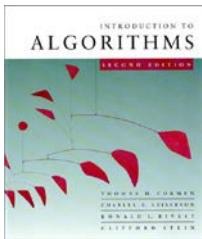
$12 > 10 \Rightarrow x \leftarrow right[x]$



# Example 2: INTERVAL-SEARCH([12,14])



$x = \text{NIL} \Rightarrow$  no interval that overlaps [12,14] exists



# Analysis

Time =  $O(h) = O(\lg n)$ , since INTERVAL-SEARCH does constant work at each level as it follows a simple path down the tree.

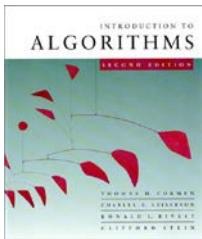
List *all* overlapping intervals:

- Search, list, delete, repeat.
- Insert them all again at the end.

Time =  $O(k \lg n)$ , where  $k$  is the total number of overlapping intervals.

This is an ***output-sensitive*** bound.

Best algorithm to date:  $O(k + \lg n)$ .



# Correctness

**Theorem.** Let  $L$  be the set of intervals in the left subtree of node  $x$ , and let  $R$  be the set of intervals in  $x$ 's right subtree.

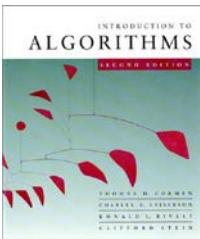
- If the search goes right, then

$$\{ i' \in L : i' \text{ overlaps } i \} = \emptyset.$$

- If the search goes left, then

$$\begin{aligned} \{ i' \in L : i' \text{ overlaps } i \} &= \emptyset \\ \Rightarrow \{ i' \in R : i' \text{ overlaps } i \} &= \emptyset. \end{aligned}$$

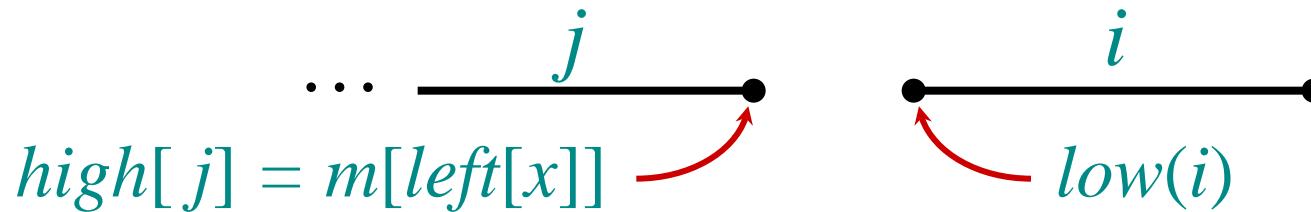
*In other words, it's always safe to take only 1 of the 2 children: we'll either find something, or nothing was to be found.*



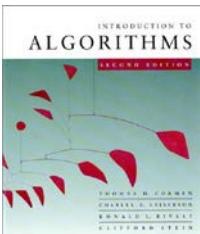
# Correctness proof

*Proof.* Suppose first that the search goes right.

- If  $\text{left}[x] = \text{NIL}$ , then we're done, since  $L = \emptyset$ .
- Otherwise, the code dictates that we must have  $\text{low}[i] > m[\text{left}[x]]$ . The value  $m[\text{left}[x]]$  corresponds to the high endpoint of some interval  $j \in L$ , and no other interval in  $L$  can have a larger high endpoint than  $\text{high}[j]$ .



- Therefore,  $\{i' \in L : i' \text{ overlaps } i\} = \emptyset$ .

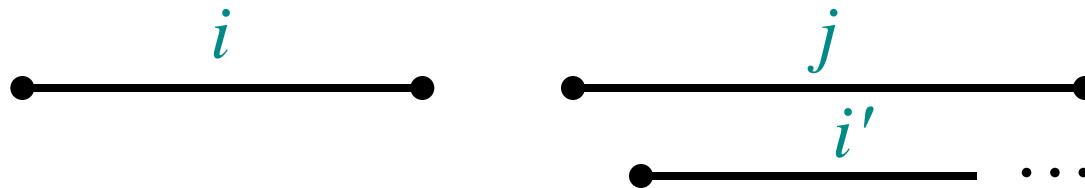


# Proof (continued)

Suppose that the search goes left, and assume that

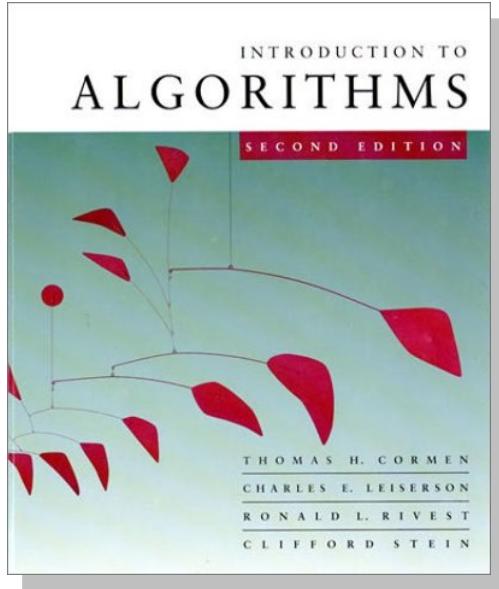
$$\{i' \in L : i' \text{ overlaps } i\} = \emptyset.$$

- Then, the code dictates that  $\text{low}[i] \leq m[\text{left}[x]] = \text{high}[j]$  for some  $j \in L$ .
- Since  $j \in L$ , it does not overlap  $i$ , and hence  $\text{high}[i] < \text{low}[j]$ .
- But, the binary-search-tree property implies that for all  $i' \in R$ , we have  $\text{low}[j] \leq \text{low}[i']$ .
- But then  $\{i' \in R : i' \text{ overlaps } i\} = \emptyset$ . □



# *Introduction to Algorithms*

## 6.046J/18.401J

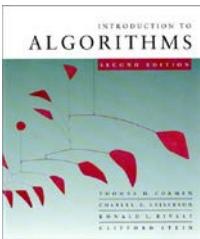


### LECTURE 12

#### Skip Lists

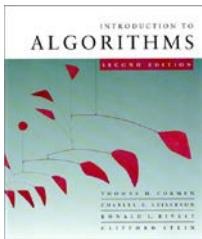
- Data structure
- Randomized insertion
- With-high-probability bound
- Analysis
- Coin flipping

Prof. Erik D. Demaine



# Skip lists

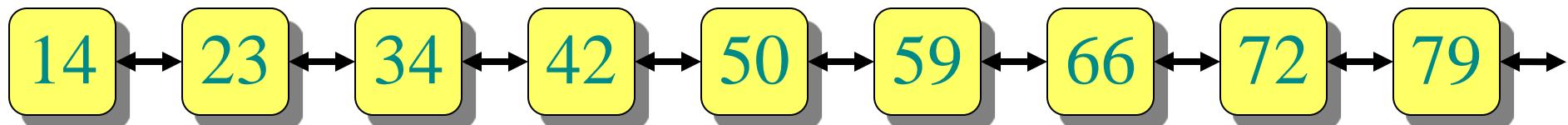
- Simple randomized dynamic search structure
  - Invented by William Pugh in 1989
  - Easy to implement
- Maintains a dynamic set of  $n$  elements in  $O(\lg n)$  time per operation in expectation and *with high probability*
  - Strong guarantee on tail of distribution of  $T(n)$
  - $O(\lg n)$  “almost always”

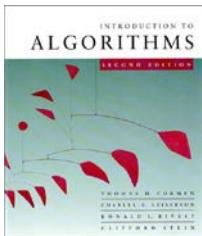


# One linked list

Start from simplest data structure:  
**(sorted) linked list**

- Searches take  $\Theta(n)$  time in worst case
- How can we speed up searches?

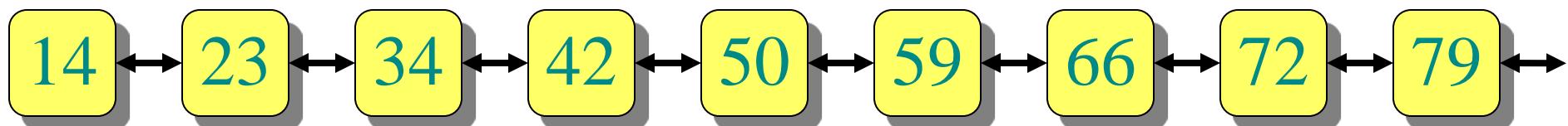


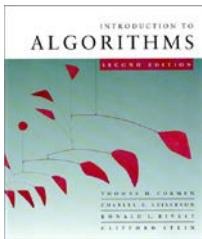


# Two linked lists

Suppose we had *two* sorted linked lists  
(on subsets of the elements)

- Each element can appear in one or both lists
- How can we speed up searches?

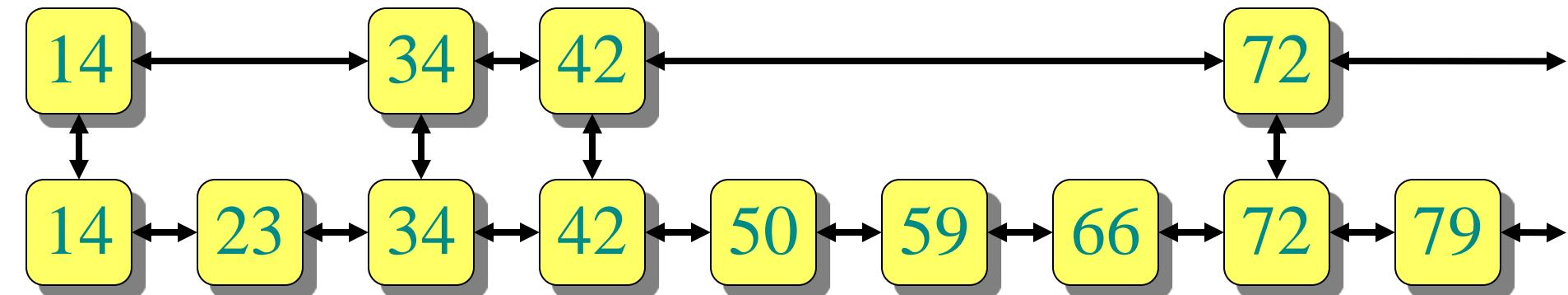


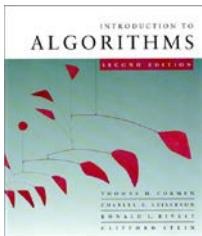


# Two linked lists as a subway

**IDEA:** Express and local subway lines  
(à la New York City 7th Avenue Line)

- Express line connects a few of the stations
- Local line connects all stations
- Links between lines at common stations

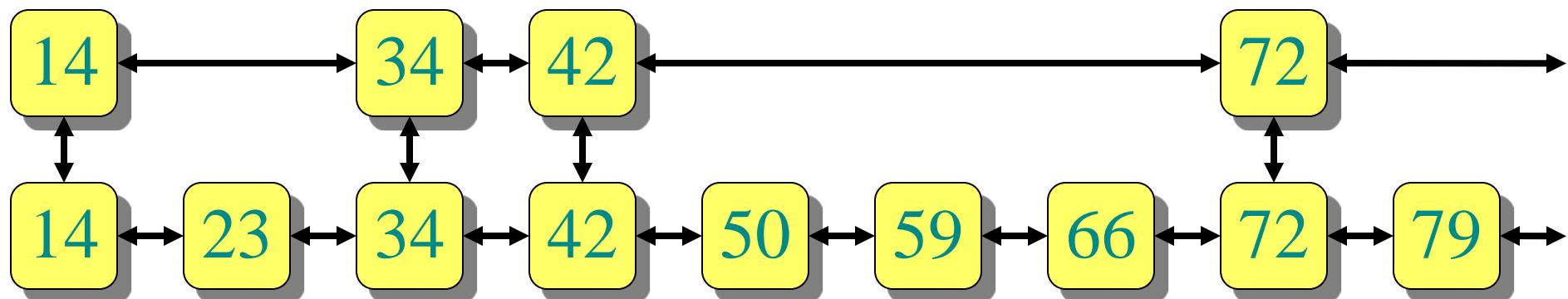


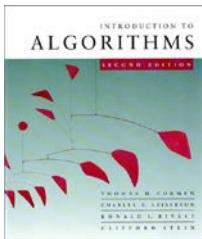


# Searching in two linked lists

SEARCH( $x$ ):

- Walk right in top linked list ( $L_1$ ) until going right would go too far
- Walk down to bottom linked list ( $L_2$ )
- Walk right in  $L_2$  until element found (or not)

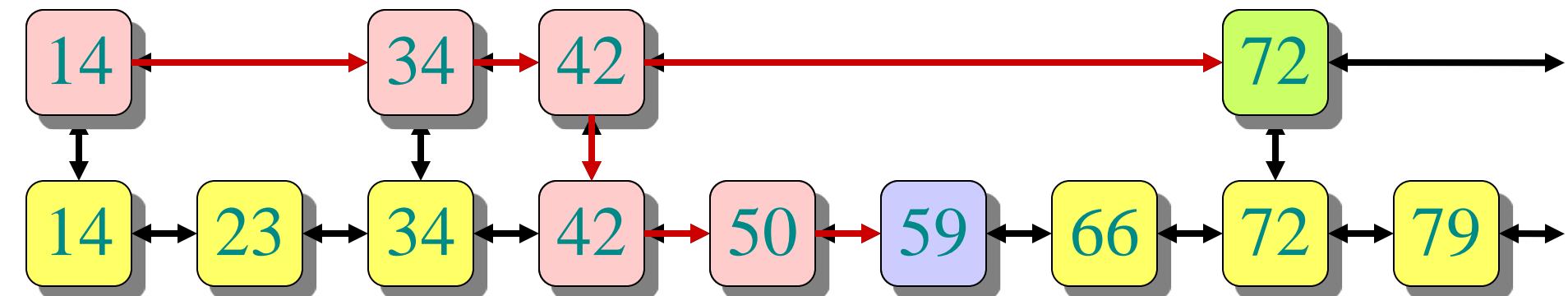


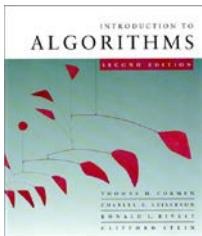


# Searching in two linked lists

EXAMPLE: SEARCH(59)

Too far:  
 $59 < 72$

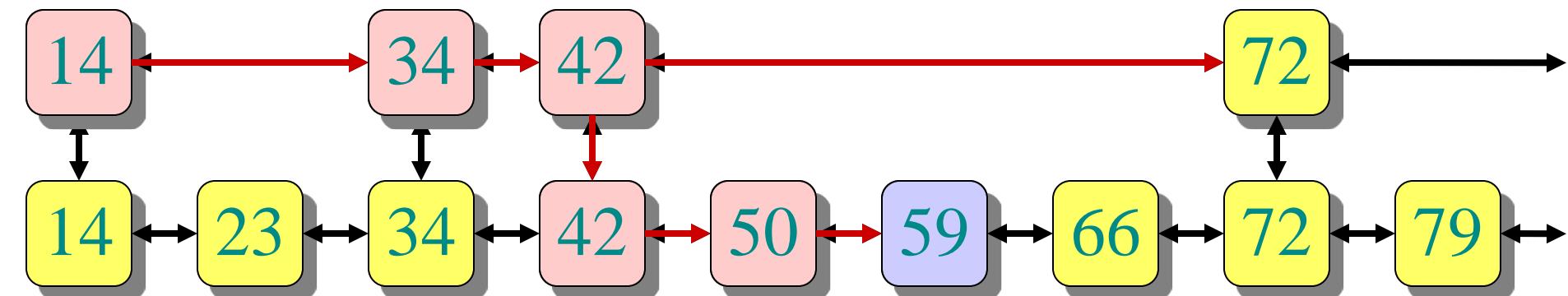


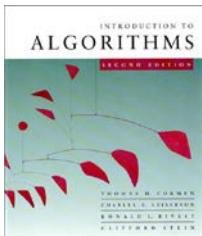


# Design of two linked lists

**QUESTION:** Which nodes should be in  $L_1$ ?

- In a subway, the “popular stations”
- Here we care about *worst-case performance*
- **Best approach:** Evenly space the nodes in  $L_1$
- But *how many nodes* should be in  $L_1$ ?

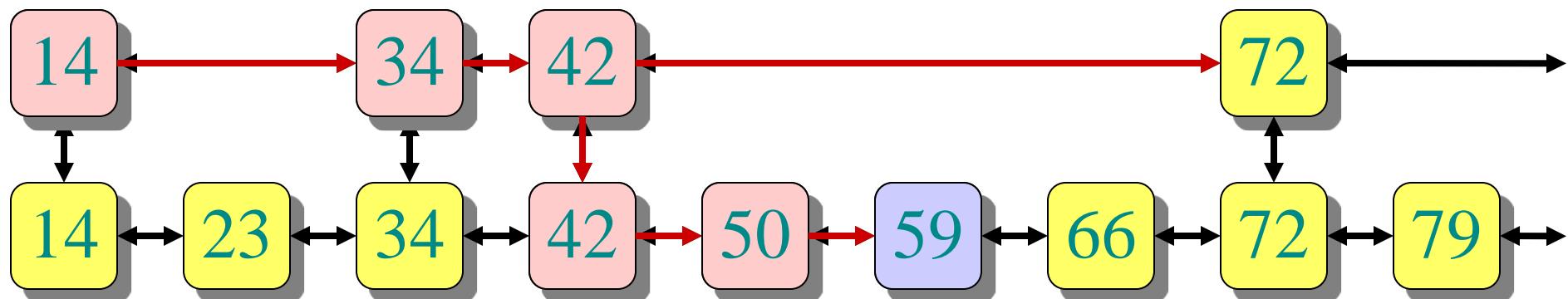


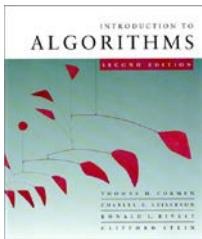


# Analysis of two linked lists

## ANALYSIS:

- Search cost is roughly  $|L_1| + \frac{|L_2|}{|L_1|}$
- Minimized (up to constant factors) when terms are equal
- $|L_1|^2 = |L_2| = n \Rightarrow |L_1| = \sqrt{n}$



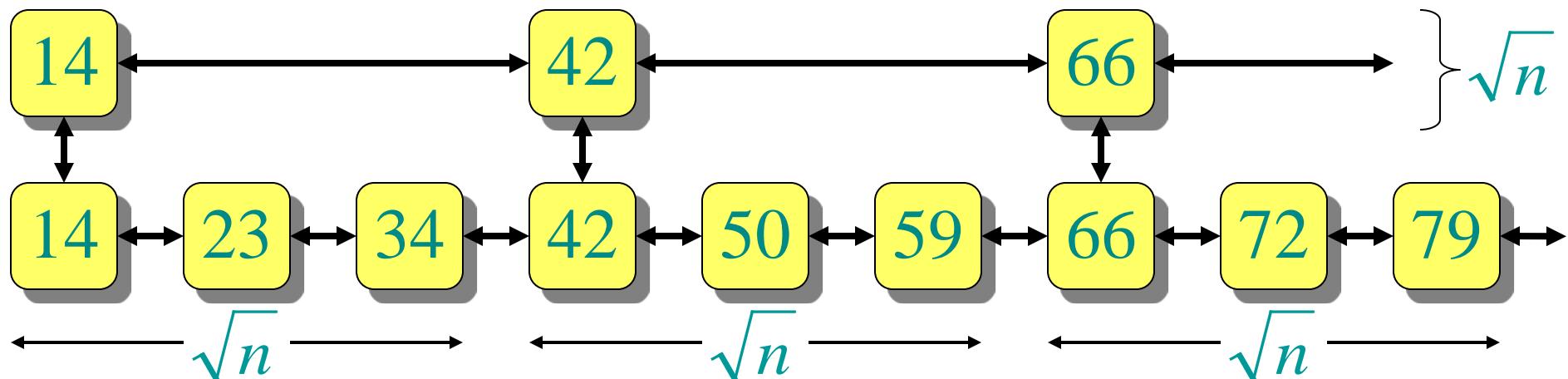


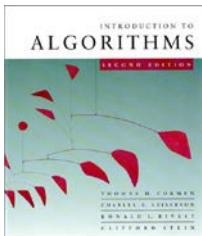
# Analysis of two linked lists

## ANALYSIS:

- $|L_1| = \sqrt{n}$ ,  $|L_2| = n$
- Search cost is roughly

$$|L_1| + \frac{|L_2|}{|L_1|} = \sqrt{n} + \frac{n}{\sqrt{n}} = 2\sqrt{n}$$

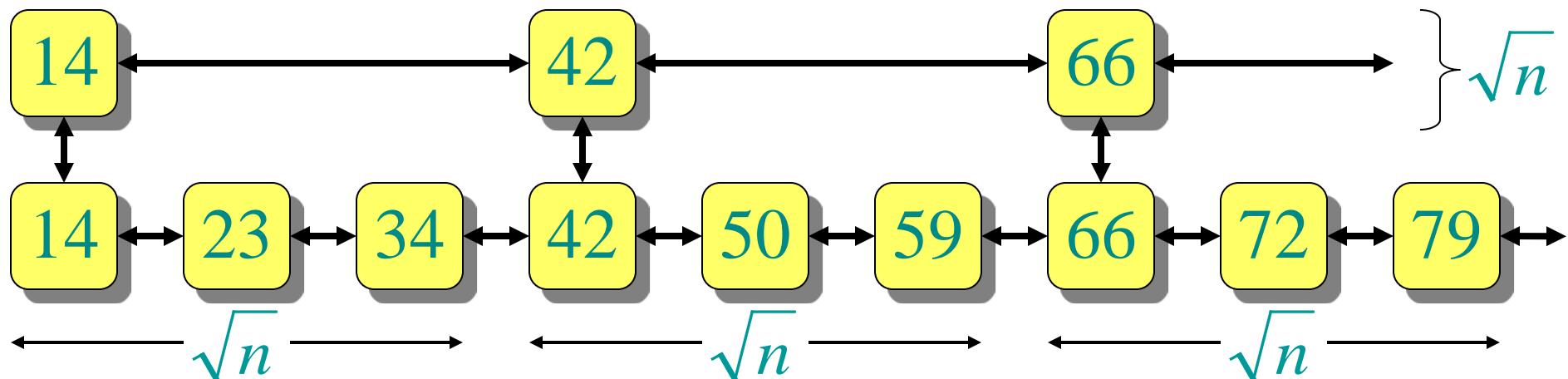


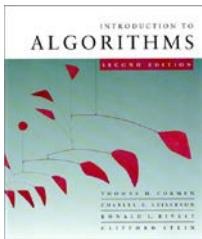


# More linked lists

What if we had more sorted linked lists?

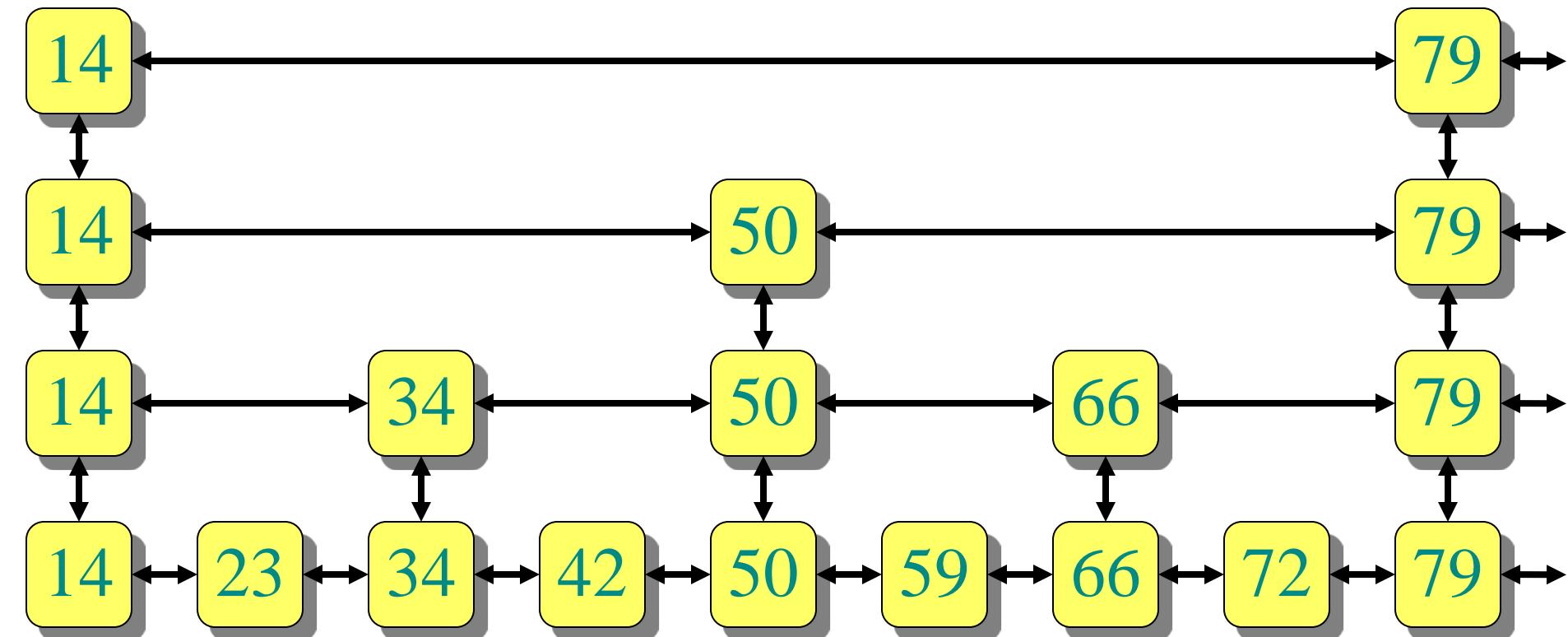
- 2 sorted lists  $\Rightarrow 2 \cdot \sqrt{n}$
- 3 sorted lists  $\Rightarrow 3 \cdot \sqrt[3]{n}$
- $k$  sorted lists  $\Rightarrow k \cdot \sqrt[k]{n}$
- $\lg n$  sorted lists  $\Rightarrow \lg n \cdot \sqrt[\lg n]{n} = 2 \lg n$

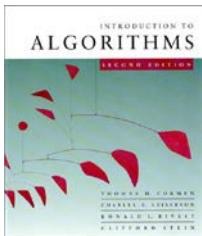




# $\lg n$ linked lists

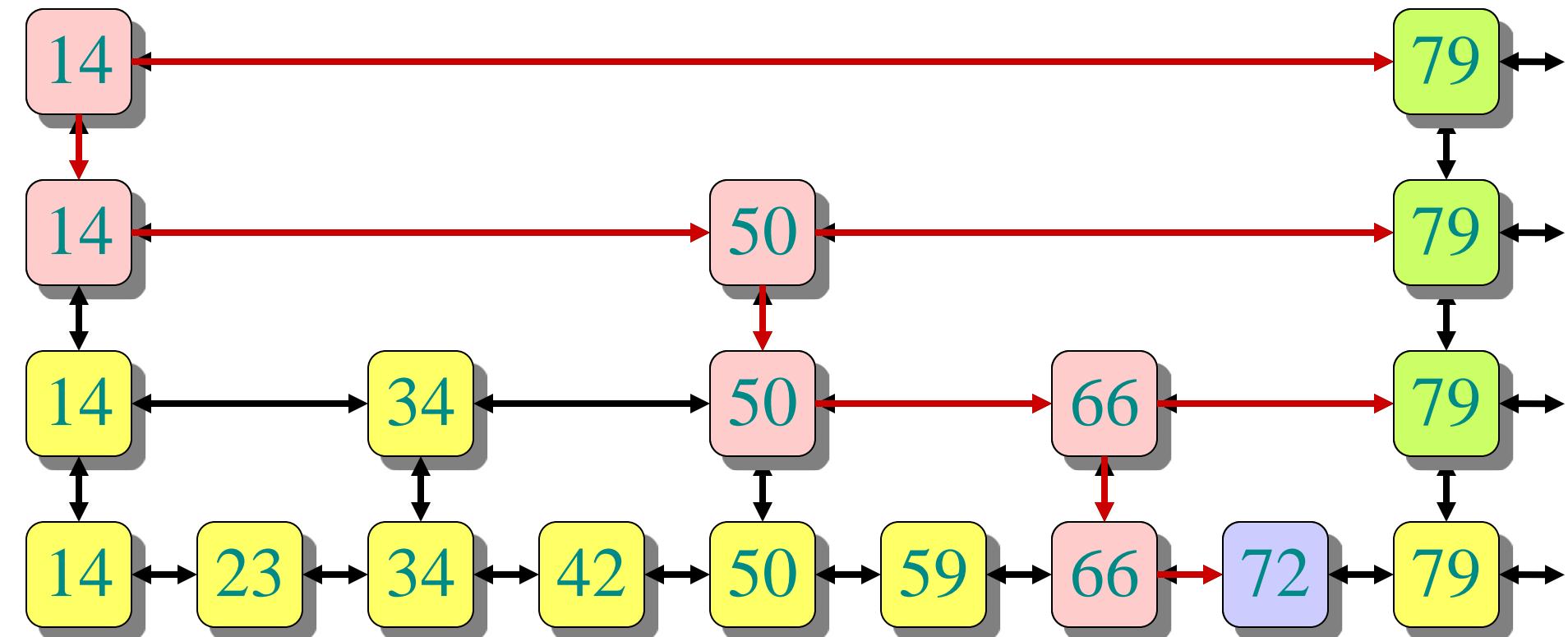
$\lg n$  sorted linked lists are like a binary tree  
(in fact, level-linked B<sup>+</sup>-tree; see Problem Set 5)

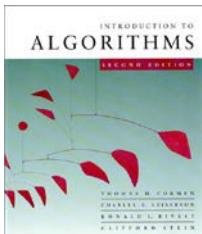




# Searching in $\lg n$ linked lists

EXAMPLE: SEARCH(72)

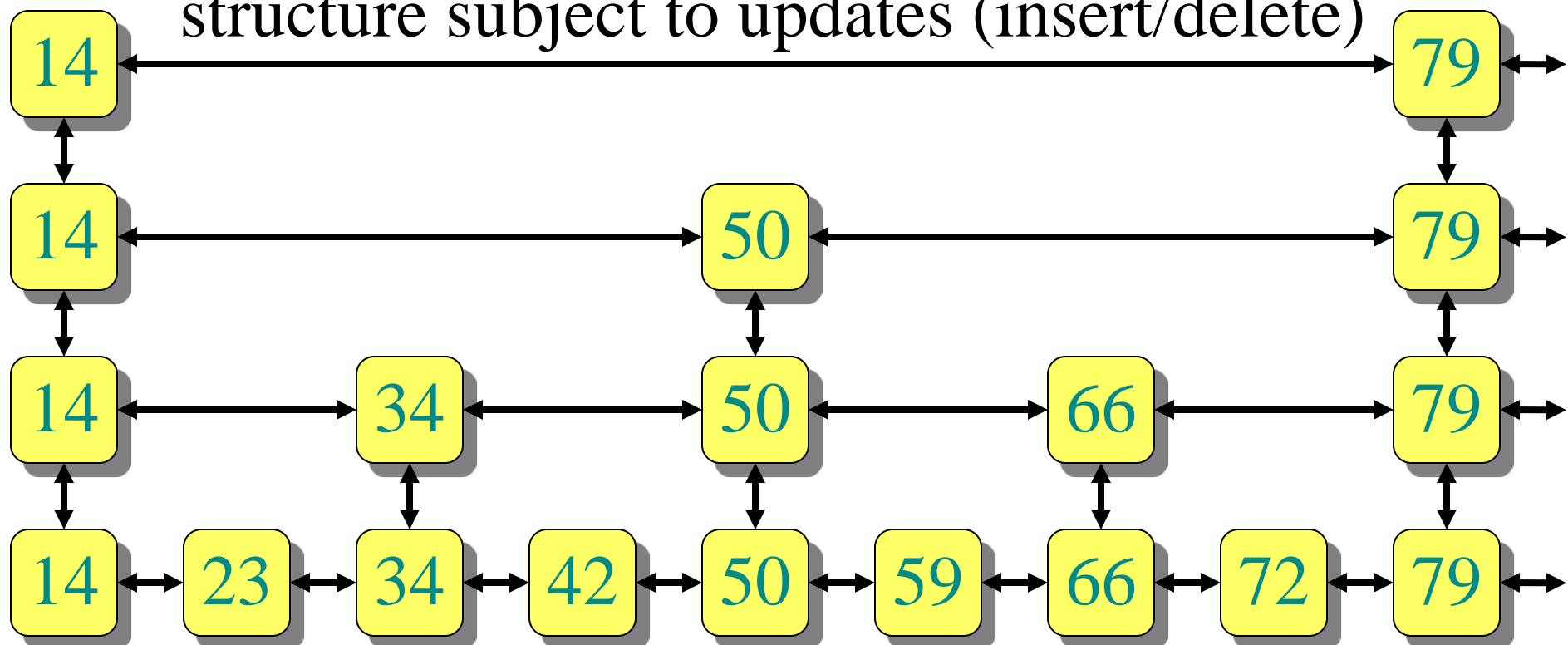


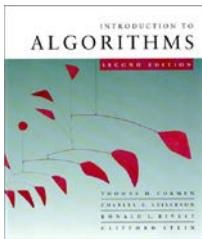


# Skip lists

*Ideal skip list* is this  $\lg n$  linked list structure

*Skip list data structure* maintains roughly this structure subject to updates (insert/delete)





# INSERT( $x$ )

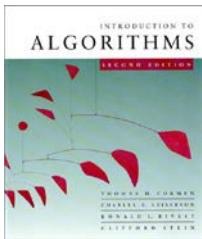
To insert an element  $x$  into a skip list:

- SEARCH( $x$ ) to see where  $x$  fits in bottom list
- Always insert into bottom list

**INVARIANT:** Bottom list contains all elements

- Insert into some of the lists above...

**QUESTION:** To which other lists should we add  $x$ ?



# INSERT( $x$ )

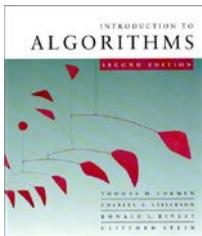
**QUESTION:** To which other lists should we add  $x$ ?

**IDEA:** Flip a (fair) coin; if HEADS,

*promote*  $x$  to next level up and flip again

- Probability of promotion to next level = 1/2
- On average:
  - 1/2 of the elements promoted 0 levels
  - 1/4 of the elements promoted 1 level
  - 1/8 of the elements promoted 2 levels
  - etc.

Approx.  
balance  
d?

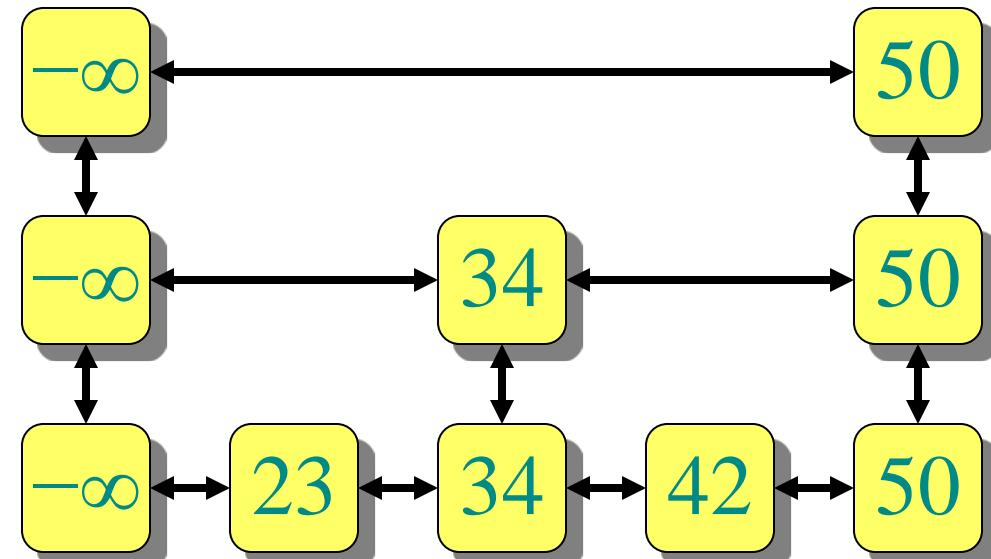


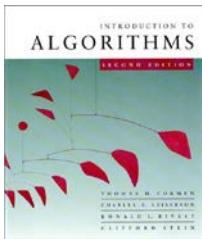
# Example of skip list

**EXERCISE:** Try building a skip list from scratch by repeated insertion using a real coin

**Small change:**

- Add special  $-\infty$  value to *every* list  
 $\Rightarrow$  can search with the same algorithm

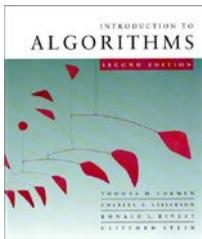




# Skip lists

A *skip list* is the result of insertions (and deletions) from an initially empty structure (containing just  $-\infty$ )

- $\text{INSERT}(x)$  uses random coin flips to decide promotion level
- $\text{DELETE}(x)$  removes  $x$  from all lists containing it



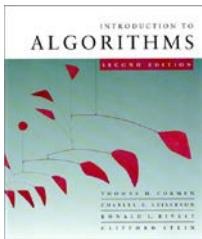
# Skip lists

A *skip list* is the result of insertions (and deletions) from an initially empty structure (containing just  $-\infty$ )

- $\text{INSERT}(x)$  uses random coin flips to decide promotion level
- $\text{DELETE}(x)$  removes  $x$  from all lists containing it

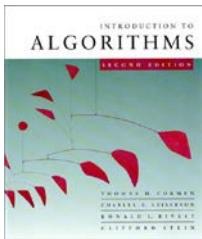
How good are skip lists? (speed/balance)

- **INTUITIVELY:** Pretty good on average
- **CLAIM:** Really, really good, almost always



# With-high-probability theorem

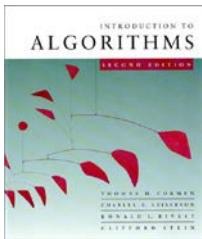
**THEOREM:** *With high probability*, every search in an  $n$ -element skip list costs  $O(\lg n)$



# With-high-probability theorem

**THEOREM:** *With high probability*, every search in a skip list costs  $O(\lg n)$

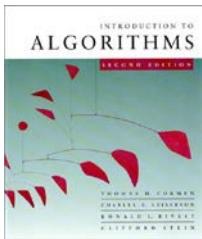
- **INFORMALLY:** Event  $E$  occurs *with high probability* (*w.h.p.*) if, for any  $\alpha \geq 1$ , there is an appropriate choice of constants for which  $E$  occurs with probability at least  $1 - O(1/n^\alpha)$ 
  - In fact, constant in  $O(\lg n)$  depends on  $\alpha$
- **FORMALLY:** Parameterized event  $E_\alpha$  occurs *with high probability* if, for any  $\alpha \geq 1$ , there is an appropriate choice of constants for which  $E_\alpha$  occurs with probability at least  $1 - c_\alpha/n^\alpha$



# With-high-probability theorem

**THEOREM:** With high probability, every search in a skip list costs  $O(\lg n)$

- **INFORMALLY:** Event  $E$  occurs *with high probability (w.h.p.)* if, for any  $\alpha \geq 1$ , there is an appropriate choice of constants for which  $E$  occurs with probability at least  $1 - O(1/n^\alpha)$
- **IDEA:** Can make *error probability*  $O(1/n^\alpha)$  very small by setting  $\alpha$  large, e.g., 100
- Almost certainly, bound remains true for entire execution of polynomial-time algorithm



# Boole's inequality / union bound

*Recall:*

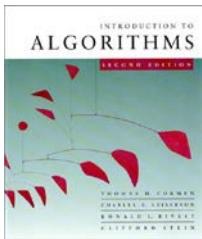
## BOOLE'S INEQUALITY / UNION BOUND:

For any random events  $E_1, E_2, \dots, E_k$ ,

$$\begin{aligned} & \Pr\{E_1 \cup E_2 \cup \dots \cup E_k\} \\ & \leq \Pr\{E_1\} + \Pr\{E_2\} + \dots + \Pr\{E_k\} \end{aligned}$$

## Application to with-high-probability events:

If  $k = n^{O(1)}$ , and each  $E_i$  occurs with high probability, then so does  $E_1 \cap E_2 \cap \dots \cap E_k$

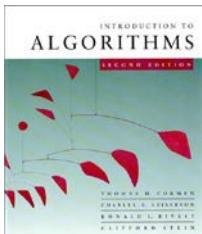


# Analysis Warmup

**LEMMA:** With high probability,  
 $n$ -element skip list has  $O(\lg n)$  levels

**PROOF:**

- Error probability for having at most  $c \lg n$  levels  
=  $\Pr\{\text{more than } c \lg n \text{ levels}\}$   
 $\leq n \cdot \Pr\{\text{element } x \text{ promoted at least } c \lg n \text{ times}\}$   
*(by Boole's Inequality)*  
=  $n \cdot (1/2^{c \lg n})$   
=  $n \cdot (1/n^c)$   
=  $1/n^{c-1}$

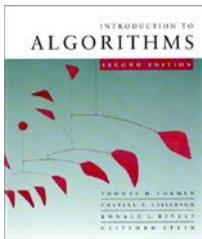


# Analysis Warmup

**LEMMA:** With high probability,  
 $n$ -element skip list has  $O(\lg n)$  levels

**PROOF:**

- Error probability for having at most  $c \lg n$  levels  
 $\leq 1/n^{c-1}$
- This probability is *polynomially small*,  
i.e., at most  $n^\alpha$  for  $\alpha = c - 1$ .
- We can make  $\alpha$  arbitrarily large by choosing the  
constant  $c$  in the  $O(\lg n)$  bound accordingly. □

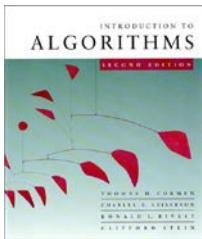


# Proof of theorem

**THEOREM:** With high probability, every search in an  $n$ -element skip list costs  $O(\lg n)$

**COOL IDEA:** Analyze search backwards—leaf to root

- Search starts [ends] at leaf (node in bottom level)
- At each node visited:
  - If node wasn't promoted higher (got TAILS here), then we go [came from] left
  - If node was promoted higher (got HEADS here), then we go [came from] up
- Search stops [starts] at the root (or  $-\infty$ )



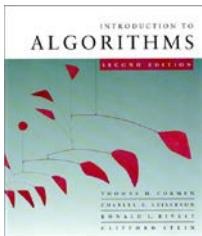
# Proof of theorem

**THEOREM:** With high probability, every search in an  $n$ -element skip list costs  $O(\lg n)$

**COOL IDEA:** Analyze search backwards—leaf to root

**PROOF:**

- Search makes “up” and “left” moves until it reaches the root (or  $-\infty$ )
- Number of “up” moves < number of levels  
 $\leq c \lg n$  w.h.p. (*Lemma*)
- $\Rightarrow$  w.h.p., number of moves is at most the number of times we need to flip a coin to get  $c \lg n$  HEADS



# Coin flipping analysis

**CLAIM:** Number of coin flips until  $c \lg n$  HEADS  
=  $\Theta(\lg n)$  with high probability

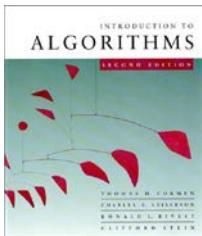
**PROOF:**

Obviously  $\Omega(\lg n)$ : at least  $c \lg n$

Prove  $O(\lg n)$  “by example”:

- Say we make  $10 c \lg n$  flips
- When are there at least  $c \lg n$  HEADS?

(Later generalize to arbitrary values of 10)

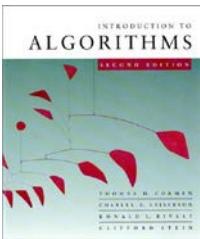


# Coin flipping analysis

**CLAIM:** Number of coin flips until  $c \lg n$  HEADS  
=  $\Theta(\lg n)$  with high probability

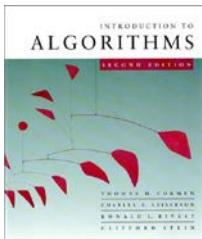
**PROOF:**

- $\Pr\{\text{exactly } c \lg n \text{ HEADS}\} = \underbrace{\binom{10c \lg n}{c \lg n}}_{\text{orders}} \cdot \underbrace{\left(\frac{1}{2}\right)^{c \lg n}}_{\text{HEADS}} \cdot \underbrace{\left(\frac{1}{2}\right)^{9c \lg n}}_{\text{TAILS}}$
- $\Pr\{\text{at most } c \lg n \text{ HEADS}\} \leq \underbrace{\binom{10c \lg n}{c \lg n}}_{\text{overestimate on orders}} \cdot \underbrace{\left(\frac{1}{2}\right)^{9c \lg n}}_{\text{TAILS}}$



# Coin flipping analysis (cont'd)

- Recall bounds on  $\binom{y}{x}$ :  $\left(\frac{y}{x}\right)^x \leq \binom{y}{x} \leq \left(e \frac{y}{x}\right)^x$
- $\Pr\{\text{at most } c \lg n \text{ HEADS}\} \leq \binom{10c \lg n}{c \lg n} \cdot \left(\frac{1}{2}\right)^{9c \lg n}$ 
$$\leq \left(e \frac{10c \lg n}{c \lg n}\right)^{c \lg n} \cdot \left(\frac{1}{2}\right)^{9c \lg n}$$
$$= (10e)^{c \lg n} 2^{-9c \lg n}$$
$$= 2^{\lg(10e) \cdot c \lg n} 2^{-9c \lg n}$$
$$= 2^{[\lg(10e) - 9] \cdot c \lg n}$$
$$= 1/n^\alpha \text{ for } \alpha = [9 - \lg(10e)] \cdot c$$



# Coin flipping analysis (cont'd)

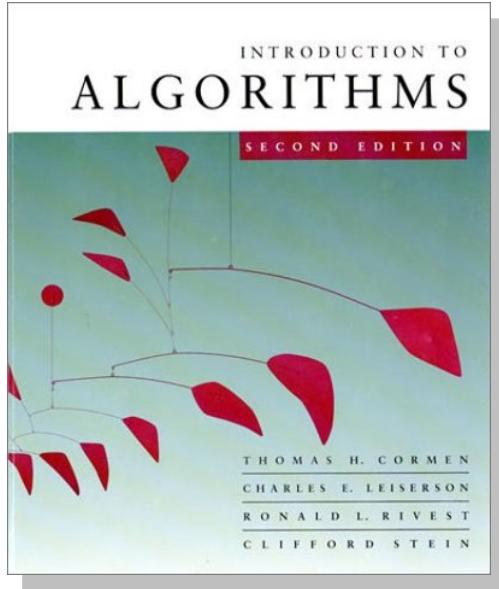
- $\Pr\{\text{at most } c \lg n \text{ HEADS}\} \leq 1/n^\alpha$  for  $\alpha = [9 - \lg(10e)]c$
- **KEY PROPERTY:**  $\alpha \rightarrow \infty$  as  $10 \rightarrow \infty$ , for any  $c$
- So set  $10$ , i.e., constant in  $O(\lg n)$  bound,  
large enough to meet desired  $\alpha$



This completes the proof of the coin-flipping claim  
and the proof of the theorem.

# *Introduction to Algorithms*

## 6.046J/18.401J

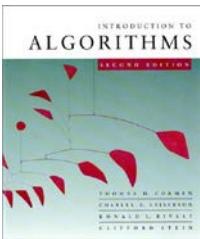


### LECTURE 13

#### Amortized Analysis

- Dynamic tables
- Aggregate method
- Accounting method
- Potential method

Prof. Charles E. Leiserson



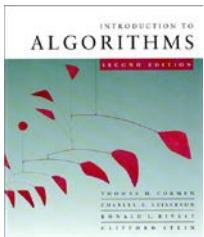
# How large should a hash table be?

**Goal:** Make the table as small as possible, but large enough so that it won't overflow (or otherwise become inefficient).

**Problem:** What if we don't know the proper size in advance?

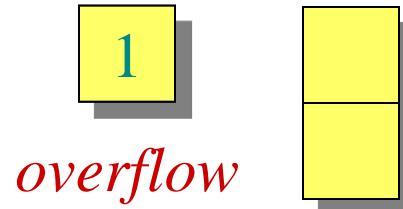
**Solution:** *Dynamic tables.*

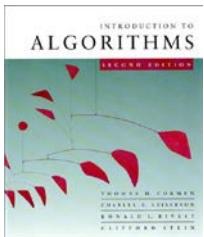
**IDEA:** Whenever the table overflows, “grow” it by allocating (via `malloc` or `new`) a new, larger table. Move all items from the old table into the new one, and free the storage for the old table.



# Example of a dynamic table

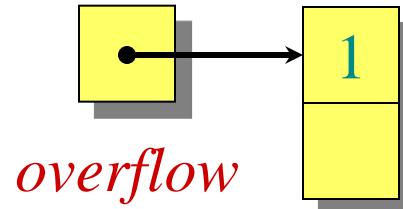
1. INSERT
2. INSERT

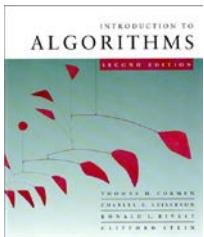




# Example of a dynamic table

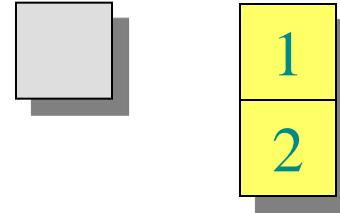
1. INSERT
2. INSERT

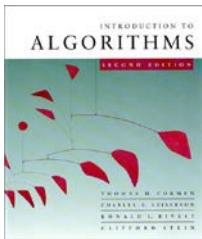




# Example of a dynamic table

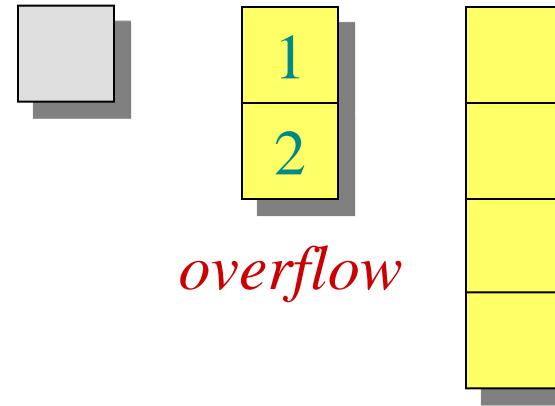
1. INSERT
2. INSERT

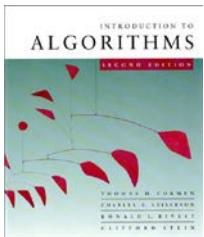




# Example of a dynamic table

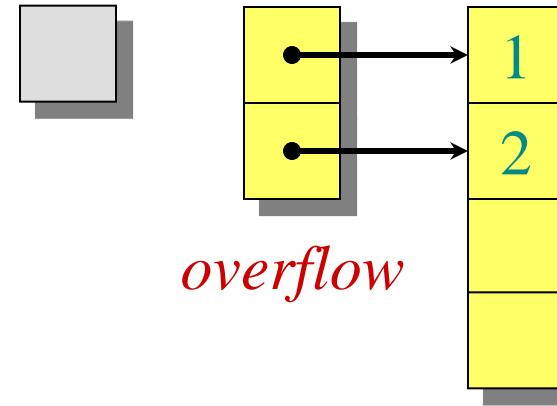
1. INSERT
2. INSERT
3. INSERT

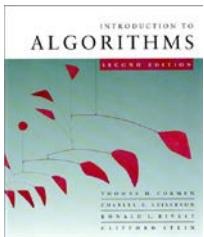




# Example of a dynamic table

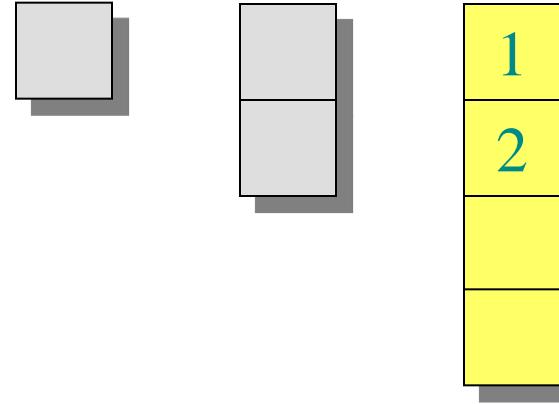
1. INSERT
2. INSERT
3. INSERT

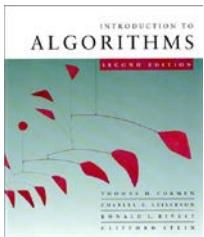




# Example of a dynamic table

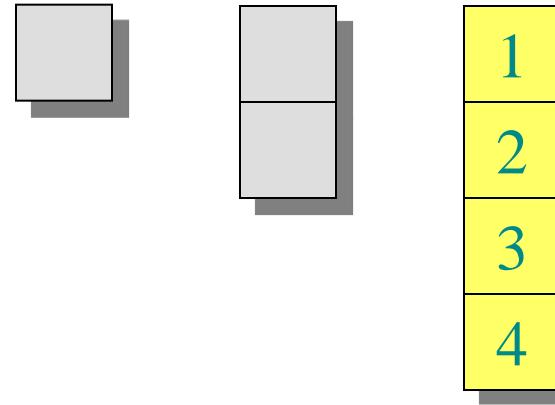
1. INSERT
2. INSERT
3. INSERT

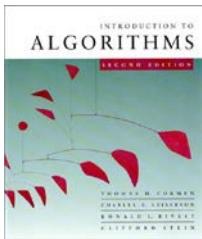




# Example of a dynamic table

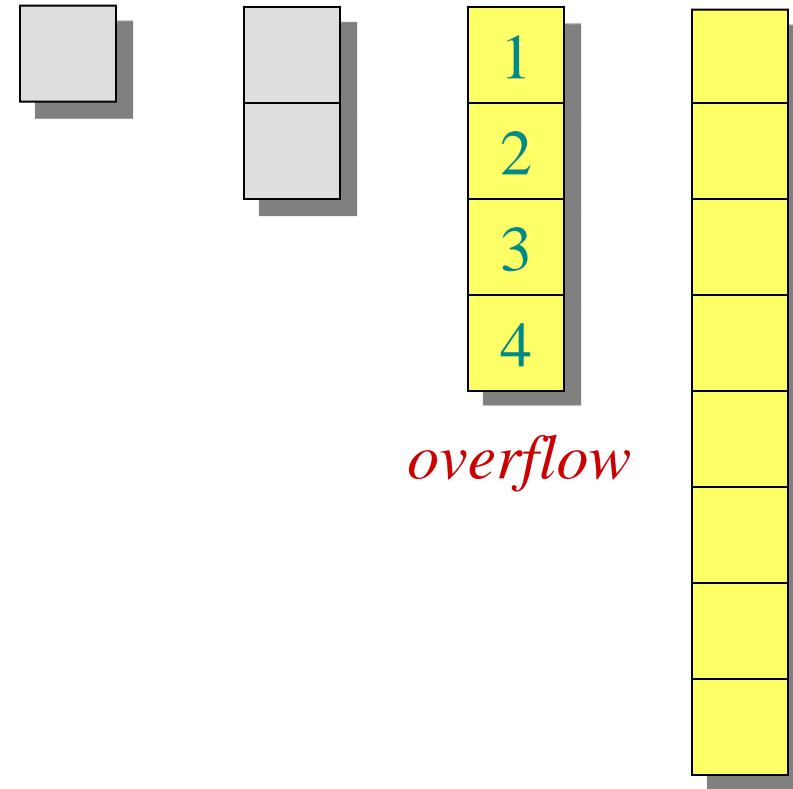
1. INSERT
2. INSERT
3. INSERT
4. INSERT

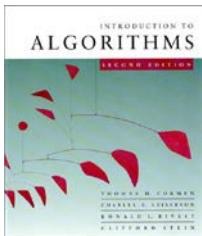




# Example of a dynamic table

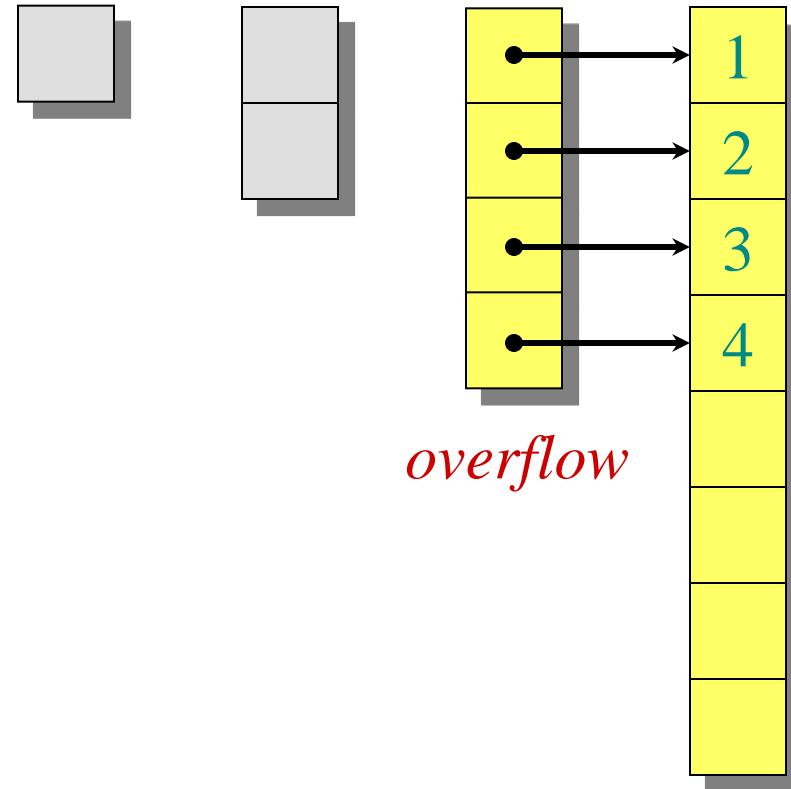
1. INSERT
2. INSERT
3. INSERT
4. INSERT
5. INSERT

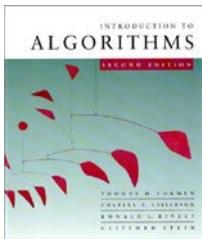




# Example of a dynamic table

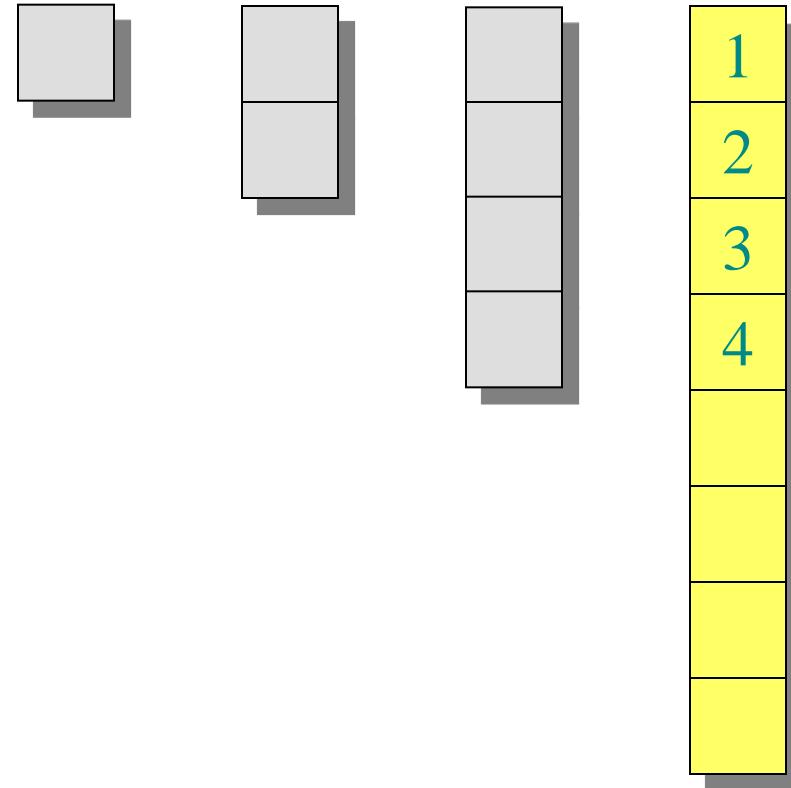
1. INSERT
2. INSERT
3. INSERT
4. INSERT
5. INSERT

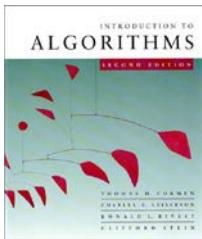




# Example of a dynamic table

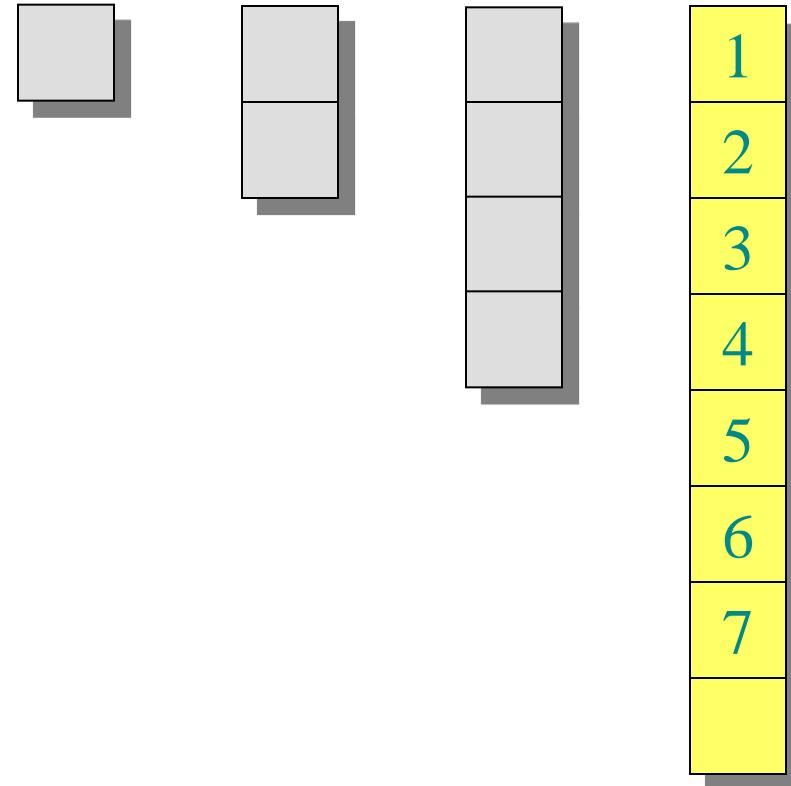
1. INSERT
2. INSERT
3. INSERT
4. INSERT
5. INSERT

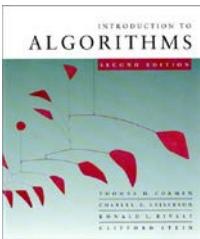




# Example of a dynamic table

1. INSERT
2. INSERT
3. INSERT
4. INSERT
5. INSERT
6. INSERT
7. INSERT



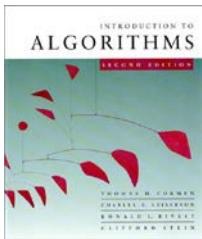


# Worst-case analysis

Consider a sequence of  $n$  insertions. The worst-case time to execute one insertion is  $\Theta(n)$ . Therefore, the worst-case time for  $n$  insertions is  $n \cdot \Theta(n) = \Theta(n^2)$ .

**WRONG!** In fact, the worst-case cost for  $n$  insertions is only  $\Theta(n) \ll \Theta(n^2)$ .

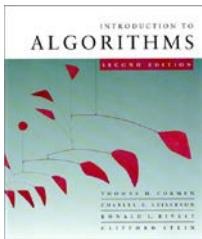
Let's see why.



# Tighter analysis

Let  $c_i =$  the cost of the  $i$ th insertion  
 $= \begin{cases} i & \text{if } i - 1 \text{ is an exact power of 2,} \\ 1 & \text{otherwise.} \end{cases}$

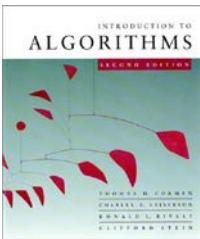
|          |   |   |   |   |   |   |   |   |    |    |
|----------|---|---|---|---|---|---|---|---|----|----|
| $i$      | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9  | 10 |
| $size_i$ | 1 | 2 | 4 | 4 | 8 | 8 | 8 | 8 | 16 | 16 |
| $c_i$    | 1 | 2 | 3 | 1 | 5 | 1 | 1 | 1 | 9  | 1  |



# Tighter analysis

Let  $c_i =$  the cost of the  $i$ th insertion  
 $= \begin{cases} i & \text{if } i - 1 \text{ is an exact power of 2,} \\ 1 & \text{otherwise.} \end{cases}$

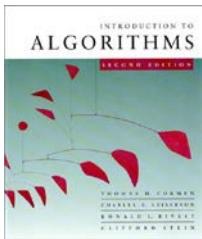
|          |   |   |   |   |   |   |   |   |    |    |
|----------|---|---|---|---|---|---|---|---|----|----|
| $i$      | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9  | 10 |
| $size_i$ | 1 | 2 | 4 | 4 | 8 | 8 | 8 | 8 | 16 | 16 |
|          | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1  | 1  |
| $c_i$    | 1 | 2 |   | 4 |   |   |   |   | 8  |    |



# Tighter analysis (continued)

$$\begin{aligned}\text{Cost of } \textcolor{teal}{n} \text{ insertions} &= \sum_{i=1}^{\textcolor{teal}{n}} c_i \\ &\leq n + \sum_{j=0}^{\lfloor \lg(n-1) \rfloor} 2^j \\ &\leq 3n \\ &= \Theta(n).\end{aligned}$$

Thus, the average cost of each dynamic-table operation is  $\Theta(n)/n = \Theta(1)$ .

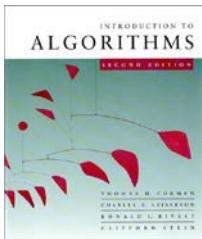


# Amortized analysis

An *amortized analysis* is any strategy for analyzing a sequence of operations to show that the average cost per operation is small, even though a single operation within the sequence might be expensive.

Even though we're taking averages, however, probability is not involved!

- An amortized analysis guarantees the average performance of each operation in the *worst case*.



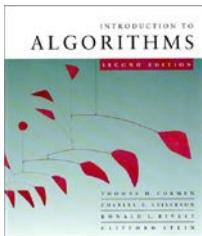
# Types of amortized analyses

Three common amortization arguments:

- the *aggregate* method,
- the *accounting* method,
- the *potential* method.

We've just seen an aggregate analysis.

The aggregate method, though simple, lacks the precision of the other two methods. In particular, the accounting and potential methods allow a specific *amortized cost* to be allocated to each operation.



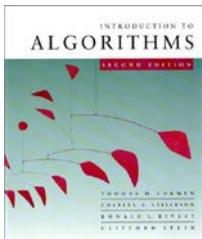
# Accounting method

- Charge  $i$ th operation a fictitious **amortized cost**  $\hat{c}_i$ , where \$1 pays for 1 unit of work (*i.e.*, time).
- This fee is consumed to perform the operation.
- Any amount not immediately consumed is stored in the **bank** for use by subsequent operations.
- The bank balance must not go negative! We must ensure that

$$\sum_{i=1}^n c_i \leq \sum_{i=1}^n \hat{c}_i$$

for all  $n$ .

- Thus, the total amortized costs provide an upper bound on the total true costs.



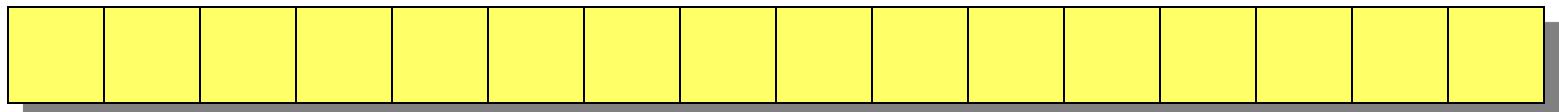
# Accounting analysis of dynamic tables

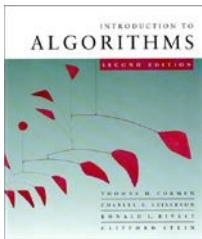
Charge an amortized cost of  $\hat{c}_i = \$3$  for the  $i$ th insertion.

- $\$1$  pays for the immediate insertion.
- $\$2$  is stored for later table doubling.

When the table doubles,  $\$1$  pays to move a recent item, and  $\$1$  pays to move an old item.

## Example:





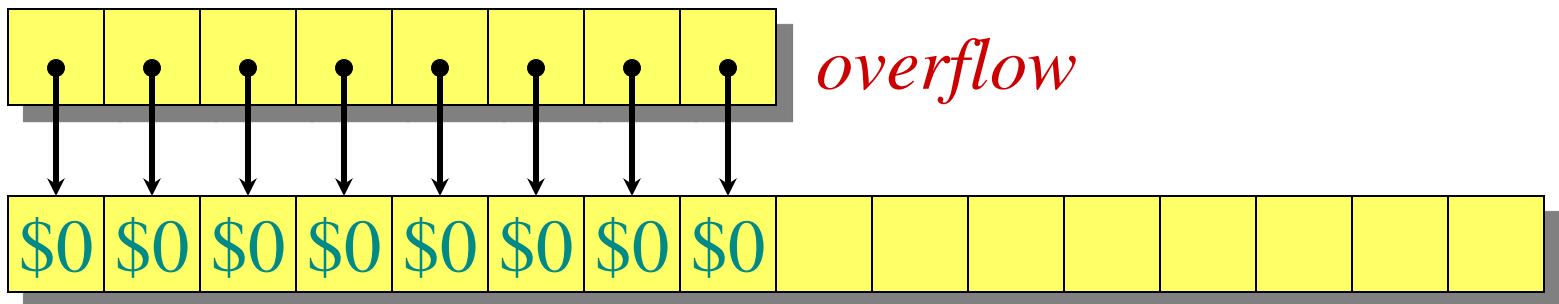
# Accounting analysis of dynamic tables

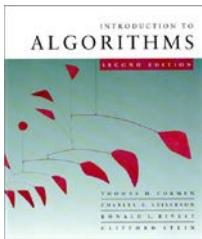
Charge an amortized cost of  $\hat{c}_i = \$3$  for the  $i$ th insertion.

- $\$1$  pays for the immediate insertion.
- $\$2$  is stored for later table doubling.

When the table doubles,  $\$1$  pays to move a recent item, and  $\$1$  pays to move an old item.

## Example:





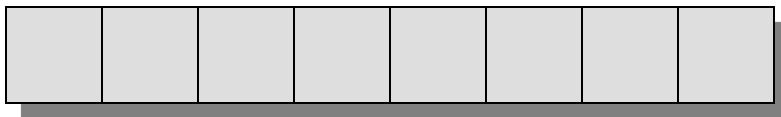
# Accounting analysis of dynamic tables

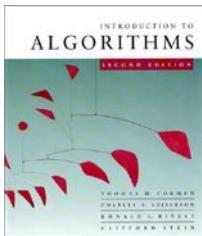
Charge an amortized cost of  $\hat{c}_i = \$3$  for the  $i$ th insertion.

- $\$1$  pays for the immediate insertion.
- $\$2$  is stored for later table doubling.

When the table doubles,  $\$1$  pays to move a recent item, and  $\$1$  pays to move an old item.

## Example:



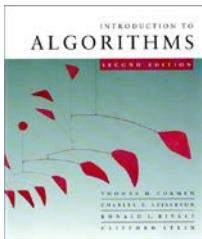


# Accounting analysis (continued)

**Key invariant:** Bank balance never drops below 0. Thus, the sum of the amortized costs provides an upper bound on the sum of the true costs.

|             |    |   |   |   |   |   |   |   |    |    |
|-------------|----|---|---|---|---|---|---|---|----|----|
| $i$         | 1  | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9  | 10 |
| $size_i$    | 1  | 2 | 4 | 4 | 8 | 8 | 8 | 8 | 16 | 16 |
| $c_i$       | 1  | 2 | 3 | 1 | 5 | 1 | 1 | 1 | 9  | 1  |
| $\hat{c}_i$ | 2* | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3  | 3  |
| $bank_i$    | 1  | 2 | 2 | 4 | 2 | 4 | 6 | 8 | 2  | 4  |

\*Okay, so I lied. The first operation costs only \$2, not \$3.

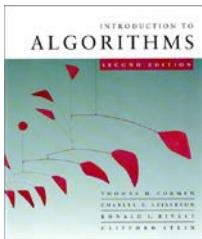


# Potential method

**IDEA:** View the bank account as the potential energy (*à la* physics) of the dynamic set.

**Framework:**

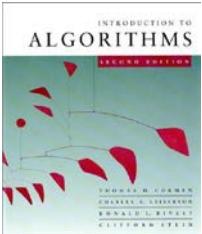
- Start with an initial data structure  $D_0$ .
- Operation  $i$  transforms  $D_{i-1}$  to  $D_i$ .
- The cost of operation  $i$  is  $c_i$ .
- Define a ***potential function***  $\Phi : \{D_i\} \rightarrow \mathbb{R}$ , such that  $\Phi(D_0) = 0$  and  $\Phi(D_i) \geq 0$  for all  $i$ .
- The ***amortized cost***  $\hat{c}_i$  with respect to  $\Phi$  is defined to be  $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$ .



# Understanding potentials

$$\hat{c}_i = c_i + \underbrace{\Phi(D_i) - \Phi(D_{i-1})}_{\textit{potential difference } \Delta\Phi_i}$$

- If  $\Delta\Phi_i > 0$ , then  $\hat{c}_i > c_i$ . Operation  $i$  stores work in the data structure for later use.
- If  $\Delta\Phi_i < 0$ , then  $\hat{c}_i < c_i$ . The data structure delivers up stored work to help pay for operation  $i$ .

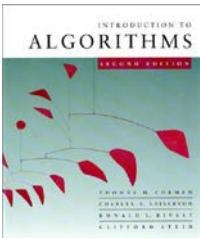


# The amortized costs bound the true costs

The total amortized cost of  $n$  operations is

$$\sum_{i=1}^n \hat{c}_i = \sum_{i=1}^n (c_i + \Phi(D_i) - \Phi(D_{i-1}))$$

Summing both sides.

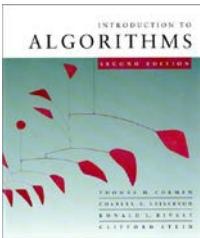


# The amortized costs bound the true costs

The total amortized cost of  $n$  operations is

$$\begin{aligned}\sum_{i=1}^n \hat{c}_i &= \sum_{i=1}^n (c_i + \Phi(D_i) - \Phi(D_{i-1})) \\ &= \sum_{i=1}^n c_i + \Phi(D_n) - \Phi(D_0)\end{aligned}$$

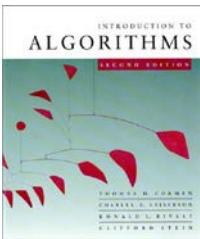
The series telescopes.



# The amortized costs bound the true costs

The total amortized cost of  $n$  operations is

$$\begin{aligned}\sum_{i=1}^n \hat{c}_i &= \sum_{i=1}^n (c_i + \Phi(D_i) - \Phi(D_{i-1})) \\ &= \sum_{i=1}^n c_i + \Phi(D_n) - \Phi(D_0) \\ &\geq \sum_{i=1}^n c_i \quad \text{since } \Phi(D_n) \geq 0 \text{ and} \\ &\quad \Phi(D_0) = 0.\end{aligned}$$



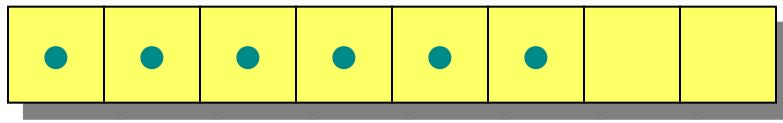
# Potential analysis of table doubling

Define the potential of the table after the  $i$ th insertion by  $\Phi(D_i) = 2i - 2^{\lceil \lg i \rceil}$ . (Assume that  $2^{\lceil \lg 0 \rceil} = 0$ .)

## Note:

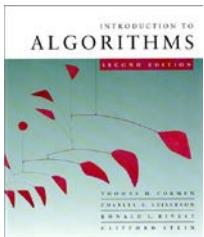
- $\Phi(D_0) = 0$ ,
- $\Phi(D_i) \geq 0$  for all  $i$ .

## Example:



$$\Phi = 2 \cdot 6 - 2^3 = 4$$

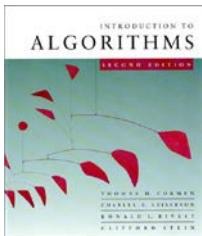
(  
A horizontal array of seven yellow rectangular boxes. The first five boxes each contain the text "\$0" in blue. The next two boxes each contain the text "\$2" in blue. The seventh box is empty. A thick grey horizontal bar is positioned directly beneath the seventh box.  
accounting method)



# Calculation of amortized costs

The amortized cost of the  $i$ th insertion is

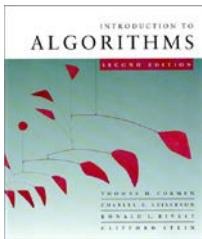
$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$



# Calculation of amortized costs

The amortized cost of the  $i$ th insertion is

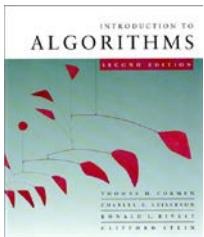
$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= \left\{ \begin{array}{ll} i & \text{if } i-1 \text{ is an exact power of 2,} \\ 1 & \text{otherwise;} \end{array} \right\} \\ &\quad + (2i - 2^{\lceil \lg i \rceil}) - (2(i-1) - 2^{\lceil \lg (i-1) \rceil})\end{aligned}$$



# Calculation of amortized costs

The amortized cost of the  $i$ th insertion is

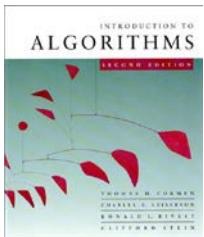
$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\&= \left\{ \begin{array}{ll} i & \text{if } i-1 \text{ is an exact power of 2,} \\ 1 & \text{otherwise;} \end{array} \right\} \\&\quad + (2i - 2^{\lceil \lg i \rceil}) - (2(i-1) - 2^{\lceil \lg (i-1) \rceil}) \\&= \left\{ \begin{array}{ll} i & \text{if } i-1 \text{ is an exact power of 2,} \\ 1 & \text{otherwise;} \end{array} \right\} \\&\quad + 2 - 2^{\lceil \lg i \rceil} + 2^{\lceil \lg (i-1) \rceil}.\end{aligned}$$



# Calculation

**Case 1:**  $i - 1$  is an exact power of 2.

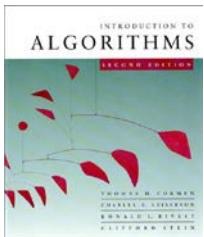
$$\hat{c}_i = i + 2 - 2^{\lceil \lg i \rceil} + 2^{\lceil \lg (i-1) \rceil}$$



# Calculation

**Case 1:**  $i - 1$  is an exact power of 2.

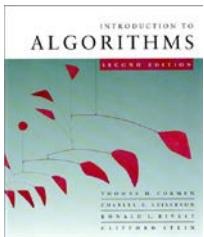
$$\begin{aligned}\hat{c}_i &= i + 2 - 2^{\lceil \lg i \rceil} + 2^{\lceil \lg (i-1) \rceil} \\ &= i + 2 - 2(i - 1) + (i - 1)\end{aligned}$$



# Calculation

**Case 1:**  $i - 1$  is an exact power of 2.

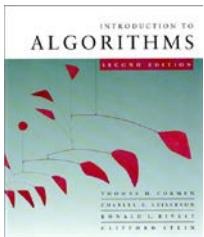
$$\begin{aligned}\hat{c}_i &= i + 2 - 2^{\lceil \lg i \rceil} + 2^{\lceil \lg (i-1) \rceil} \\ &= i + 2 - 2(i-1) + (i-1) \\ &= i + 2 - 2i + 2 + i - 1\end{aligned}$$



# Calculation

**Case 1:**  $i - 1$  is an exact power of 2.

$$\begin{aligned}\hat{c}_i &= i + 2 - 2^{\lceil \lg i \rceil} + 2^{\lceil \lg (i-1) \rceil} \\&= i + 2 - 2(i-1) + (i-1) \\&= i + 2 - 2i + 2 + i - 1 \\&= 3\end{aligned}$$



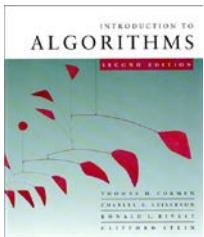
# Calculation

**Case 1:**  $i - 1$  is an exact power of 2.

$$\begin{aligned}\hat{c}_i &= i + 2 - 2^{\lceil \lg i \rceil} + 2^{\lceil \lg (i-1) \rceil} \\&= i + 2 - 2(i-1) + (i-1) \\&= i + 2 - 2i + 2 + i - 1 \\&= 3\end{aligned}$$

**Case 2:**  $i - 1$  is *not* an exact power of 2.

$$\hat{c}_i = 1 + 2 - 2^{\lceil \lg i \rceil} + 2^{\lceil \lg (i-1) \rceil}$$



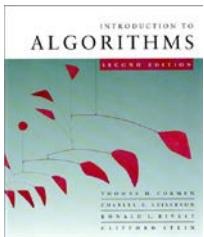
# Calculation

**Case 1:**  $i - 1$  is an exact power of 2.

$$\begin{aligned}\hat{c}_i &= i + 2 - 2^{\lceil \lg i \rceil} + 2^{\lceil \lg (i-1) \rceil} \\&= i + 2 - 2(i-1) + (i-1) \\&= i + 2 - 2i + 2 + i - 1 \\&= 3\end{aligned}$$

**Case 2:**  $i - 1$  is *not* an exact power of 2.

$$\begin{aligned}\hat{c}_i &= 1 + 2 - 2^{\lceil \lg i \rceil} + 2^{\lceil \lg (i-1) \rceil} \\&= 3 \quad (\text{since } 2^{\lceil \lg i \rceil} = 2^{\lceil \lg (i-1) \rceil})\end{aligned}$$



# Calculation

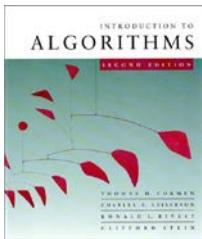
**Case 1:**  $i - 1$  is an exact power of 2.

$$\begin{aligned}\hat{c}_i &= i + 2 - 2^{\lceil \lg i \rceil} + 2^{\lceil \lg (i-1) \rceil} \\&= i + 2 - 2(i-1) + (i-1) \\&= i + 2 - 2i + 2 + i - 1 \\&= 3\end{aligned}$$

**Case 2:**  $i - 1$  is *not* an exact power of 2.

$$\begin{aligned}\hat{c}_i &= 1 + 2 - 2^{\lceil \lg i \rceil} + 2^{\lceil \lg (i-1) \rceil} \\&= 3\end{aligned}$$

Therefore,  $n$  insertions cost  $\Theta(n)$  in the worst case.



# Calculation

**Case 1:**  $i - 1$  is an exact power of 2.

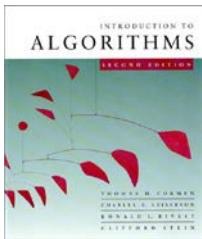
$$\begin{aligned}\hat{c}_i &= i + 2 - 2^{\lceil \lg i \rceil} + 2^{\lceil \lg (i-1) \rceil} \\ &= i + 2 - 2(i-1) + (i-1) \\ &= i + 2 - 2i + 2 + i - 1 \\ &= 3\end{aligned}$$

**Case 2:**  $i - 1$  is *not* an exact power of 2.

$$\begin{aligned}\hat{c}_i &= 1 + 2 - 2^{\lceil \lg i \rceil} + 2^{\lceil \lg (i-1) \rceil} \\ &= 3\end{aligned}$$

Therefore,  $n$  insertions cost  $\Theta(n)$  in the worst case.

**Exercise:** Fix the bug in this analysis to show that the amortized cost of the first insertion is only 2.

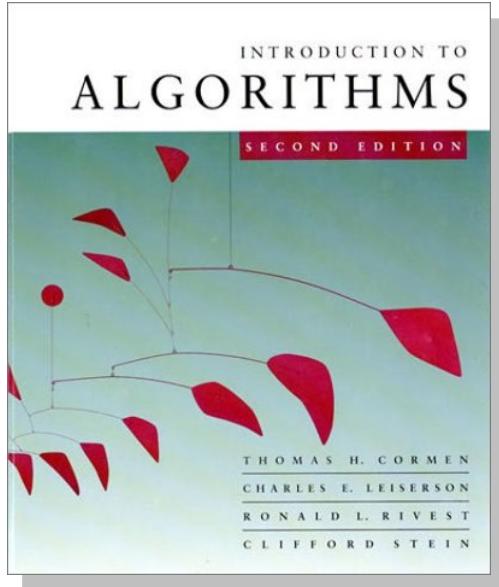


# Conclusions

- Amortized costs can provide a clean abstraction of data-structure performance.
- Any of the analysis methods can be used when an amortized analysis is called for, but each method has some situations where it is arguably the simplest or most precise.
- Different schemes may work for assigning amortized costs in the accounting method, or potentials in the potential method, sometimes yielding radically different bounds.

# *Introduction to Algorithms*

## 6.046J/18.401J

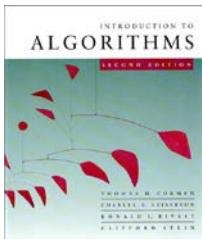


### LECTURE 14

#### Competitive Analysis

- Self-organizing lists
- Move-to-front heuristic
- Competitive analysis of MTF

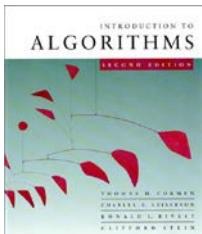
Prof. Charles E. Leiserson



# Self-organizing lists

List  $L$  of  $n$  elements

- The operation  $\text{ACCESS}(x)$  costs  $\text{rank}_L(x) =$  distance of  $x$  from the head of  $L$ .
- $L$  can be reordered by transposing adjacent elements at a cost of 1.

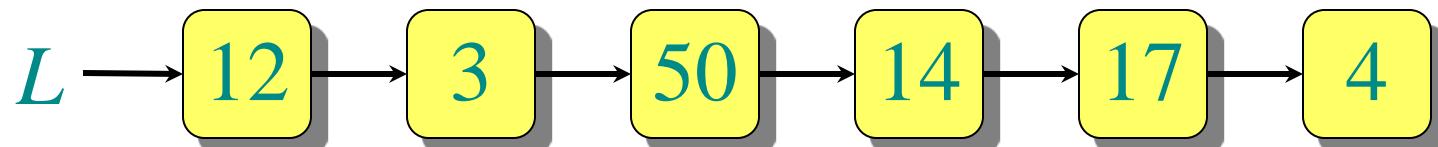


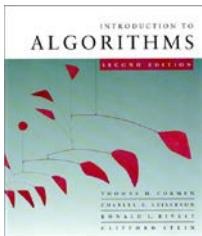
# Self-organizing lists

List  $L$  of  $n$  elements

- The operation  $\text{ACCESS}(x)$  costs  $\text{rank}_L(x) =$  distance of  $x$  from the head of  $L$ .
- $L$  can be reordered by transposing adjacent elements at a cost of 1.

**Example:**



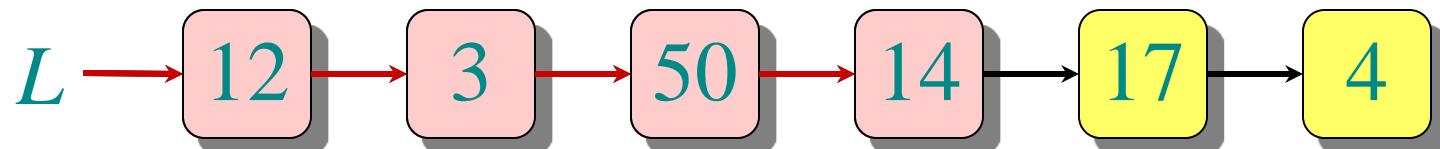


# Self-organizing lists

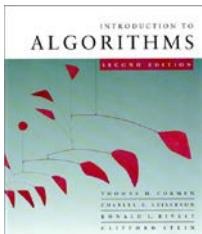
List  $L$  of  $n$  elements

- The operation  $\text{ACCESS}(x)$  costs  $\text{rank}_L(x) =$  distance of  $x$  from the head of  $L$ .
- $L$  can be reordered by transposing adjacent elements at a cost of 1.

**Example:**



Accessing the element with key 14 costs 4.

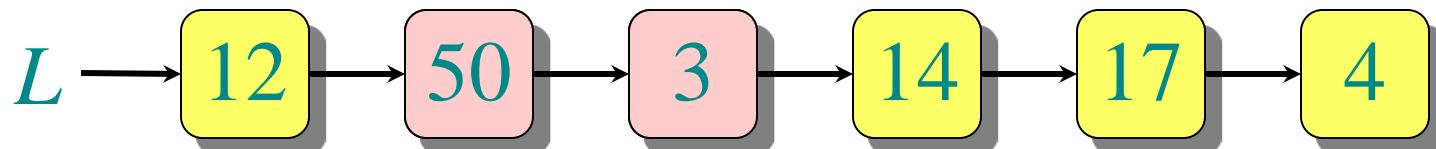


# Self-organizing lists

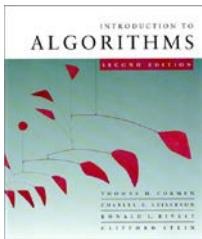
List  $L$  of  $n$  elements

- The operation  $\text{ACCESS}(x)$  costs  $\text{rank}_L(x) =$  distance of  $x$  from the head of  $L$ .
- $L$  can be reordered by transposing adjacent elements at a cost of 1.

**Example:**



Transposing 3 and 50 costs 1.

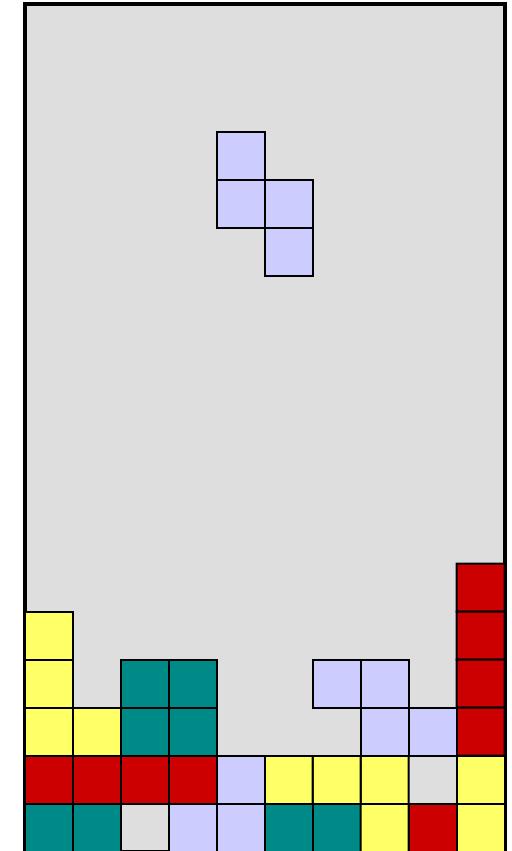


# On-line and off-line problems

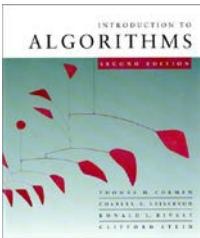
**Definition.** A sequence  $S$  of operations is provided one at a time. For each operation, an *on-line* algorithm  $A$  must execute the operation immediately without any knowledge of future operations (e.g., *Tetris*).

An *off-line* algorithm may see the whole sequence  $S$  in advance.

**Goal:** Minimize the total cost  $C_A(S)$ .



*The game of Tetris*

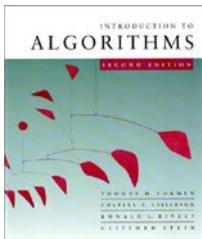


# Worst-case analysis of self-organizing lists

An adversary always accesses the tail ( $n$ th) element of  $L$ . Then, for any on-line algorithm  $A$ , we have

$$C_A(S) = \Omega(|S| \cdot n)$$

in the worst case.



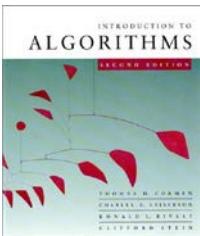
# Average-case analysis of self-organizing lists

Suppose that element  $x$  is accessed with probability  $p(x)$ . Then, we have

$$\mathbb{E}[C_A(S)] = \sum_{x \in L} p(x) \cdot \text{rank}_L(x),$$

which is minimized when  $L$  is sorted in decreasing order with respect to  $p$ .

**Heuristic:** Keep a count of the number of times each element is accessed, and maintain  $L$  in order of decreasing count.



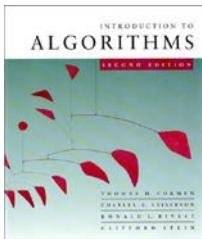
# The move-to-front heuristic

**Practice:** Implementers discovered that the ***move-to-front (MTF)*** heuristic empirically yields good results.

**IDEA:** After accessing  $x$ , move  $x$  to the head of  $L$  using transposes:

$$\text{cost} = 2 \cdot \text{rank}_L(x) .$$

The MTF heuristic responds well to locality in the access sequence  $S$ .

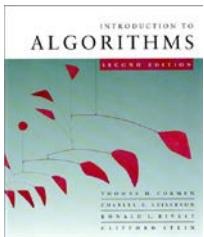


# Competitive analysis

**Definition.** An on-line algorithm  $A$  is  **$\alpha$ -competitive** if there exists a constant  $k$  such that for any sequence  $S$  of operations,

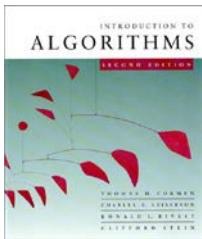
$$C_A(S) \leq \alpha \cdot C_{\text{OPT}}(S) + k ,$$

where  $\text{OPT}$  is the optimal off-line algorithm (“God’s algorithm”).



# MTF is $O(1)$ -competitive

**Theorem.** MTF is 4-competitive for self-organizing lists.



# MTF is $O(1)$ -competitive

**Theorem.** MTF is  $4$ -competitive for self-organizing lists.

*Proof.* Let  $L_i$  be MTF's list after the  $i$ th access, and let  $L_i^*$  be OPT's list after the  $i$ th access.

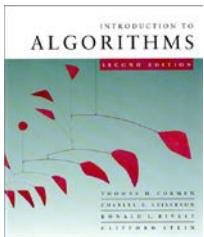
Let  $c_i =$  MTF's cost for the  $i$ th operation

$$= 2 \cdot \text{rank}_{L_{i-1}}(x) \text{ if it accesses } x;$$

$c_i^* =$  OPT's cost for the  $i$ th operation

$$= \text{rank}_{L_{i-1}^*}(x) + t_i,$$

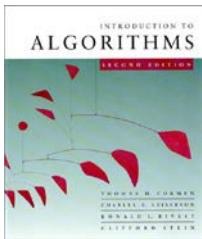
where  $t_i$  is the number of transposes that OPT performs.



# Potential function

Define the potential function  $\Phi: \{L_i\} \rightarrow \mathbb{R}$  by

$$\begin{aligned}\Phi(L_i) &= 2 \cdot |\{(x, y) : x \prec_{L_i} y \text{ and } y \prec_{L_i^*} x\}| \\ &= 2 \cdot \# \text{ inversions } .\end{aligned}$$

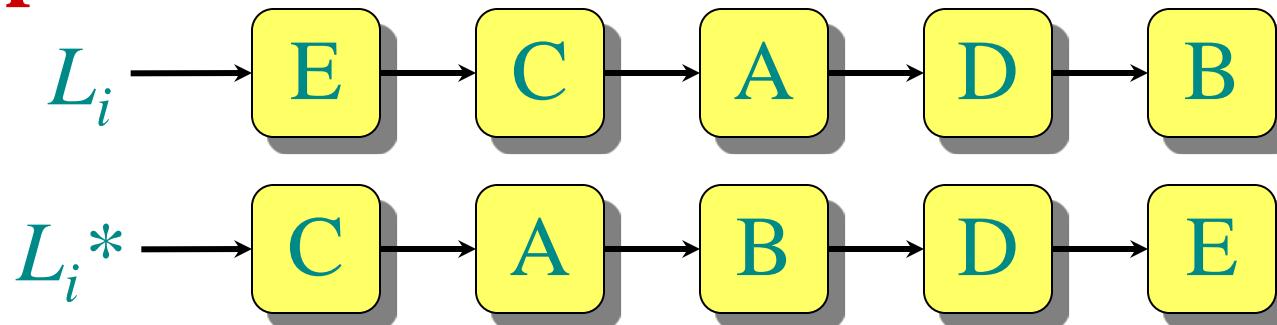


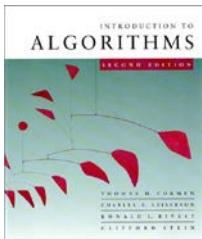
# Potential function

Define the potential function  $\Phi: \{L_i\} \rightarrow \mathbb{R}$  by

$$\begin{aligned}\Phi(L_i) &= 2 \cdot |\{(x, y) : x \prec_{L_i} y \text{ and } y \prec_{L_i^*} x\}| \\ &= 2 \cdot \# \text{ inversions } .\end{aligned}$$

**Example.**



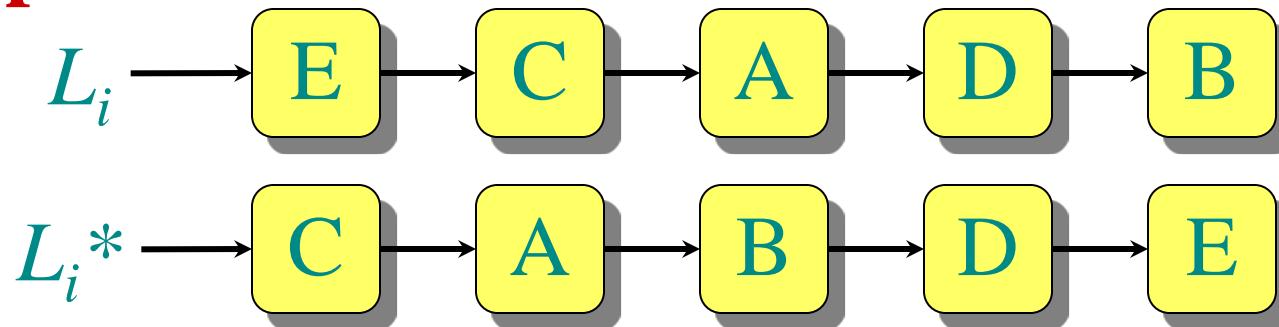


# Potential function

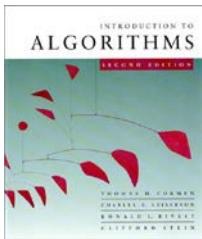
Define the potential function  $\Phi: \{L_i\} \rightarrow \mathbb{R}$  by

$$\begin{aligned}\Phi(L_i) &= 2 \cdot |\{(x, y) : x \prec_{L_i} y \text{ and } y \prec_{L_i^*} x\}| \\ &= 2 \cdot \# \text{ inversions } .\end{aligned}$$

**Example.**



$$\Phi(L_i) = 2 \cdot |\{\dots\}|$$

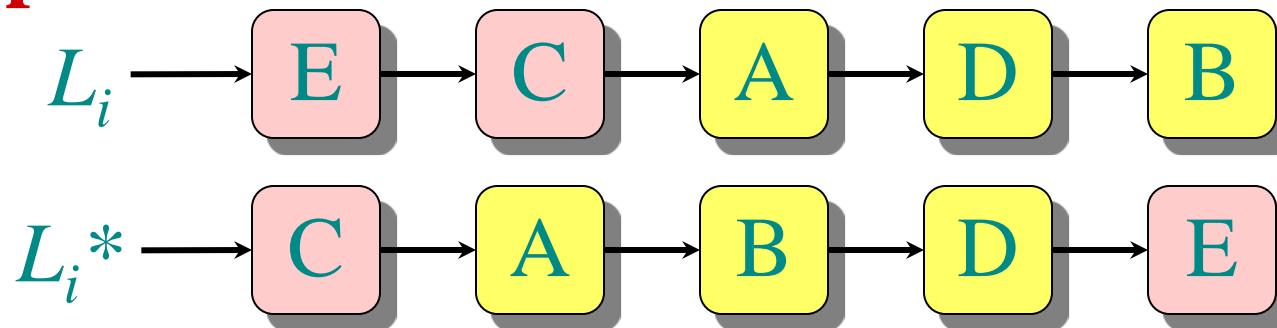


# Potential function

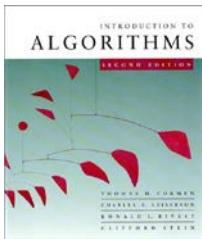
Define the potential function  $\Phi: \{L_i\} \rightarrow \mathbb{R}$  by

$$\begin{aligned}\Phi(L_i) &= 2 \cdot |\{(x, y) : x \prec_{L_i} y \text{ and } y \prec_{L_i^*} x\}| \\ &= 2 \cdot \# \text{ inversions } .\end{aligned}$$

**Example.**



$$\Phi(L_i) = 2 \cdot |\{(E, C), \dots\}|$$

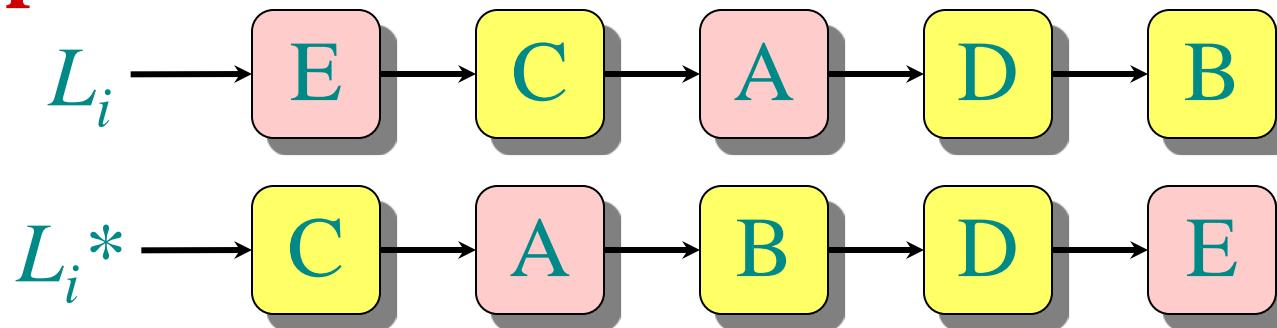


# Potential function

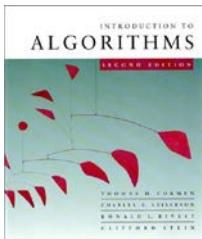
Define the potential function  $\Phi: \{L_i\} \rightarrow \mathbb{R}$  by

$$\begin{aligned}\Phi(L_i) &= 2 \cdot |\{(x, y) : x \prec_{L_i} y \text{ and } y \prec_{L_i^*} x\}| \\ &= 2 \cdot \# \text{ inversions } .\end{aligned}$$

**Example.**



$$\Phi(L_i) = 2 \cdot |\{(E,C), (E,A), \dots\}|$$

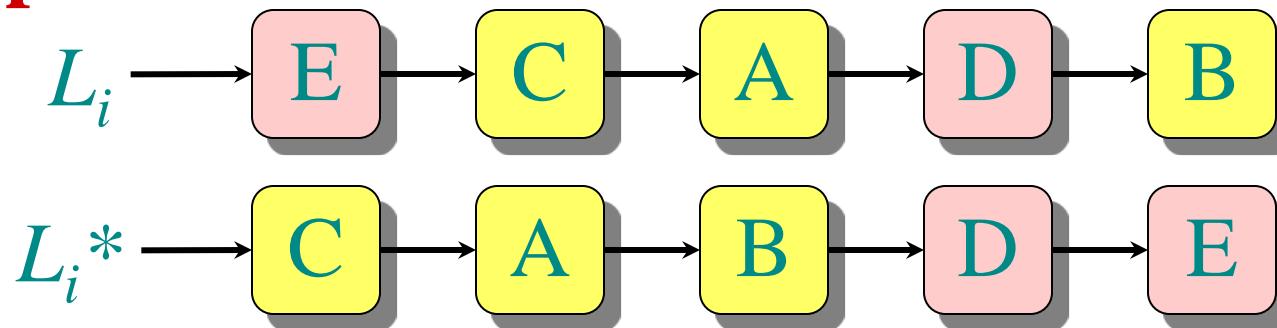


# Potential function

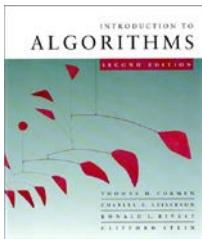
Define the potential function  $\Phi: \{L_i\} \rightarrow \mathbb{R}$  by

$$\begin{aligned}\Phi(L_i) &= 2 \cdot |\{(x, y) : x \prec_{L_i} y \text{ and } y \prec_{L_i^*} x\}| \\ &= 2 \cdot \# \text{ inversions } .\end{aligned}$$

**Example.**



$$\Phi(L_i) = 2 \cdot |\{(E,C), (E,A), (E,D), \dots\}|$$

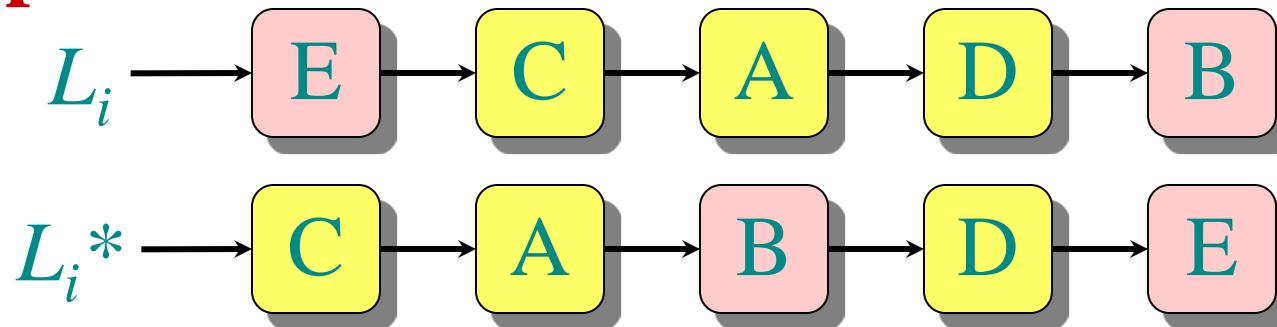


# Potential function

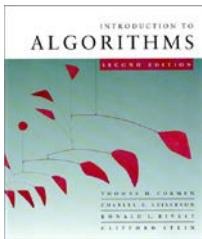
Define the potential function  $\Phi: \{L_i\} \rightarrow \mathbb{R}$  by

$$\begin{aligned}\Phi(L_i) &= 2 \cdot |\{(x, y) : x \prec_{L_i} y \text{ and } y \prec_{L_i^*} x\}| \\ &= 2 \cdot \# \text{ inversions } .\end{aligned}$$

**Example.**



$$\Phi(L_i) = 2 \cdot |\{(E,C), (E,A), (E,D), (E,B), \dots\}|$$

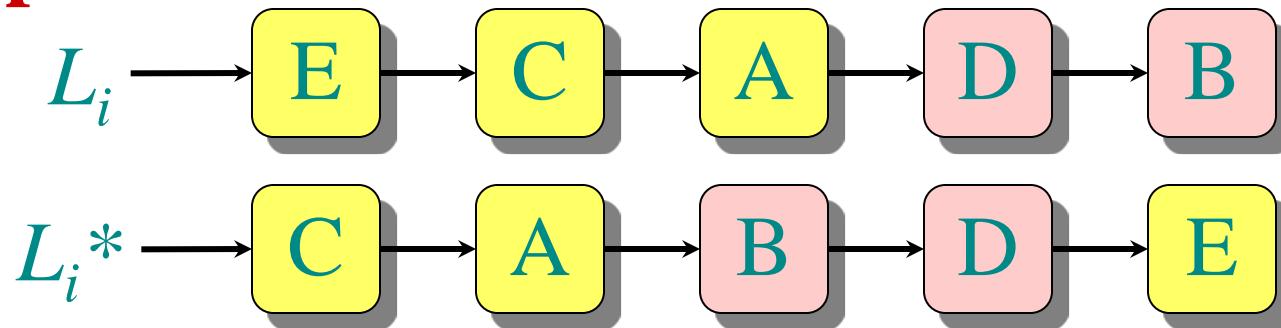


# Potential function

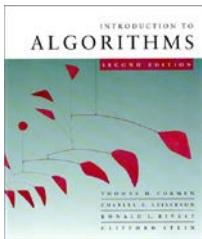
Define the potential function  $\Phi: \{L_i\} \rightarrow \mathbb{R}$  by

$$\begin{aligned}\Phi(L_i) &= 2 \cdot |\{(x, y) : x \prec_{L_i} y \text{ and } y \prec_{L_i^*} x\}| \\ &= 2 \cdot \# \text{ inversions } .\end{aligned}$$

**Example.**



$$\Phi(L_i) = 2 \cdot | \{ (E,C), (E,A), (E,D), (E,B), (D,B) \} |$$

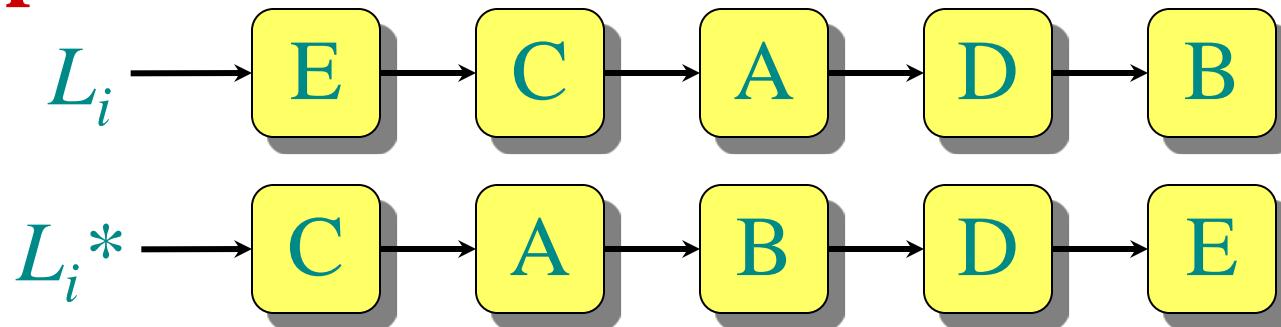


# Potential function

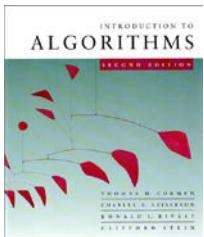
Define the potential function  $\Phi: \{L_i\} \rightarrow \mathbb{R}$  by

$$\begin{aligned}\Phi(L_i) &= 2 \cdot |\{(x, y) : x \prec_{L_i} y \text{ and } y \prec_{L_i^*} x\}| \\ &= 2 \cdot \# \text{ inversions } .\end{aligned}$$

**Example.**



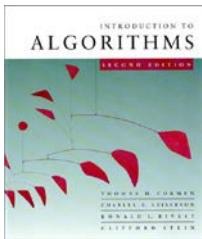
$$\begin{aligned}\Phi(L_i) &= 2 \cdot |\{(E,C), (E,A), (E,D), (E,B), (D,B)\}| \\ &= 10 .\end{aligned}$$



# Potential function

Define the potential function  $\Phi: \{L_i\} \rightarrow \mathbb{R}$  by

$$\begin{aligned}\Phi(L_i) &= 2 \cdot |\{(x, y) : x \prec_{L_i} y \text{ and } y \prec_{L_i^*} x\}| \\ &= 2 \cdot \# \text{ inversions } .\end{aligned}$$



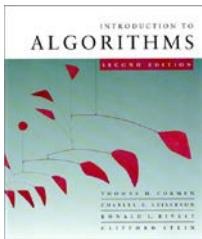
# Potential function

Define the potential function  $\Phi: \{L_i\} \rightarrow \mathbb{R}$  by

$$\begin{aligned}\Phi(L_i) &= 2 \cdot |\{(x, y) : x \prec_{L_i} y \text{ and } y \prec_{L_i^*} x\}| \\ &= 2 \cdot \# \text{ inversions } .\end{aligned}$$

Note that

- $\Phi(L_i) \geq 0$  for  $i = 0, 1, \dots,$
- $\Phi(L_0) = 0$  if MTF and OPT start with the same list.



# Potential function

Define the potential function  $\Phi: \{L_i\} \rightarrow \mathbb{R}$  by

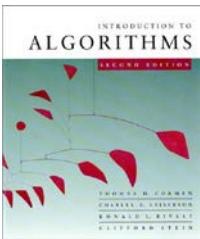
$$\begin{aligned}\Phi(L_i) &= 2 \cdot |\{(x, y) : x \prec_{L_i} y \text{ and } y \prec_{L_i^*} x\}| \\ &= 2 \cdot \# \text{ inversions } .\end{aligned}$$

Note that

- $\Phi(L_i) \geq 0$  for  $i = 0, 1, \dots,$
- $\Phi(L_0) = 0$  if MTF and OPT start with the same list.

How much does  $\Phi$  change from 1 transpose?

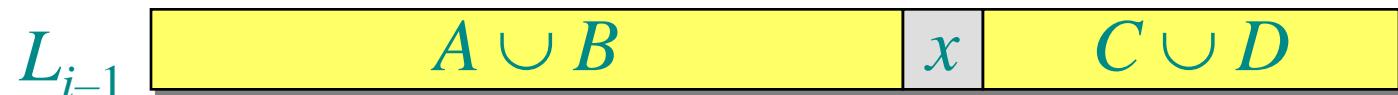
- A transpose creates/destroys 1 inversion.
- $\Delta\Phi = \pm 2$  .

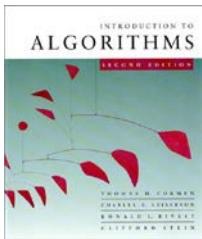


# What happens on an access?

Suppose that operation  $i$  accesses element  $x$ , and define

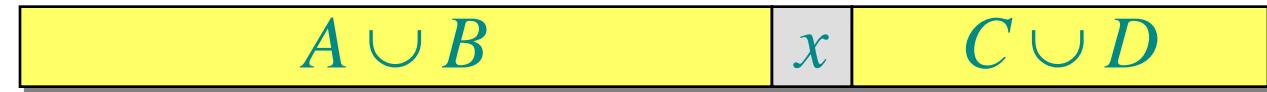
$$\begin{aligned}A &= \{y \in L_{i-1} : y \prec_{L_{i-1}} x \text{ and } y \prec_{L_{i-1}*} x\}, \\B &= \{y \in L_{i-1} : y \prec_{L_{i-1}} x \text{ and } y \succ_{L_{i-1}*} x\}, \\C &= \{y \in L_{i-1} : y \succ_{L_{i-1}} x \text{ and } y \prec_{L_{i-1}*} x\}, \\D &= \{y \in L_{i-1} : y \succ_{L_{i-1}} x \text{ and } y \succ_{L_{i-1}*} x\}.\end{aligned}$$





# What happens on an access?

$L_{i-1}$



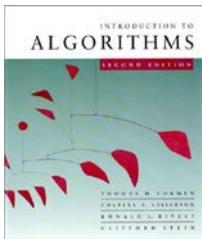
$$r = \text{rank}_{L_{i-1}}(x)$$

$L_{i-1}^*$

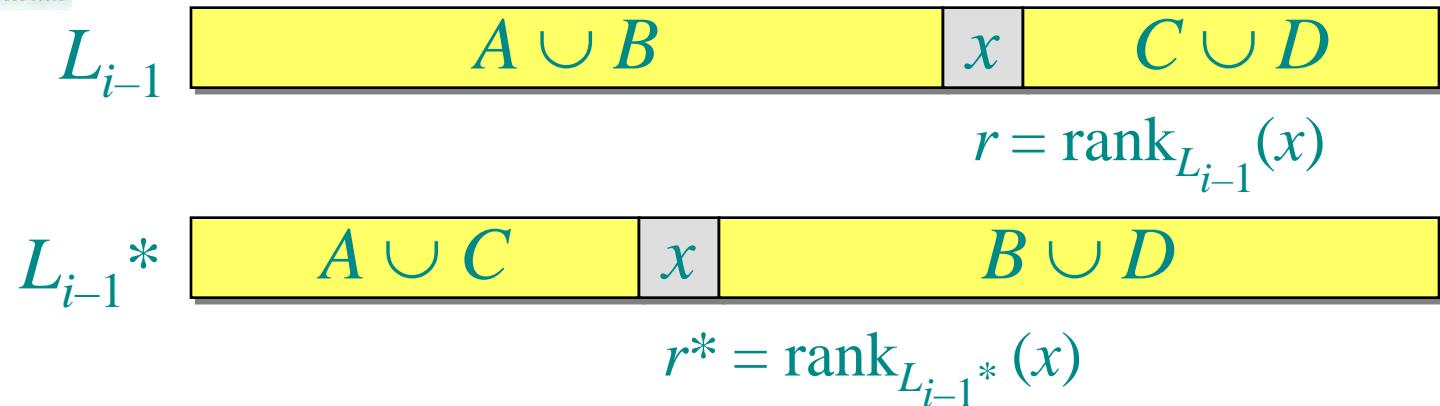


$$r^* = \text{rank}_{L_{i-1}^*}(x)$$

We have  $r = |A| + |B| + 1$  and  $r^* = |A| + |C| + 1$ .



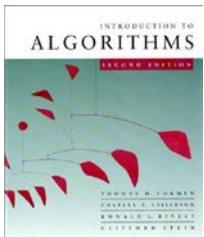
# What happens on an access?



We have  $r = |A| + |B| + 1$  and  $r^* = |A| + |C| + 1$ .

When MTF moves  $x$  to the front, it creates  $|A|$  inversions and destroys  $|B|$  inversions. Each transpose by OPT creates  $\leq 1$  inversion. Thus, we have

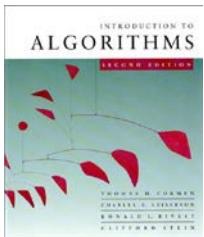
$$\Phi(L_i) - \Phi(L_{i-1}) \leq 2(|A| - |B| + t_i) .$$



# Amortized cost

The amortized cost for the  $i$ th operation of MTF with respect to  $\Phi$  is

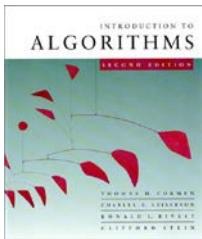
$$\hat{c}_i = c_i + \Phi(L_i) - \Phi(L_{i-1})$$



# Amortized cost

The amortized cost for the  $i$ th operation of MTF with respect to  $\Phi$  is

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(L_i) - \Phi(L_{i-1}) \\ &\leq 2r + 2(|A| - |B| + t_i)\end{aligned}$$

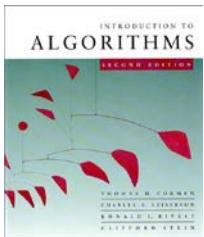


# Amortized cost

The amortized cost for the  $i$ th operation of MTF with respect to  $\Phi$  is

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(L_i) - \Phi(L_{i-1}) \\ &\leq 2r + 2(|A| - |B| + t_i) \\ &= 2r + 2(|A| - (r - 1 - |A|)) + t_i\end{aligned}$$

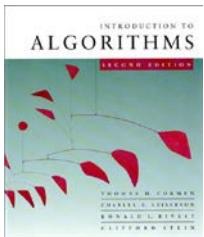
(since  $r = |A| + |B| + 1$ )



# Amortized cost

The amortized cost for the  $i$ th operation of MTF with respect to  $\Phi$  is

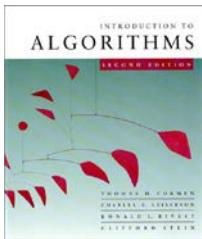
$$\begin{aligned}\hat{c}_i &= c_i + \Phi(L_i) - \Phi(L_{i-1}) \\&\leq 2r + 2(|A| - |B| + t_i) \\&= 2r + 2(|A| - (r - 1 - |A|)) + t_i \\&= 2r + 4|A| - 2r + 2 + 2t_i\end{aligned}$$



# Amortized cost

The amortized cost for the  $i$ th operation of MTF with respect to  $\Phi$  is

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(L_i) - \Phi(L_{i-1}) \\&\leq 2r + 2(|A| - |B| + t_i) \\&= 2r + 2(|A| - (r - 1 - |A|)) + t_i \\&= 2r + 4|A| - 2r + 2 + 2t_i \\&= 4|A| + 2 + 2t_i\end{aligned}$$

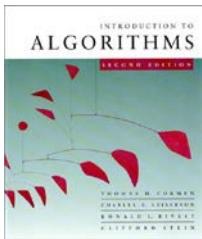


# Amortized cost

The amortized cost for the  $i$ th operation of MTF with respect to  $\Phi$  is

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(L_i) - \Phi(L_{i-1}) \\&\leq 2r + 2(|A| - |B| + t_i) \\&= 2r + 2(|A| - (r - 1 - |A|) + t_i) \\&= 2r + 4|A| - 2r + 2 + 2t_i \\&= 4|A| + 2 + 2t_i \\&\leq 4(r^* + t_i)\end{aligned}$$

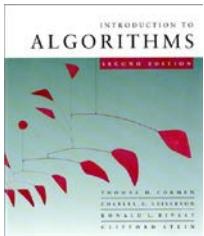
(since  $r^* = |A| + |C| + 1 \geq |A| + 1$ )



# Amortized cost

The amortized cost for the  $i$ th operation of MTF with respect to  $\Phi$  is

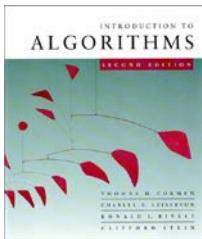
$$\begin{aligned}\hat{c}_i &= c_i + \Phi(L_i) - \Phi(L_{i-1}) \\&\leq 2r + 2(|A| - |B| + t_i) \\&= 2r + 2(|A| - (r - 1 - |A|)) + t_i \\&= 2r + 4|A| - 2r + 2 + 2t_i \\&= 4|A| + 2 + 2t_i \\&\leq 4(r^* + t_i) \\&= 4c_i^*.\end{aligned}$$



# The grand finale

Thus, we have

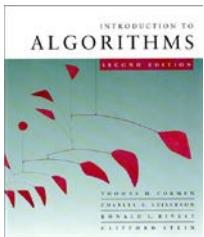
$$C_{\text{MTF}}(S) = \sum_{i=1}^{|S|} c_i$$



# The grand finale

Thus, we have

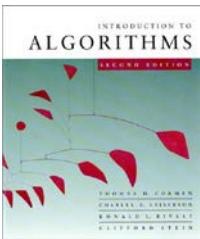
$$\begin{aligned} C_{\text{MTF}}(S) &= \sum_{i=1}^{|S|} c_i \\ &= \sum_{i=1}^{|S|} (\hat{c}_i + \Phi(L_{i-1}) - \Phi(L_i)) \end{aligned}$$



# The grand finale

Thus, we have

$$\begin{aligned} C_{\text{MTF}}(S) &= \sum_{i=1}^{|S|} c_i \\ &= \sum_{i=1}^{|S|} (\hat{c}_i + \Phi(L_{i-1}) - \Phi(L_i)) \\ &\leq \left( \sum_{i=1}^{|S|} 4c_i^* \right) + \Phi(L_0) - \Phi(L_{|S|}) \end{aligned}$$

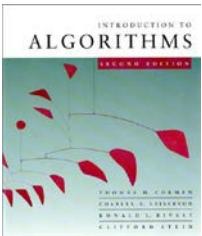


# The grand finale

Thus, we have

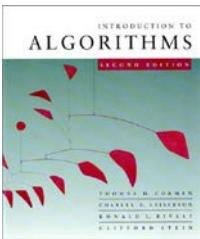
$$\begin{aligned} C_{\text{MTF}}(S) &= \sum_{i=1}^{|S|} c_i \\ &= \sum_{i=1}^{|S|} (\hat{c}_i + \Phi(L_{i-1}) - \Phi(L_i)) \\ &\leq \left( \sum_{i=1}^{|S|} 4c_i^* \right) + \Phi(L_0) - \Phi(L_{|S|}) \\ &\leq 4 \cdot C_{\text{OPT}}(S), \end{aligned}$$

since  $\Phi(L_0) = 0$  and  $\Phi(L_{|S|}) \geq 0$ . □



# Addendum

If we count transpositions that move  $x$  toward the front as “free” (models splicing  $x$  in and out of  $L$  in constant time), then MTF is 2-competitive.



# Addendum

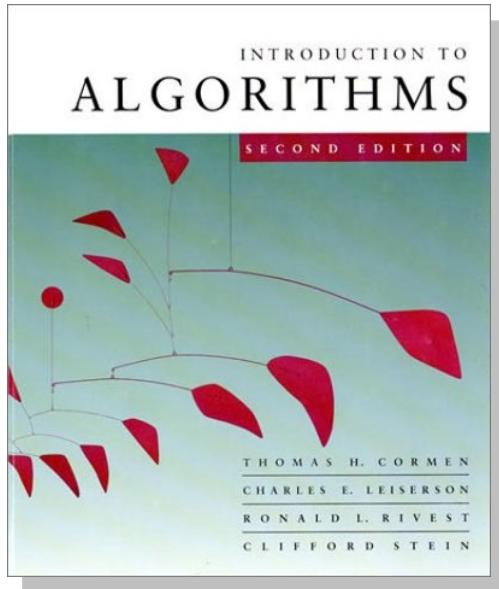
If we count transpositions that move  $x$  toward the front as “free” (models splicing  $x$  in and out of  $L$  in constant time), then MTF is 2-competitive.

What if  $L_0 \neq L_0^*$ ?

- Then,  $\Phi(L_0)$  might be  $\Theta(n^2)$  in the worst case.
- Thus,  $C_{\text{MTF}}(S) \leq 4 \cdot C_{\text{OPT}}(S) + \Theta(n^2)$ , which is still 4-competitive, since  $n^2$  is constant as  $|S| \rightarrow \infty$ .

# *Introduction to Algorithms*

## 6.046J/18.401J

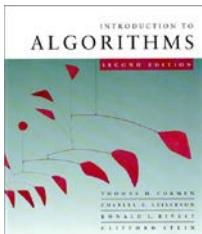


### LECTURE 15

#### Dynamic Programming

- Longest common subsequence
- Optimal substructure
- Overlapping subproblems

Prof. Charles E. Leiserson

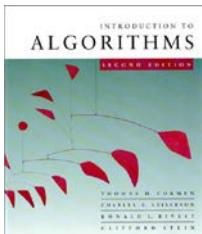


# Dynamic programming

*Design technique, like divide-and-conquer.*

**Example: *Longest Common Subsequence (LCS)***

- Given two sequences  $x[1 \dots m]$  and  $y[1 \dots n]$ , find a longest subsequence common to them both.



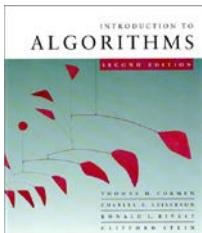
# Dynamic programming

*Design technique, like divide-and-conquer.*

**Example: *Longest Common Subsequence (LCS)***

- Given two sequences  $x[1 \dots m]$  and  $y[1 \dots n]$ , find a longest subsequence common to them both.  

- “a” *not* “the”



# Dynamic programming

*Design technique, like divide-and-conquer.*

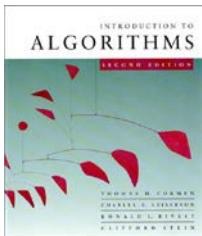
**Example: *Longest Common Subsequence (LCS)***

- Given two sequences  $x[1 \dots m]$  and  $y[1 \dots n]$ , find a longest subsequence common to them both.

“a” *not* “the”

$x$ : A    B    C    B    D    A    B

$y$ : B    D    C    A    B    A



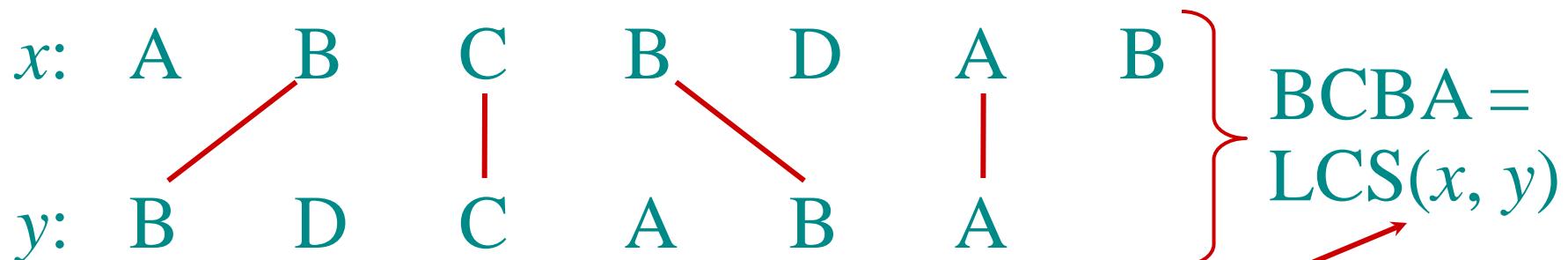
# Dynamic programming

*Design technique, like divide-and-conquer.*

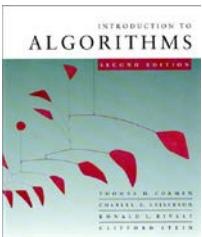
**Example: *Longest Common Subsequence (LCS)***

- Given two sequences  $x[1 \dots m]$  and  $y[1 \dots n]$ , find a longest subsequence common to them both.

“a” not “the”

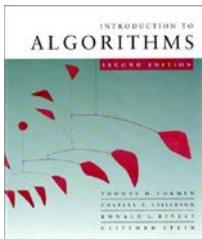


functional notation,  
but not a function



# Brute-force LCS algorithm

Check every subsequence of  $x[1 \dots m]$  to see if it is also a subsequence of  $y[1 \dots n]$ .



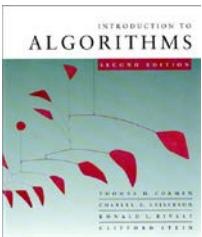
# Brute-force LCS algorithm

Check every subsequence of  $x[1 \dots m]$  to see if it is also a subsequence of  $y[1 \dots n]$ .

## Analysis

- Checking =  $O(n)$  time per subsequence.
- $2^m$  subsequences of  $x$  (each bit-vector of length  $m$  determines a distinct subsequence of  $x$ ).

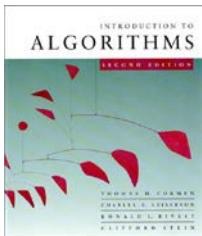
Worst-case running time =  $O(n2^m)$   
= exponential time.



# Towards a better algorithm

## Simplification:

1. Look at the *length* of a longest-common subsequence.
2. Extend the algorithm to find the LCS itself.

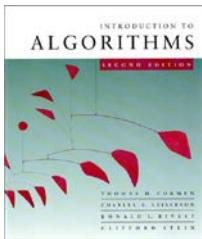


# Towards a better algorithm

## Simplification:

1. Look at the *length* of a longest-common subsequence.
2. Extend the algorithm to find the LCS itself.

**Notation:** Denote the length of a sequence  $s$  by  $|s|$ .



# Towards a better algorithm

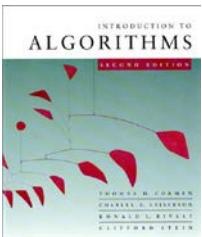
## Simplification:

1. Look at the *length* of a longest-common subsequence.
2. Extend the algorithm to find the LCS itself.

**Notation:** Denote the length of a sequence  $s$  by  $|s|$ .

**Strategy:** Consider *prefixes* of  $x$  and  $y$ .

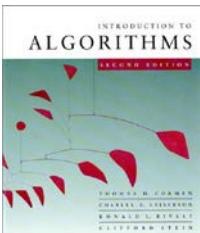
- Define  $c[i, j] = |\text{LCS}(x[1 \dots i], y[1 \dots j])|$ .
- Then,  $c[m, n] = |\text{LCS}(x, y)|$ .



# Recursive formulation

**Theorem.**

$$c[i, j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\ \max\{c[i-1, j], c[i, j-1]\} & \text{otherwise.} \end{cases}$$

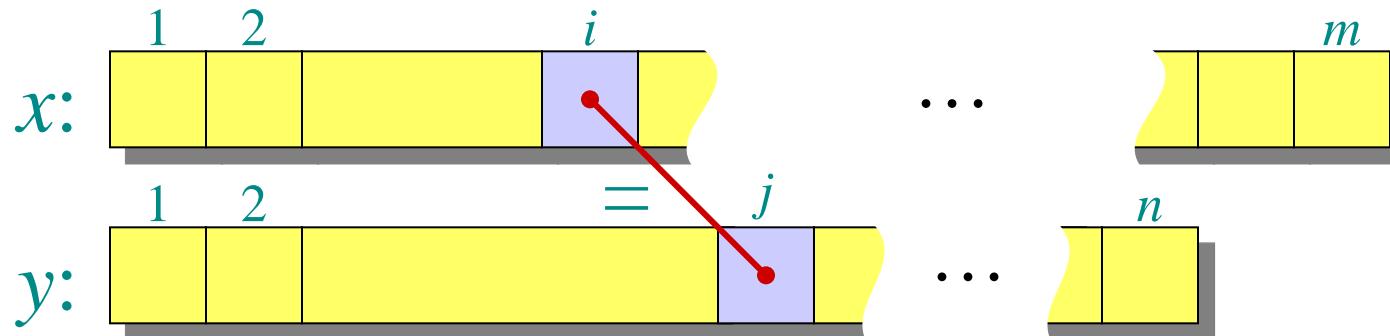


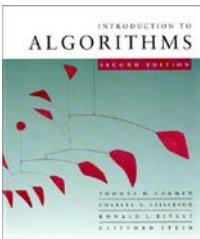
# Recursive formulation

**Theorem.**

$$c[i, j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\ \max\{c[i-1, j], c[i, j-1]\} & \text{otherwise.} \end{cases}$$

*Proof.* Case  $x[i] = y[j]$ :



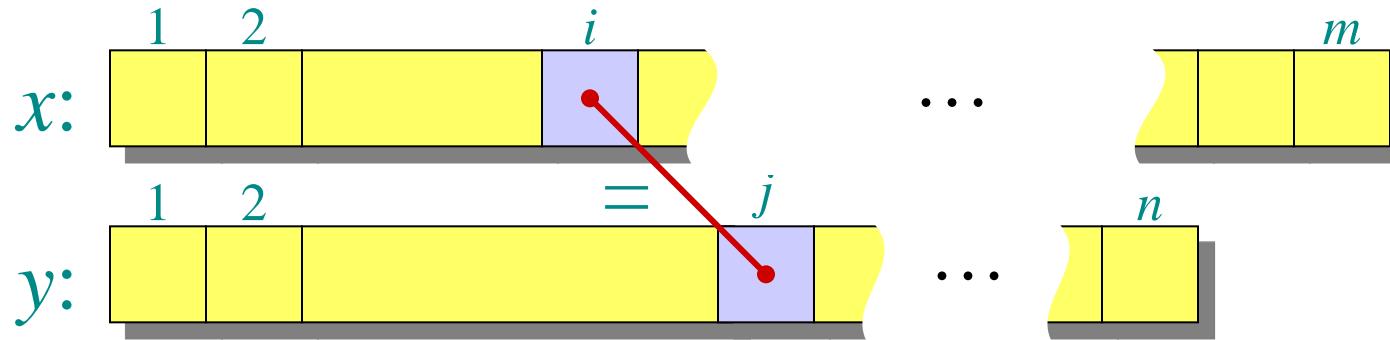


# Recursive formulation

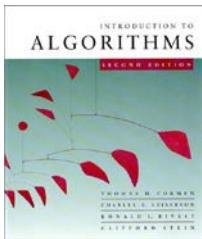
**Theorem.**

$$c[i, j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\ \max\{c[i-1, j], c[i, j-1]\} & \text{otherwise.} \end{cases}$$

*Proof.* Case  $x[i] = y[j]$ :



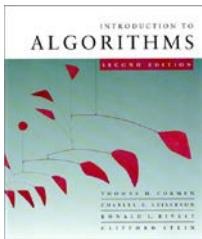
Let  $z[1 \dots k] = \text{LCS}(x[1 \dots i], y[1 \dots j])$ , where  $c[i, j] = k$ . Then,  $z[k] = x[i]$ , or else  $z$  could be extended. Thus,  $z[1 \dots k-1]$  is CS of  $x[1 \dots i-1]$  and  $y[1 \dots j-1]$ .



# Proof (continued)

**Claim:**  $z[1 \dots k-1] = \text{LCS}(x[1 \dots i-1], y[1 \dots j-1])$ .

Suppose  $w$  is a longer CS of  $x[1 \dots i-1]$  and  $y[1 \dots j-1]$ , that is,  $|w| > k-1$ . Then, ***cut and paste***:  $w \parallel z[k]$  ( $w$  concatenated with  $z[k]$ ) is a common subsequence of  $x[1 \dots i]$  and  $y[1 \dots j]$  with  $|w \parallel z[k]| > k$ . Contradiction, proving the claim.



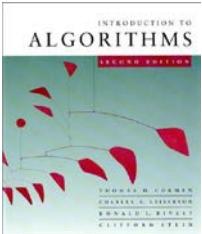
# Proof (continued)

**Claim:**  $z[1 \dots k-1] = \text{LCS}(x[1 \dots i-1], y[1 \dots j-1])$ .

Suppose  $w$  is a longer CS of  $x[1 \dots i-1]$  and  $y[1 \dots j-1]$ , that is,  $|w| > k-1$ . Then, ***cut and paste***:  $w \parallel z[k]$  ( $w$  concatenated with  $z[k]$ ) is a common subsequence of  $x[1 \dots i]$  and  $y[1 \dots j]$  with  $|w \parallel z[k]| > k$ . Contradiction, proving the claim.

Thus,  $c[i-1, j-1] = k-1$ , which implies that  $c[i, j] = c[i-1, j-1] + 1$ .

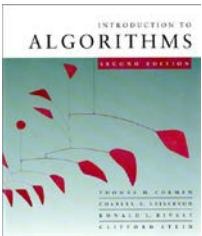
Other cases are similar.



# Dynamic-programming hallmark #1

## *Optimal substructure*

*An optimal solution to a problem (instance) contains optimal solutions to subproblems.*

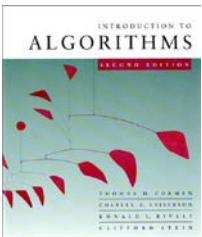


# Dynamic-programming hallmark #1

## *Optimal substructure*

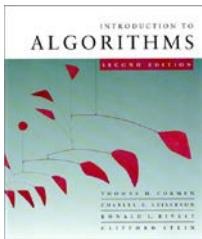
*An optimal solution to a problem (instance) contains optimal solutions to subproblems.*

If  $z = \text{LCS}(x, y)$ , then any prefix of  $z$  is an LCS of a prefix of  $x$  and a prefix of  $y$ .



# Recursive algorithm for LCS

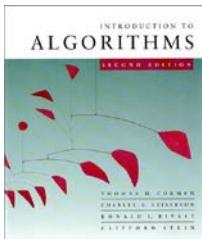
```
LCS( $x, y, i, j$ )    // ignoring base cases
if  $x[i] = y[j]$ 
  then  $c[i, j] \leftarrow \text{LCS}(x, y, i-1, j-1) + 1$ 
  else  $c[i, j] \leftarrow \max \{ \text{LCS}(x, y, i-1, j),$ 
                                 $\text{LCS}(x, y, i, j-1) \}$ 
return  $c[i, j]$ 
```



# Recursive algorithm for LCS

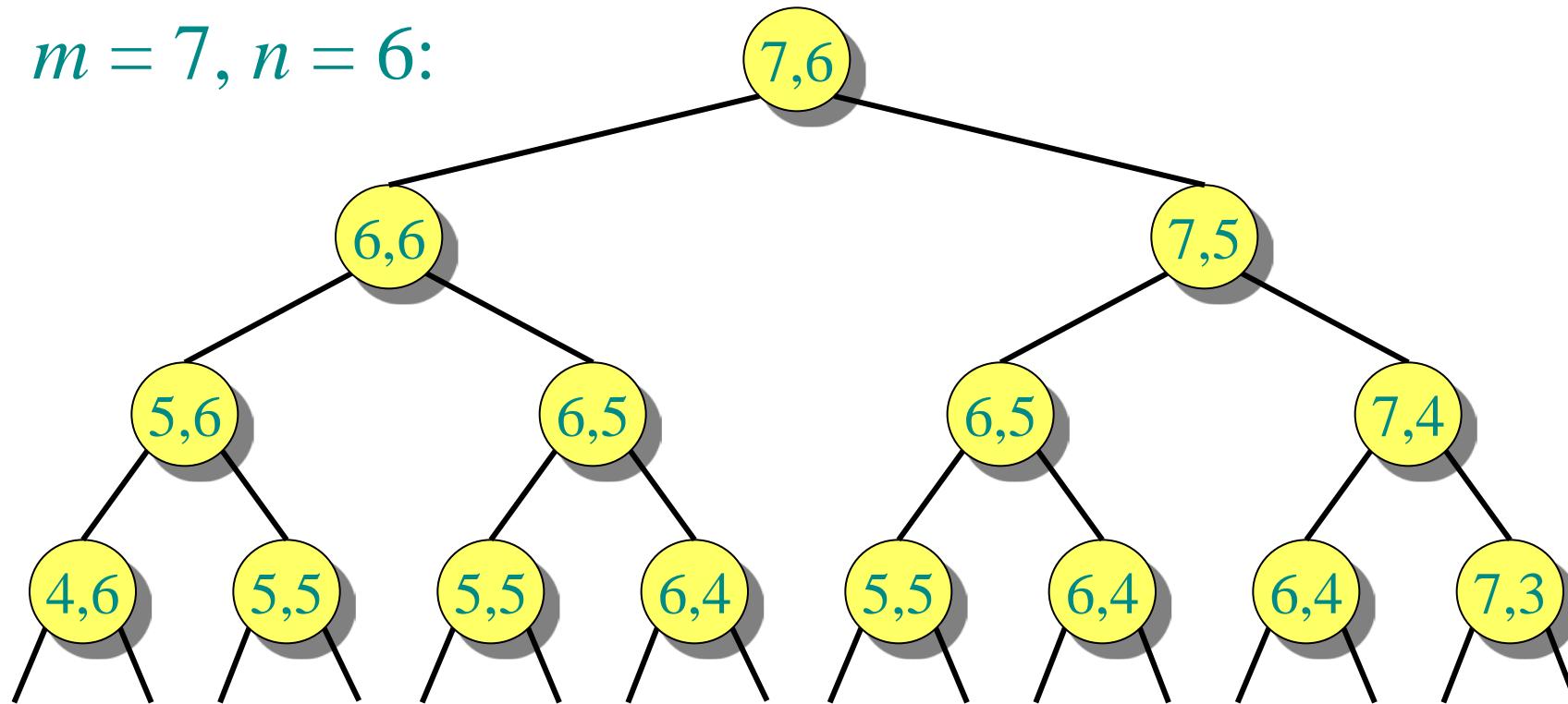
```
LCS( $x, y, i, j$ )    // ignoring base cases
  if  $x[i] = y[j]$ 
    then  $c[i, j] \leftarrow \text{LCS}(x, y, i-1, j-1) + 1$ 
  else  $c[i, j] \leftarrow \max \{ \text{LCS}(x, y, i-1, j),$ 
                                 $\text{LCS}(x, y, i, j-1) \}$ 
  return  $c[i, j]$ 
```

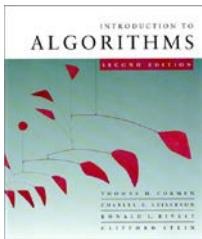
**Worse case:**  $x[i] \neq y[j]$ , in which case the algorithm evaluates two subproblems, each with only one parameter decremented.



# Recursion tree

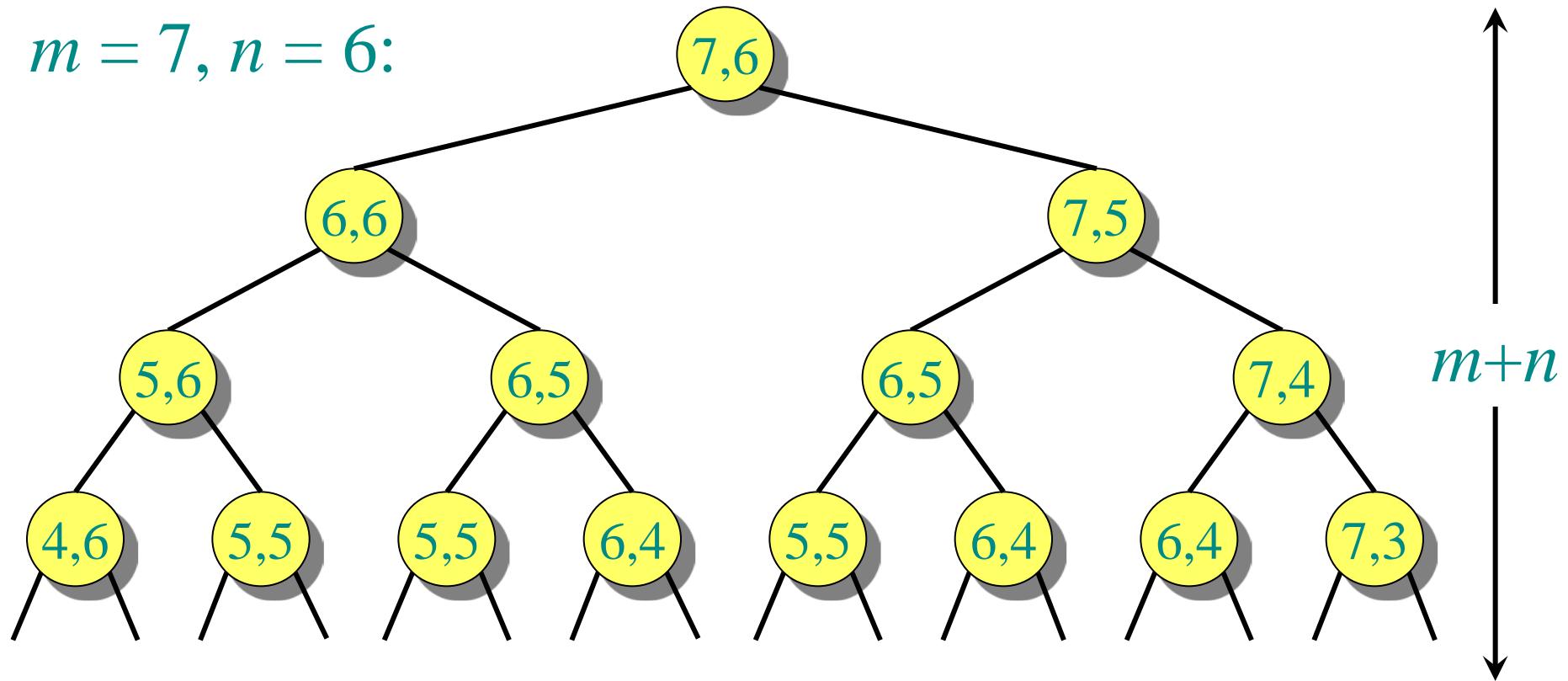
$m = 7, n = 6$ :



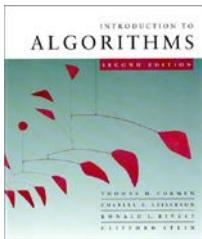


# Recursion tree

$m = 7, n = 6:$

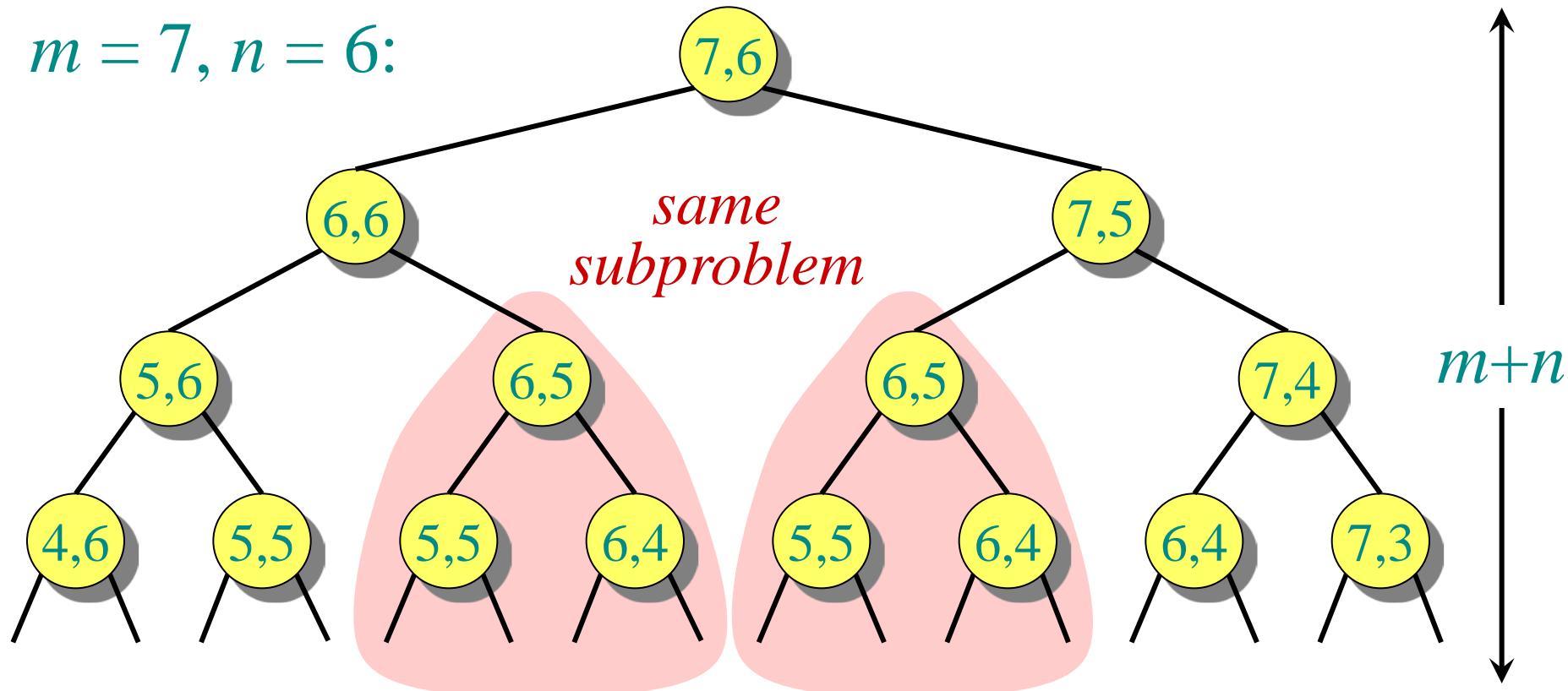


Height  $= m + n \Rightarrow$  work potentially exponential.

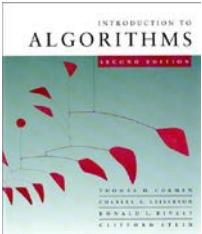


# Recursion tree

$m = 7, n = 6:$



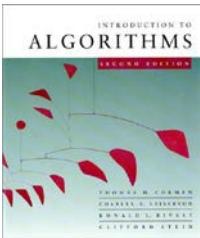
Height  $= m + n \Rightarrow$  work potentially exponential,  
but we're solving subproblems already solved!



# Dynamic-programming hallmark #2

## *Overlapping subproblems*

*A recursive solution contains a “small” number of distinct subproblems repeated many times.*

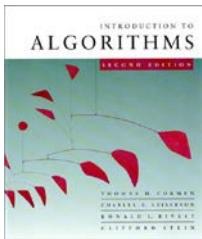


# Dynamic-programming hallmark #2

## *Overlapping subproblems*

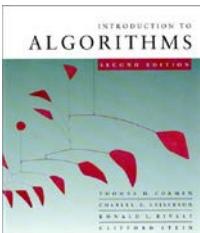
*A recursive solution contains a “small” number of distinct subproblems repeated many times.*

The number of distinct LCS subproblems for two strings of lengths  $m$  and  $n$  is only  $mn$ .



# Memoization algorithm

***Memoization:*** After computing a solution to a subproblem, store it in a table. Subsequent calls check the table to avoid redoing work.



# Memoization algorithm

**Memoization:** After computing a solution to a subproblem, store it in a table. Subsequent calls check the table to avoid redoing work.

$\text{LCS}(x, y, i, j)$

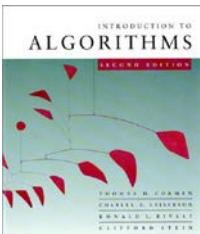
**if**  $c[i, j] = \text{NIL}$

**then if**  $x[i] = y[j]$

**then**  $c[i, j] \leftarrow \text{LCS}(x, y, i-1, j-1) + 1$

**else**  $c[i, j] \leftarrow \max \{ \text{LCS}(x, y, i-1, j), \text{LCS}(x, y, i, j-1) \}$

*same  
as  
before*



# Memoization algorithm

**Memoization:** After computing a solution to a subproblem, store it in a table. Subsequent calls check the table to avoid redoing work.

$\text{LCS}(x, y, i, j)$

**if**  $c[i, j] = \text{NIL}$

**then if**  $x[i] = y[j]$

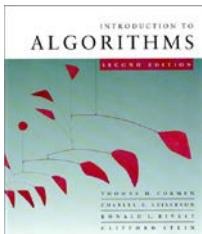
**then**  $c[i, j] \leftarrow \text{LCS}(x, y, i-1, j-1) + 1$

**else**  $c[i, j] \leftarrow \max \{ \text{LCS}(x, y, i-1, j), \text{LCS}(x, y, i, j-1) \}$

*same  
as  
before*

Time =  $\Theta(mn)$  = constant work per table entry.

Space =  $\Theta(mn)$ .

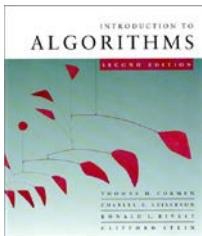


# Dynamic-programming algorithm

**IDEA:**

Compute the table bottom-up.

|   | A | B | C | B | D | A | B |
|---|---|---|---|---|---|---|---|
| A | 0 | 1 | 1 | 2 | 2 | 2 | 3 |
| B | 0 | 1 | 2 | 2 | 3 | 3 | 4 |
| C | 0 | 0 | 1 | 2 | 2 | 2 | 2 |
| D | 0 | 0 | 1 | 1 | 1 | 2 | 2 |
|   | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| B | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| A | 0 | 1 | 2 | 2 | 3 | 4 | 4 |



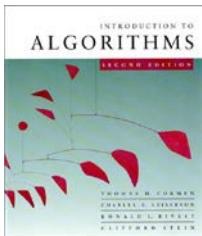
# Dynamic-programming algorithm

**IDEA:**

Compute the table bottom-up.

Time =  $\Theta(mn)$ .

|   | A | B | C | B | D | A | B |
|---|---|---|---|---|---|---|---|
| B | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| D | 0 | 0 | 1 | 1 | 1 | 2 | 2 |
| C | 0 | 0 | 1 | 2 | 2 | 2 | 2 |
| A | 0 | 1 | 1 | 2 | 2 | 2 | 3 |
| B | 0 | 1 | 2 | 2 | 3 | 3 | 3 |
| A | 0 | 1 | 2 | 2 | 3 | 3 | 4 |



# Dynamic-programming algorithm

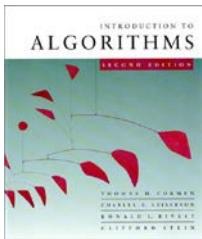
## IDEA:

Compute the table bottom-up.

Time =  $\Theta(mn)$ .

Reconstruct LCS by tracing backwards.

|   | A | B | C | B | D | A | B |   |
|---|---|---|---|---|---|---|---|---|
| B | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| D | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| C | 0 | 0 | 1 | 2 | 2 | 2 | 2 | 2 |
| A | 0 | 1 | 1 | 2 | 2 | 2 | 3 | 3 |
| B | 0 | 1 | 2 | 2 | 3 | 3 | 3 | 4 |
| A | 0 | 1 | 2 | 2 | 3 | 3 | 4 | 4 |



# Dynamic-programming algorithm

## IDEA:

Compute the table bottom-up.

Time =  $\Theta(mn)$ .

Reconstruct LCS by tracing backwards.

Space =  $\Theta(mn)$ .

## Exercise:

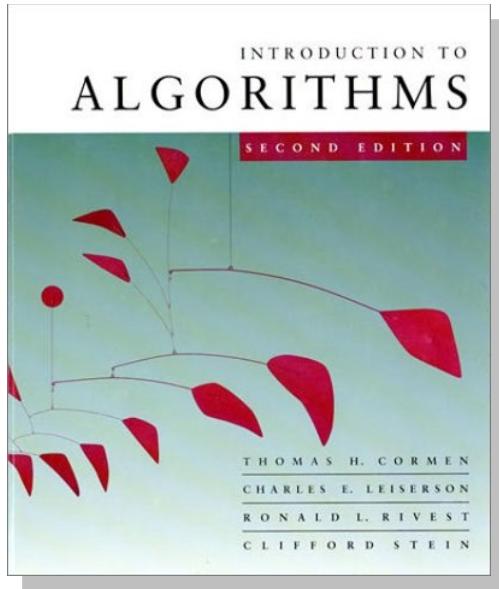
$O(\min\{m, n\})$ .

|   | A | B | C | B | D | A | B |
|---|---|---|---|---|---|---|---|
| B | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| D | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| C | 0 | 0 | 1 | 2 | 2 | 2 | 2 |
| A | 0 | 1 | 1 | 2 | 2 | 2 | 3 |
| B | 0 | 1 | 2 | 2 | 3 | 3 | 3 |
| A | 0 | 1 | 2 | 2 | 3 | 3 | 4 |

The table illustrates the computation of the Longest Common Subsequence (LCS) between two sequences, A and B. The rows represent sequence A and the columns represent sequence B. The diagonal elements (0,0), (1,1), (2,2), (3,3), and (4,4) are marked with red numbers. Other cells contain blue numbers, indicating the length of the LCS up to that point. Diagonal arrows point from the bottom-left towards the top-right, tracing the path of matches between the two sequences.

# *Introduction to Algorithms*

## 6.046J/18.401J

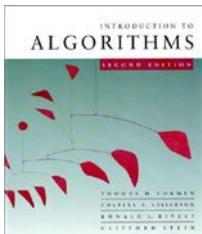


## LECTURE 16

### Greedy Algorithms (and Graphs)

- Graph representation
- Minimum spanning trees
- Optimal substructure
- Greedy choice
- Prim's greedy MST algorithm

Prof. Charles E. Leiserson



# Graphs (review)

**Definition.** A *directed graph (digraph)*

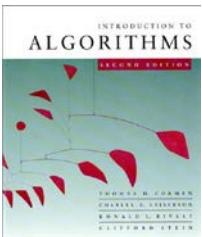
$G = (V, E)$  is an ordered pair consisting of

- a set  $V$  of *vertices* (singular: *vertex*),
- a set  $E \subseteq V \times V$  of *edges*.

In an *undirected graph*  $G = (V, E)$ , the edge set  $E$  consists of *unordered* pairs of vertices.

In either case, we have  $|E| = O(V^2)$ . Moreover, if  $G$  is connected, then  $|E| \geq |V| - 1$ , which implies that  $\lg |E| = \Theta(\lg V)$ .

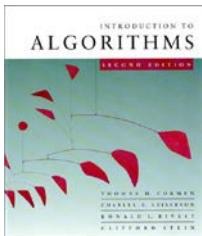
(Review CLRS, Appendix B.)



# Adjacency-matrix representation

The **adjacency matrix** of a graph  $G = (V, E)$ , where  $V = \{1, 2, \dots, n\}$ , is the matrix  $A[1 \dots n, 1 \dots n]$  given by

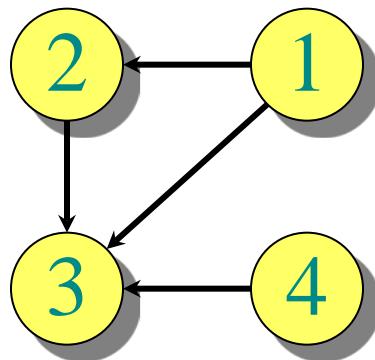
$$A[i, j] = \begin{cases} 1 & \text{if } (i, j) \in E, \\ 0 & \text{if } (i, j) \notin E. \end{cases}$$



# Adjacency-matrix representation

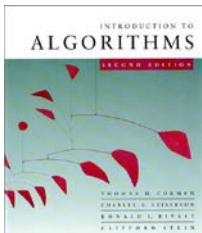
The **adjacency matrix** of a graph  $G = (V, E)$ , where  $V = \{1, 2, \dots, n\}$ , is the matrix  $A[1 \dots n, 1 \dots n]$  given by

$$A[i, j] = \begin{cases} 1 & \text{if } (i, j) \in E, \\ 0 & \text{if } (i, j) \notin E. \end{cases}$$



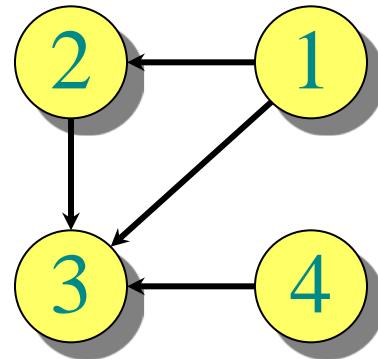
| A | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 0 |
| 2 | 0 | 0 | 1 | 0 |
| 3 | 0 | 0 | 0 | 0 |
| 4 | 0 | 0 | 1 | 0 |

$\Theta(V^2)$  storage  
 $\Rightarrow$  **dense**  
representation.



# Adjacency-list representation

An *adjacency list* of a vertex  $v \in V$  is the list  $\text{Adj}[v]$  of vertices adjacent to  $v$ .

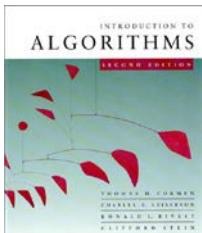


$$\text{Adj}[1] = \{2, 3\}$$

$$\text{Adj}[2] = \{3\}$$

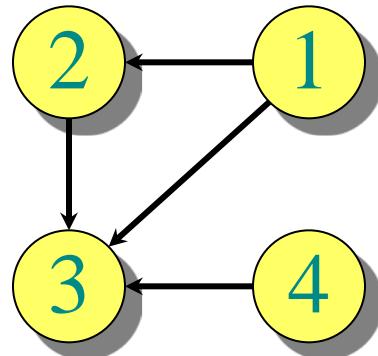
$$\text{Adj}[3] = \{\}$$

$$\text{Adj}[4] = \{3\}$$



# Adjacency-list representation

An *adjacency list* of a vertex  $v \in V$  is the list  $\text{Adj}[v]$  of vertices adjacent to  $v$ .



$$\text{Adj}[1] = \{2, 3\}$$

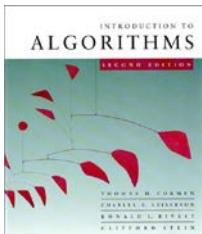
$$\text{Adj}[2] = \{3\}$$

$$\text{Adj}[3] = \{\}$$

$$\text{Adj}[4] = \{3\}$$

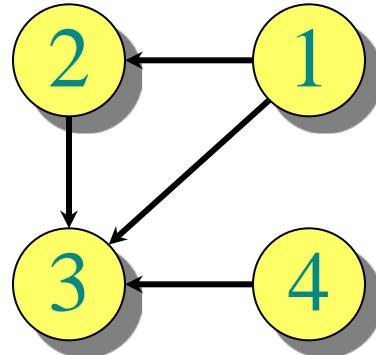
For undirected graphs,  $|\text{Adj}[v]| = \text{degree}(v)$ .

For digraphs,  $|\text{Adj}[v]| = \text{out-degree}(v)$ .



# Adjacency-list representation

An *adjacency list* of a vertex  $v \in V$  is the list  $\text{Adj}[v]$  of vertices adjacent to  $v$ .

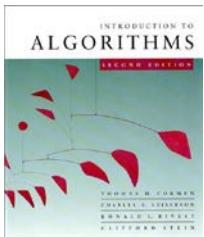


$$\begin{aligned}\text{Adj}[1] &= \{2, 3\} \\ \text{Adj}[2] &= \{3\} \\ \text{Adj}[3] &= \{\} \\ \text{Adj}[4] &= \{3\}\end{aligned}$$

For undirected graphs,  $|\text{Adj}[v]| = \text{degree}(v)$ .

For digraphs,  $|\text{Adj}[v]| = \text{out-degree}(v)$ .

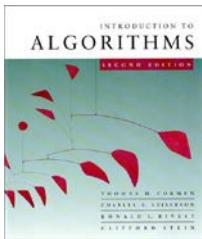
**Handshaking Lemma:**  $\sum_{v \in V} \text{degree}(v) = 2|E|$  for undirected graphs  $\Rightarrow$  adjacency lists use  $\Theta(V + E)$  storage — a *sparse* representation.



# Minimum spanning trees

**Input:** A connected, undirected graph  $G = (V, E)$  with weight function  $w : E \rightarrow \mathbb{R}$ .

- For simplicity, assume that all edge weights are distinct. (CLRS covers the general case.)



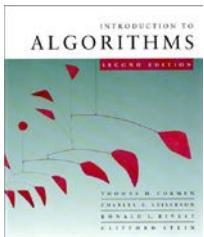
# Minimum spanning trees

**Input:** A connected, undirected graph  $G = (V, E)$  with weight function  $w : E \rightarrow \mathbb{R}$ .

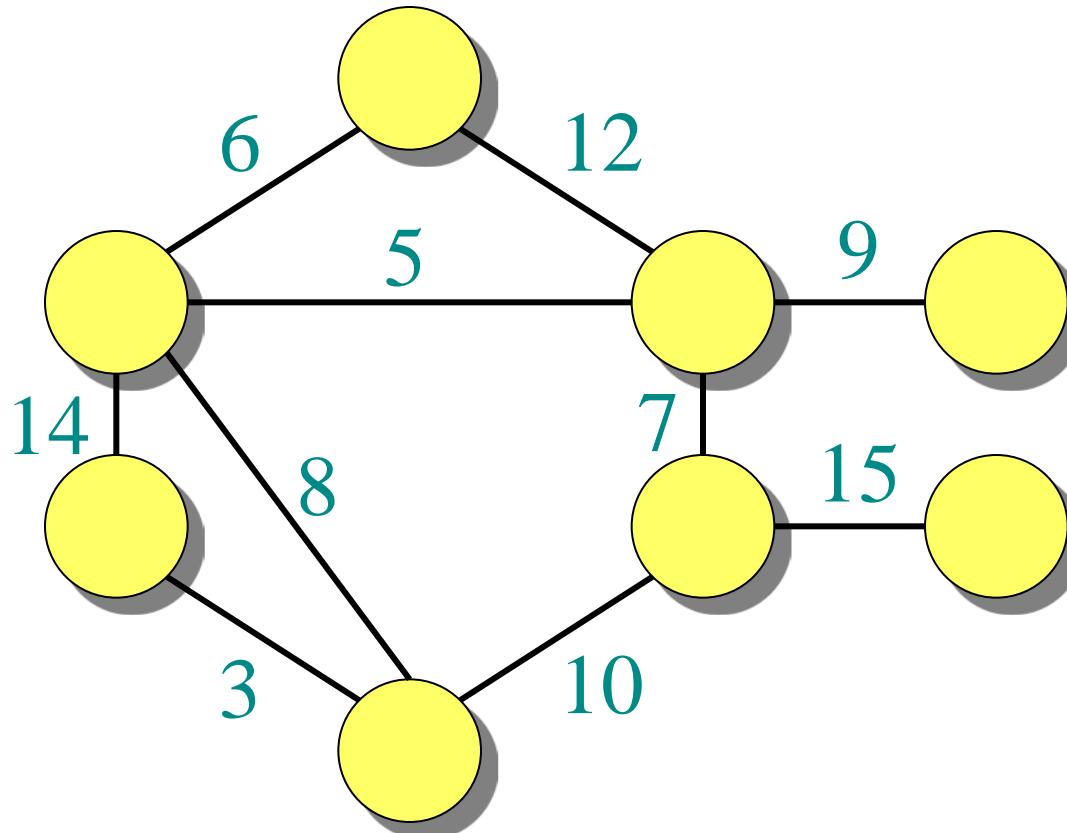
- For simplicity, assume that all edge weights are distinct. (CLRS covers the general case.)

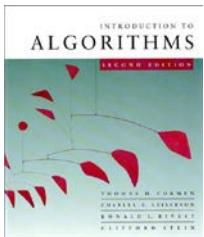
**Output:** A *spanning tree*  $T$  — a tree that connects all vertices — of minimum weight:

$$w(T) = \sum_{(u,v) \in T} w(u,v).$$

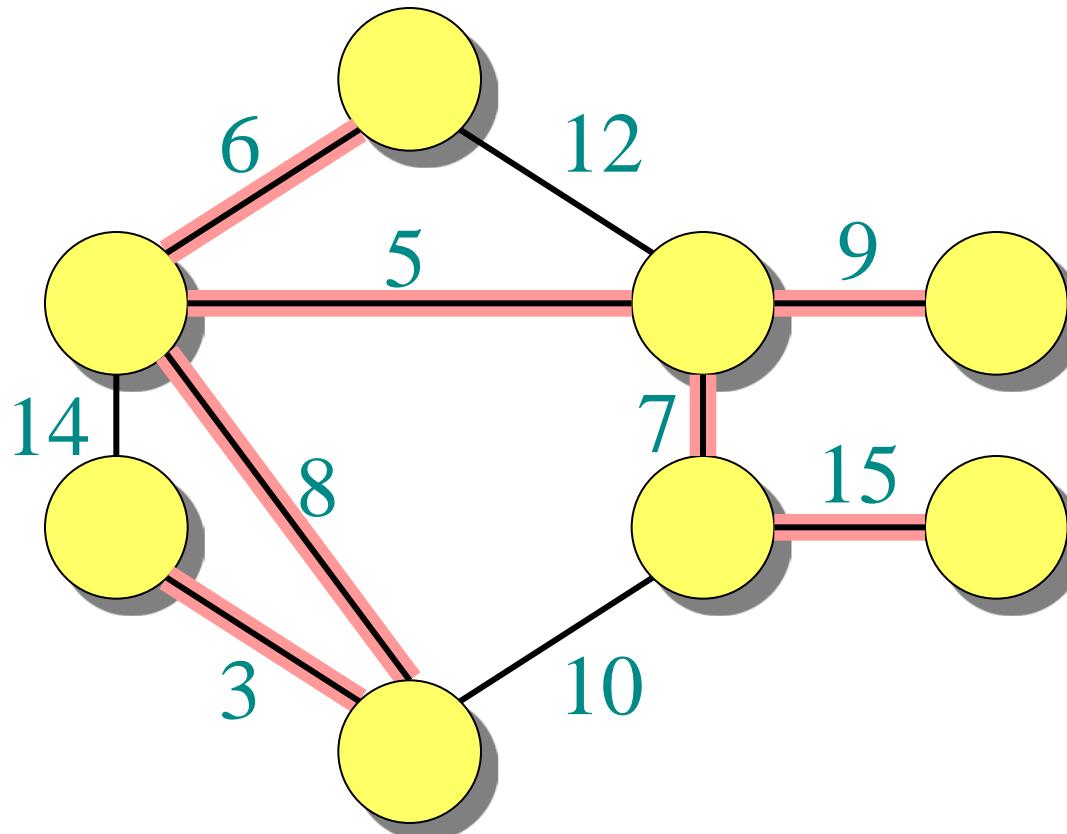


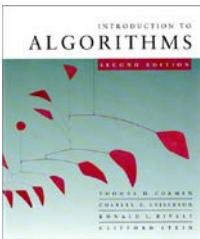
# Example of MST





# Example of MST

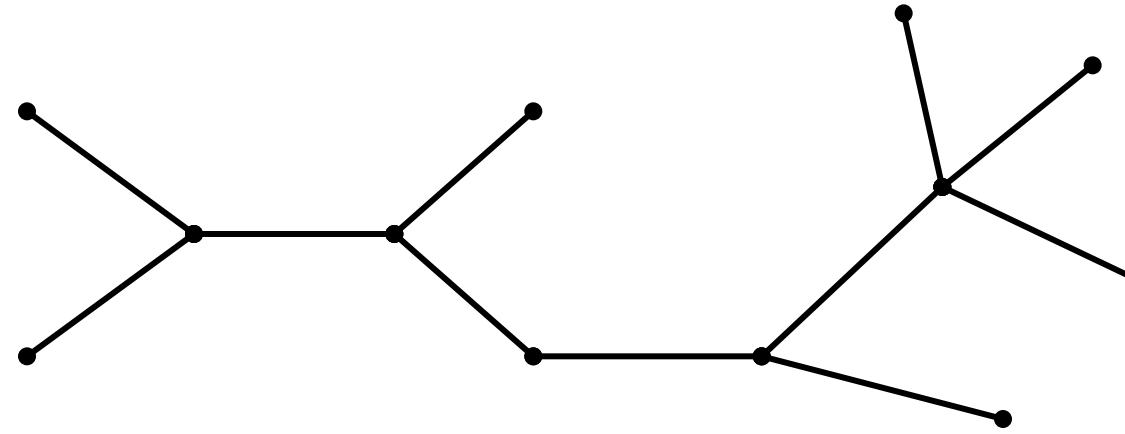


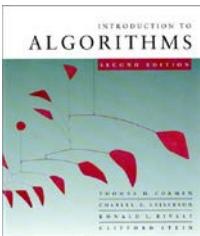


# Optimal substructure

MST  $T$ :

(Other edges of  $G$   
are not shown.)

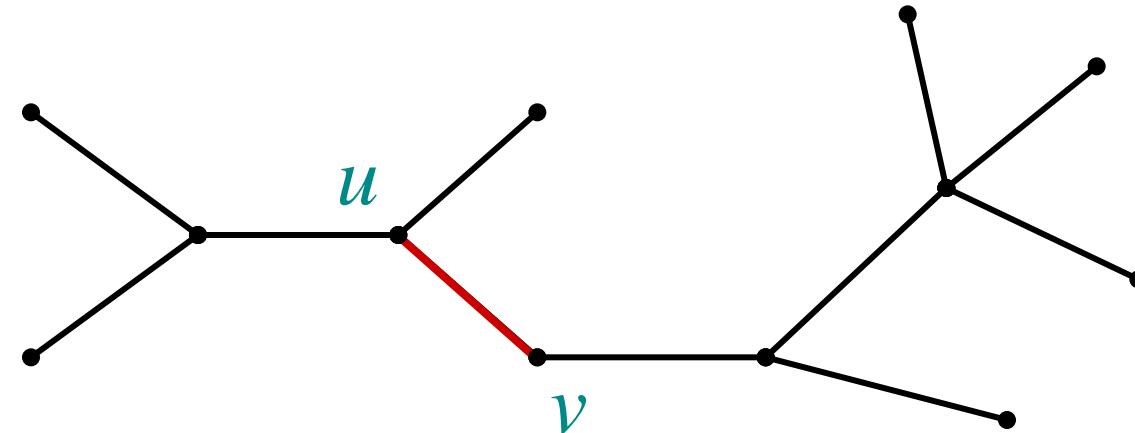




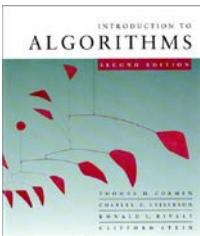
# Optimal substructure

MST  $T$ :

(Other edges of  $G$   
are not shown.)



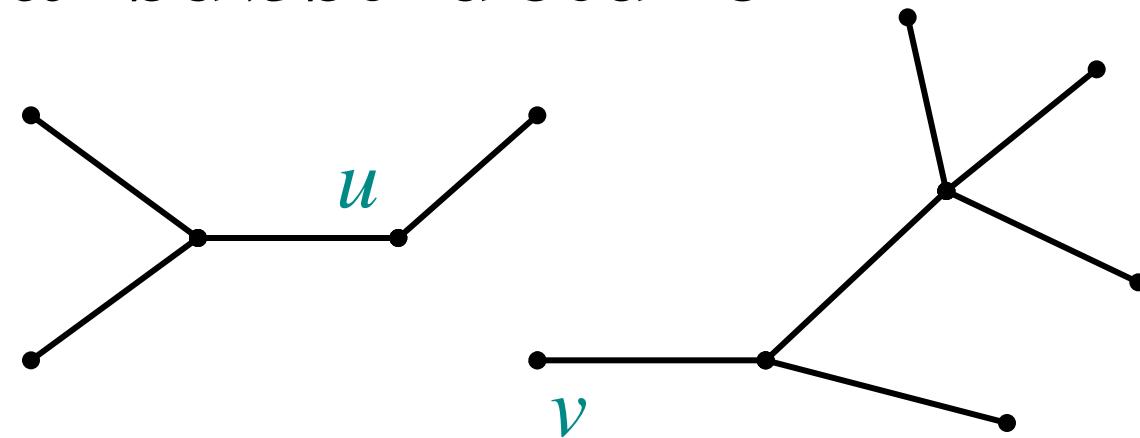
Remove any edge  $(u, v) \in T$ .



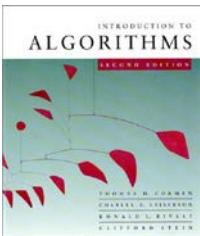
# Optimal substructure

MST  $T$ :

(Other edges of  $G$   
are not shown.)



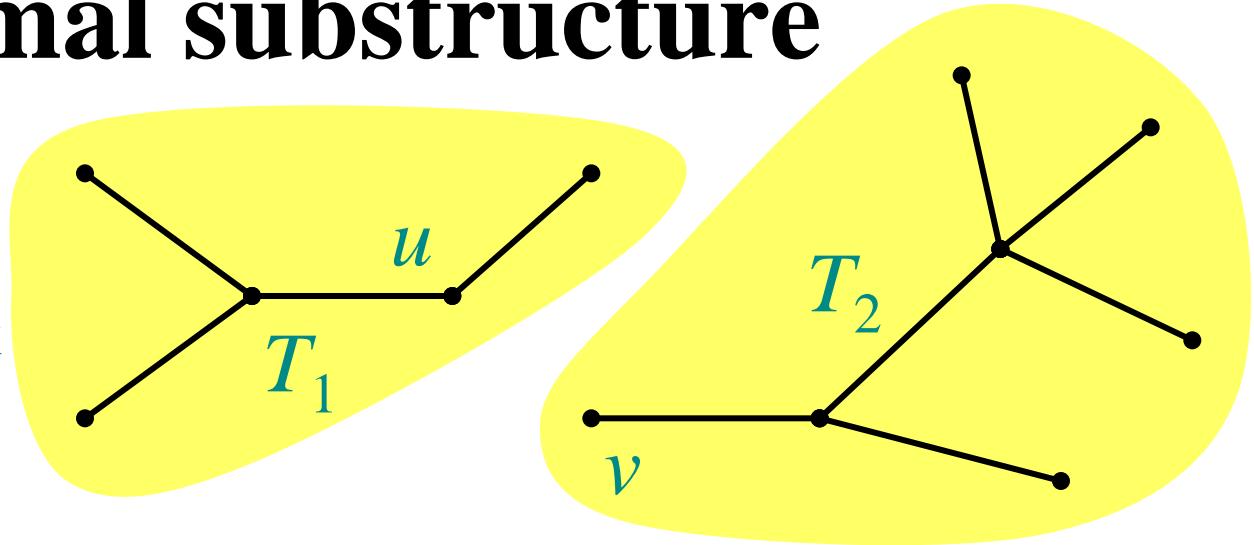
Remove any edge  $(u, v) \in T$ .



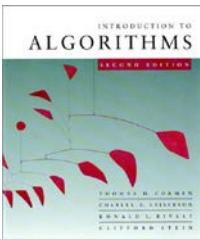
# Optimal substructure

MST  $T$ :

(Other edges of  $G$   
are not shown.)



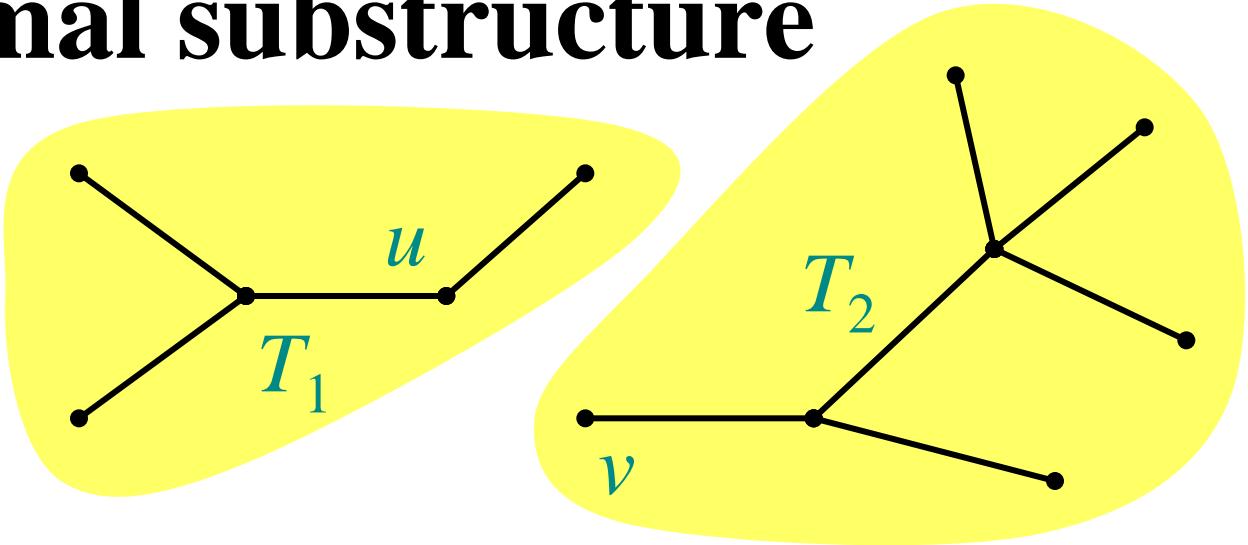
Remove any edge  $(u, v) \in T$ . Then,  $T$  is partitioned into two subtrees  $T_1$  and  $T_2$ .



# Optimal substructure

MST  $T$ :

(Other edges of  $G$   
are not shown.)



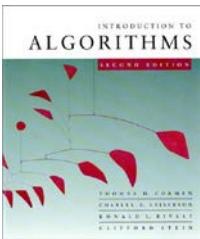
Remove any edge  $(u, v) \in T$ . Then,  $T$  is partitioned into two subtrees  $T_1$  and  $T_2$ .

**Theorem.** The subtree  $T_1$  is an MST of  $G_1 = (V_1, E_1)$ , the subgraph of  $G$  *induced* by the vertices of  $T_1$ :

$$V_1 = \text{vertices of } T_1,$$

$$E_1 = \{ (x, y) \in E : x, y \in V_1 \}.$$

Similarly for  $T_2$ .

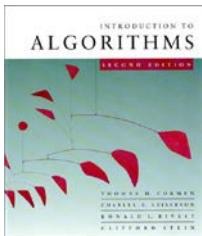


# Proof of optimal substructure

*Proof.* Cut and paste:

$$w(T) = w(u, v) + w(T_1) + w(T_2).$$

If  $T'_1$  were a lower-weight spanning tree than  $T_1$  for  $G_1$ , then  $T' = \{(u, v)\} \cup T'_1 \cup T_2$  would be a lower-weight spanning tree than  $T$  for  $G$ . □



# Proof of optimal substructure

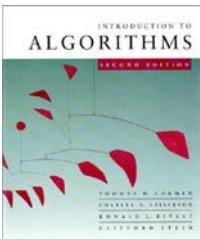
*Proof.* Cut and paste:

$$w(T) = w(u, v) + w(T_1) + w(T_2).$$

If  $T'_1$  were a lower-weight spanning tree than  $T_1$  for  $G_1$ , then  $T' = \{(u, v)\} \cup T'_1 \cup T_2$  would be a lower-weight spanning tree than  $T$  for  $G$ . □

Do we also have overlapping subproblems?

- Yes.



# Proof of optimal substructure

*Proof.* Cut and paste:

$$w(T) = w(u, v) + w(T_1) + w(T_2).$$

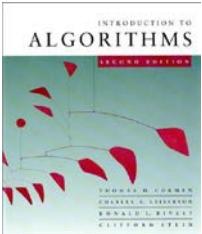
If  $T'_1$  were a lower-weight spanning tree than  $T_1$  for  $G_1$ , then  $T' = \{(u, v)\} \cup T'_1 \cup T_2$  would be a lower-weight spanning tree than  $T$  for  $G$ . □

Do we also have overlapping subproblems?

- Yes.

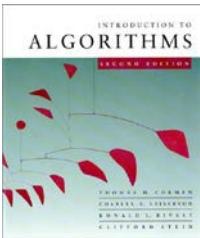
Great, then dynamic programming may work!

- Yes, but MST exhibits another powerful property which leads to an even more efficient algorithm.



# Hallmark for “greedy” algorithms

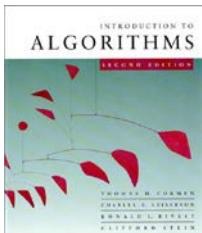
*Greedy-choice property*  
A locally optimal choice  
is globally optimal.



# Hallmark for “greedy” algorithms

***Greedy-choice property***  
*A locally optimal choice  
is globally optimal.*

**Theorem.** Let  $T$  be the MST of  $G = (V, E)$ , and let  $A \subseteq V$ . Suppose that  $(u, v) \in E$  is the least-weight edge connecting  $A$  to  $V - A$ . Then,  $(u, v) \in T$ .

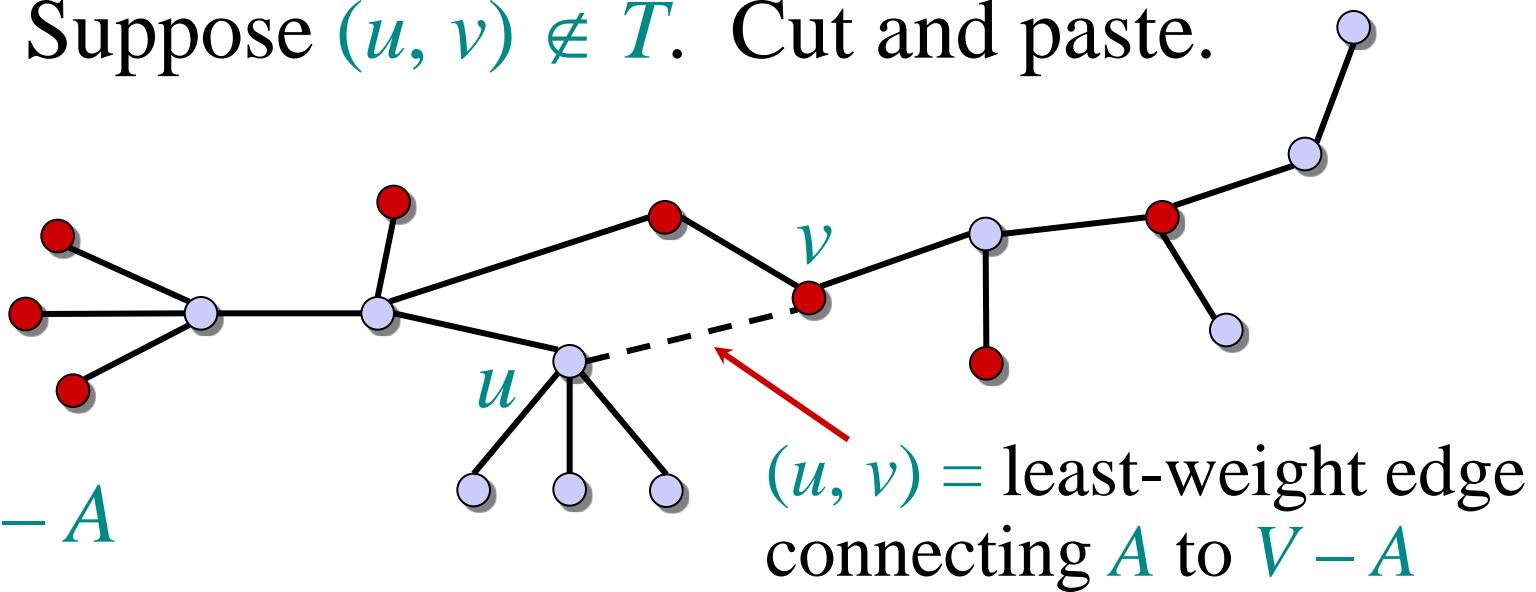


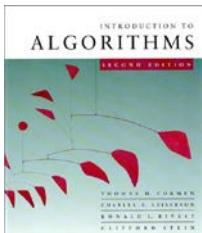
# Proof of theorem

*Proof.* Suppose  $(u, v) \notin T$ . Cut and paste.

$T$ :

- $\in A$
- $\in V - A$



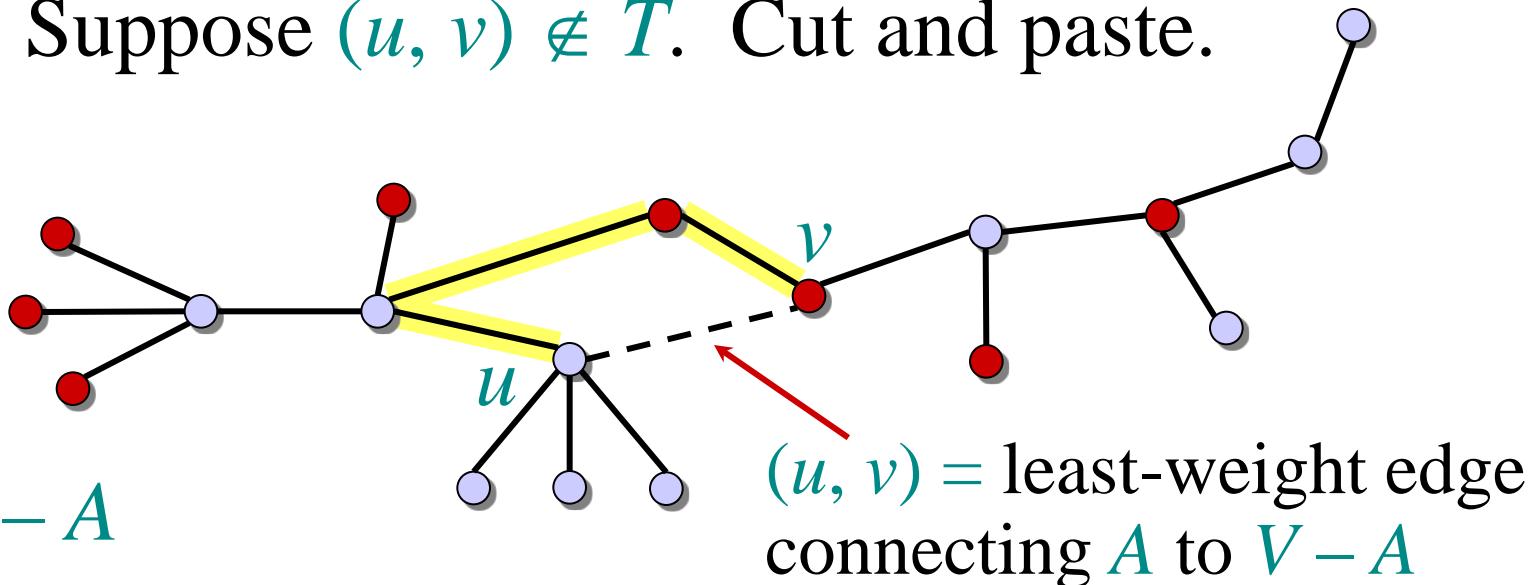


# Proof of theorem

*Proof.* Suppose  $(u, v) \notin T$ . Cut and paste.

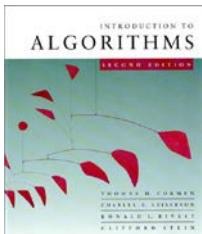
$T$ :

- $\in A$
- $\in V - A$



$(u, v)$  = least-weight edge  
connecting  $A$  to  $V - A$

Consider the unique simple path from  $u$  to  $v$  in  $T$ .

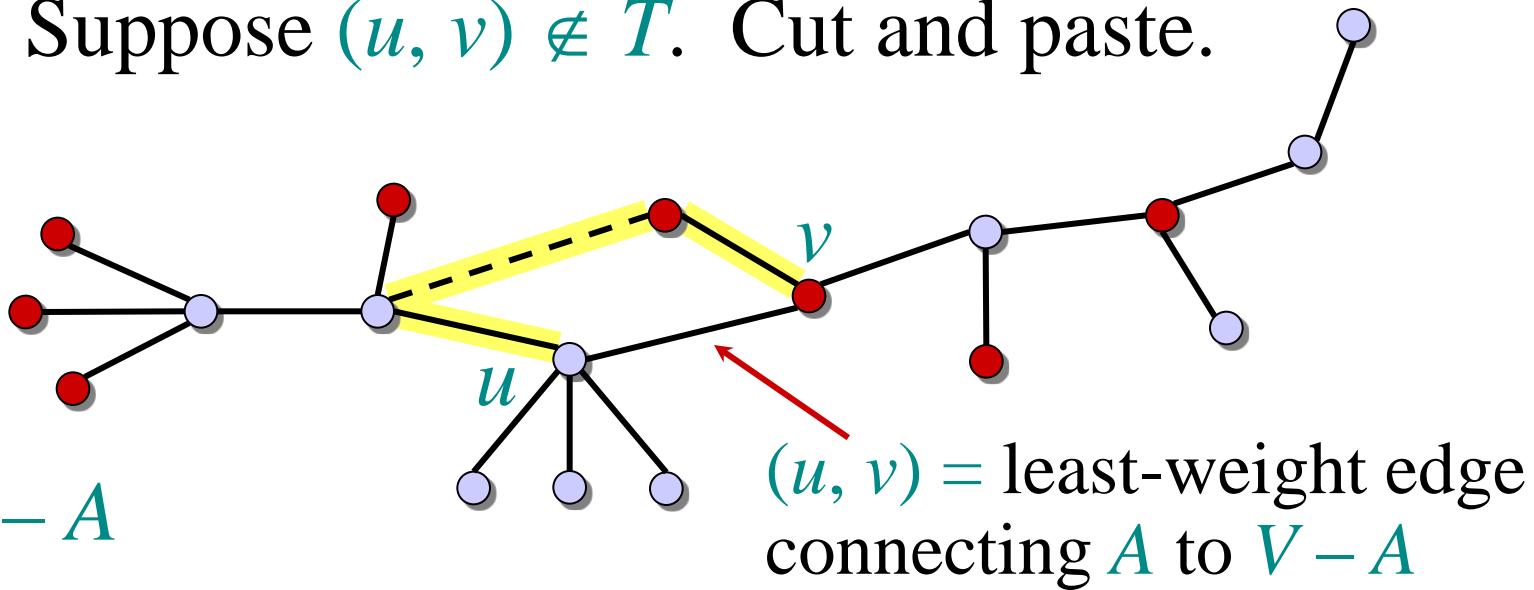


# Proof of theorem

*Proof.* Suppose  $(u, v) \notin T$ . Cut and paste.

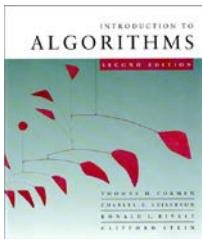
$T$ :

- $\in A$
- $\in V - A$



Consider the unique simple path from  $u$  to  $v$  in  $T$ .

Swap  $(u, v)$  with the first edge on this path that connects a vertex in  $A$  to a vertex in  $V - A$ .

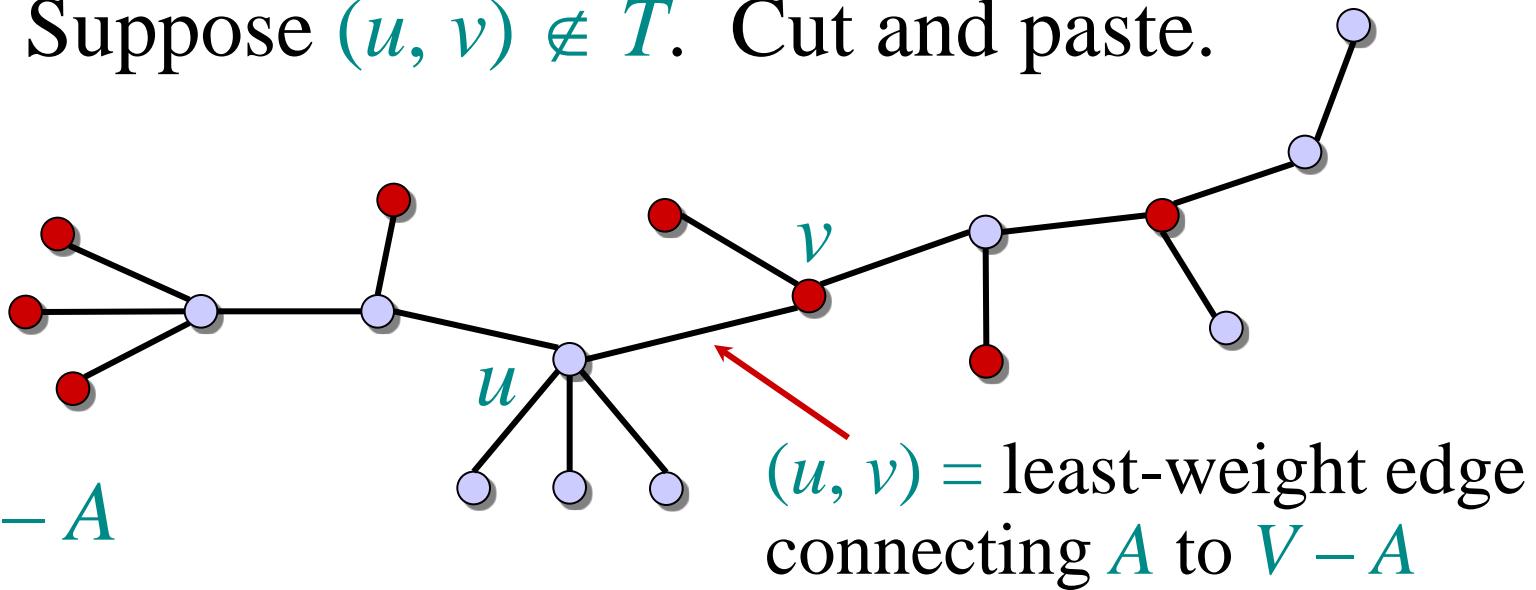


# Proof of theorem

*Proof.* Suppose  $(u, v) \notin T$ . Cut and paste.

$T'$ :

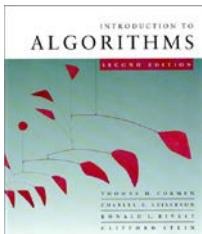
- $\in A$
- $\in V - A$



Consider the unique simple path from  $u$  to  $v$  in  $T$ .

Swap  $(u, v)$  with the first edge on this path that connects a vertex in  $A$  to a vertex in  $V - A$ .

A lighter-weight spanning tree than  $T$  results. □



# Prim's algorithm

**IDEA:** Maintain  $V - A$  as a priority queue  $Q$ . Key each vertex in  $Q$  with the weight of the least-weight edge connecting it to a vertex in  $A$ .

$Q \leftarrow V$

$key[v] \leftarrow \infty$  for all  $v \in V$

$key[s] \leftarrow 0$  for some arbitrary  $s \in V$

**while**  $Q \neq \emptyset$

**do**  $u \leftarrow \text{EXTRACT-MIN}(Q)$

**for** each  $v \in \text{Adj}[u]$

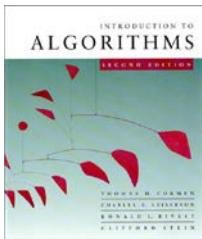
**do if**  $v \in Q$  and  $w(u, v) < key[v]$

**then**  $key[v] \leftarrow w(u, v)$

                ▶ DECREASE-KEY

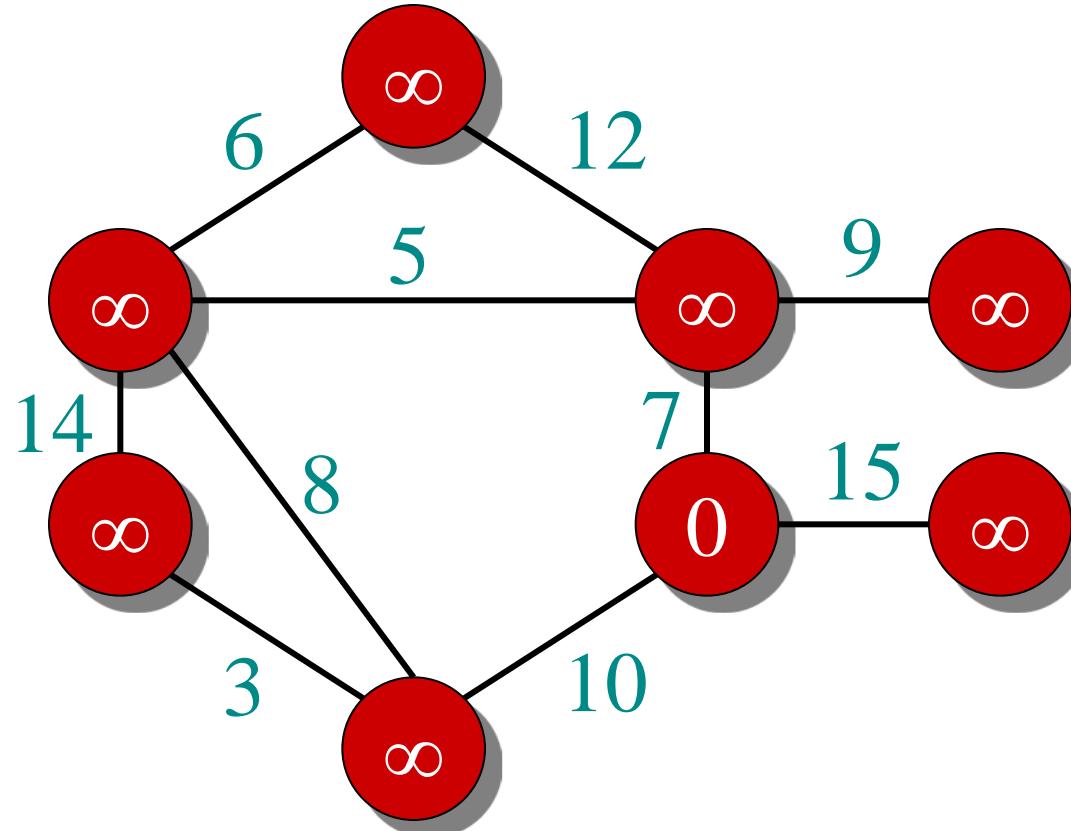
$\pi[v] \leftarrow u$

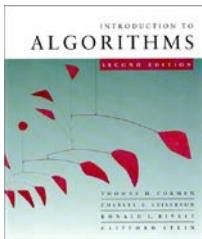
At the end,  $\{(v, \pi[v])\}$  forms the MST.



# Example of Prim's algorithm

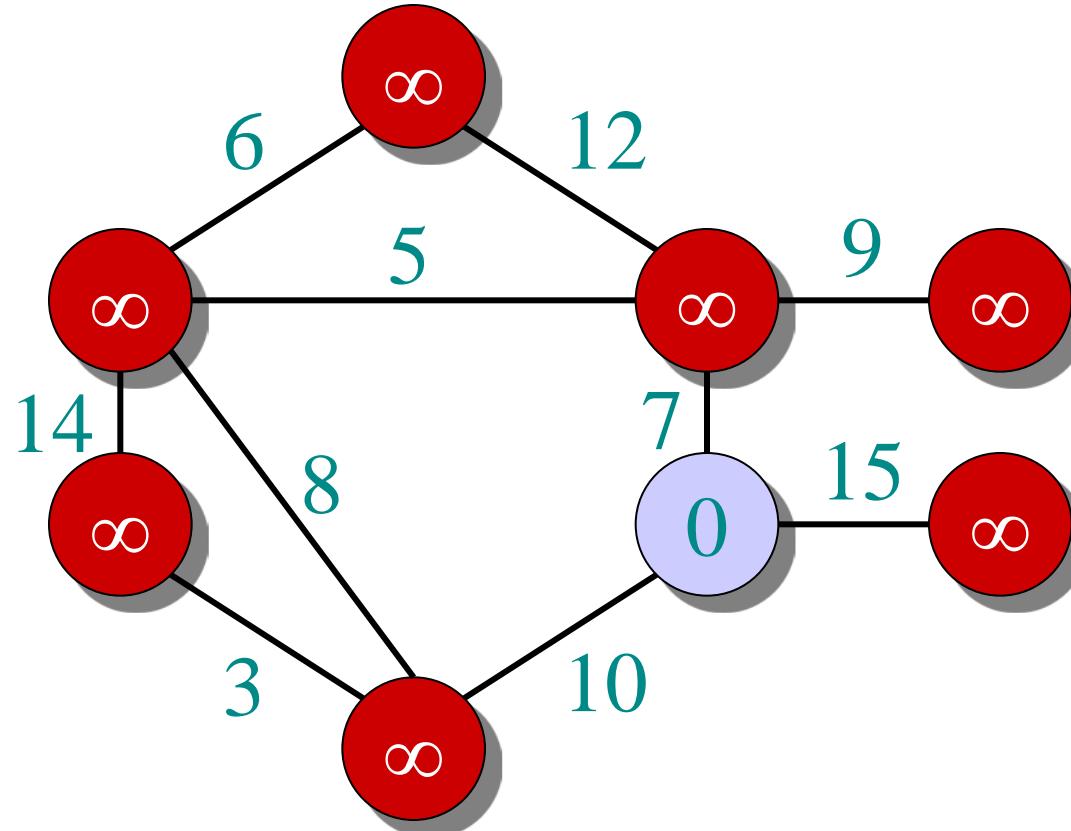
- $\in A$
- $\in V - A$

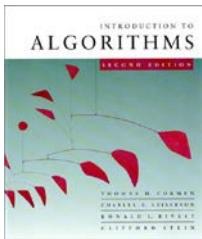




# Example of Prim's algorithm

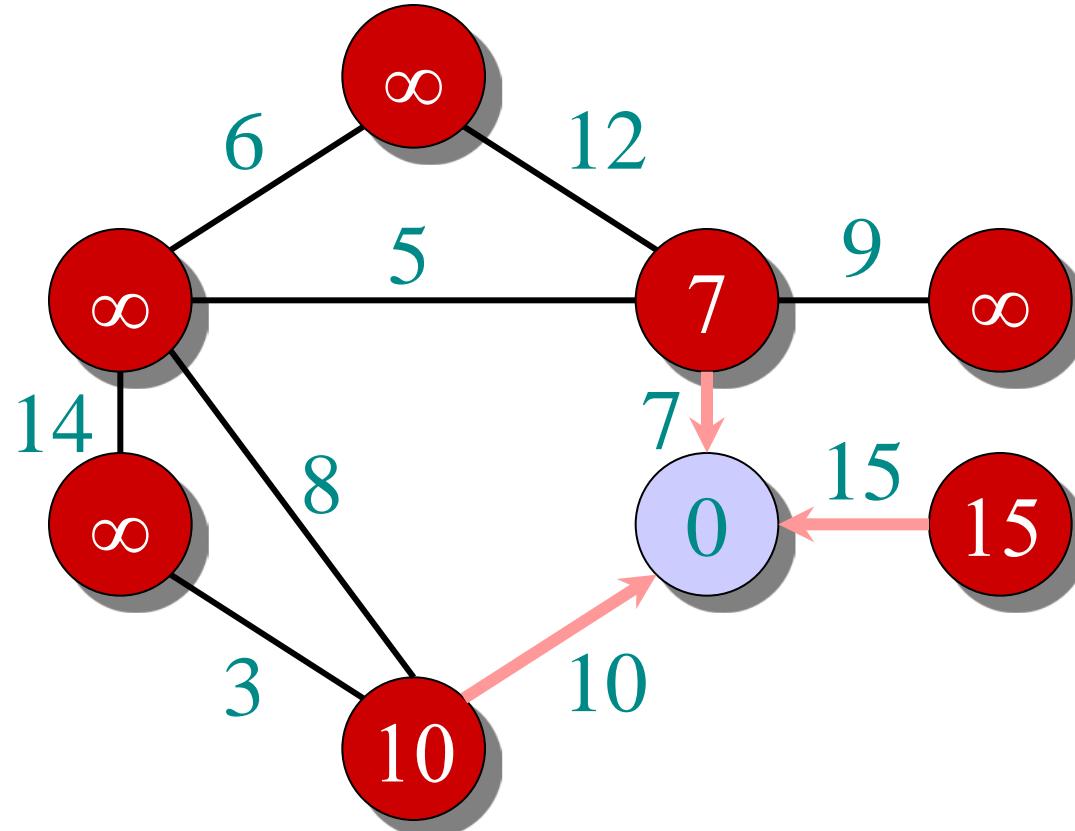
- $\in A$
- $\in V - A$

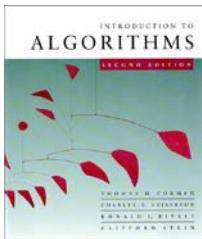




# Example of Prim's algorithm

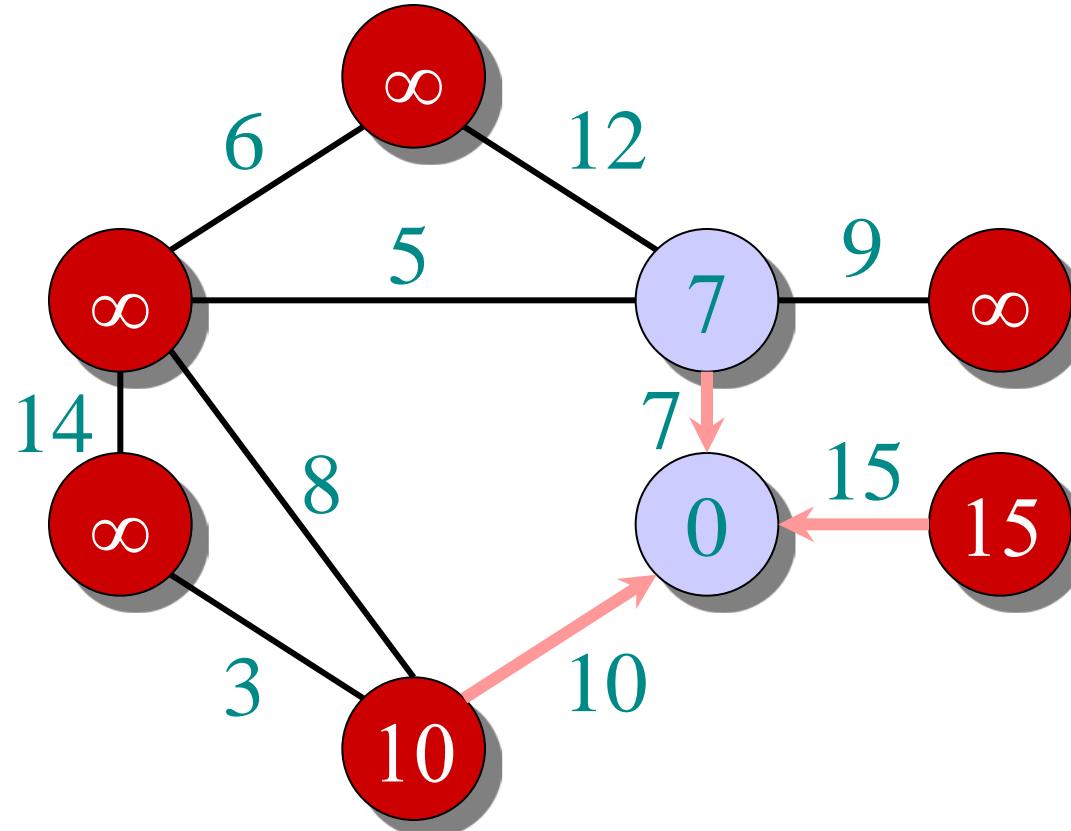
- $\in A$
- $\in V - A$

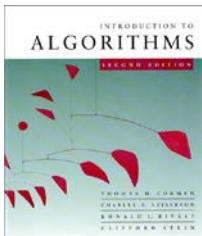




# Example of Prim's algorithm

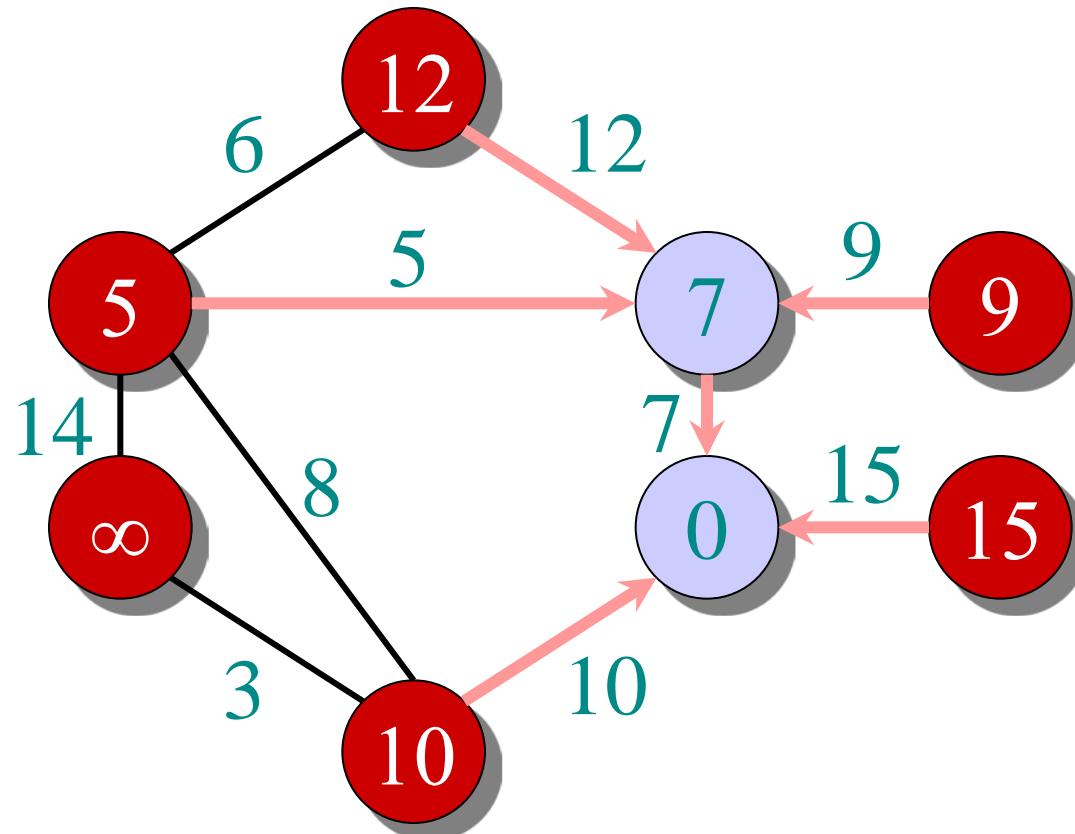
- $\in A$
- $\in V - A$

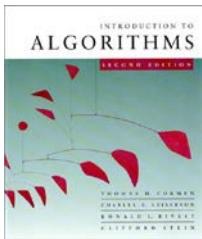




# Example of Prim's algorithm

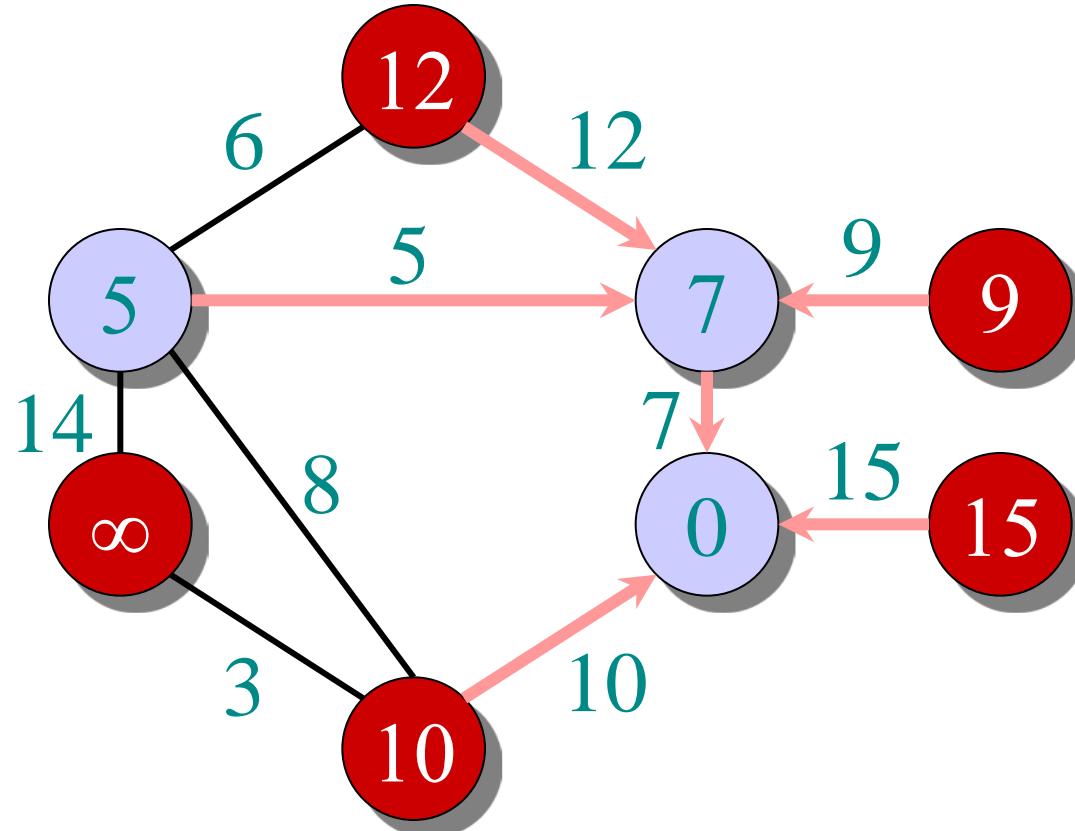
- $\in A$
- $\in V - A$

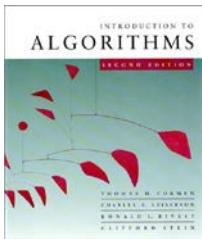




# Example of Prim's algorithm

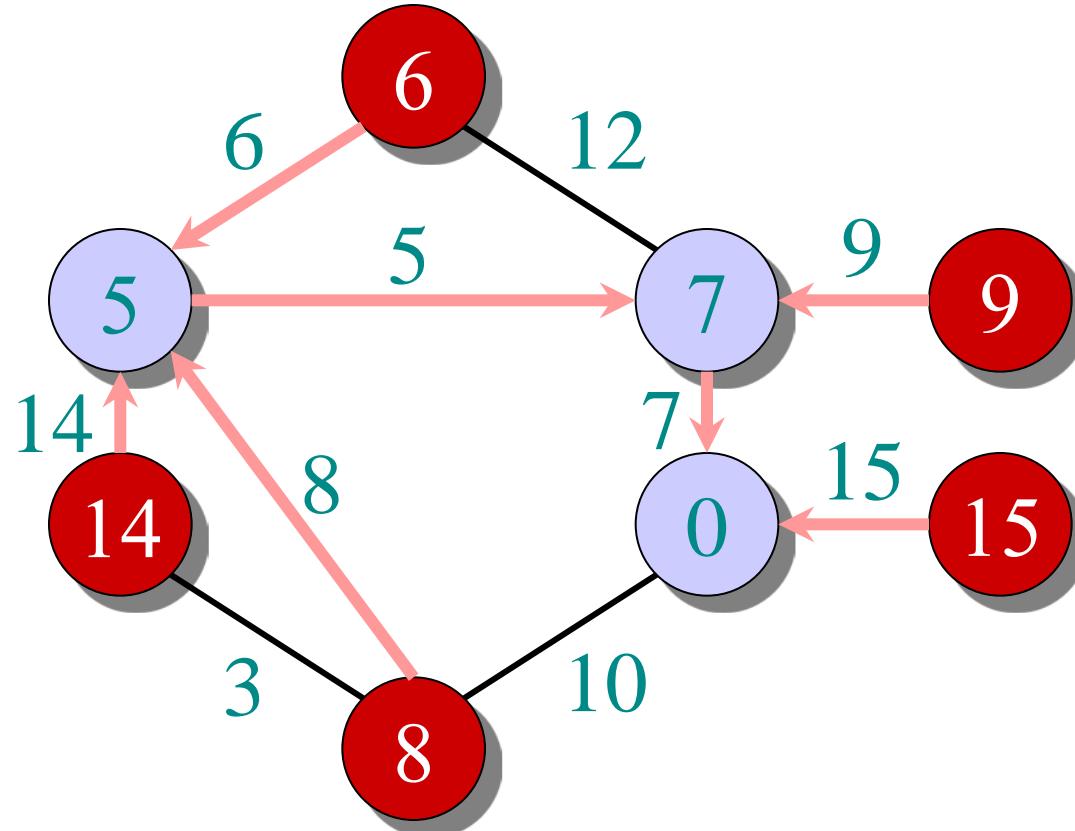
- $\in A$
- $\in V - A$

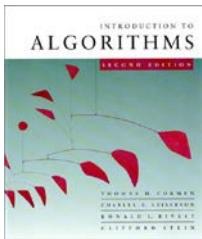




# Example of Prim's algorithm

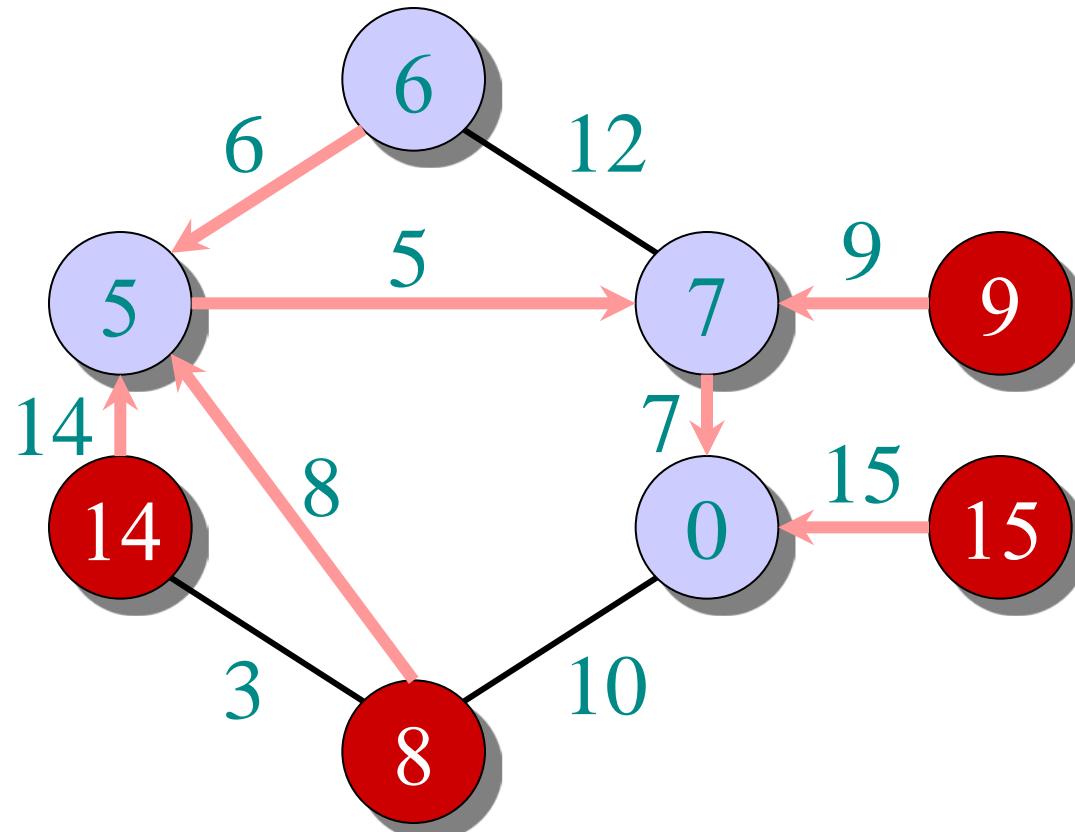
- $\in A$
- $\in V - A$

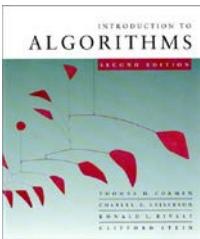




# Example of Prim's algorithm

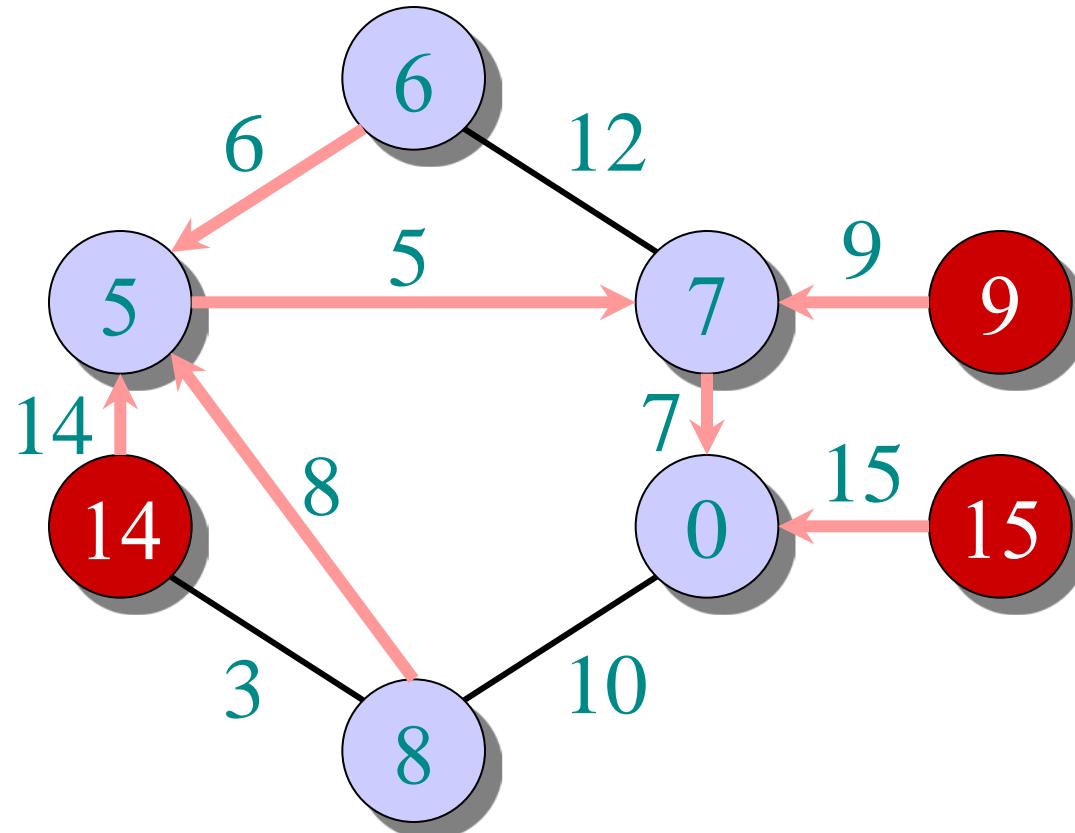
- $\in A$
- $\in V - A$

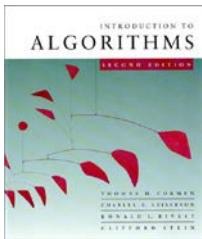




# Example of Prim's algorithm

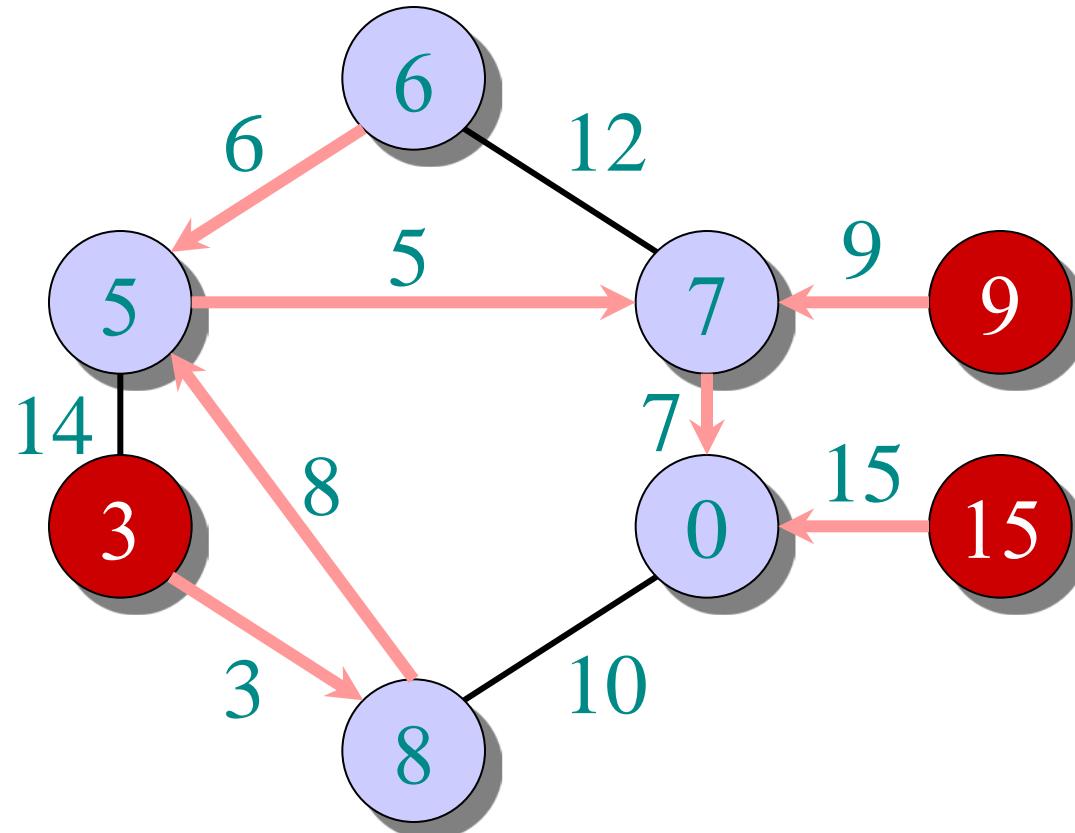
- $\in A$
- $\in V - A$

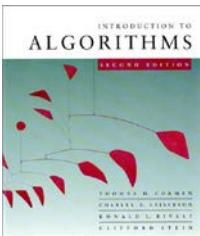




# Example of Prim's algorithm

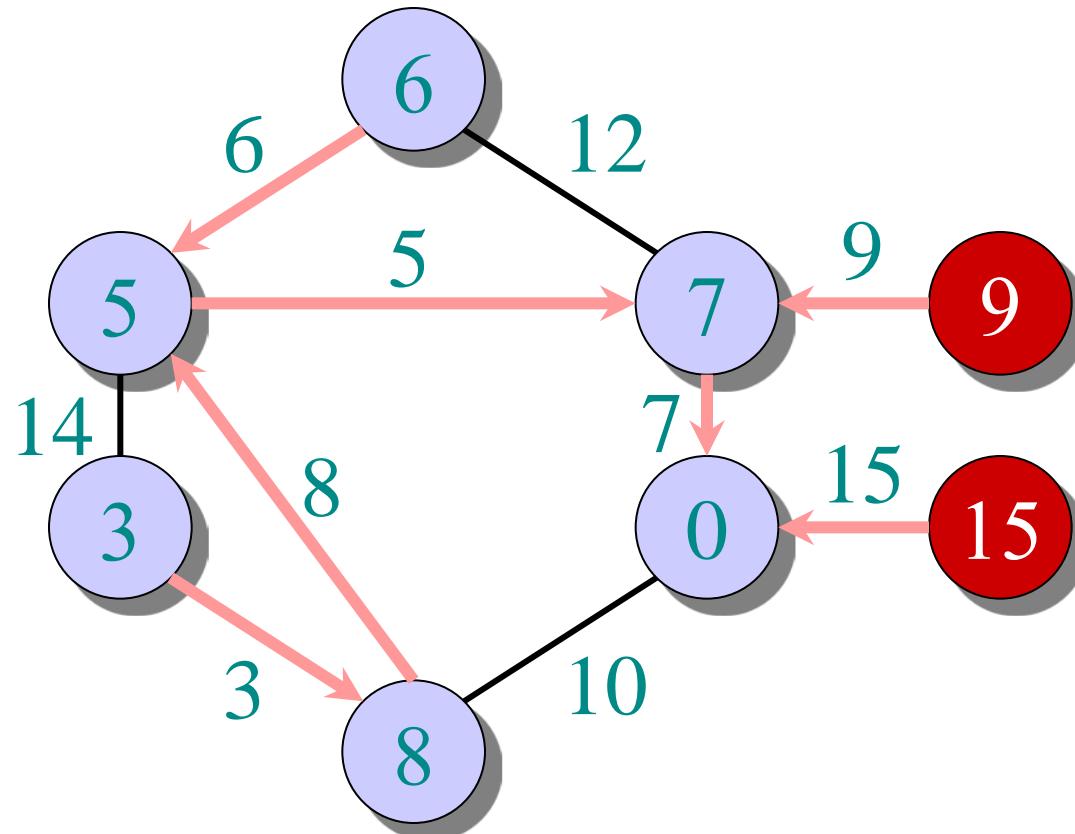
- $\in A$
- $\in V - A$

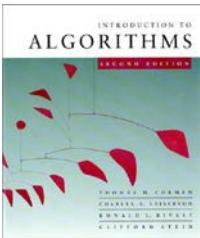




# Example of Prim's algorithm

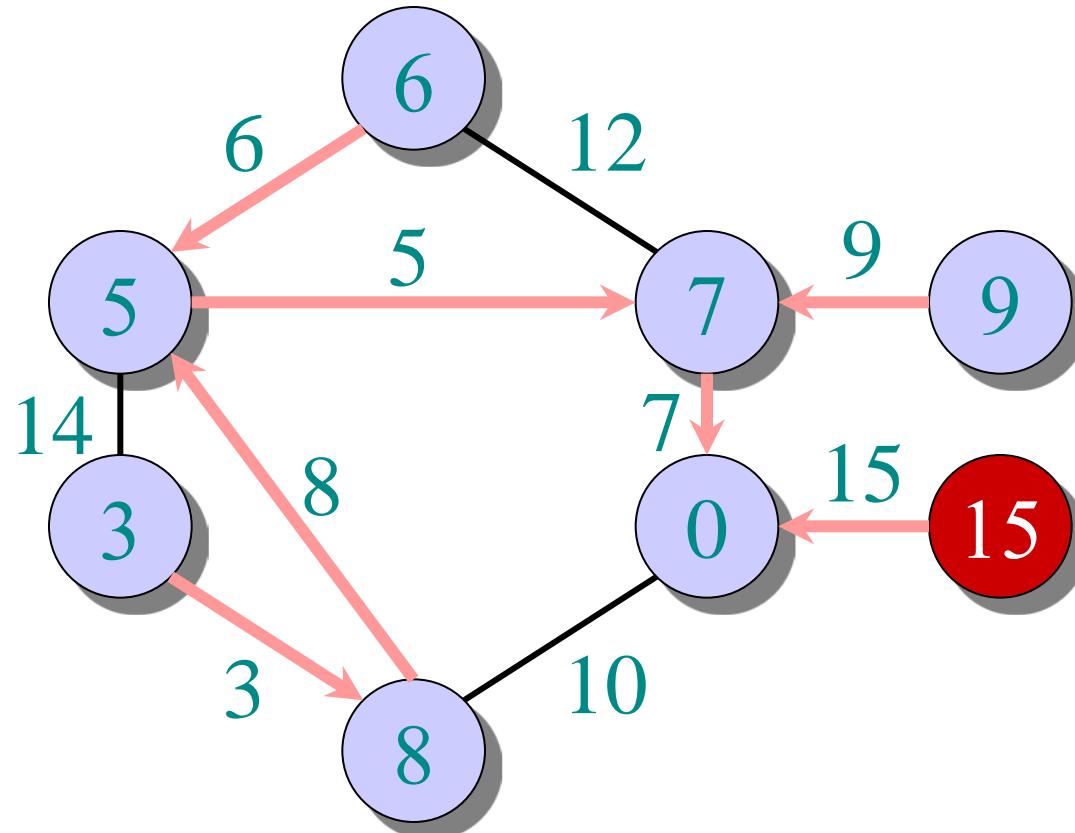
- $\in A$
- $\in V - A$

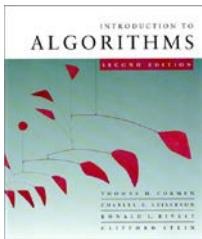




# Example of Prim's algorithm

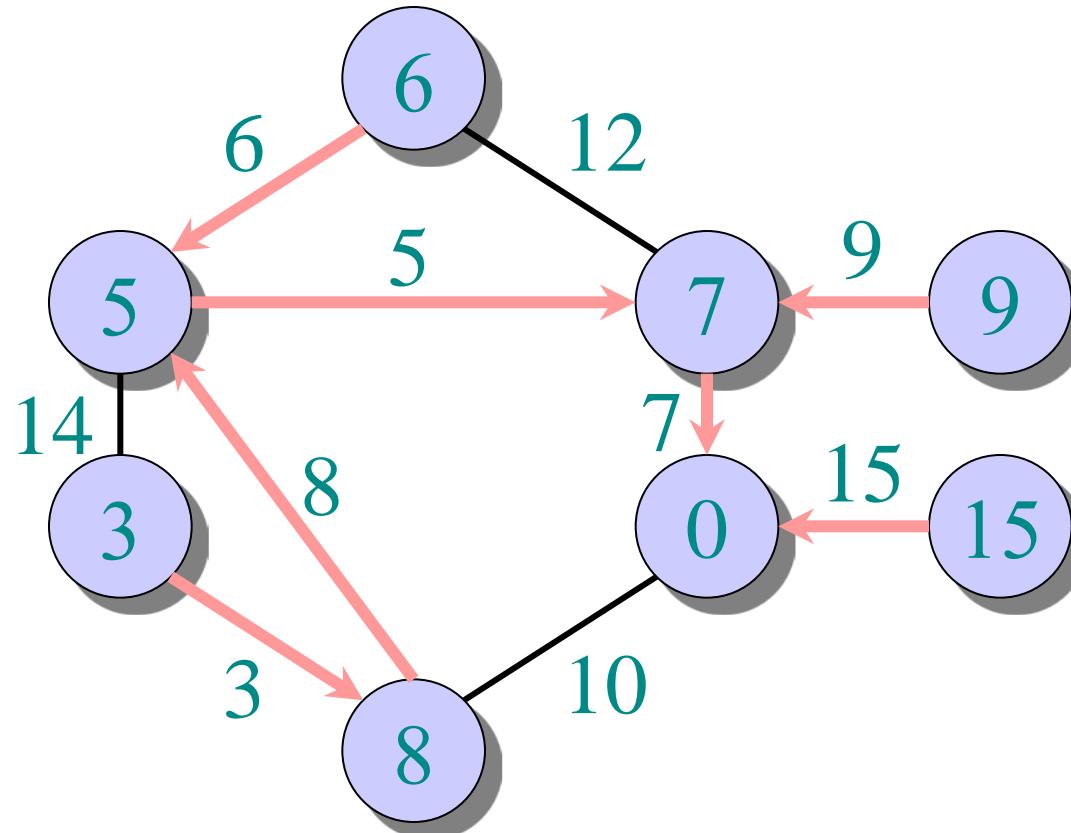
- $\in A$
- $\in V - A$

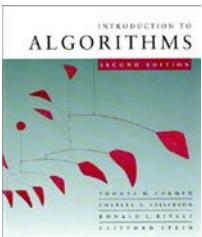




# Example of Prim's algorithm

- $\in A$
- $\in V - A$





# Analysis of Prim

$Q \leftarrow V$

$key[v] \leftarrow \infty$  for all  $v \in V$

$key[s] \leftarrow 0$  for some arbitrary  $s \in V$

**while**  $Q \neq \emptyset$

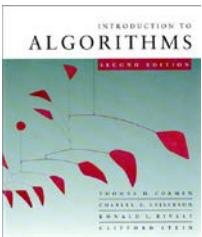
**do**  $u \leftarrow \text{EXTRACT-MIN}(Q)$

**for** each  $v \in Adj[u]$

**do if**  $v \in Q$  and  $w(u, v) < key[v]$

**then**  $key[v] \leftarrow w(u, v)$

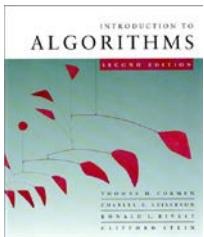
$\pi[v] \leftarrow u$



# Analysis of Prim

$\Theta(V)$  total

$$\left\{ \begin{array}{l} Q \leftarrow V \\ key[v] \leftarrow \infty \text{ for all } v \in V \\ key[s] \leftarrow 0 \text{ for some arbitrary } s \in V \\ \textbf{while } Q \neq \emptyset \\ \quad \textbf{do } u \leftarrow \text{EXTRACT-MIN}(Q) \\ \quad \textbf{for each } v \in Adj[u] \\ \quad \quad \textbf{do if } v \in Q \text{ and } w(u, v) < key[v] \\ \quad \quad \quad \textbf{then } key[v] \leftarrow w(u, v) \\ \quad \quad \quad \pi[v] \leftarrow u \end{array} \right.$$



# Analysis of Prim

$\Theta(V)$  total {

$Q \leftarrow V$

$key[v] \leftarrow \infty$  for all  $v \in V$

$key[s] \leftarrow 0$  for some arbitrary  $s \in V$

**while**  $Q \neq \emptyset$

**do**  $u \leftarrow \text{EXTRACT-MIN}(Q)$

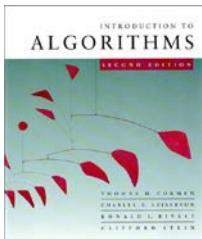
**for each**  $v \in \text{Adj}[u]$

**do if**  $v \in Q$  and  $w(u, v) < key[v]$

**then**  $key[v] \leftarrow w(u, v)$

$\pi[v] \leftarrow u$

$|V|$  times {



# Analysis of Prim

$\Theta(V)$  total {

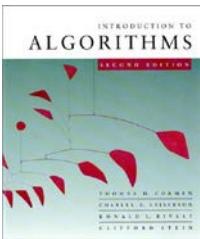
- $Q \leftarrow V$
- $key[v] \leftarrow \infty$  for all  $v \in V$
- $key[s] \leftarrow 0$  for some arbitrary  $s \in V$

**while**  $Q \neq \emptyset$

- do**  $u \leftarrow \text{EXTRACT-MIN}(Q)$
- for each**  $v \in \text{Adj}[u]$
- do if**  $v \in Q$  and  $w(u, v) < key[v]$
- then**  $key[v] \leftarrow w(u, v)$
- $\pi[v] \leftarrow u$

$|V|$  times {

*degree( $u$ )* times {



# Analysis of Prim

$\Theta(V)$  total {

- $Q \leftarrow V$
- $key[v] \leftarrow \infty$  for all  $v \in V$
- $key[s] \leftarrow 0$  for some arbitrary  $s \in V$

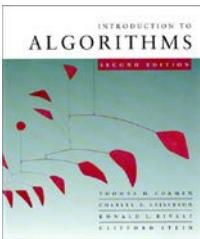
**while**  $Q \neq \emptyset$

- do**  $u \leftarrow \text{EXTRACT-MIN}(Q)$
- for each**  $v \in \text{Adj}[u]$
- do if**  $v \in Q$  and  $w(u, v) < key[v]$
- then**  $key[v] \leftarrow w(u, v)$

$|V|$  times {

- $degree(u)$  times {

Handshaking Lemma  $\Rightarrow \Theta(E)$  implicit DECREASE-KEY's.

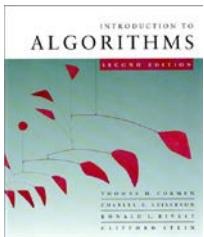


# Analysis of Prim

$\Theta(V)$  total {  
 $|V|$  times {  
   $Q \leftarrow V$   
   $key[v] \leftarrow \infty$  for all  $v \in V$   
   $key[s] \leftarrow 0$  for some arbitrary  $s \in V$   
  **while**  $Q \neq \emptyset$   
    **do**  $u \leftarrow \text{EXTRACT-MIN}(Q)$   
    **for each**  $v \in \text{Adj}[u]$   
      **do if**  $v \in Q$  and  $w(u, v) < key[v]$   
      **then**  $key[v] \leftarrow w(u, v)$   
           $\pi[v] \leftarrow u$      ↑  
  }

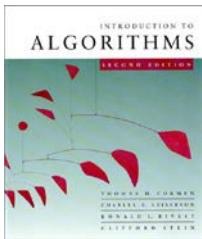
Handshaking Lemma  $\Rightarrow \Theta(E)$  implicit DECREASE-KEY's.

Time =  $\Theta(V) \cdot T_{\text{EXTRACT-MIN}} + \Theta(E) \cdot T_{\text{DECREASE-KEY}}$



# Analysis of Prim (continued)

Time =  $\Theta(V) \cdot T_{\text{EXTRACT-MIN}} + \Theta(E) \cdot T_{\text{DECREASE-KEY}}$

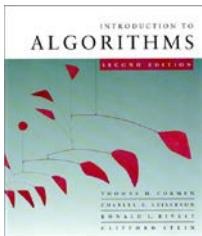


# Analysis of Prim (continued)

Time =  $\Theta(V) \cdot T_{\text{EXTRACT-MIN}} + \Theta(E) \cdot T_{\text{DECREASE-KEY}}$

| $Q$ | $T_{\text{EXTRACT-MIN}}$ | $T_{\text{DECREASE-KEY}}$ | Total |
|-----|--------------------------|---------------------------|-------|
|-----|--------------------------|---------------------------|-------|

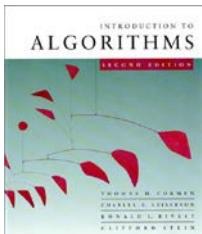
---



# Analysis of Prim (continued)

$$\text{Time} = \Theta(V) \cdot T_{\text{EXTRACT-MIN}} + \Theta(E) \cdot T_{\text{DECREASE-KEY}}$$

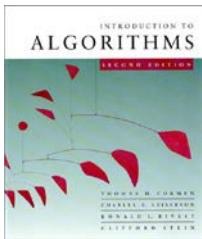
| $Q$   | $T_{\text{EXTRACT-MIN}}$ | $T_{\text{DECREASE-KEY}}$ | Total    |
|-------|--------------------------|---------------------------|----------|
| array | $O(V)$                   | $O(1)$                    | $O(V^2)$ |



# Analysis of Prim (continued)

Time =  $\Theta(V) \cdot T_{\text{EXTRACT-MIN}} + \Theta(E) \cdot T_{\text{DECREASE-KEY}}$

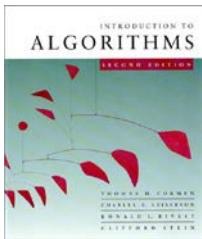
| $Q$         | $T_{\text{EXTRACT-MIN}}$ | $T_{\text{DECREASE-KEY}}$ | Total        |
|-------------|--------------------------|---------------------------|--------------|
| array       | $O(V)$                   | $O(1)$                    | $O(V^2)$     |
| binary heap | $O(\lg V)$               | $O(\lg V)$                | $O(E \lg V)$ |



# Analysis of Prim (continued)

$$\text{Time} = \Theta(V) \cdot T_{\text{EXTRACT-MIN}} + \Theta(E) \cdot T_{\text{DECREASE-KEY}}$$

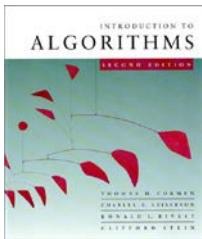
| $Q$            | $T_{\text{EXTRACT-MIN}}$ | $T_{\text{DECREASE-KEY}}$ | Total                          |
|----------------|--------------------------|---------------------------|--------------------------------|
| array          | $O(V)$                   | $O(1)$                    | $O(V^2)$                       |
| binary heap    | $O(\lg V)$               | $O(\lg V)$                | $O(E \lg V)$                   |
| Fibonacci heap | $O(\lg V)$<br>amortized  | $O(1)$<br>amortized       | $O(E + V \lg V)$<br>worst case |



# MST algorithms

Kruskal's algorithm (see CLRS):

- Uses the *disjoint-set data structure* (see CLRS, Ch. 21).
- Running time =  $O(E \lg V)$ .



# MST algorithms

Kruskal's algorithm (see CLRS):

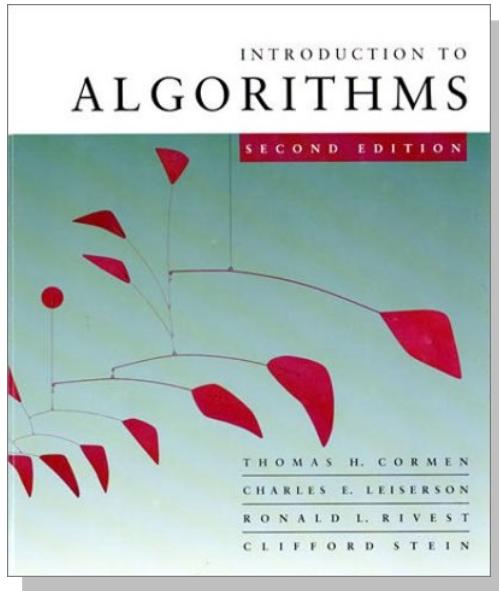
- Uses the *disjoint-set data structure* (see CLRS, Ch. 21).
- Running time =  $O(E \lg V)$ .

Best to date:

- Karger, Klein, and Tarjan [1993].
- Randomized algorithm.
- $O(V + E)$  expected time.

# *Introduction to Algorithms*

## 6.046J/18.401J

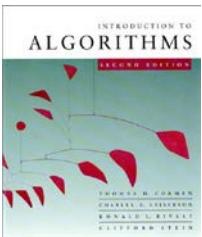


### LECTURE 17

#### Shortest Paths I

- Properties of shortest paths
- Dijkstra's algorithm
- Correctness
- Analysis
- Breadth-first search

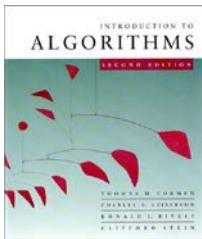
Prof. Erik Demaine



# Paths in graphs

Consider a digraph  $G = (V, E)$  with edge-weight function  $w : E \rightarrow \mathbb{R}$ . The **weight** of path  $p = v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_k$  is defined to be

$$w(p) = \sum_{i=1}^{k-1} w(v_i, v_{i+1}).$$

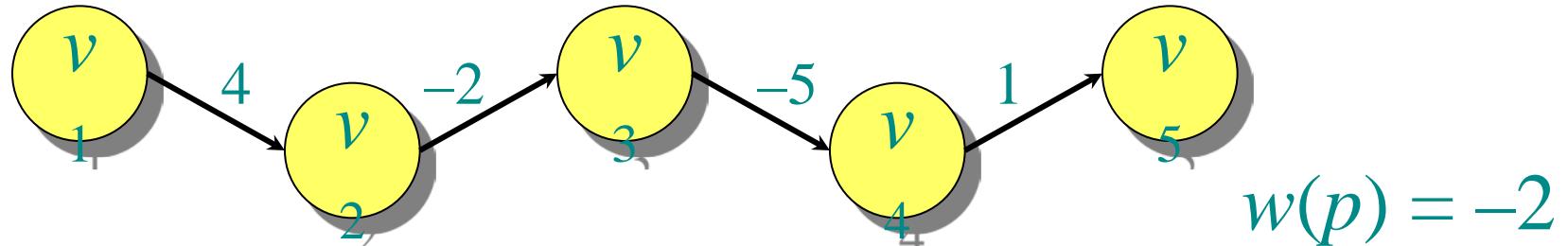


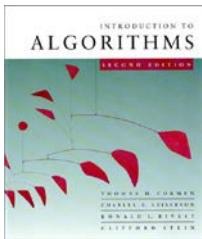
# Paths in graphs

Consider a digraph  $G = (V, E)$  with edge-weight function  $w : E \rightarrow \mathbb{R}$ . The **weight** of path  $p = v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_k$  is defined to be

$$w(p) = \sum_{i=1}^{k-1} w(v_i, v_{i+1}).$$

**Example:**



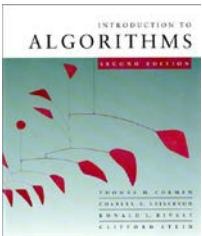


# Shortest paths

A *shortest path* from  $u$  to  $v$  is a path of minimum weight from  $u$  to  $v$ . The *shortest-path weight* from  $u$  to  $v$  is defined as

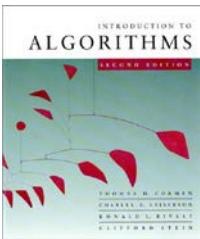
$$\delta(u, v) = \min\{w(p) : p \text{ is a path from } u \text{ to } v\}.$$

**Note:**  $\delta(u, v) = \infty$  if no path from  $u$  to  $v$  exists.



# Well-definedness of shortest paths

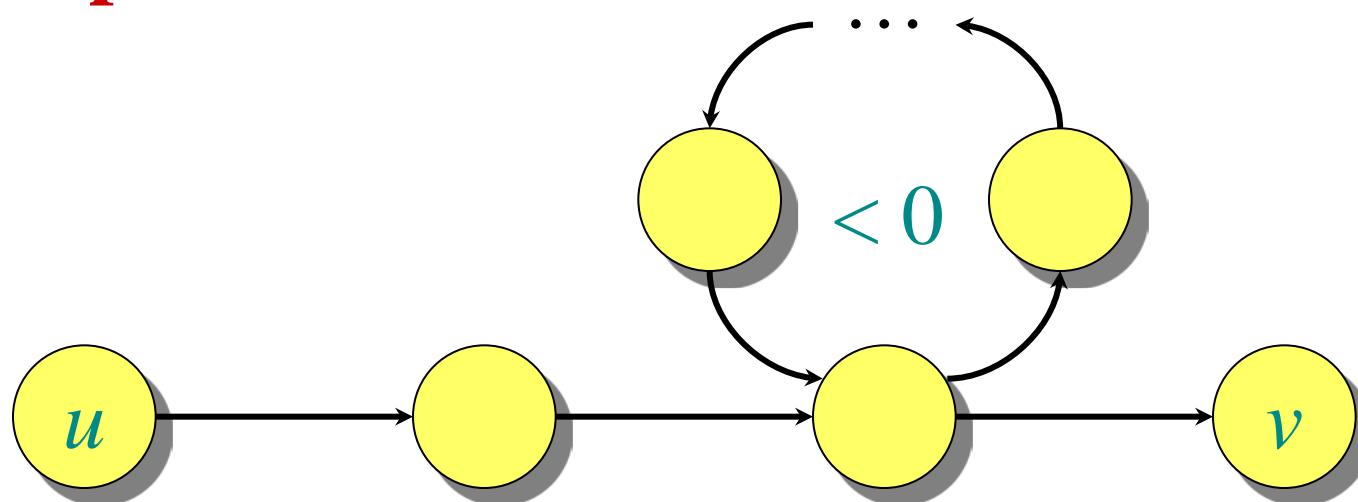
If a graph  $G$  contains a negative-weight cycle, then some shortest paths do not exist.

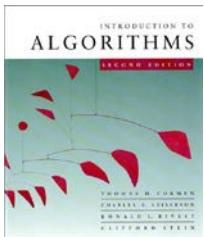


# Well-definedness of shortest paths

If a graph  $G$  contains a negative-weight cycle, then some shortest paths do not exist.

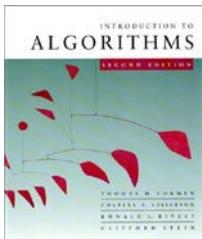
**Example:**





# Optimal substructure

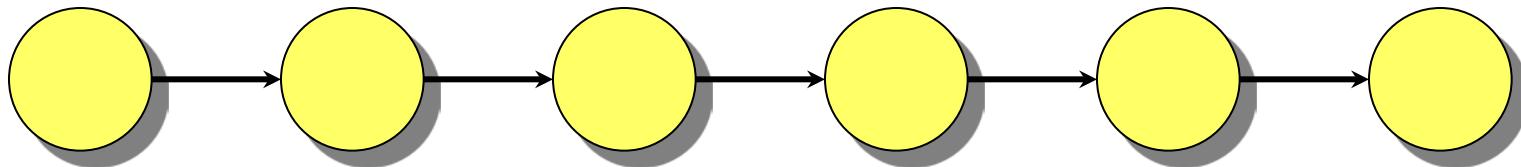
**Theorem.** A subpath of a shortest path is a shortest path.

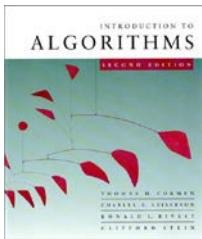


# Optimal substructure

**Theorem.** A subpath of a shortest path is a shortest path.

*Proof.* Cut and paste:

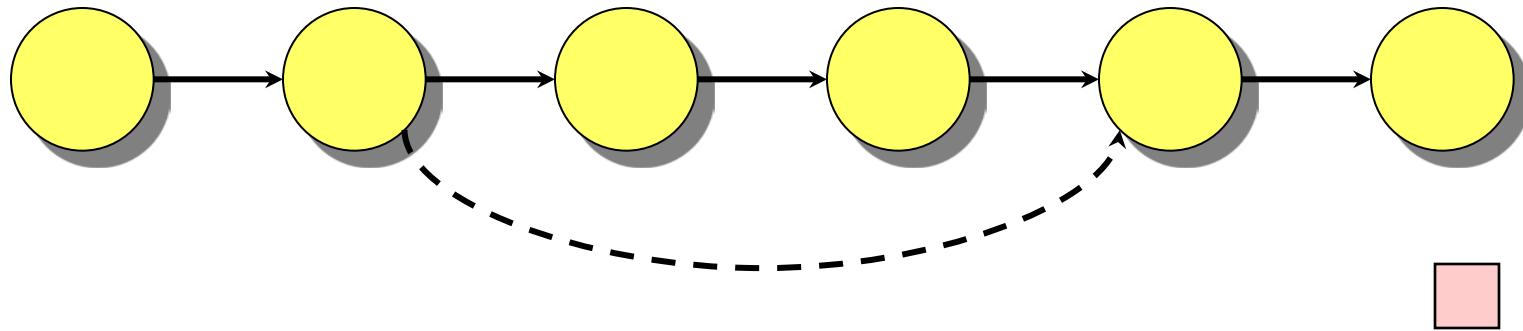


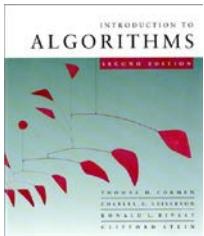


# Optimal substructure

**Theorem.** A subpath of a shortest path is a shortest path.

*Proof.* Cut and paste:

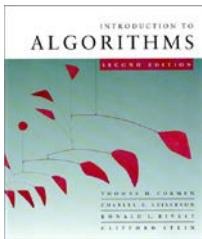




# Triangle inequality

**Theorem.** For all  $u, v, x \in V$ , we have

$$\delta(u, v) \leq \delta(u, x) + \delta(x, v).$$

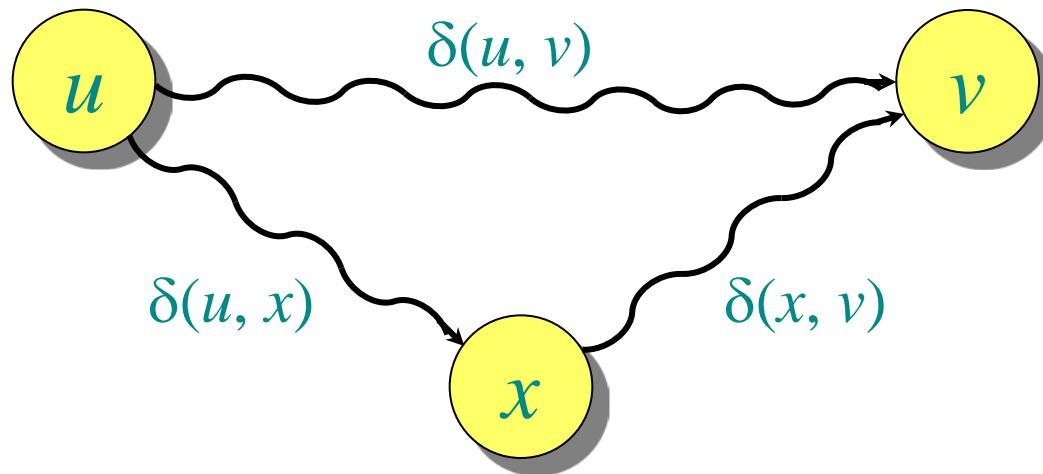


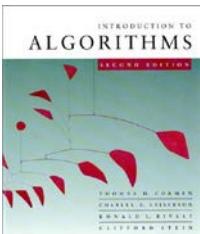
# Triangle inequality

**Theorem.** For all  $u, v, x \in V$ , we have

$$\delta(u, v) \leq \delta(u, x) + \delta(x, v).$$

*Proof.*





# Single-source shortest paths (nonnegative edge weights)

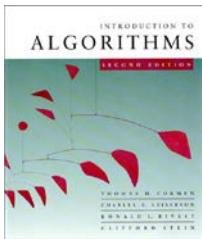
**Problem.** Assume that  $w(u, v) \geq 0$  for all  $(u, v) \in E$ . (Hence, all shortest-path weights must exist.) From a given source vertex  $s \in V$ , find the shortest-path weights  $\delta(s, v)$  for all  $v \in V$ .

---

---

**IDEA:** Greedy.

1. Maintain a set  $S$  of vertices whose shortest-path distances from  $s$  are known.
2. At each step, add to  $S$  the vertex  $v \in V - S$  whose distance estimate from  $s$  is minimum.
3. Update the distance estimates of vertices adjacent to  $v$ .



# Dijkstra's algorithm

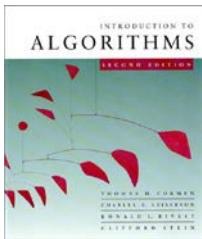
$d[s] \leftarrow 0$

**for** each  $v \in V - \{s\}$

**do**  $d[v] \leftarrow \infty$

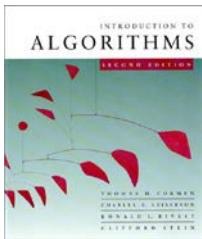
$S \leftarrow \emptyset$

$Q \leftarrow V$       ▷  $Q$  is a priority queue maintaining  $V - S$ ,  
keyed on  $d[v]$



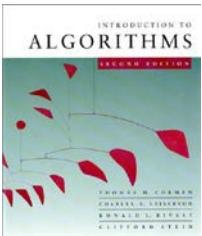
# Dijkstra's algorithm

```
d[s] ← 0
for each  $v \in V - \{s\}$ 
  do  $d[v] \leftarrow \infty$ 
 $S \leftarrow \emptyset$ 
 $Q \leftarrow V$       ▷  $Q$  is a priority queue maintaining  $V - S$ ,
                    keyed on  $d[v]$ 
while  $Q \neq \emptyset$ 
  do  $u \leftarrow \text{EXTRACT-MIN}(Q)$ 
     $S \leftarrow S \cup \{u\}$ 
    for each  $v \in \text{Adj}[u]$ 
      do if  $d[v] > d[u] + w(u, v)$ 
        then  $d[v] \leftarrow d[u] + w(u, v)$ 
```



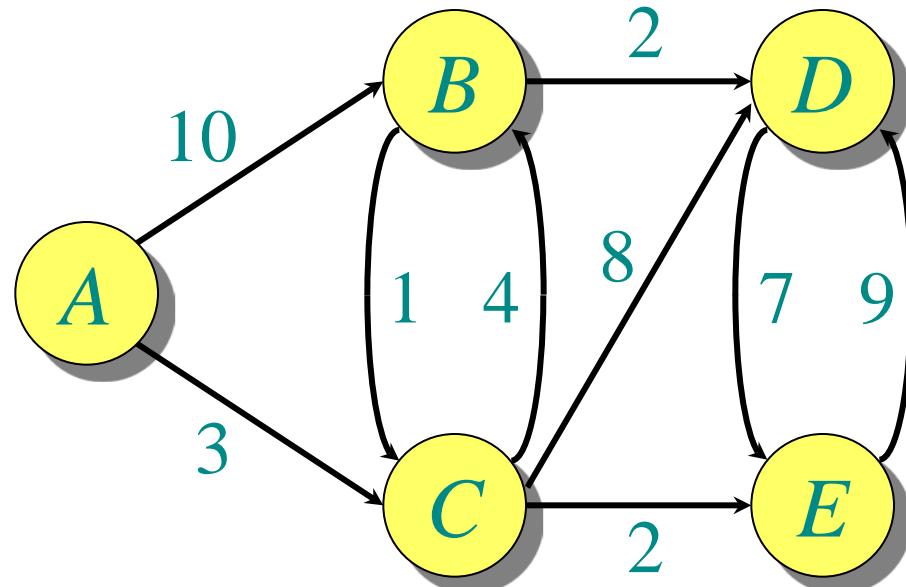
# Dijkstra's algorithm

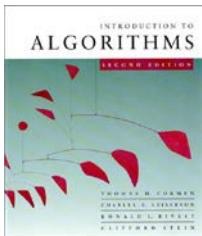
```
d[s] ← 0
for each  $v \in V - \{s\}$ 
  do  $d[v] \leftarrow \infty$ 
 $S \leftarrow \emptyset$ 
 $Q \leftarrow V$       ▷  $Q$  is a priority queue maintaining  $V - S$ ,
                    keyed on  $d[v]$ 
while  $Q \neq \emptyset$ 
  do  $u \leftarrow \text{EXTRACT-MIN}(Q)$ 
     $S \leftarrow S \cup \{u\}$ 
    for each  $v \in \text{Adj}[u]$ 
      do if  $d[v] > d[u] + w(u, v)$ 
        then  $d[v] \leftarrow d[u] + w(u, v)$       relaxation step
          ↗ Implicit DECREASE-KEY
```



# Example of Dijkstra's algorithm

Graph with  
nonnegative  
edge weights:

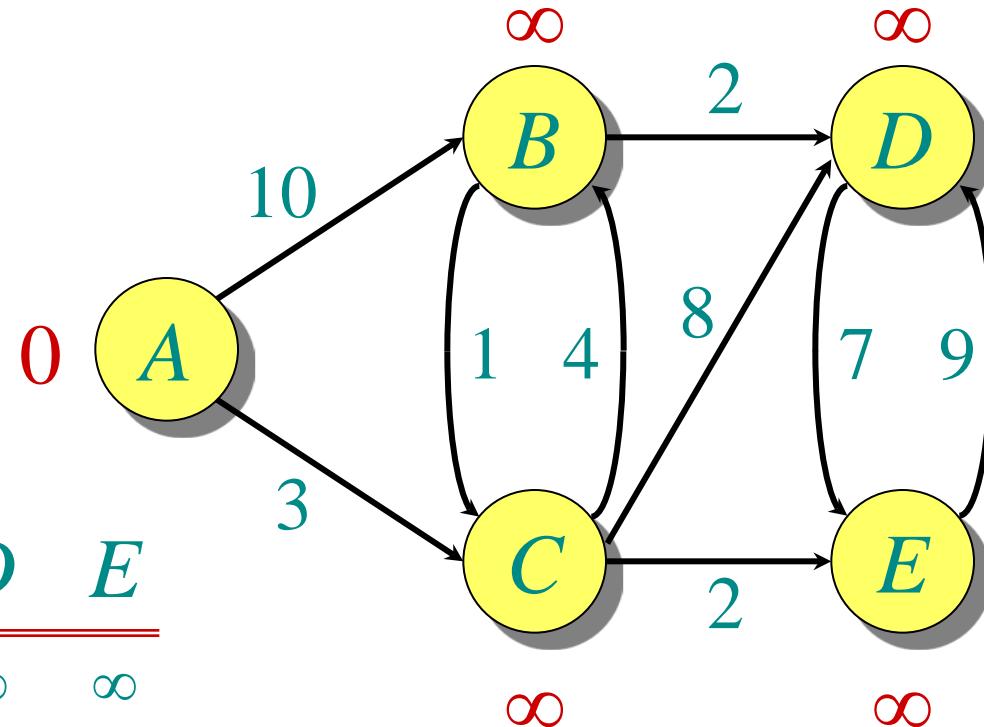




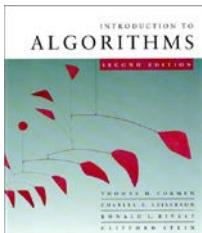
# Example of Dijkstra's algorithm

Initialize:

|      |     |          |          |          |          |
|------|-----|----------|----------|----------|----------|
| $Q:$ | $A$ | $B$      | $C$      | $D$      | $E$      |
|      | 0   | $\infty$ | $\infty$ | $\infty$ | $\infty$ |

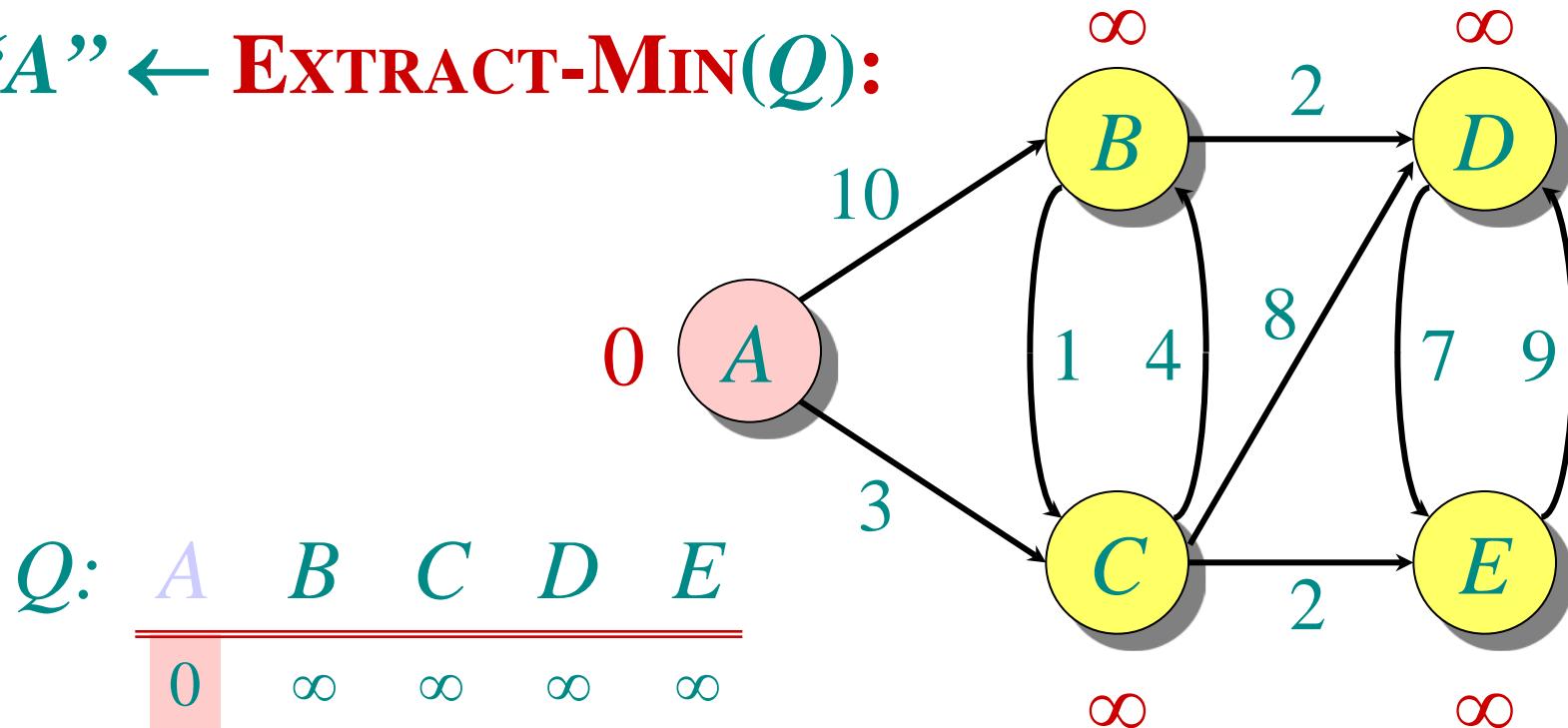


$S: \{\}$

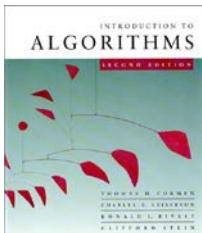


# Example of Dijkstra's algorithm

“A”  $\leftarrow \text{EXTRACT-MIN}(Q)$ :

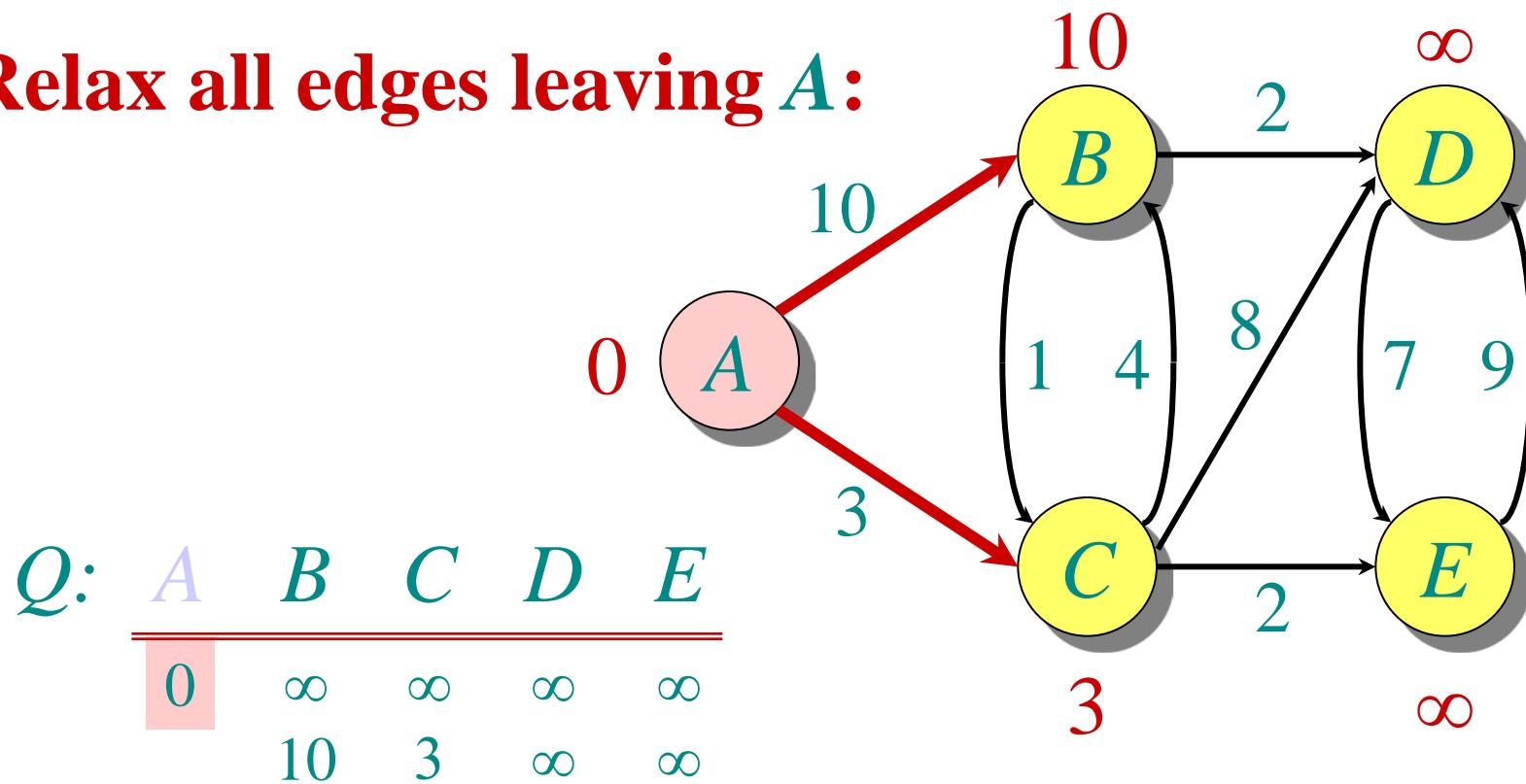


$S: \{ A \}$

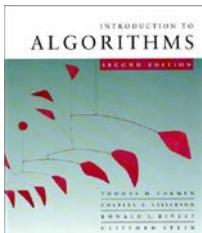


# Example of Dijkstra's algorithm

Relax all edges leaving  $A$ :



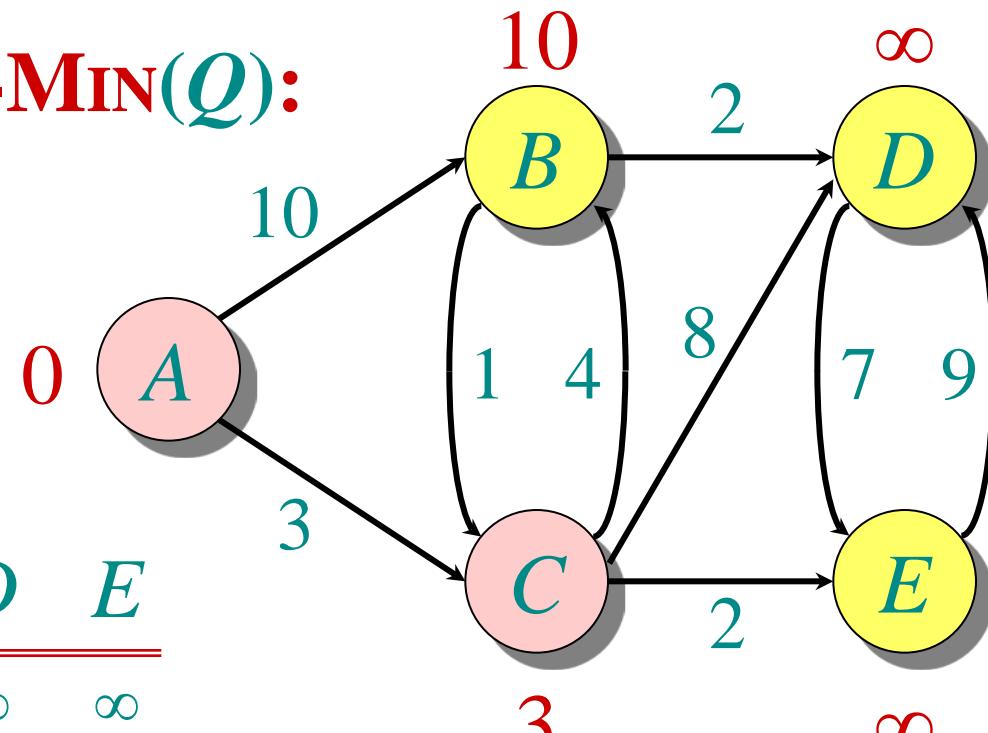
$S: \{ A \}$



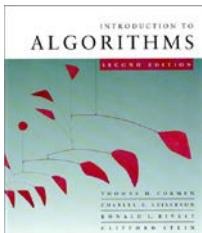
# Example of Dijkstra's algorithm

“C”  $\leftarrow \text{EXTRACT-MIN}(Q)$ :

| $A$ | $B$      | $C$      | $D$      | $E$      |
|-----|----------|----------|----------|----------|
| 0   | $\infty$ | $\infty$ | $\infty$ | $\infty$ |



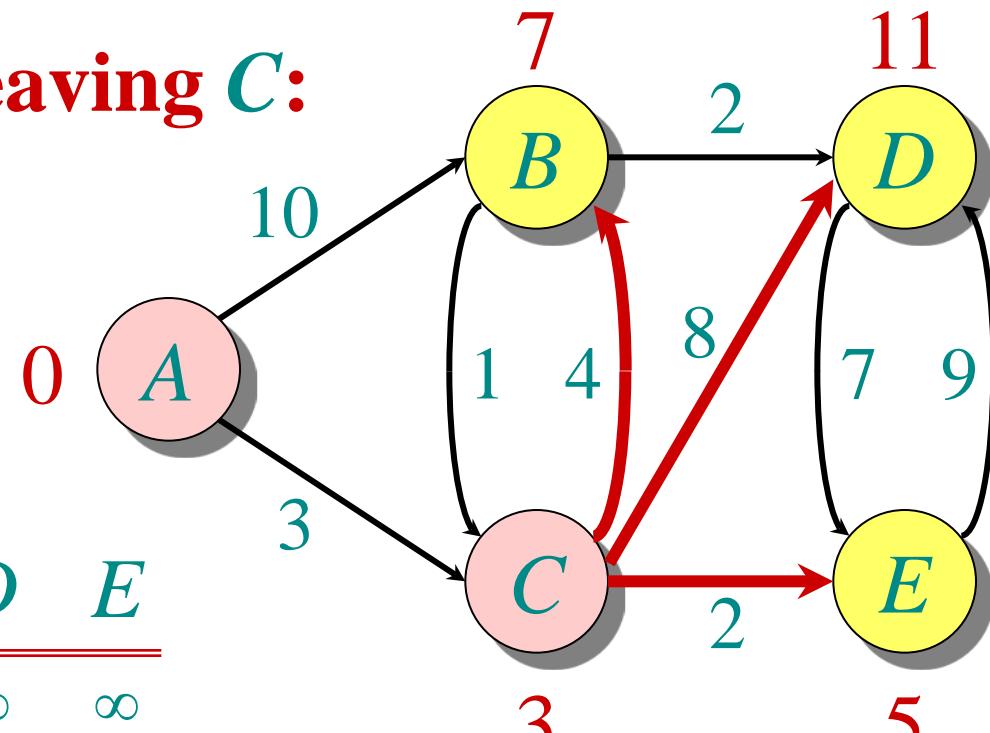
$S: \{ A, C \}$



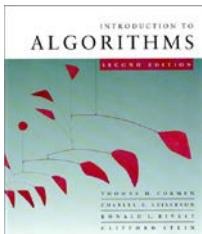
# Example of Dijkstra's algorithm

Relax all edges leaving  $C$ :

| $Q:$ | $A$ | $B$      | $C$      | $D$      | $E$      |
|------|-----|----------|----------|----------|----------|
|      | 0   | $\infty$ | $\infty$ | $\infty$ | $\infty$ |
|      | 10  |          | 3        | $\infty$ | $\infty$ |
|      | 7   |          | 11       | 5        |          |



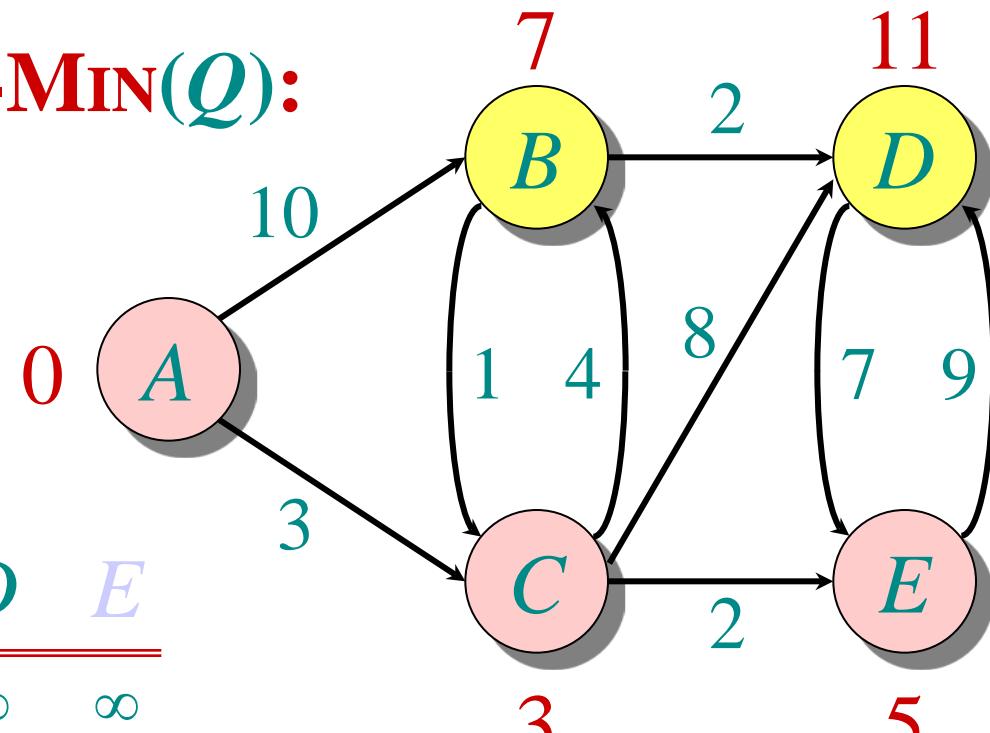
$S: \{ A, C \}$



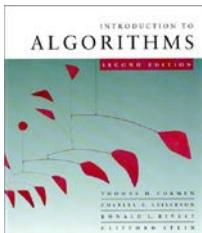
# Example of Dijkstra's algorithm

“E”  $\leftarrow$  EXTRACT-MIN( $Q$ ):

| $Q:$ | $A$ | $B$      | $C$      | $D$      | $E$      |
|------|-----|----------|----------|----------|----------|
|      | 0   | $\infty$ | $\infty$ | $\infty$ | $\infty$ |
|      | 10  |          | 3        | $\infty$ | $\infty$ |
|      | 7   |          | 11       |          | 5        |

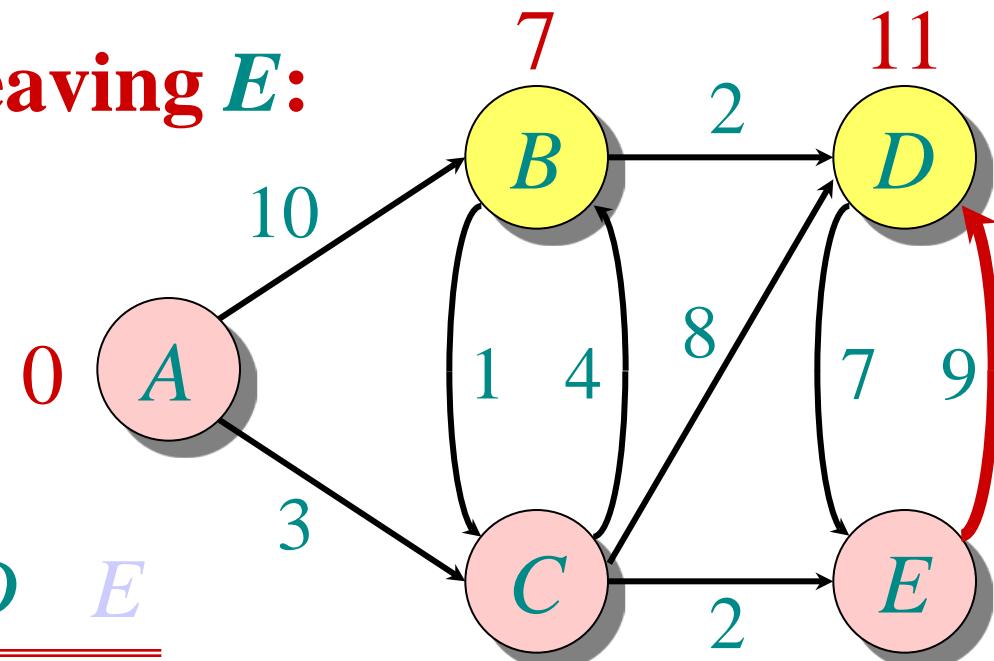


$S: \{ A, C, E \}$



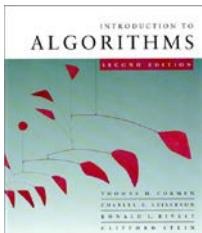
# Example of Dijkstra's algorithm

Relax all edges leaving  $E$ :



| $Q:$ | $A$ | $B$      | $C$      | $D$      | $E$      |
|------|-----|----------|----------|----------|----------|
|      | 0   | $\infty$ | $\infty$ | $\infty$ | $\infty$ |
|      | 10  | 3        | $\infty$ | $\infty$ |          |
|      | 7   |          | 11       |          | 5        |
|      | 7   |          | 11       |          |          |

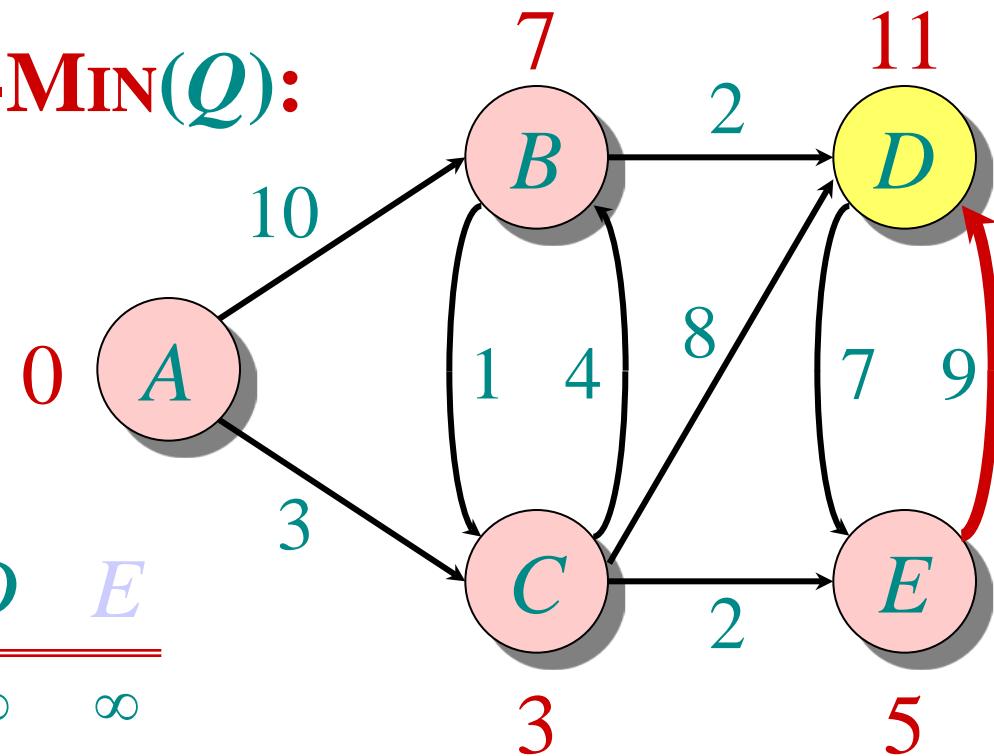
$S: \{ A, C, E \}$



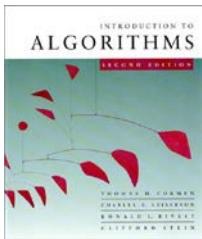
# Example of Dijkstra's algorithm

“*B*”  $\leftarrow$  EXTRACT-MIN(*Q*):

| <i>A</i> | <i>B</i> | <i>C</i> | <i>D</i> | <i>E</i> |
|----------|----------|----------|----------|----------|
| 0        | $\infty$ | $\infty$ | $\infty$ | $\infty$ |
| 10       |          | 3        | $\infty$ | $\infty$ |
| 7        |          | 11       | 5        | 11       |

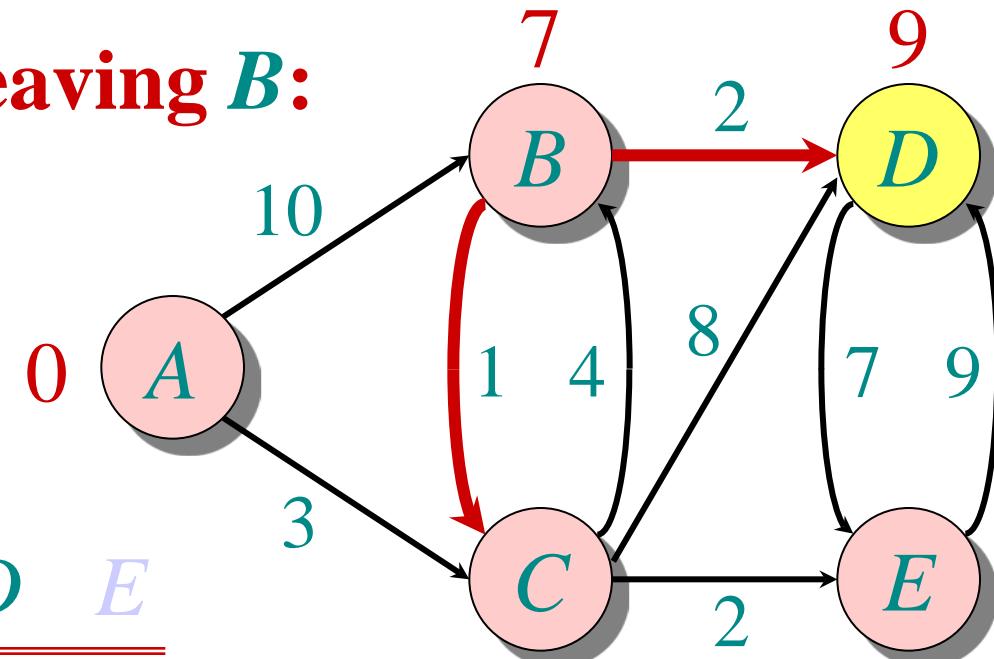


*S*: { *A, C, E, B* }



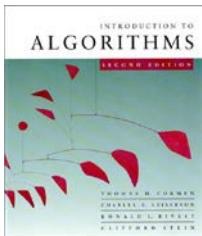
# Example of Dijkstra's algorithm

Relax all edges leaving  $B$ :



| $Q:$ | $A$ | $B$      | $C$      | $D$      | $E$      |
|------|-----|----------|----------|----------|----------|
|      | 0   | $\infty$ | $\infty$ | $\infty$ | $\infty$ |
|      | 10  |          | 3        | $\infty$ | $\infty$ |
|      | 7   |          | 11       | 5        |          |
|      | 7   |          | 11       |          | 9        |

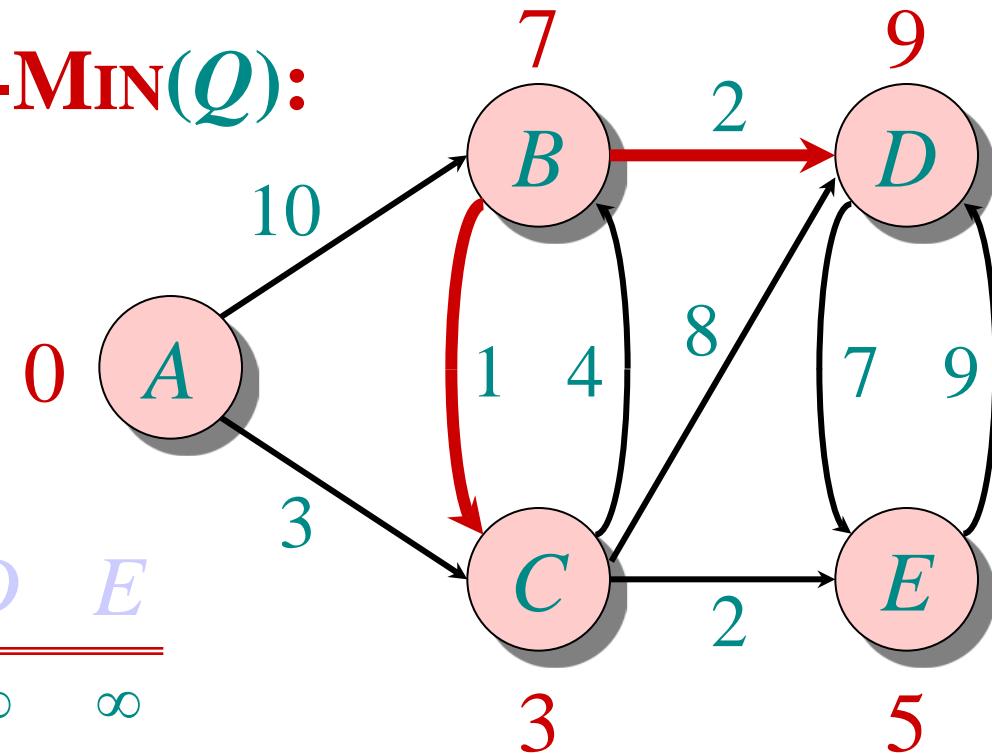
$S: \{ A, C, E, B \}$



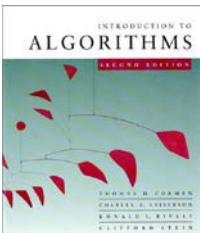
# Example of Dijkstra's algorithm

“ $D$ ”  $\leftarrow$  EXTRACT-MIN( $Q$ ):

| $A$ | $B$      | $C$      | $D$      | $E$      |
|-----|----------|----------|----------|----------|
| 0   | $\infty$ | $\infty$ | $\infty$ | $\infty$ |
| 10  |          | 3        | $\infty$ | $\infty$ |
| 7   |          | 11       | 5        | 9        |

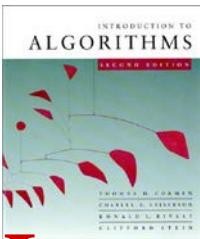


$S: \{ A, C, E, B, D \}$



# Correctness — Part I

**Lemma.** Initializing  $d[s] \leftarrow 0$  and  $d[v] \leftarrow \infty$  for all  $v \in V - \{s\}$  establishes  $d[v] \geq \delta(s, v)$  for all  $v \in V$ , and this invariant is maintained over any sequence of relaxation steps.



# Correctness — Part I

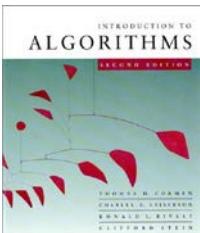
**Lemma.** Initializing  $d[s] \leftarrow 0$  and  $d[v] \leftarrow \infty$  for all  $v \in V - \{s\}$  establishes  $d[v] \geq \delta(s, v)$  for all  $v \in V$ , and this invariant is maintained over any sequence of relaxation steps.

**Proof.** Suppose not. Let  $v$  be the first vertex for which  $d[v] < \delta(s, v)$ , and let  $u$  be the vertex that caused  $d[v]$  to change:  $d[v] = d[u] + w(u, v)$ . Then,

$$\begin{aligned}d[v] &< \delta(s, v) \\&\leq \delta(s, u) + \delta(u, v) \\&\leq \delta(s, u) + w(u, v) \\&\leq d[u] + w(u, v)\end{aligned}$$

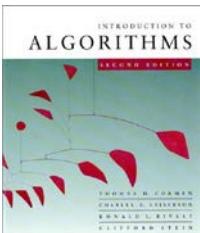
supposition  
triangle inequality  
sh. path  $\leq$  specific path  
 $v$  is first violation

Contradiction. 



# Correctness — Part II

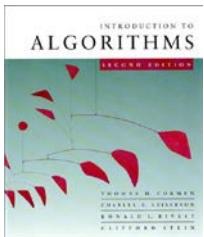
**Lemma.** Let  $u$  be  $v$ 's predecessor on a shortest path from  $s$  to  $v$ . Then, if  $d[u] = \delta(s, u)$  and edge  $(u, v)$  is relaxed, we have  $d[v] = \delta(s, v)$  after the relaxation.



# Correctness — Part II

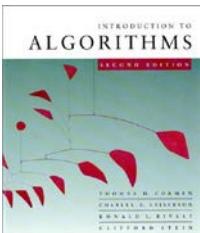
**Lemma.** Let  $u$  be  $v$ 's predecessor on a shortest path from  $s$  to  $v$ . Then, if  $d[u] = \delta(s, u)$  and edge  $(u, v)$  is relaxed, we have  $d[v] = \delta(s, v)$  after the relaxation.

**Proof.** Observe that  $\delta(s, v) = \delta(s, u) + w(u, v)$ . Suppose that  $d[v] > \delta(s, v)$  before the relaxation. (Otherwise, we're done.) Then, the test  $d[v] > d[u] + w(u, v)$  succeeds, because  $d[v] > \delta(s, v) = \delta(s, u) + w(u, v) = d[u] + w(u, v)$ , and the algorithm sets  $d[v] = d[u] + w(u, v) = \delta(s, v)$ . □



# Correctness — Part III

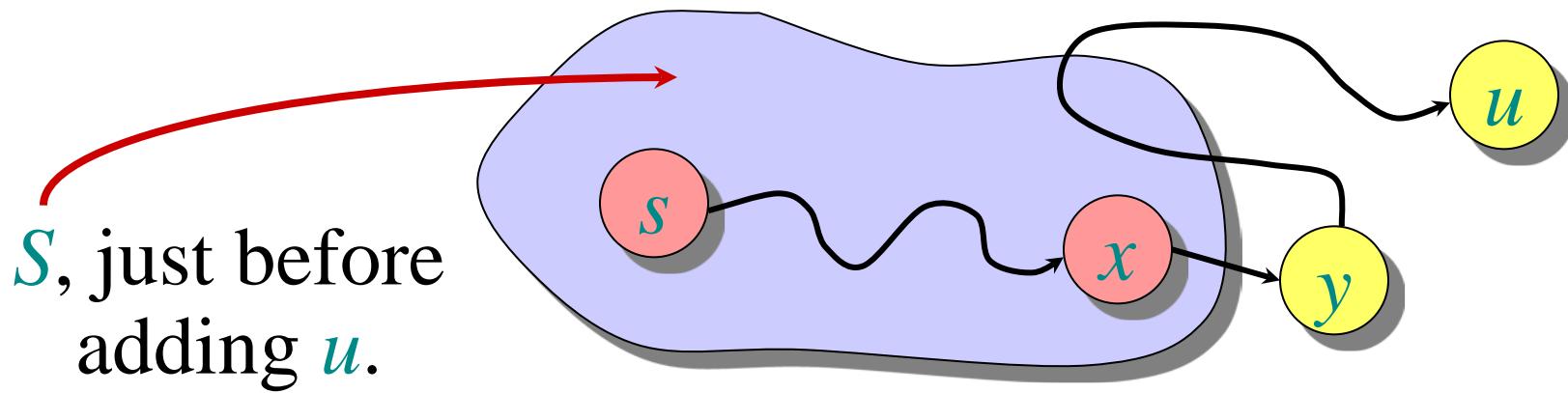
**Theorem.** Dijkstra's algorithm terminates with  $d[v] = \delta(s, v)$  for all  $v \in V$ .

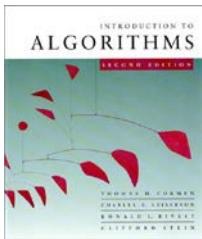


# Correctness — Part III

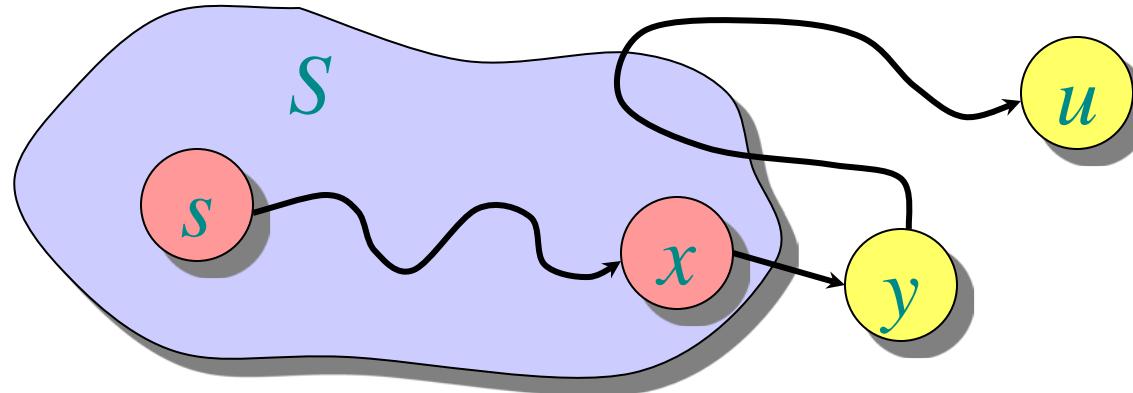
**Theorem.** Dijkstra's algorithm terminates with  $d[v] = \delta(s, v)$  for all  $v \in V$ .

*Proof.* It suffices to show that  $d[v] = \delta(s, v)$  for every  $v \in V$  when  $v$  is added to  $S$ . Suppose  $u$  is the first vertex added to  $S$  for which  $d[u] > \delta(s, u)$ . Let  $y$  be the first vertex in  $V - S$  along a shortest path from  $s$  to  $u$ , and let  $x$  be its predecessor:

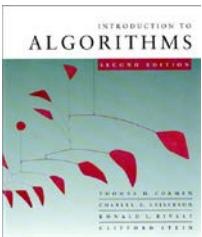




# Correctness — Part III (continued)

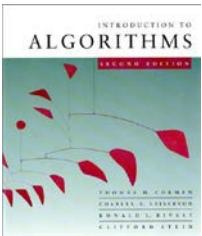


Since  $u$  is the first vertex violating the claimed invariant, we have  $d[x] = \delta(s, x)$ . When  $x$  was added to  $S$ , the edge  $(x, y)$  was relaxed, which implies that  $d[y] = \delta(s, y) \leq \delta(s, u) < d[u]$ . But,  $d[u] \leq d[y]$  by our choice of  $u$ . Contradiction. □



# Analysis of Dijkstra

```
while  $Q \neq \emptyset$ 
    do  $u \leftarrow \text{EXTRACT-MIN}(Q)$ 
         $S \leftarrow S \cup \{u\}$ 
        for each  $v \in \text{Adj}[u]$ 
            do if  $d[v] > d[u] + w(u, v)$ 
                then  $d[v] \leftarrow d[u] + w(u, v)$ 
```

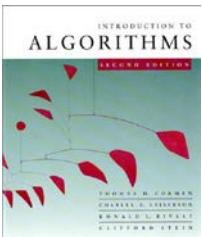


# Analysis of Dijkstra

$|V|$   
times

{

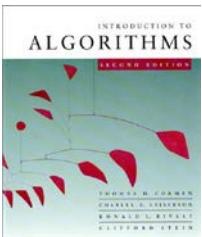
```
while  $Q \neq \emptyset$ 
    do  $u \leftarrow \text{EXTRACT-MIN}(Q)$ 
         $S \leftarrow S \cup \{u\}$ 
        for each  $v \in \text{Adj}[u]$ 
            do if  $d[v] > d[u] + w(u, v)$ 
                then  $d[v] \leftarrow d[u] + w(u, v)$ 
```



# Analysis of Dijkstra

$|V|$   
times }       $degree(u)$   
              times {

```
while  $Q \neq \emptyset$ 
    do  $u \leftarrow \text{EXTRACT-MIN}(Q)$ 
         $S \leftarrow S \cup \{u\}$ 
        for each  $v \in Adj[u]$ 
            do if  $d[v] > d[u] + w(u, v)$ 
                then  $d[v] \leftarrow d[u] + w(u, v)$ 
```



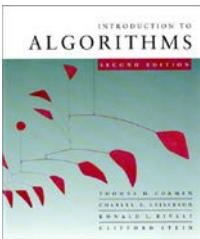
# Analysis of Dijkstra

|V| times {  
*degree(u)* times {

```
while  $Q \neq \emptyset$ 
    do  $u \leftarrow \text{EXTRACT-MIN}(Q)$ 
         $S \leftarrow S \cup \{u\}$ 
        for each  $v \in \text{Adj}[u]$ 
            do if  $d[v] > d[u] + w(u, v)$ 
                then  $d[v] \leftarrow d[u] + w(u, v)$ 
```

↑

Handshaking Lemma  $\Rightarrow \Theta(E)$  implicit DECREASE-KEY's.



# Analysis of Dijkstra

$|V|$   
times }       $degree(u)$   
                times {

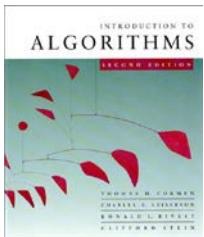
```
while  $Q \neq \emptyset$ 
    do  $u \leftarrow \text{EXTRACT-MIN}(Q)$ 
         $S \leftarrow S \cup \{u\}$ 
        for each  $v \in Adj[u]$ 
            do if  $d[v] > d[u] + w(u, v)$ 
                then  $d[v] \leftarrow d[u] + w(u, v)$ 
```

A red curly brace on the left groups the outer loop's execution by  $|V|$ . Another red curly brace groups the inner loop's execution by  $degree(u)$ . A red arrow points from the  $d[v] \leftarrow d[u] + w(u, v)$  assignment statement to the  $d[v]$  term, highlighting it.

Handshaking Lemma  $\Rightarrow \Theta(E)$  implicit DECREASE-KEY's.

$$\text{Time} = \Theta(V \cdot T_{\text{EXTRACT-MIN}} + E \cdot T_{\text{DECREASE-KEY}})$$

**Note:** Same formula as in the analysis of Prim's minimum spanning tree algorithm.

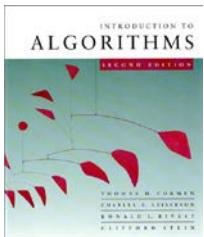


# Analysis of Dijkstra (continued)

Time =  $\Theta(V) \cdot T_{\text{EXTRACT-MIN}} + \Theta(E) \cdot T_{\text{DECREASE-KEY}}$

| $Q$ | $T_{\text{EXTRACT-MIN}}$ | $T_{\text{DECREASE-KEY}}$ | Total |
|-----|--------------------------|---------------------------|-------|
|-----|--------------------------|---------------------------|-------|

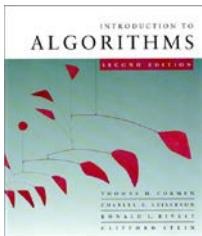
---



# Analysis of Dijkstra (continued)

Time =  $\Theta(V) \cdot T_{\text{EXTRACT-MIN}} + \Theta(E) \cdot T_{\text{DECREASE-KEY}}$

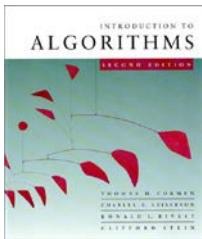
| $Q$   | $T_{\text{EXTRACT-MIN}}$ | $T_{\text{DECREASE-KEY}}$ | Total    |
|-------|--------------------------|---------------------------|----------|
| array | $O(V)$                   | $O(1)$                    | $O(V^2)$ |



# Analysis of Dijkstra (continued)

Time =  $\Theta(V) \cdot T_{\text{EXTRACT-MIN}} + \Theta(E) \cdot T_{\text{DECREASE-KEY}}$

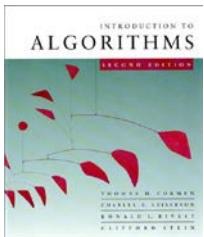
| $Q$         | $T_{\text{EXTRACT-MIN}}$ | $T_{\text{DECREASE-KEY}}$ | Total        |
|-------------|--------------------------|---------------------------|--------------|
| array       | $O(V)$                   | $O(1)$                    | $O(V^2)$     |
| binary heap | $O(\lg V)$               | $O(\lg V)$                | $O(E \lg V)$ |



# Analysis of Dijkstra (continued)

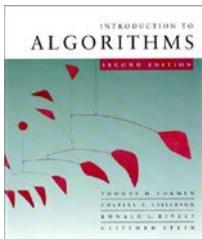
Time =  $\Theta(V) \cdot T_{\text{EXTRACT-MIN}} + \Theta(E) \cdot T_{\text{DECREASE-KEY}}$

| $Q$            | $T_{\text{EXTRACT-MIN}}$ | $T_{\text{DECREASE-KEY}}$ | Total                          |
|----------------|--------------------------|---------------------------|--------------------------------|
| array          | $O(V)$                   | $O(1)$                    | $O(V^2)$                       |
| binary heap    | $O(\lg V)$               | $O(\lg V)$                | $O(E \lg V)$                   |
| Fibonacci heap | $O(\lg V)$<br>amortized  | $O(1)$<br>amortized       | $O(E + V \lg V)$<br>worst case |



# Unweighted graphs

Suppose that  $w(u, v) = 1$  for all  $(u, v) \in E$ .  
Can Dijkstra's algorithm be improved?

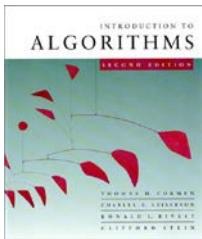


# Unweighted graphs

Suppose that  $w(u, v) = 1$  for all  $(u, v) \in E$ .

Can Dijkstra's algorithm be improved?

- Use a simple FIFO queue instead of a priority queue.



# Unweighted graphs

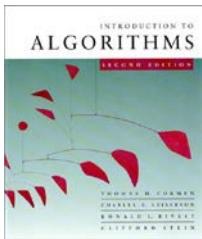
Suppose that  $w(u, v) = 1$  for all  $(u, v) \in E$ .

Can Dijkstra's algorithm be improved?

- Use a simple FIFO queue instead of a priority queue.

## *Breadth-first search*

```
while  $Q \neq \emptyset$ 
    do  $u \leftarrow \text{DEQUEUE}(Q)$ 
        for each  $v \in \text{Adj}[u]$ 
            do if  $d[v] = \infty$ 
                then  $d[v] \leftarrow d[u] + 1$ 
                    ENQUEUE( $Q, v$ )
```



# Unweighted graphs

Suppose that  $w(u, v) = 1$  for all  $(u, v) \in E$ .

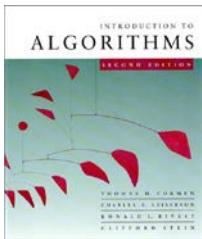
Can Dijkstra's algorithm be improved?

- Use a simple FIFO queue instead of a priority queue.

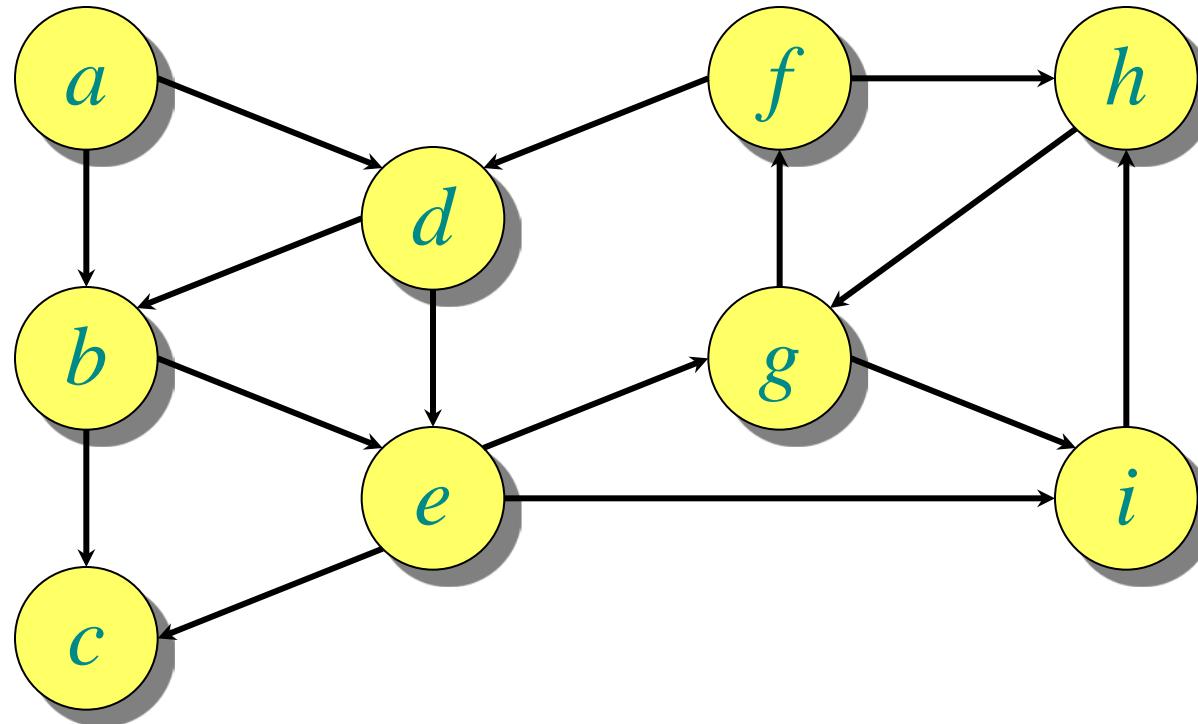
## *Breadth-first search*

```
while  $Q \neq \emptyset$ 
    do  $u \leftarrow \text{DEQUEUE}(Q)$ 
        for each  $v \in \text{Adj}[u]$ 
            do if  $d[v] = \infty$ 
                then  $d[v] \leftarrow d[u] + 1$ 
                    ENQUEUE( $Q, v$ )
```

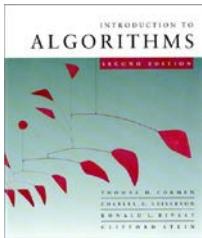
**Analysis:** Time =  $O(V + E)$ .



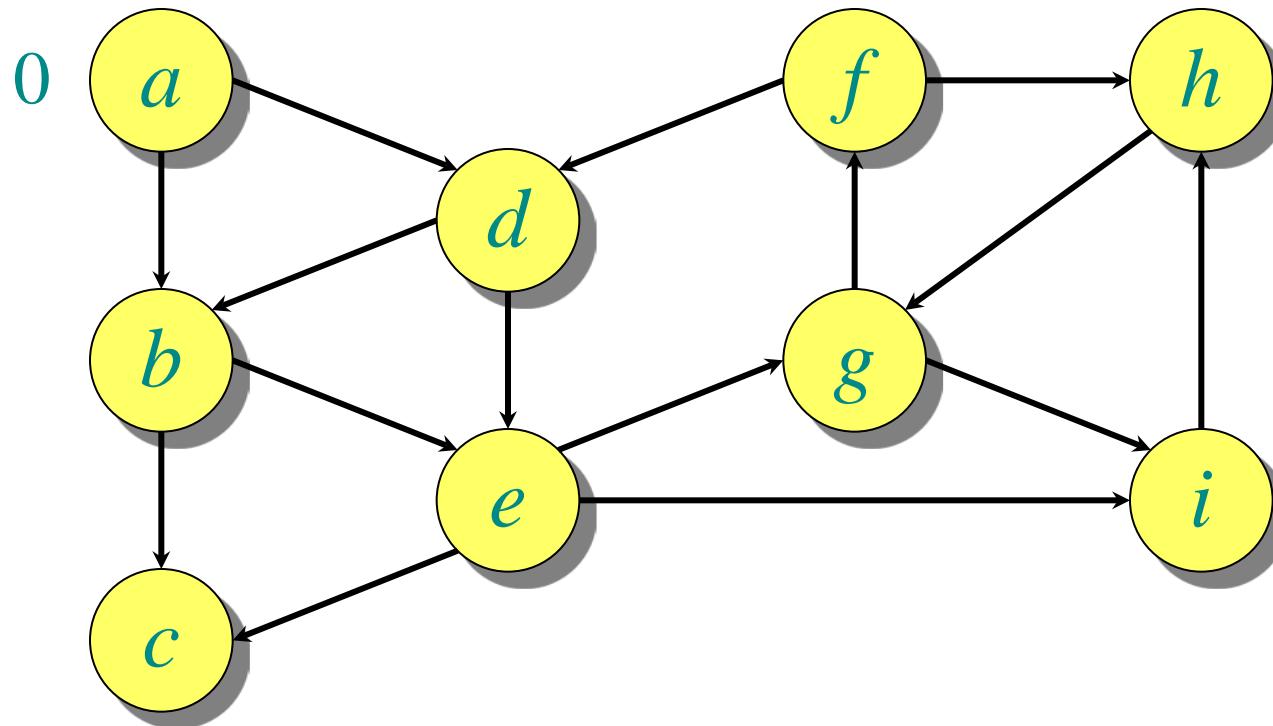
# Example of breadth-first search



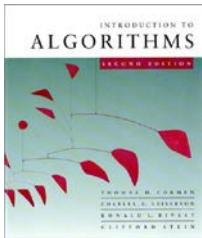
*Q:*



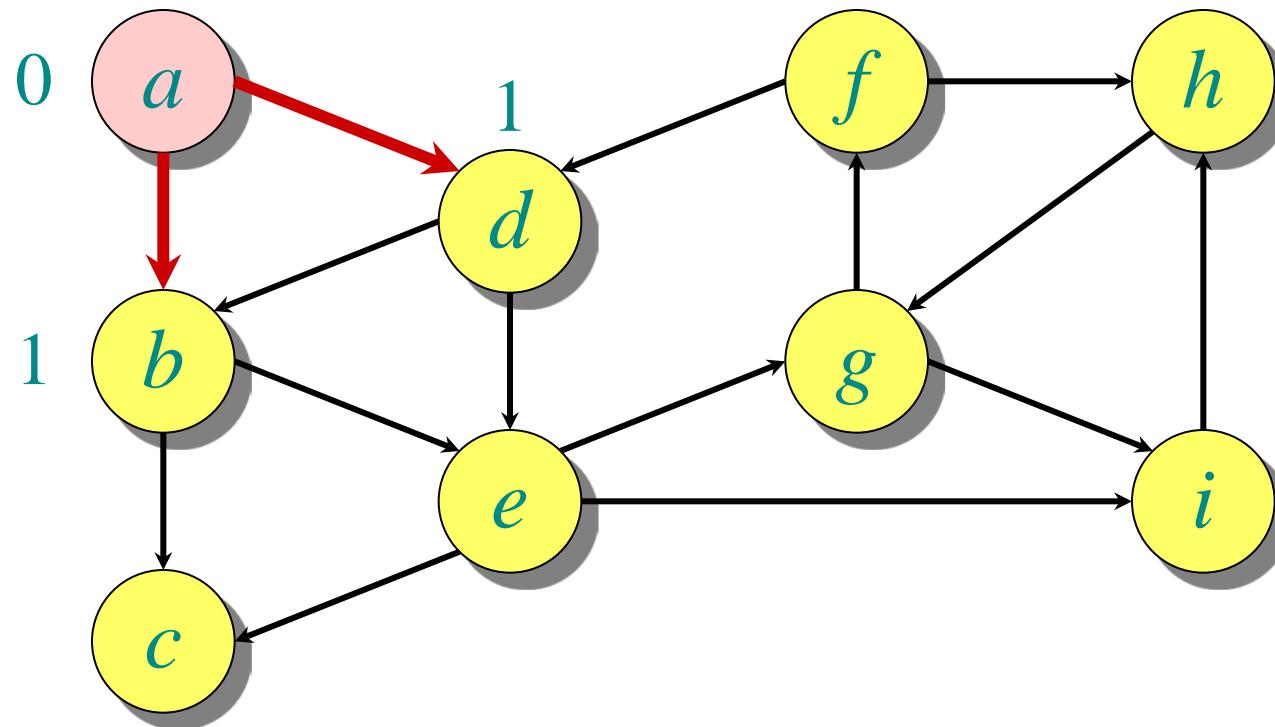
# Example of breadth-first search



0  
 $Q: a$

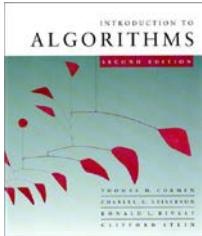


# Example of breadth-first search

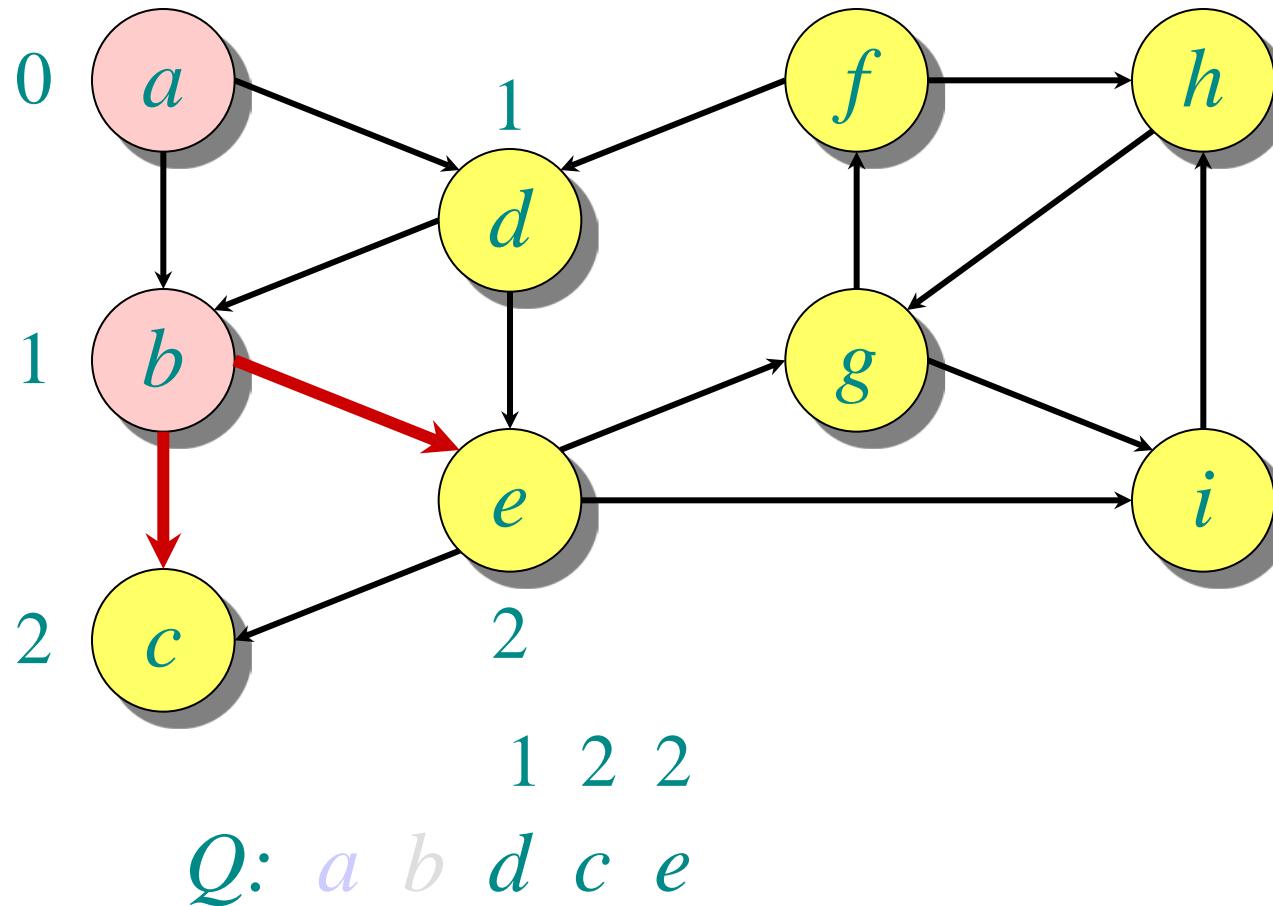


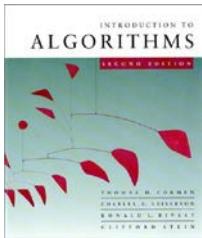
1 1

$Q: a \ b \ d$

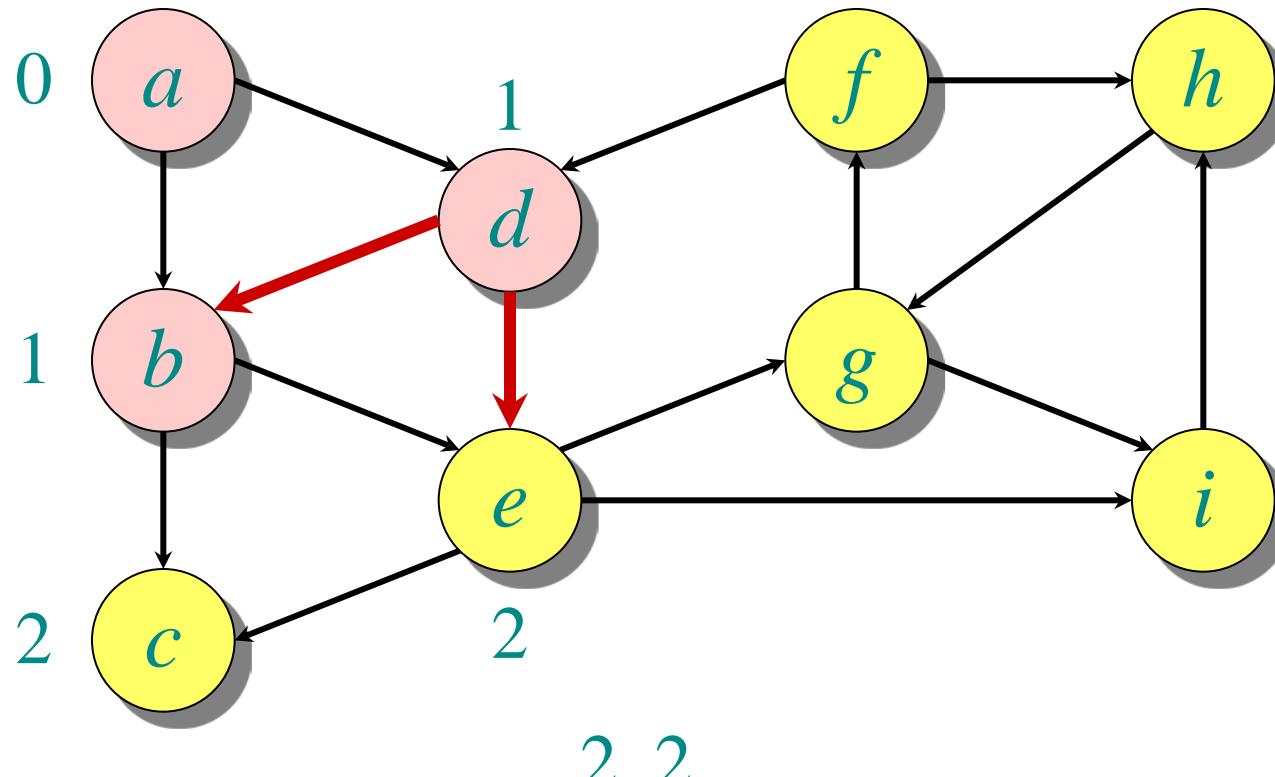


# Example of breadth-first search

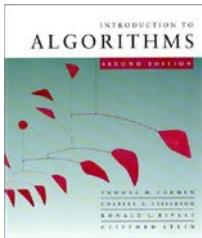




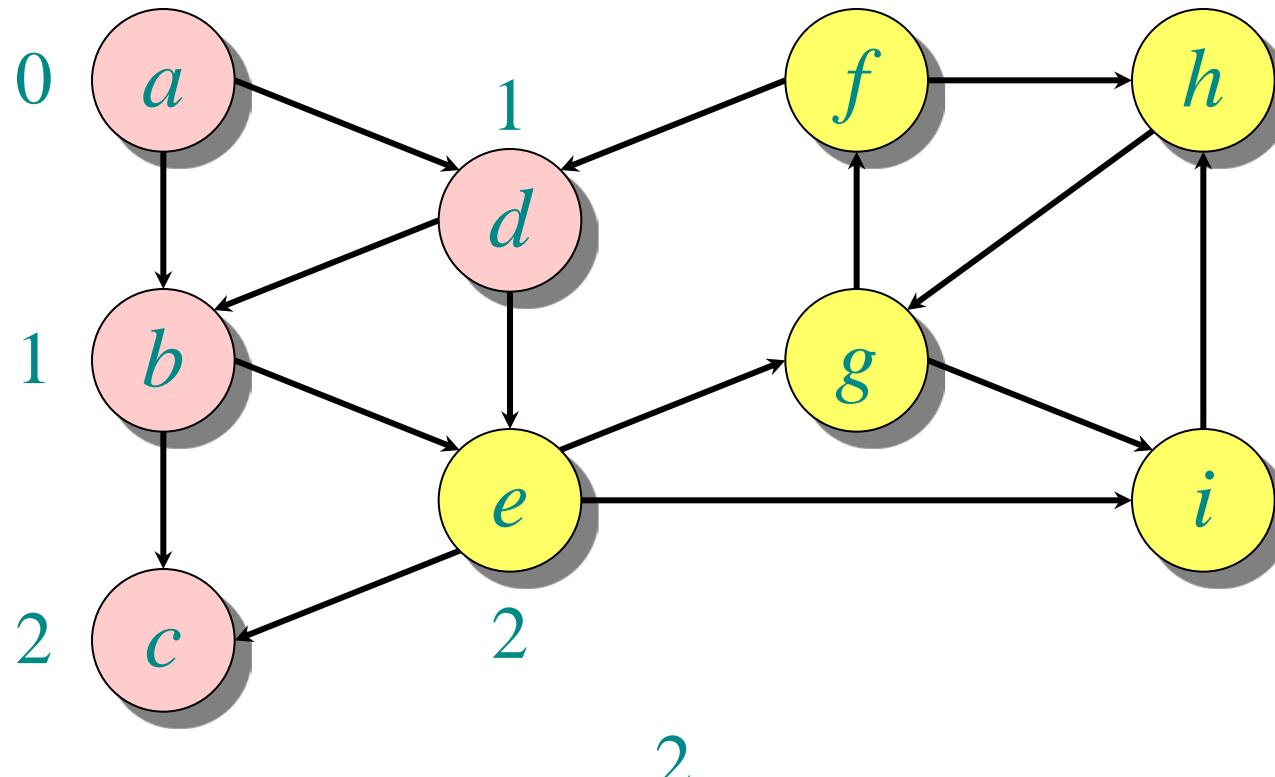
# Example of breadth-first search



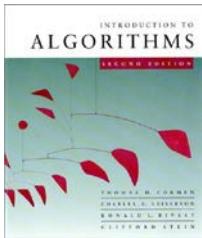
$Q: a \ b \ d \ c \ e$



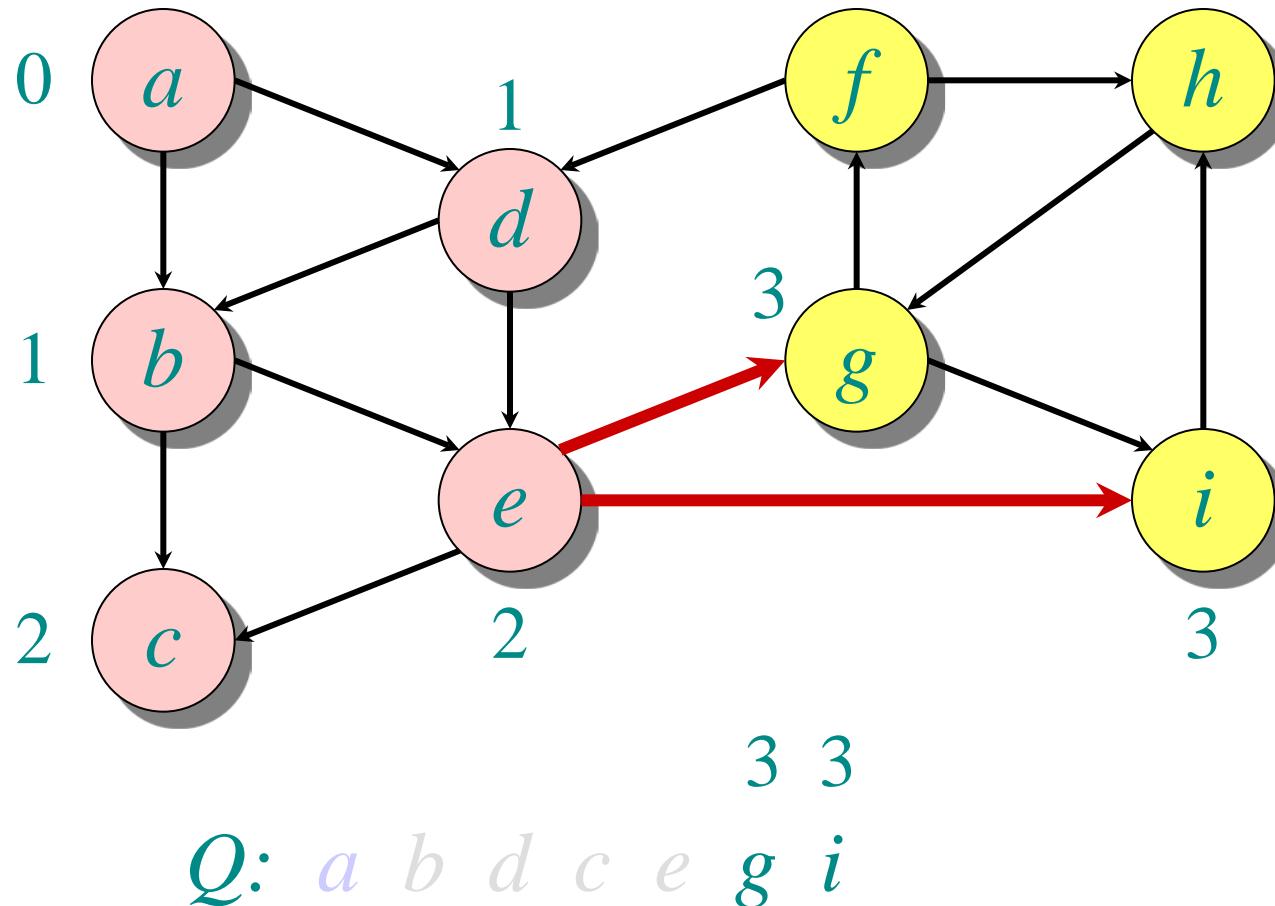
# Example of breadth-first search

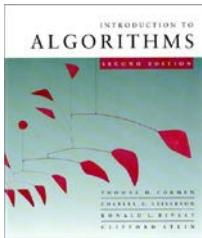


*Q:* *a b d c e*

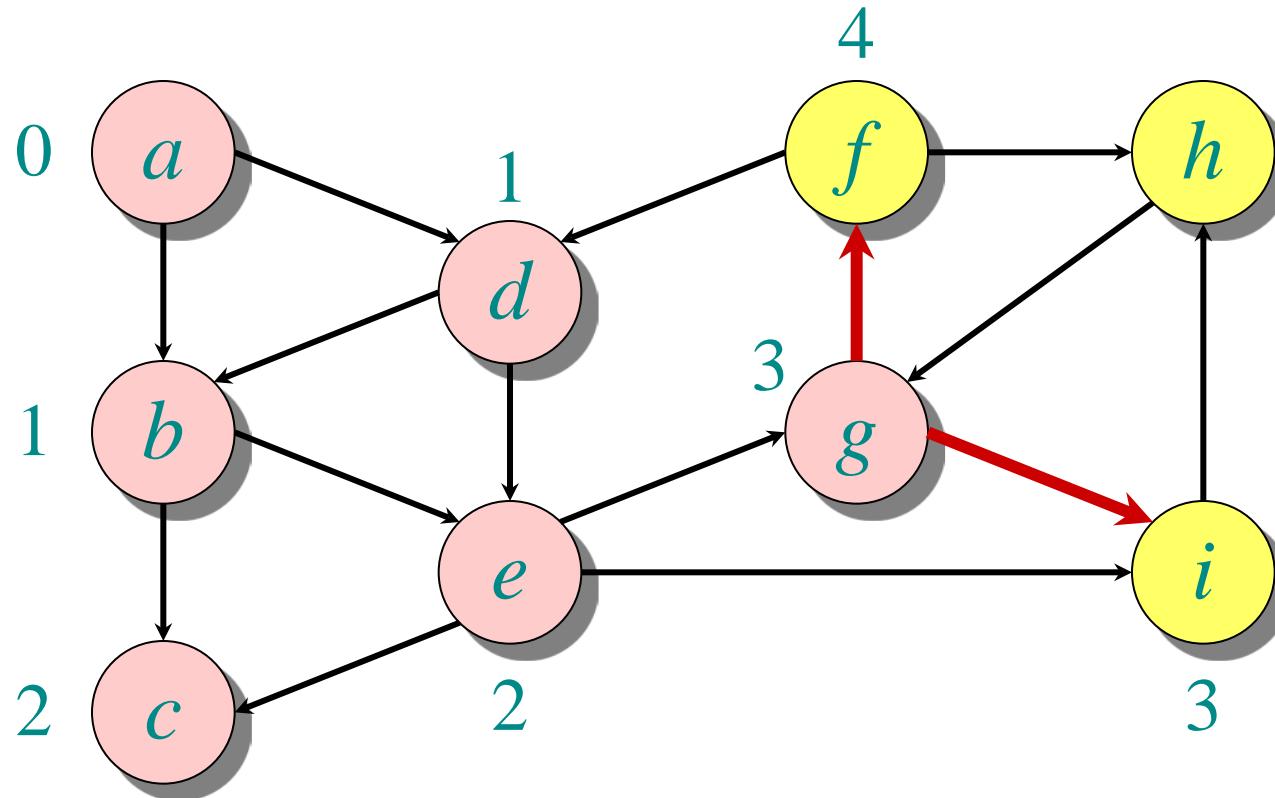


# Example of breadth-first search

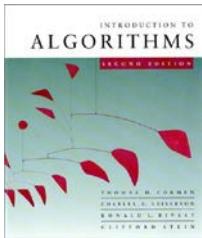




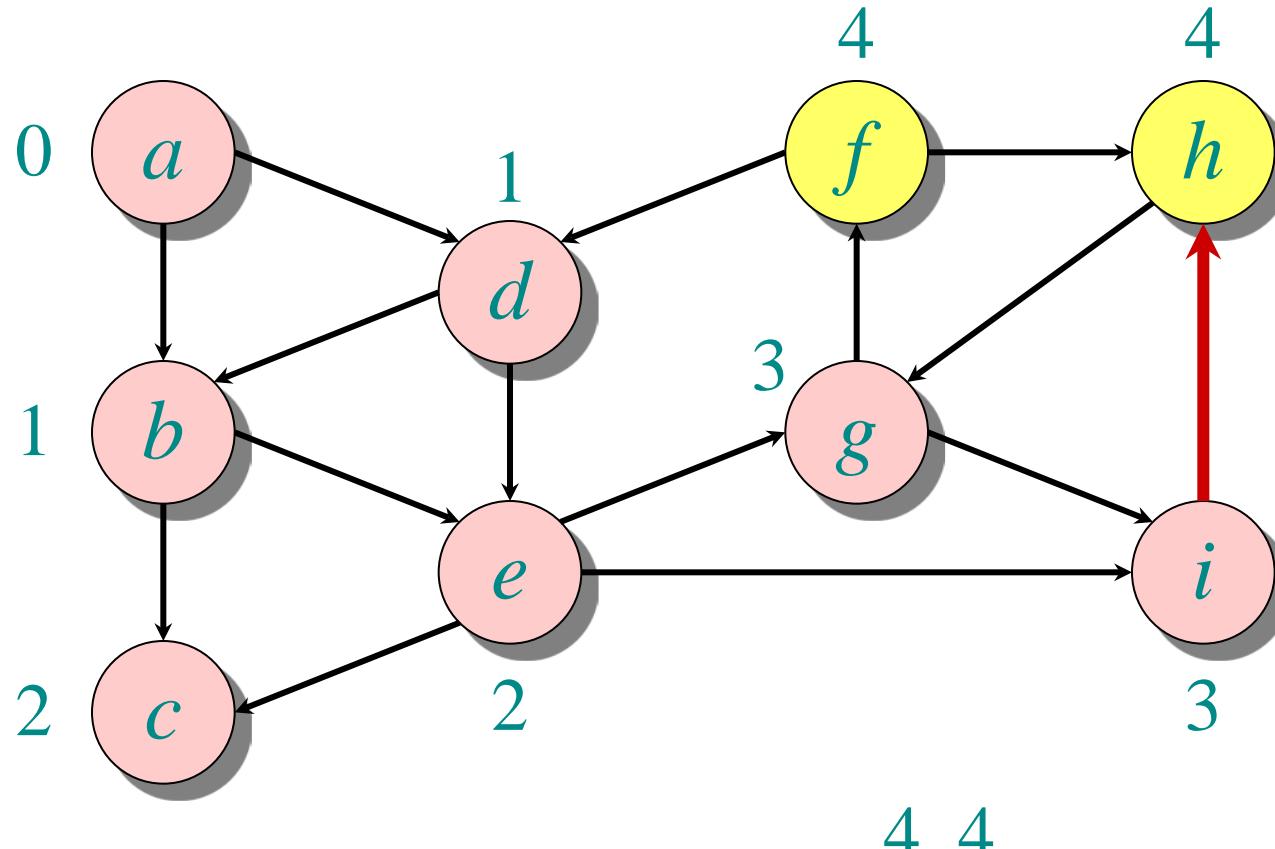
# Example of breadth-first search



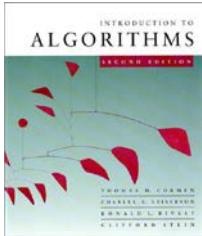
*Q:* *a b d c e g i f*



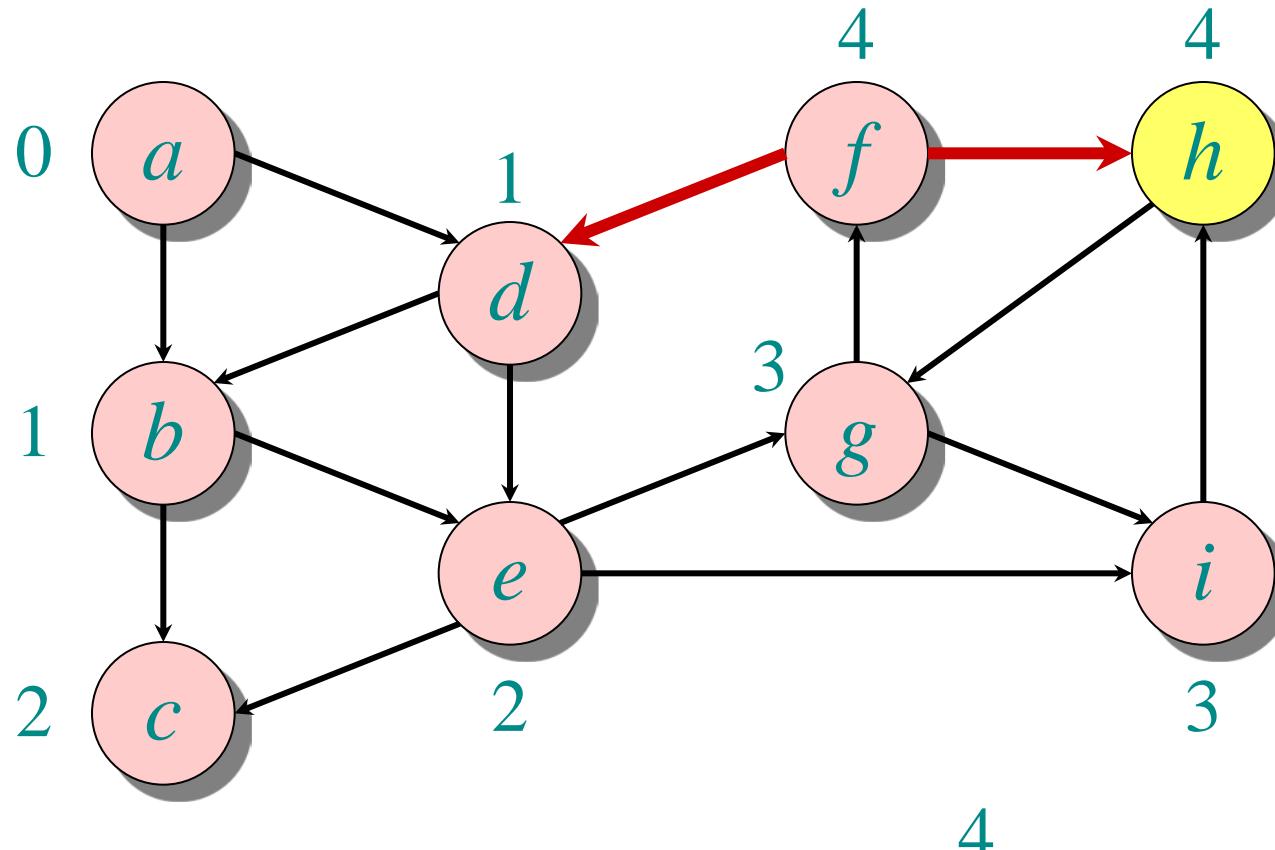
# Example of breadth-first search



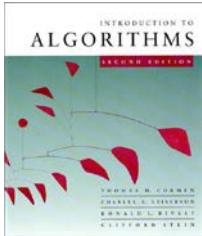
*Q:* *a b d c e g i f h*



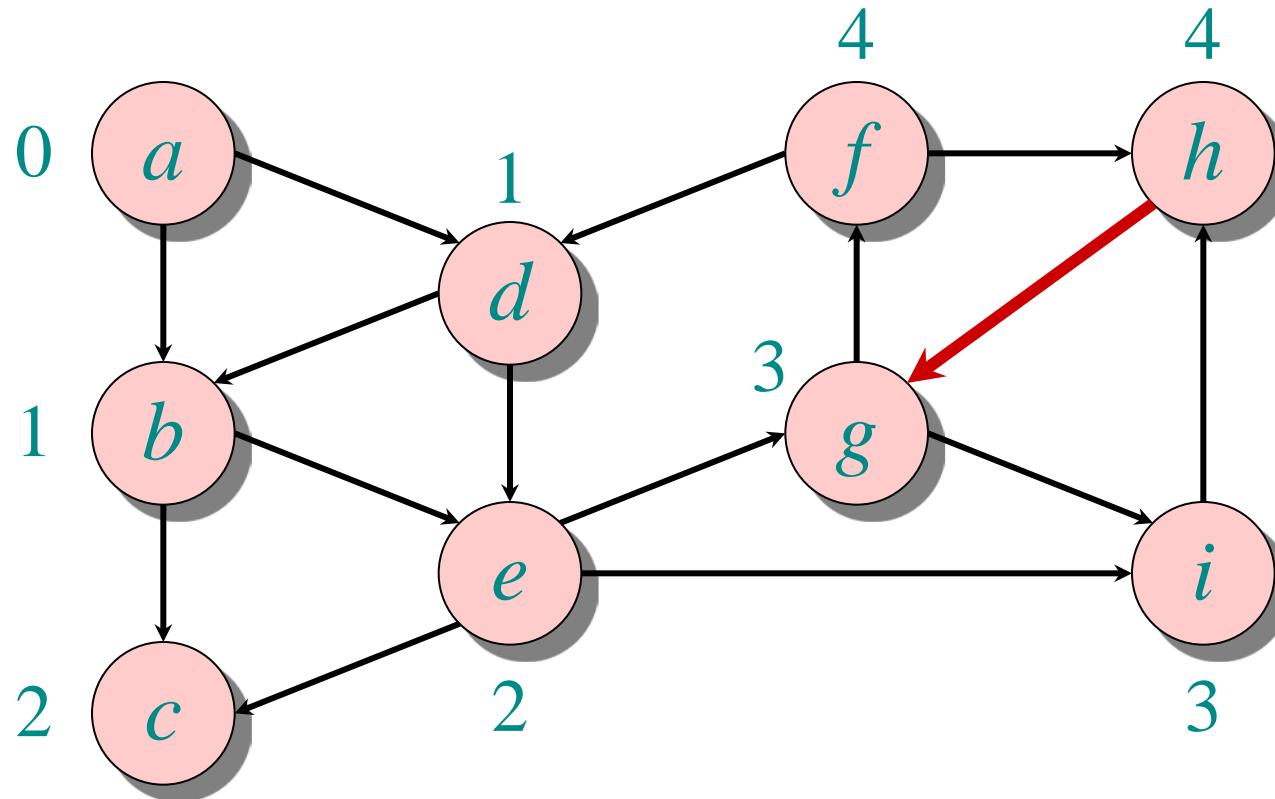
# Example of breadth-first search



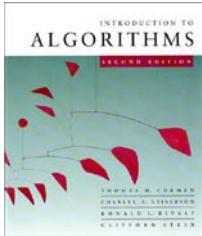
*Q:* *a b d c e g i f h*



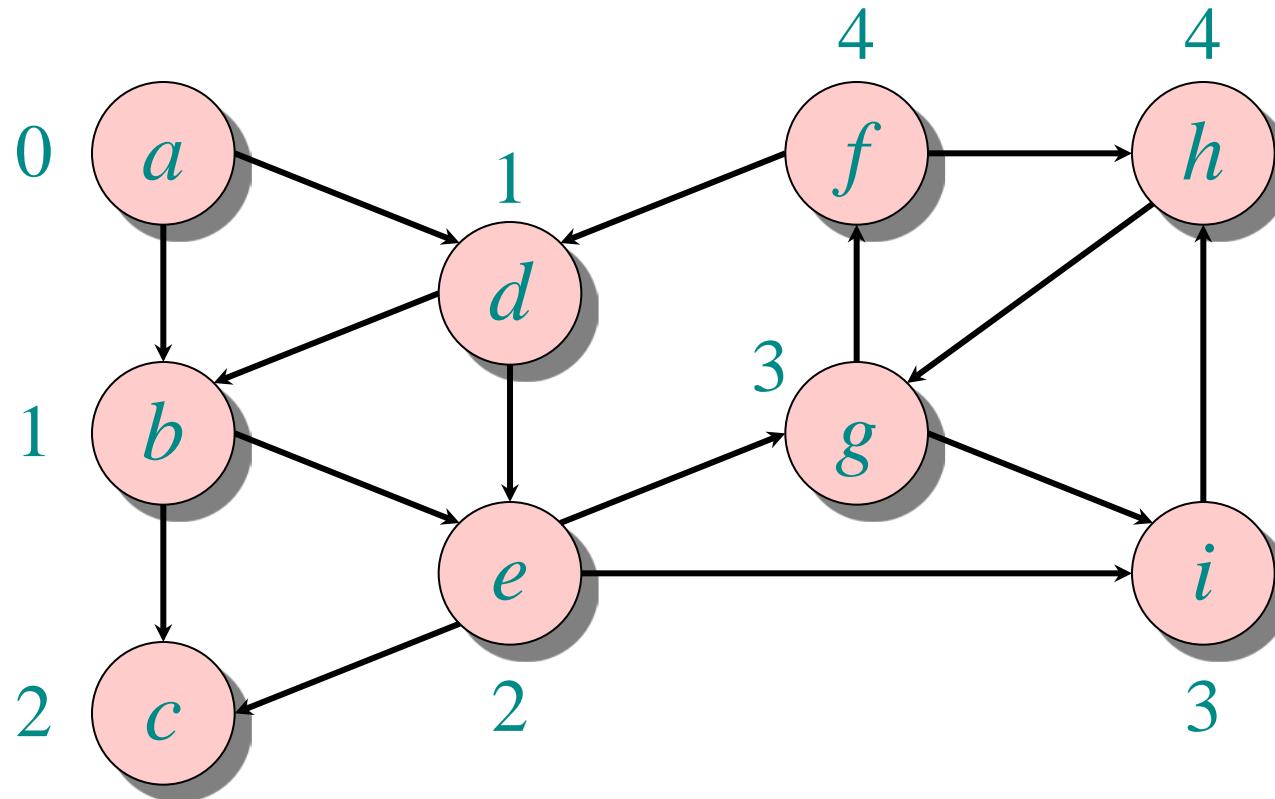
# Example of breadth-first search



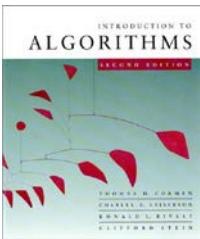
*Q:* *a b d c e g i f h*



# Example of breadth-first search



*Q:* *a b d c e g i f h*



# Correctness of BFS

```
while  $Q \neq \emptyset$ 
    do  $u \leftarrow \text{DEQUEUE}(Q)$ 
        for each  $v \in \text{Adj}[u]$ 
            do if  $d[v] = \infty$ 
                then  $d[v] \leftarrow d[u] + 1$ 
                    ENQUEUE( $Q, v$ )
```

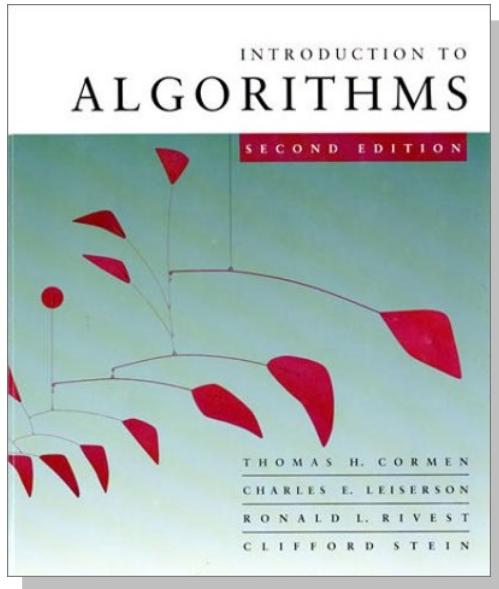
## Key idea:

The FIFO  $Q$  in breadth-first search mimics the priority queue  $Q$  in Dijkstra.

- **Invariant:**  $v$  comes after  $u$  in  $Q$  implies that  $d[v] = d[u]$  or  $d[v] = d[u] + 1$ .

# *Introduction to Algorithms*

## 6.046J/18.401J

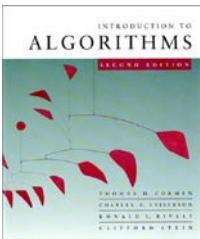


### LECTURE 18

#### Shortest Paths II

- Bellman-Ford algorithm
- Linear programming and difference constraints
- VLSI layout compaction

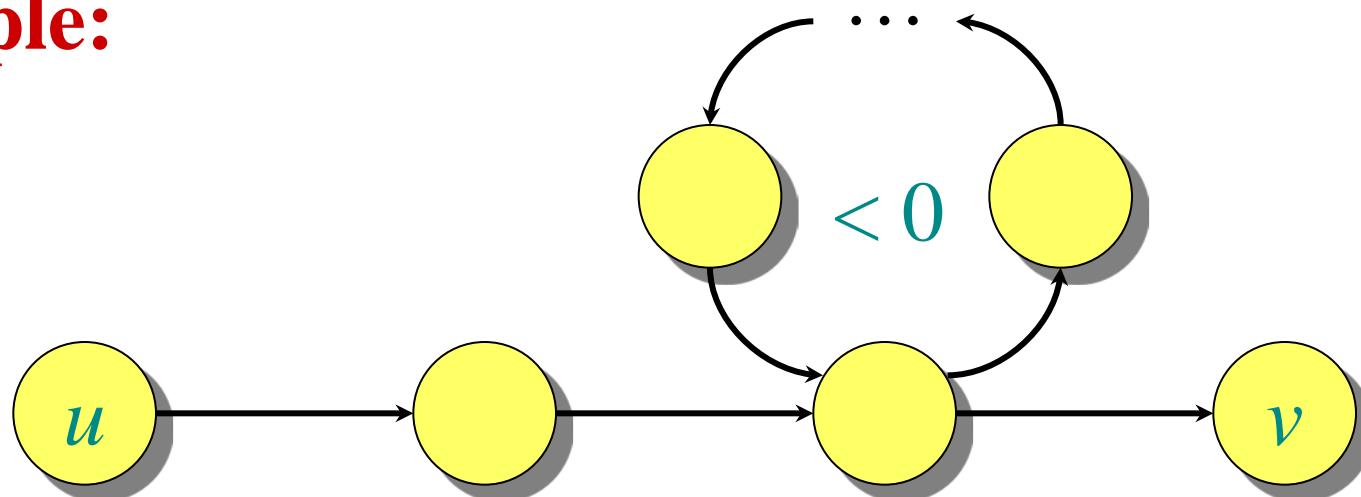
Prof. Erik Demaine

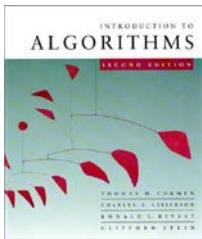


# Negative-weight cycles

**Recall:** If a graph  $G = (V, E)$  contains a negative-weight cycle, then some shortest paths may not exist.

**Example:**

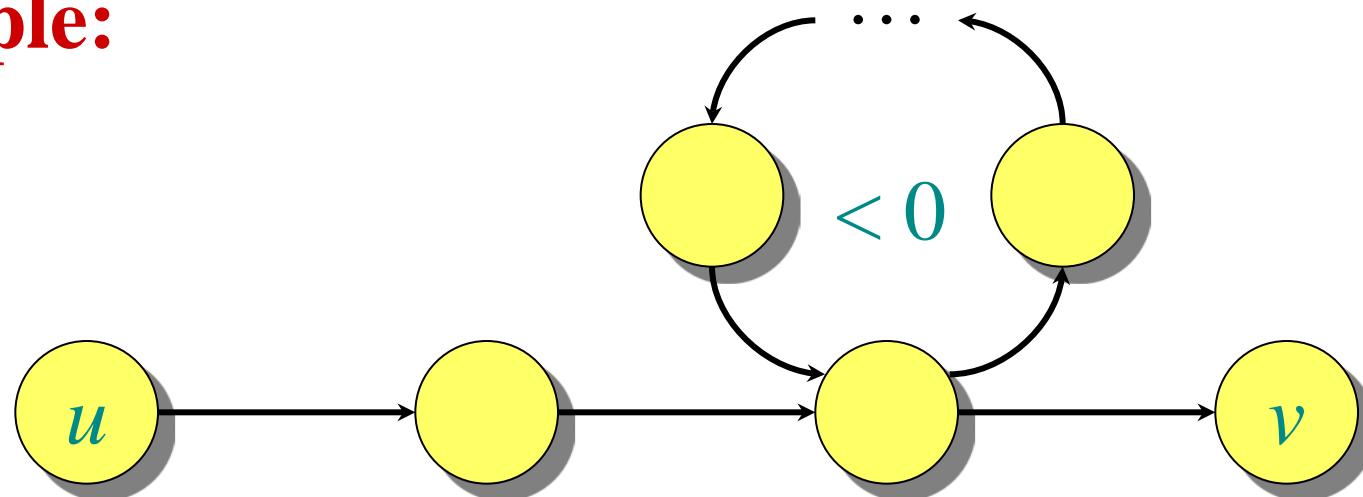




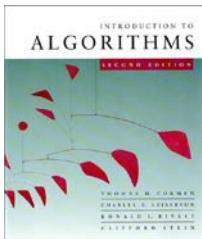
# Negative-weight cycles

**Recall:** If a graph  $G = (V, E)$  contains a negative-weight cycle, then some shortest paths may not exist.

**Example:**



**Bellman-Ford algorithm:** Finds all shortest-path lengths from a **source**  $s \in V$  to all  $v \in V$  or determines that a negative-weight cycle exists.



# Bellman-Ford algorithm

```
 $d[s] \leftarrow 0$ 
for each  $v \in V - \{s\}$ 
  do  $d[v] \leftarrow \infty$ 
```

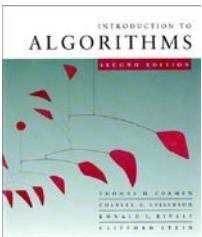
}

**for**  $i \leftarrow 1$  **to**  $|V| - 1$ 
**do for** each edge  $(u, v) \in E$ 
**do if**  $d[v] > d[u] + w(u, v)$ 
**then**  $d[v] \leftarrow d[u] + w(u, v)$

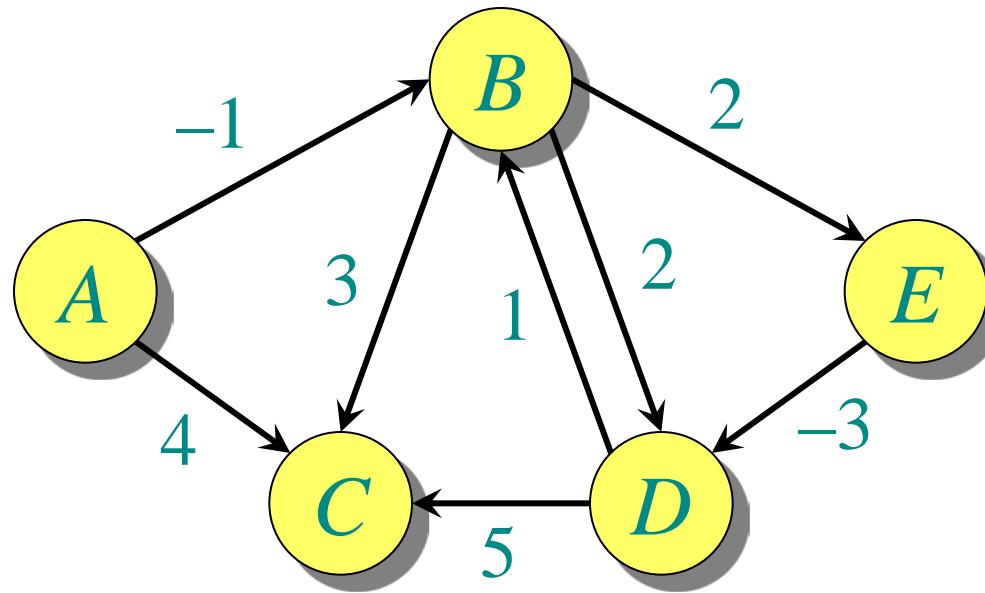
*relaxation step*

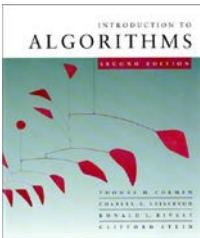
```
for each edge  $(u, v) \in E$ 
  do if  $d[v] > d[u] + w(u, v)$ 
    then report that a negative-weight cycle exists
```

At the end,  $d[v] = \delta(s, v)$ , if no negative-weight cycles.  
Time =  $O(VE)$ .

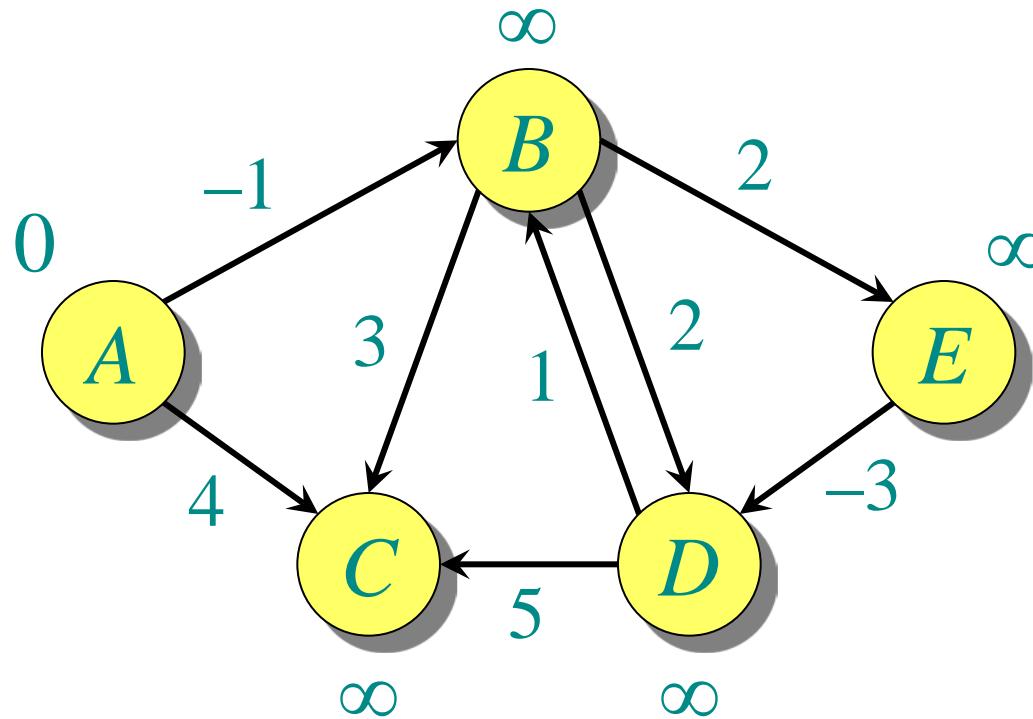


# Example of Bellman-Ford

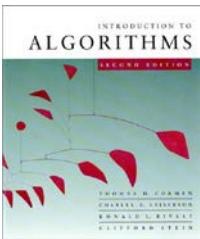




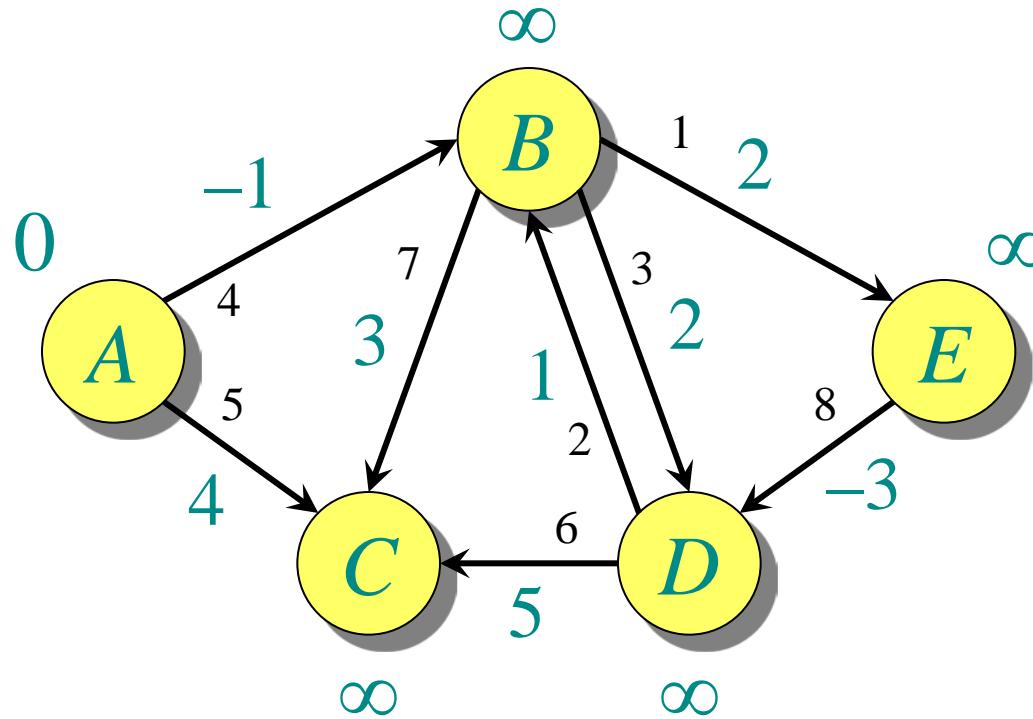
# Example of Bellman-Ford



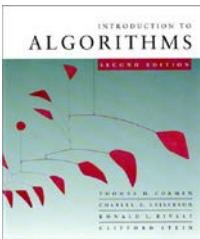
Initialization.



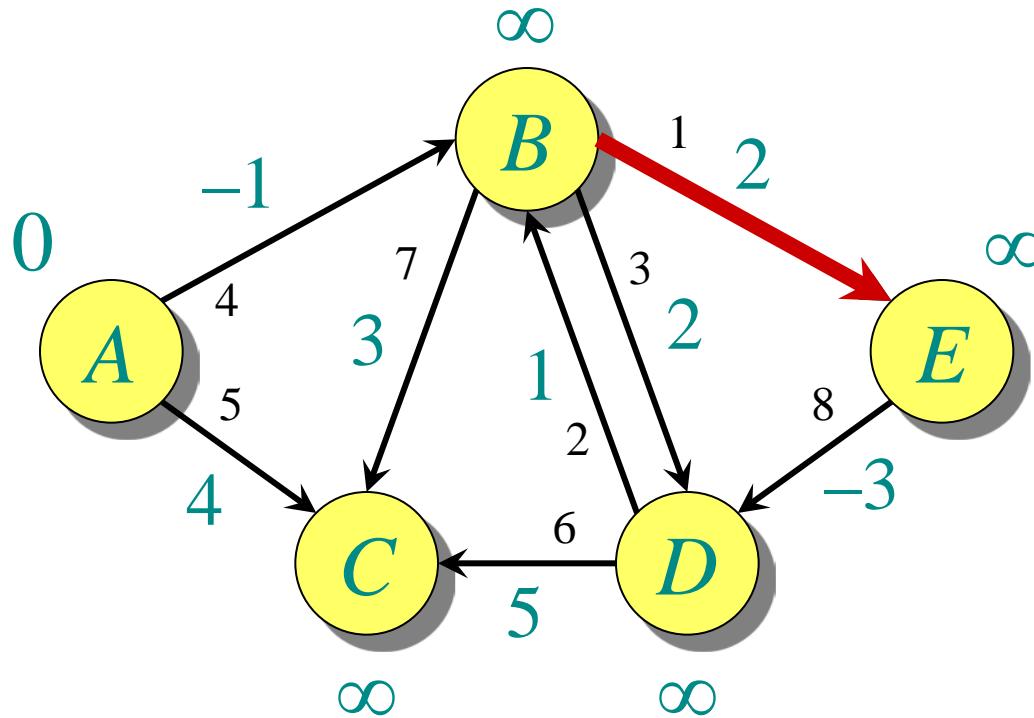
# Example of Bellman-Ford

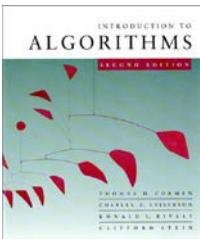


Order of edge relaxation.

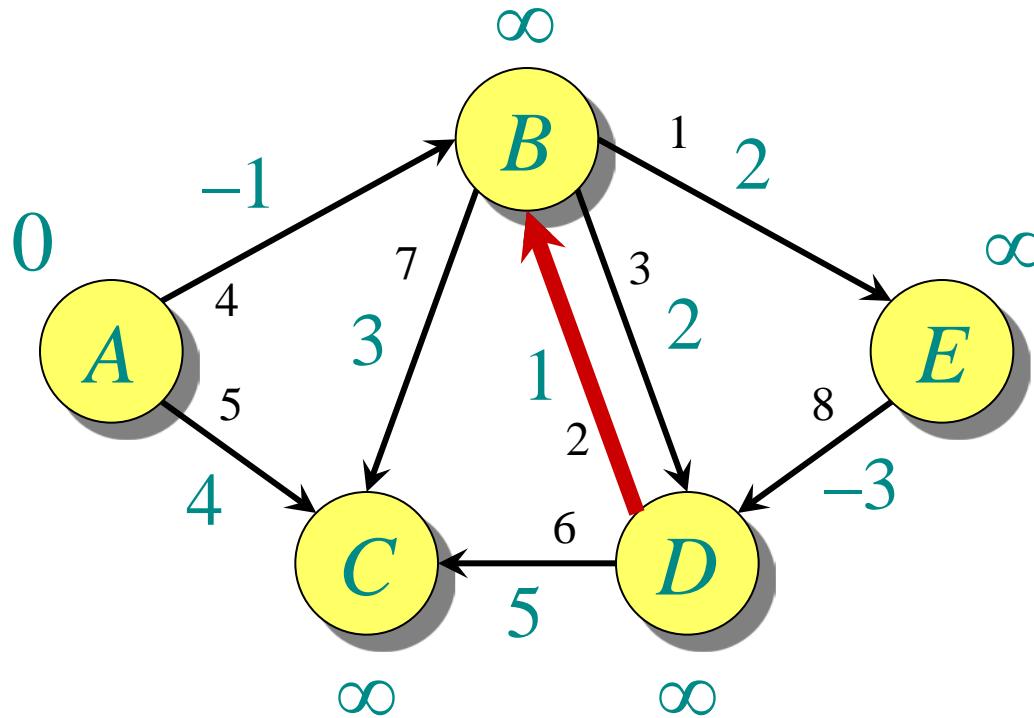


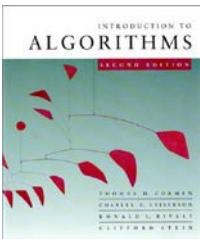
# Example of Bellman-Ford



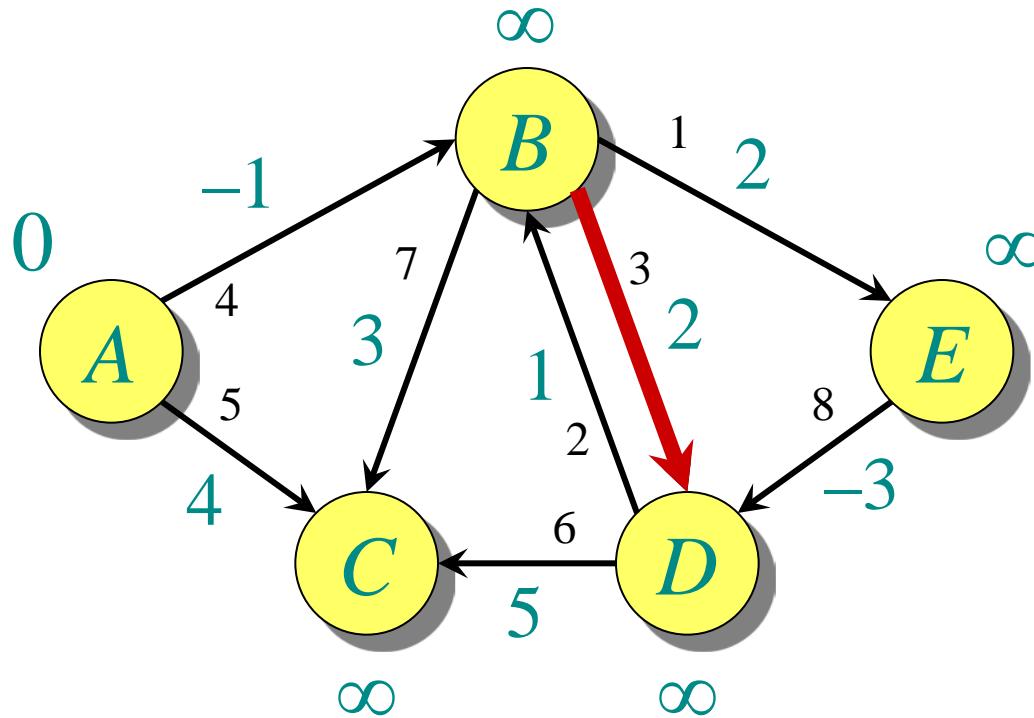


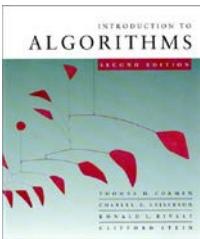
# Example of Bellman-Ford



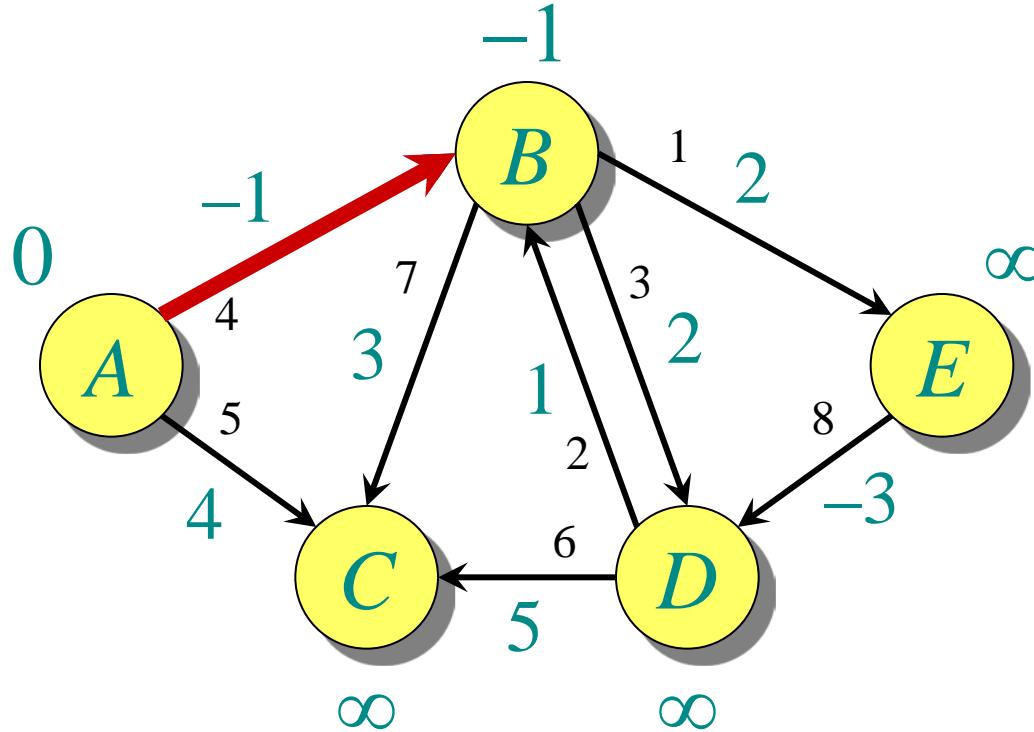


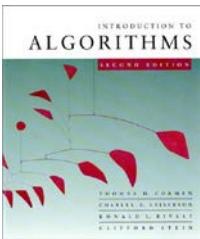
# Example of Bellman-Ford



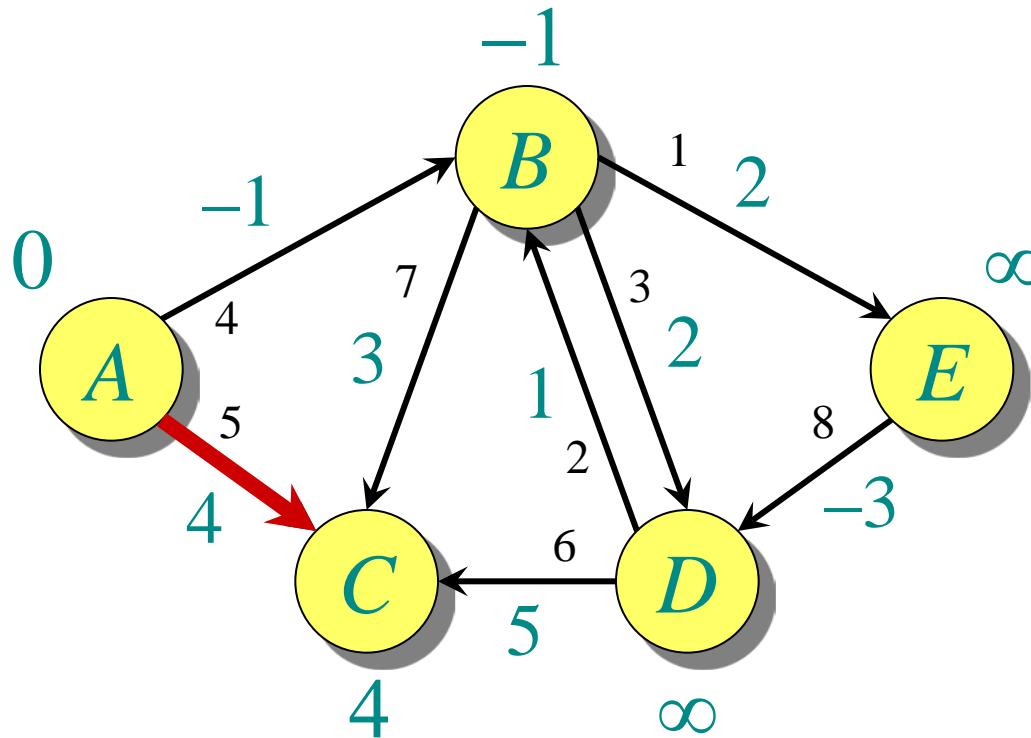


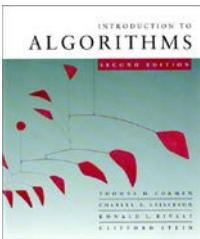
# Example of Bellman-Ford



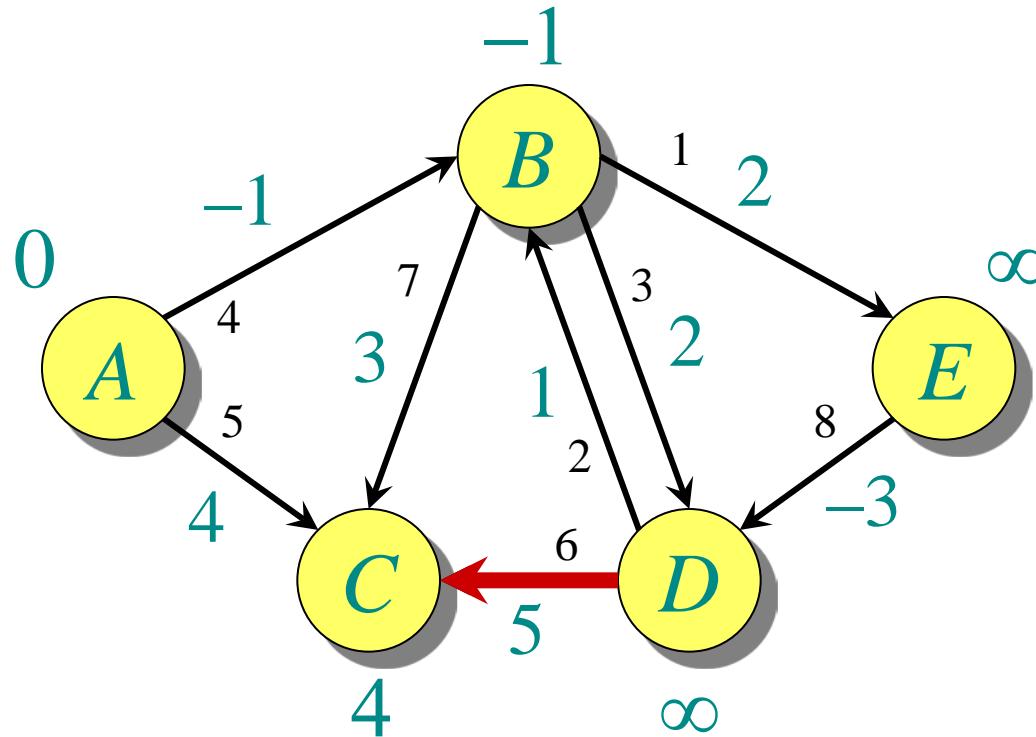


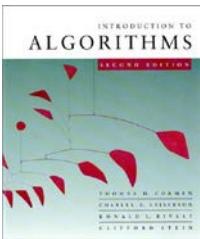
# Example of Bellman-Ford



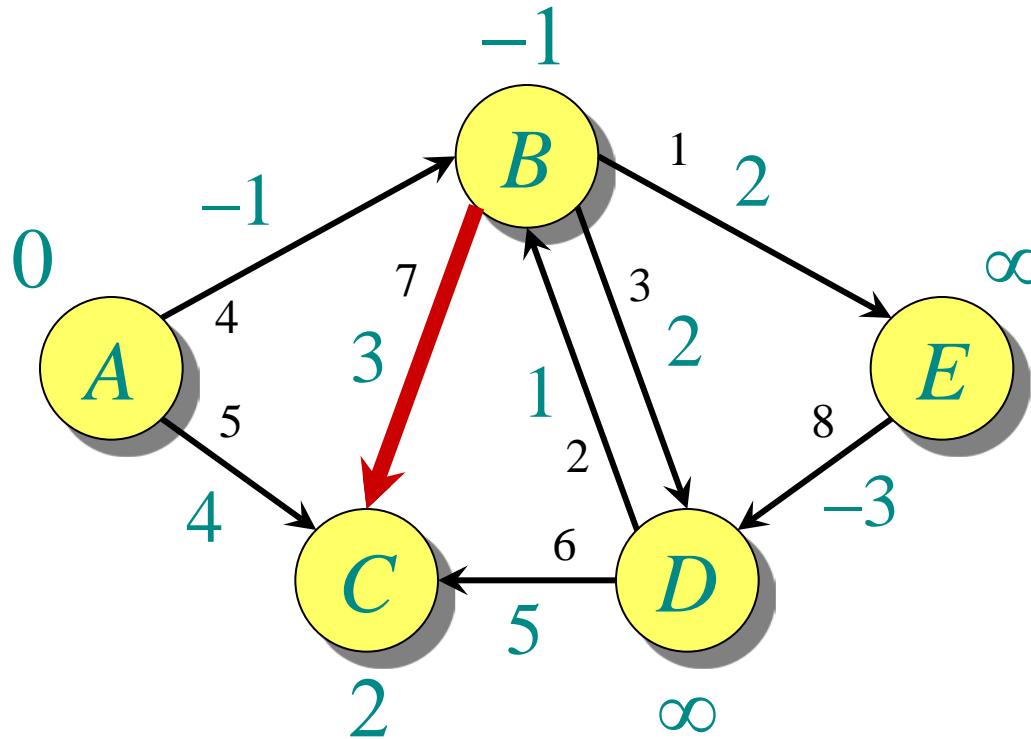


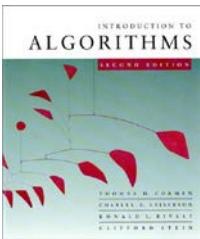
# Example of Bellman-Ford



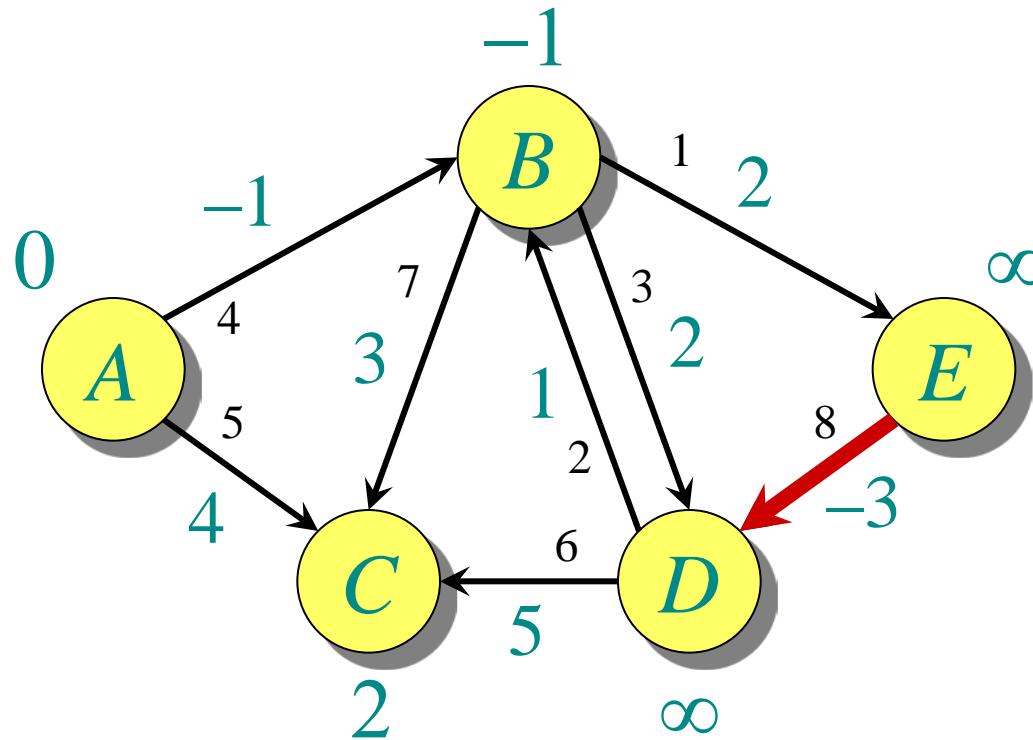


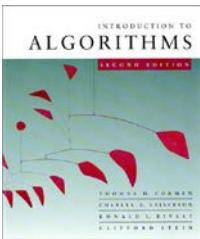
# Example of Bellman-Ford



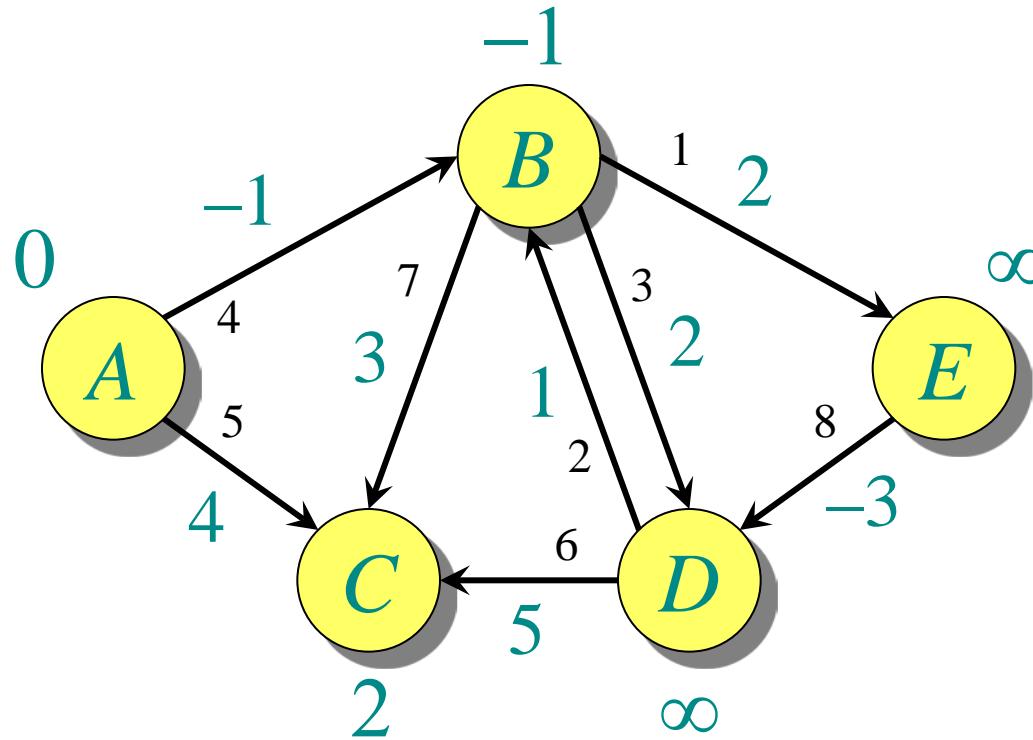


# Example of Bellman-Ford

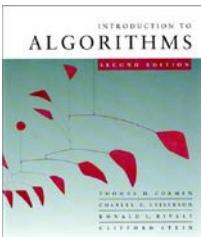




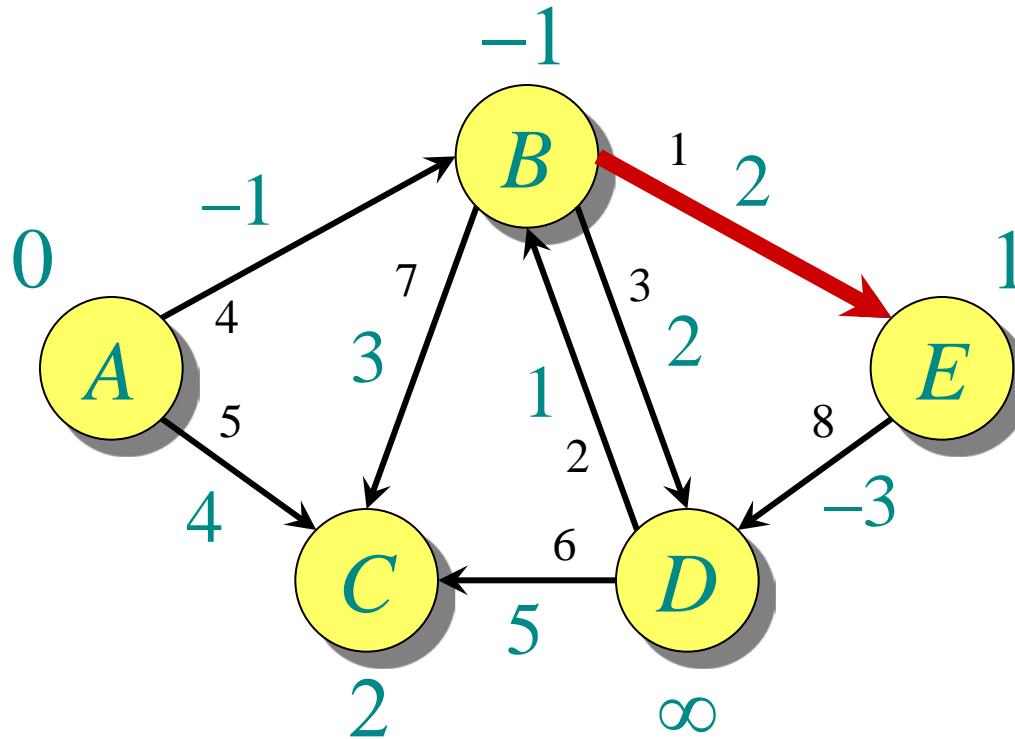
# Example of Bellman-Ford

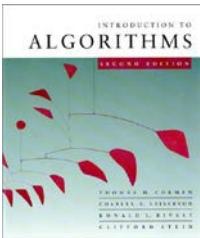


End of pass 1.

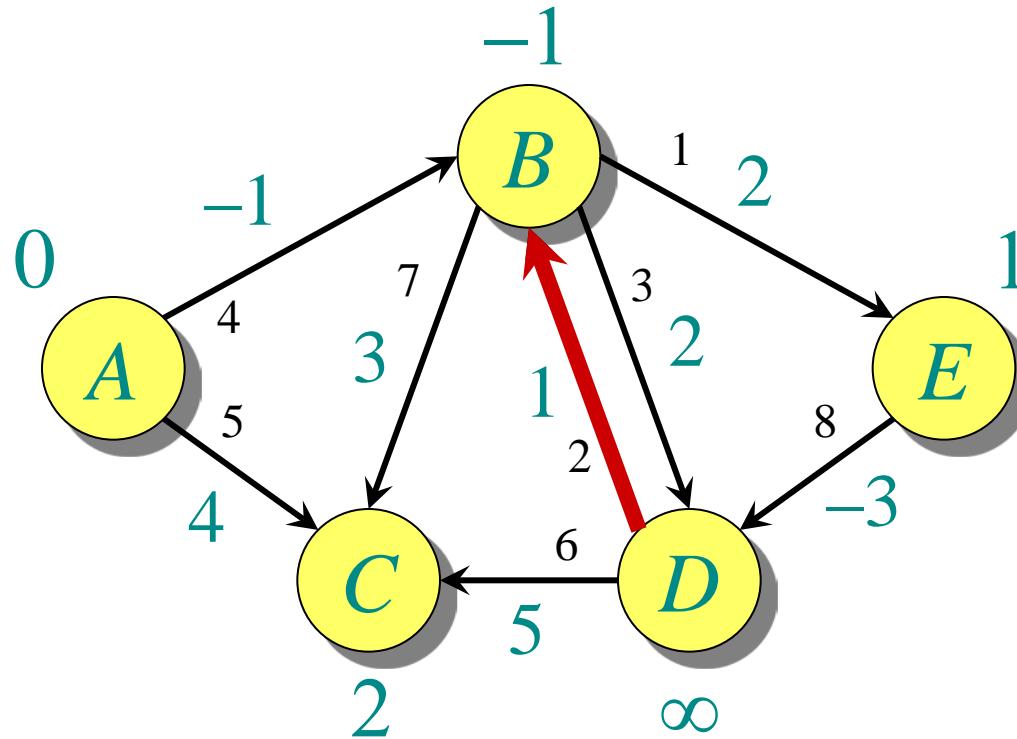


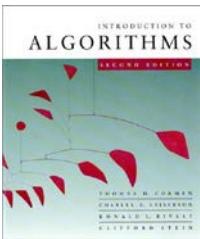
# Example of Bellman-Ford



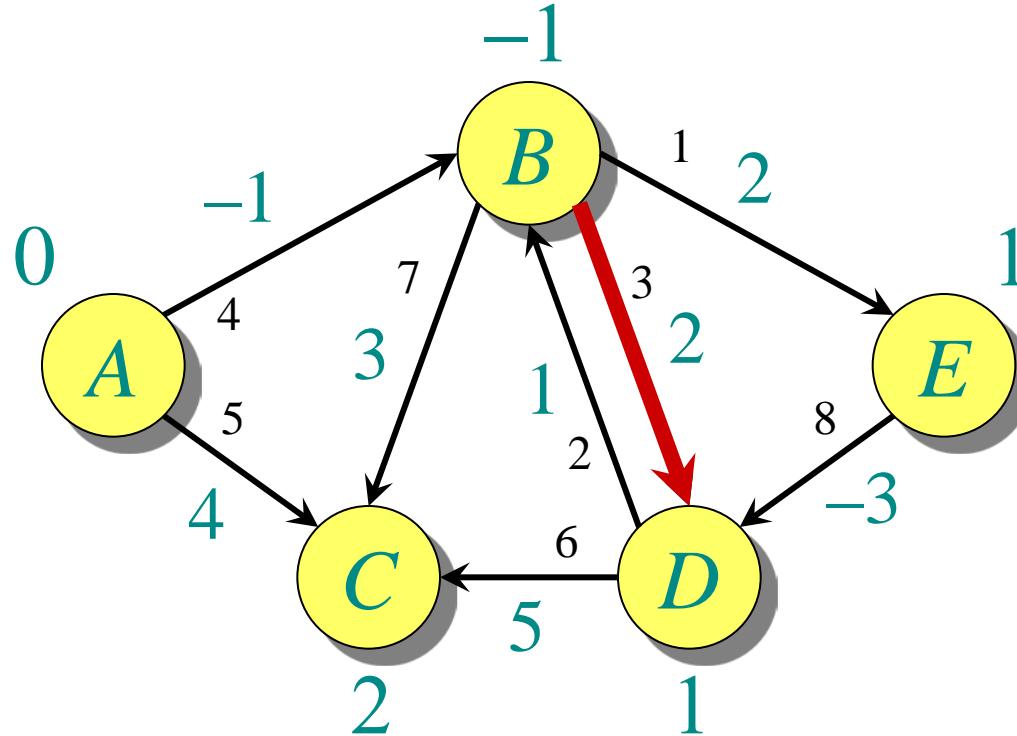


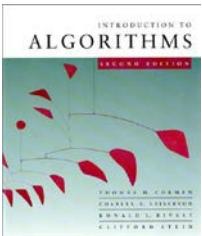
# Example of Bellman-Ford



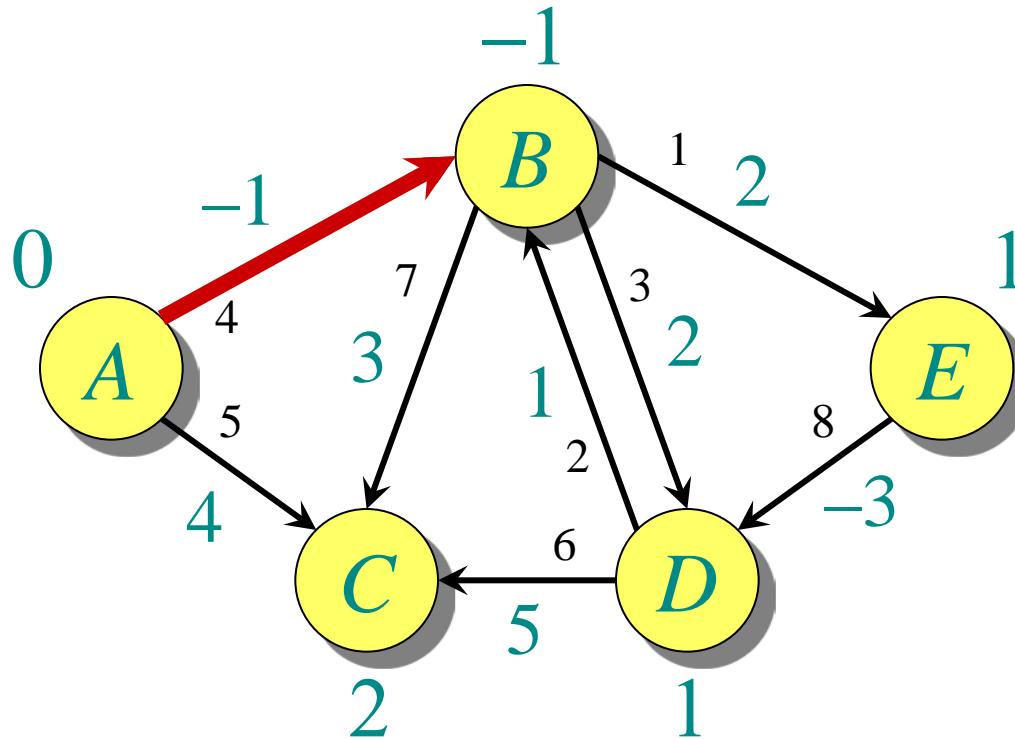


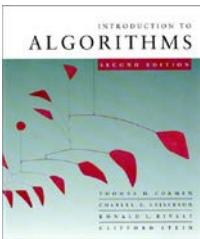
# Example of Bellman-Ford



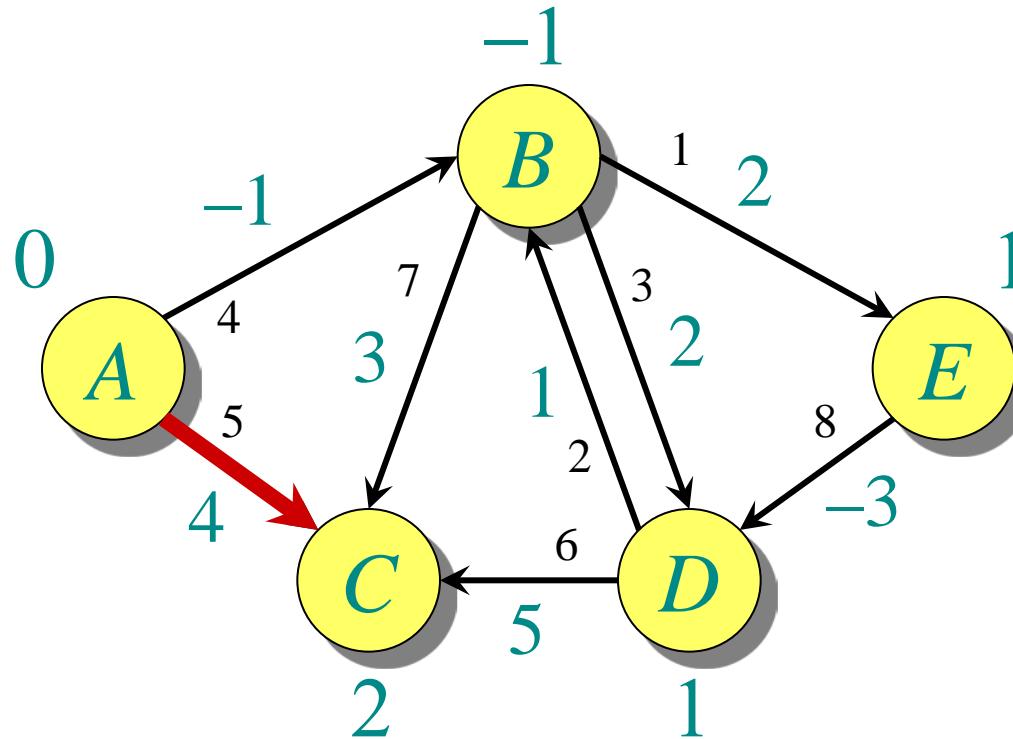


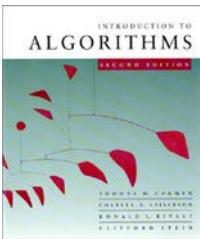
# Example of Bellman-Ford



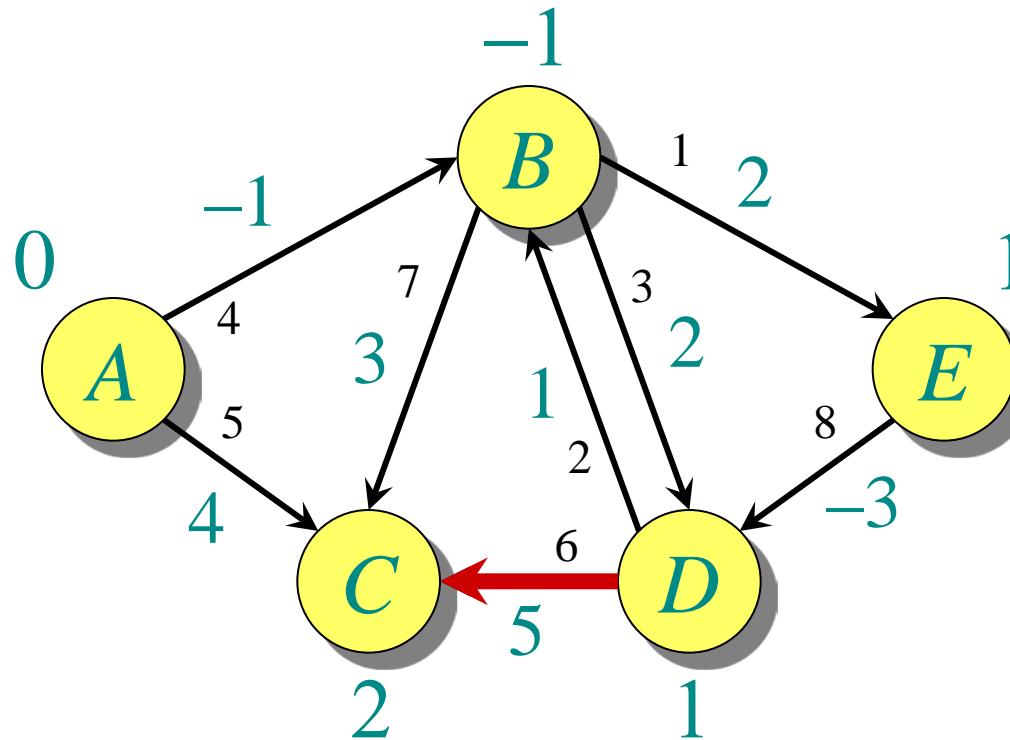


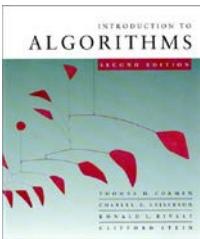
# Example of Bellman-Ford



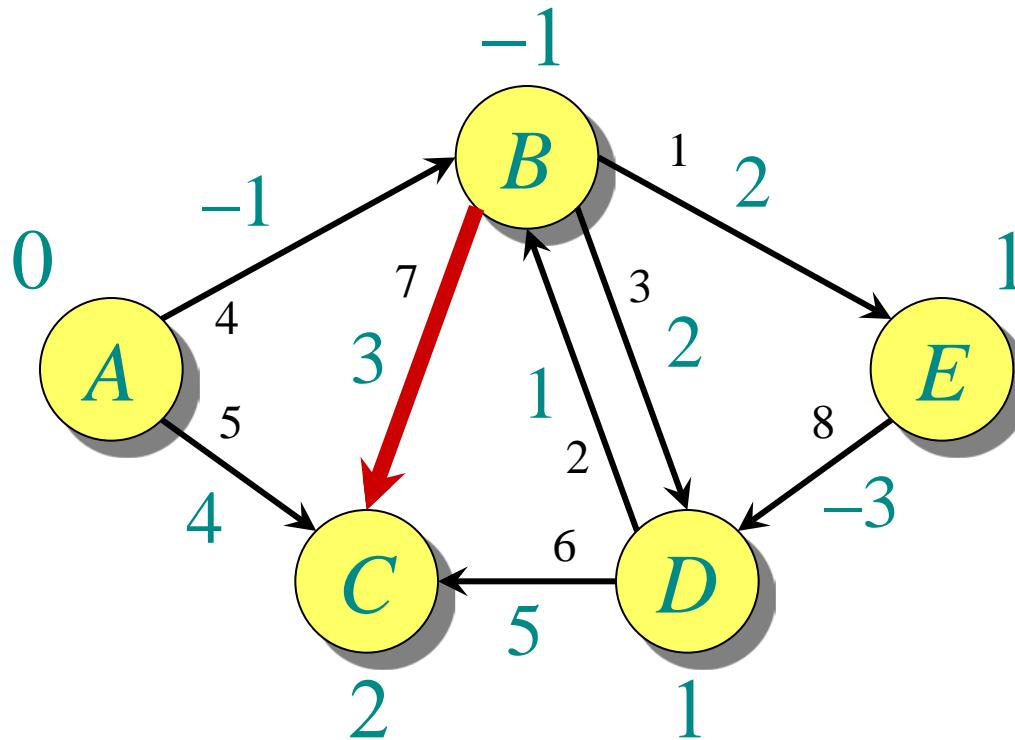


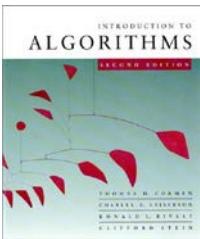
# Example of Bellman-Ford



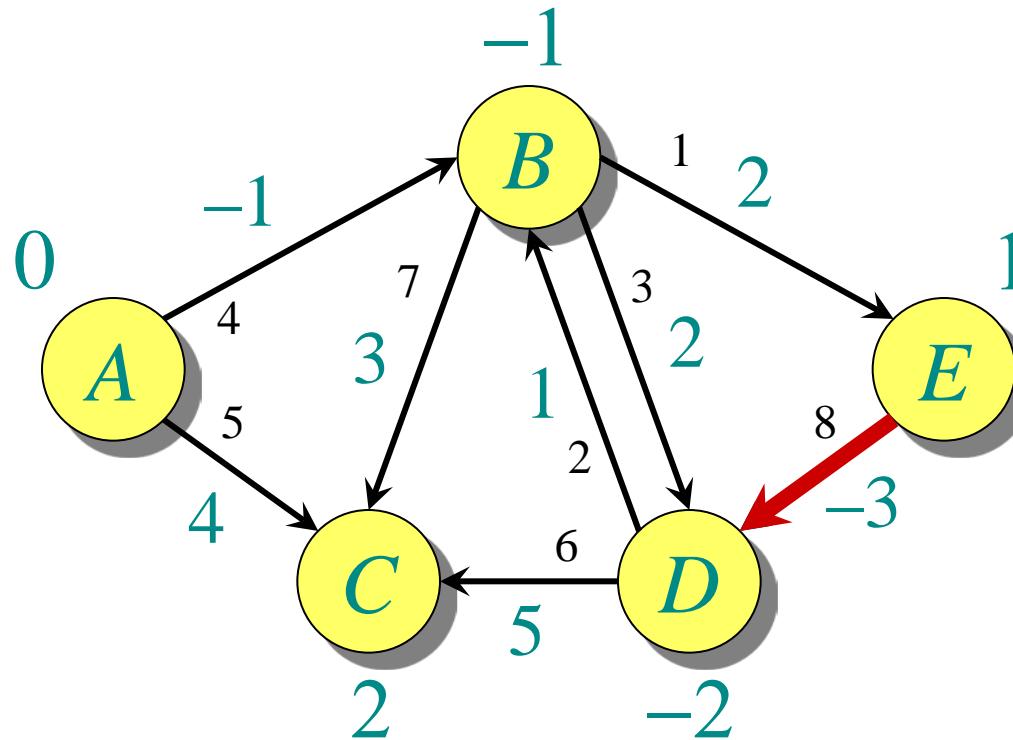


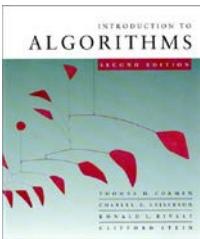
# Example of Bellman-Ford



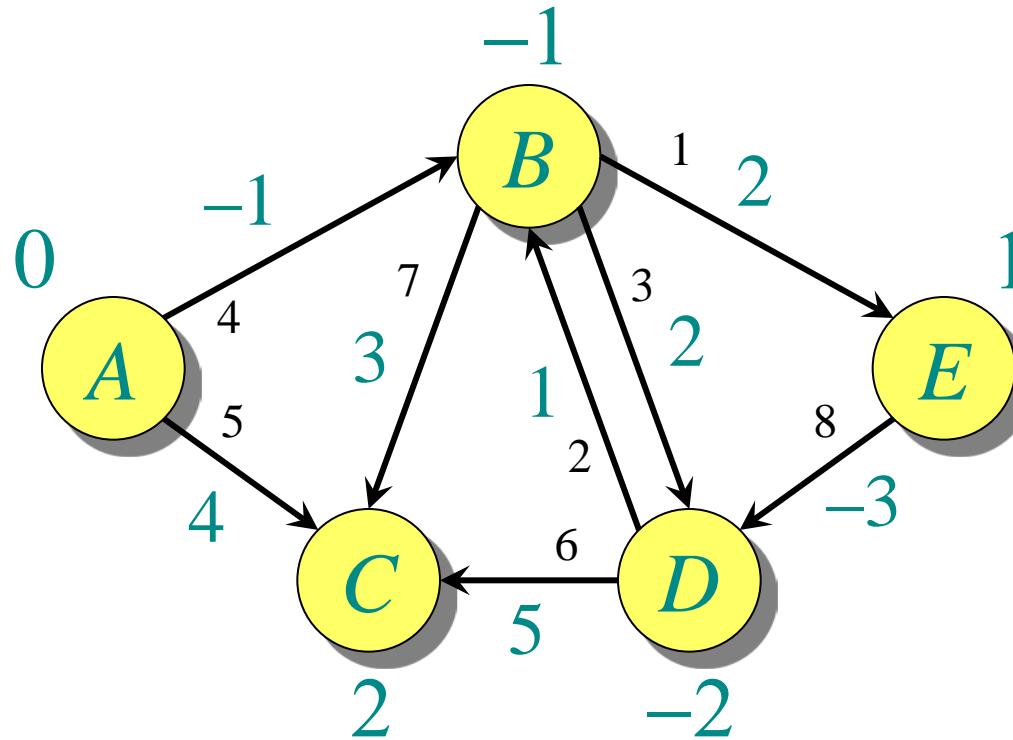


# Example of Bellman-Ford

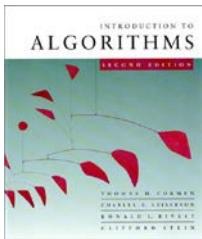




# Example of Bellman-Ford

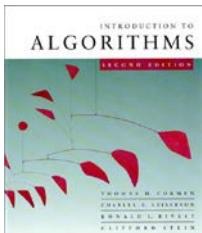


End of pass 2 (and 3 and 4).



# Correctness

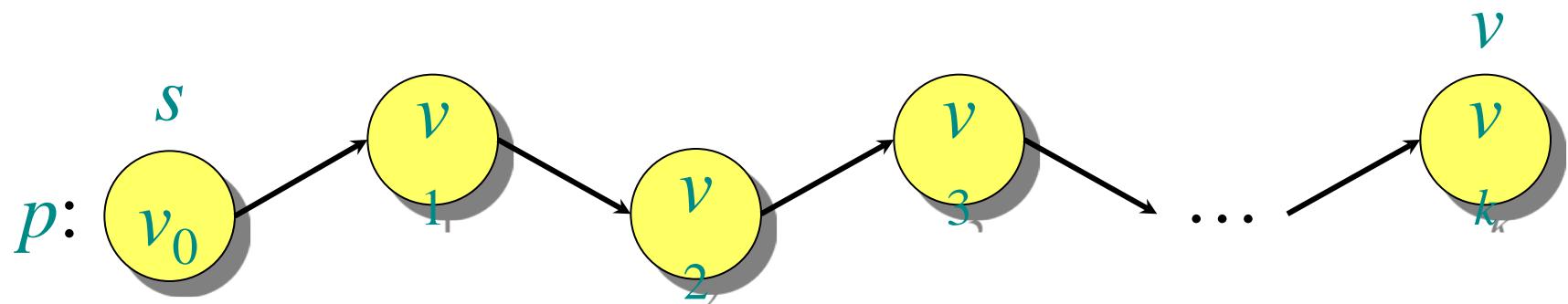
**Theorem.** If  $G = (V, E)$  contains no negative-weight cycles, then after the Bellman-Ford algorithm executes,  $d[v] = \delta(s, v)$  for all  $v \in V$ .



# Correctness

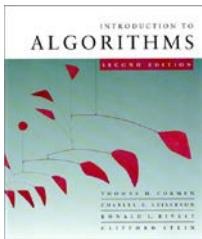
**Theorem.** If  $G = (V, E)$  contains no negative-weight cycles, then after the Bellman-Ford algorithm executes,  $d[v] = \delta(s, v)$  for all  $v \in V$ .

*Proof.* Let  $v \in V$  be any vertex, and consider a shortest path  $p$  from  $s$  to  $v$  with the minimum number of edges.

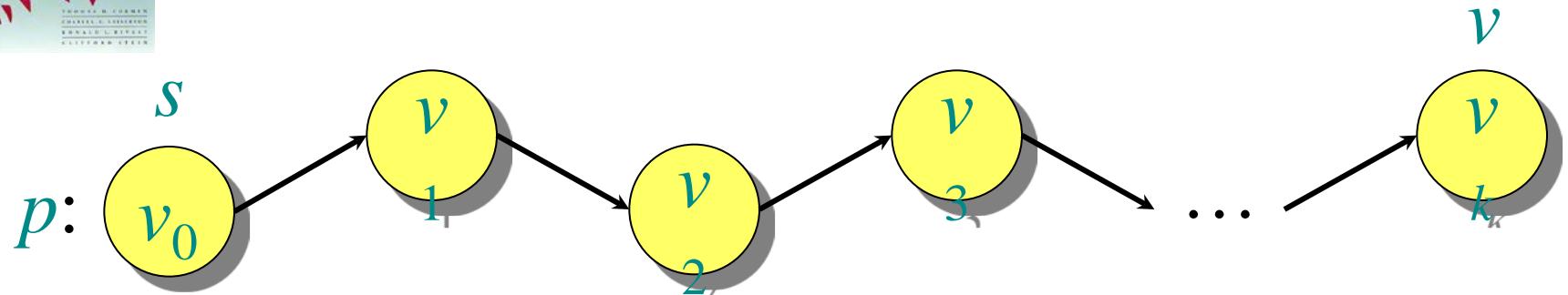


Since  $p$  is a shortest path, we have

$$\delta(s, v_i) = \delta(s, v_{i-1}) + w(v_{i-1}, v_i) .$$



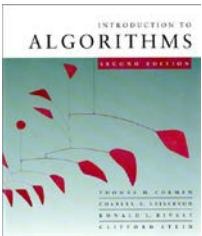
# Correctness (continued)



Initially,  $d[v_0] = 0 = \delta(s, v_0)$ , and  $d[v_0]$  is unchanged by subsequent relaxations (because of the lemma from *Shortest Paths I* that  $d[v] \geq \delta(s, v)$ ).

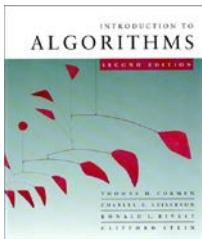
- After 1 pass through  $E$ , we have  $d[v_1] = \delta(s, v_1)$ .
- After 2 passes through  $E$ , we have  $d[v_2] = \delta(s, v_2)$ .
- $\vdots$
- After  $k$  passes through  $E$ , we have  $d[v_k] = \delta(s, v_k)$ .

Since  $G$  contains no negative-weight cycles,  $p$  is simple. Longest simple path has  $\leq |V| - 1$  edges. □



# Detection of negative-weight cycles

**Corollary.** If a value  $d[v]$  fails to converge after  $|V| - 1$  passes, there exists a negative-weight cycle in  $G$  reachable from  $s$ . □

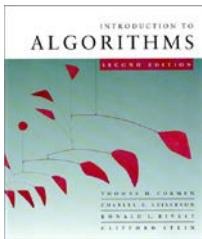


# Linear programming

Let  $A$  be an  $m \times n$  matrix,  $b$  be an  $m$ -vector, and  $c$  be an  $n$ -vector. Find an  $n$ -vector  $x$  that maximizes  $c^T x$  subject to  $Ax \leq b$ , or determine that no such solution exists.

$$\begin{matrix} & n \\ A & \cdot & x & \leq & b \end{matrix} \quad \text{maximizing} \quad \begin{matrix} c^T & \cdot & x \end{matrix}$$

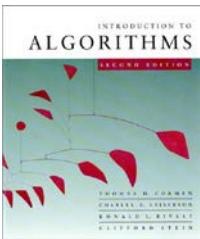
The diagram illustrates the linear programming problem. On the left, a pink rectangle labeled  $A$  represents an  $m \times n$  matrix, and a yellow vertical bar labeled  $x$  represents an  $n$ -vector. The inequality symbol  $\leq$  is positioned between them. To the right, a pink horizontal bar labeled  $c^T$  represents the transpose of the vector  $c$ , and another yellow vertical bar labeled  $x$  is shown, indicating the objective function to be maximized.



# Linear-programming algorithms

## Algorithms for the general problem

- Simplex methods — practical, but worst-case exponential time.
- Interior-point methods — polynomial time and competes with simplex.



# Linear-programming algorithms

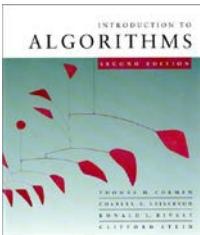
## Algorithms for the general problem

- Simplex methods — practical, but worst-case exponential time.
- Interior-point methods — polynomial time and competes with simplex.

***Feasibility problem:*** No optimization criterion.

Just find  $\mathbf{x}$  such that  $A\mathbf{x} \leq \mathbf{b}$ .

- In general, just as hard as ordinary LP.

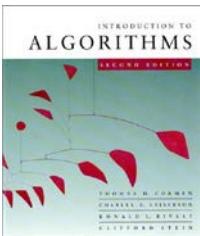


# Solving a system of difference constraints

Linear programming where each row of  $A$  contains exactly one  $1$ , one  $-1$ , and the rest  $0$ 's.

## Example:

$$\left. \begin{array}{l} x_1 - x_2 \leq 3 \\ x_2 - x_3 \leq -2 \\ x_1 - x_3 \leq 2 \end{array} \right\} x_j - x_i \leq w_{ij}$$



# Solving a system of difference constraints

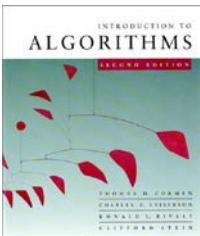
Linear programming where each row of  $A$  contains exactly one  $1$ , one  $-1$ , and the rest  $0$ 's.

**Example:**

$$\left. \begin{array}{l} x_1 - x_2 \leq 3 \\ x_2 - x_3 \leq -2 \\ x_1 - x_3 \leq 2 \end{array} \right\} \quad x_j - x_i \leq w_{ij}$$

**Solution:**

$$\begin{array}{ll} x_1 = 3 & \\ x_2 = 0 & \\ x_3 = 2 & \end{array}$$



# Solving a system of difference constraints

Linear programming where each row of  $A$  contains exactly one  $1$ , one  $-1$ , and the rest  $0$ 's.

**Example:**

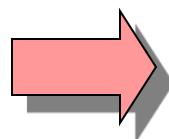
$$\left. \begin{array}{l} x_1 - x_2 \leq 3 \\ x_2 - x_3 \leq -2 \\ x_1 - x_3 \leq 2 \end{array} \right\} \quad x_j - x_i \leq w_{ij}$$

**Solution:**

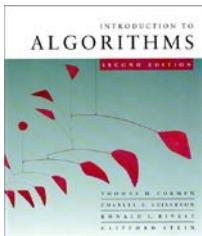
$$\begin{aligned} x_1 &= 3 \\ x_2 &= 0 \\ x_3 &= 2 \end{aligned}$$

**Constraint graph:**

$$x_j - x_i \leq w_{ij}$$

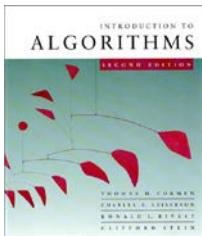


(The “ $A$ ” matrix has dimensions  $|E| \times |V|$ .)



# Unsatisfiable constraints

**Theorem.** If the constraint graph contains a negative-weight cycle, then the system of differences is unsatisfiable.



# Unsatisfiable constraints

**Theorem.** If the constraint graph contains a negative-weight cycle, then the system of differences is unsatisfiable.

*Proof.* Suppose that the negative-weight cycle is  $v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_k \rightarrow v_1$ . Then, we have

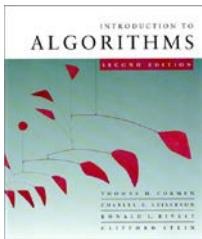
$$x_2 - x_1 \leq w_{12}$$

$$x_3 - x_2 \leq w_{23}$$

⋮

$$x_k - x_{k-1} \leq w_{k-1, k}$$

$$x_1 - x_k \leq w_{k1}$$



# Unsatisfiable constraints

**Theorem.** If the constraint graph contains a negative-weight cycle, then the system of differences is unsatisfiable.

*Proof.* Suppose that the negative-weight cycle is  $v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_k \rightarrow v_1$ . Then, we have

$$x_2 - x_1 \leq w_{12}$$

$$x_3 - x_2 \leq w_{23}$$

⋮

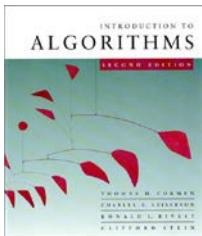
$$x_k - x_{k-1} \leq w_{k-1, k}$$

$$x_1 - x_k \leq w_{k1}$$

---

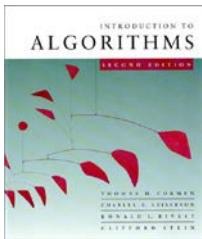
$$\begin{array}{rcl} 0 & \leq & \text{weight of cycle} \\ & & < 0 \end{array}$$

Therefore, no values for the  $x_i$  can satisfy the constraints. □



# Satisfying the constraints

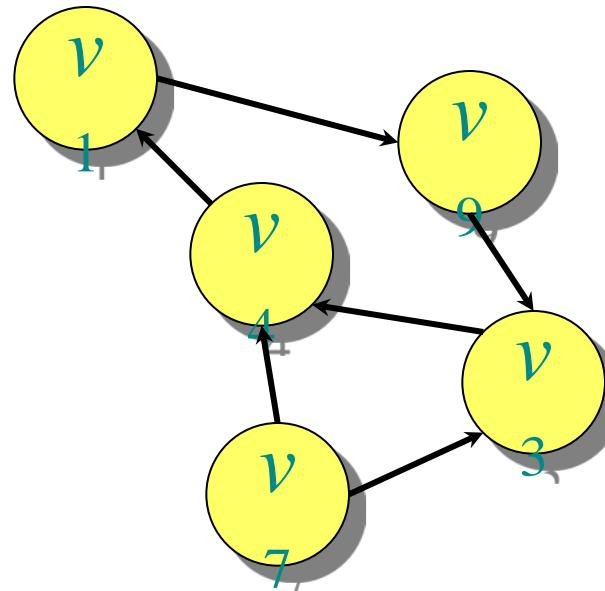
**Theorem.** Suppose no negative-weight cycle exists in the constraint graph. Then, the constraints are satisfiable.

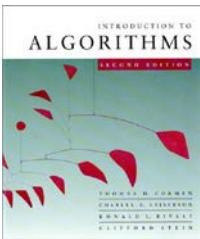


# Satisfying the constraints

**Theorem.** Suppose no negative-weight cycle exists in the constraint graph. Then, the constraints are satisfiable.

*Proof.* Add a new vertex  $s$  to  $V$  with a 0-weight edge to each vertex  $v_i \in V$ .

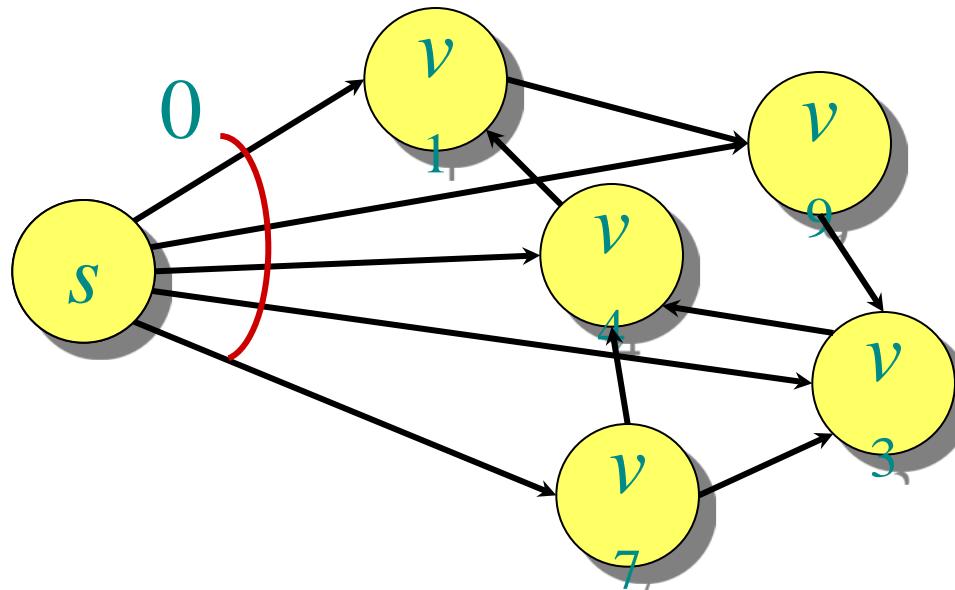




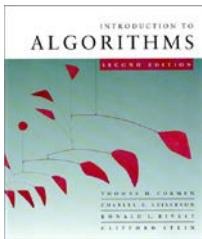
# Satisfying the constraints

**Theorem.** Suppose no negative-weight cycle exists in the constraint graph. Then, the constraints are satisfiable.

*Proof.* Add a new vertex  $s$  to  $V$  with a 0-weight edge to each vertex  $v_i \in V$ .



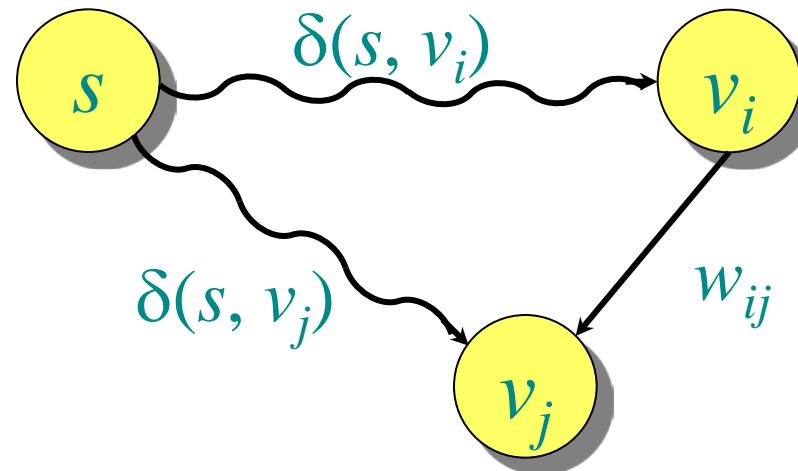
**Note:**  
No negative-weight cycles introduced  $\Rightarrow$  shortest paths exist.



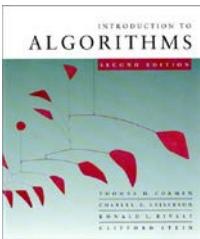
# Proof (continued)

**Claim:** The assignment  $x_i = \delta(s, v_i)$  solves the constraints.

Consider any constraint  $x_j - x_i \leq w_{ij}$ , and consider the shortest paths from  $s$  to  $v_j$  and  $v_i$ :



The triangle inequality gives us  $\delta(s, v_j) \leq \delta(s, v_i) + w_{ij}$ . Since  $x_i = \delta(s, v_i)$  and  $x_j = \delta(s, v_j)$ , the constraint  $x_j - x_i \leq w_{ij}$  is satisfied. □



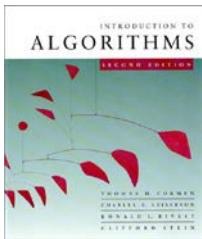
# Bellman-Ford and linear programming

**Corollary.** The Bellman-Ford algorithm can solve a system of  $m$  difference constraints on  $n$  variables in  $O(mn)$  time. □

Single-source shortest paths is a simple LP problem.

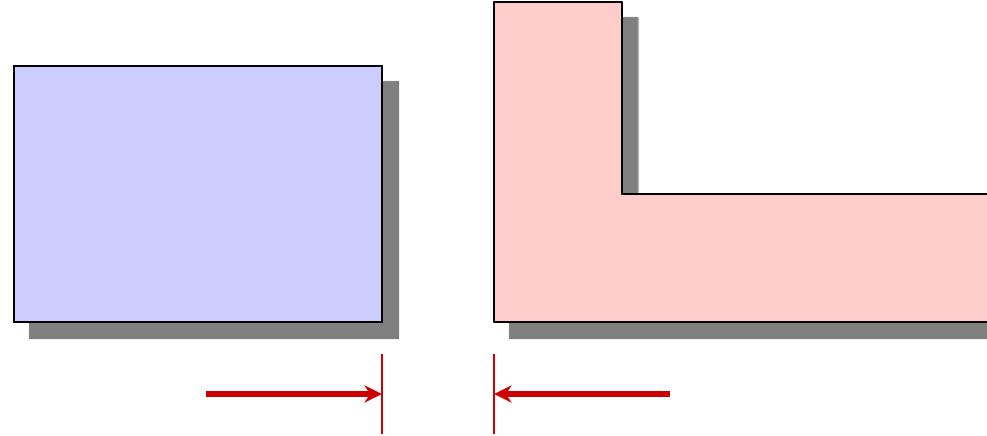
In fact, Bellman-Ford maximizes  $x_1 + x_2 + \dots + x_n$  subject to the constraints  $x_j - x_i \leq w_{ij}$  and  $x_i \leq 0$  (exercise).

Bellman-Ford also minimizes  $\max_i\{x_i\} - \min_i\{x_i\}$  (exercise).

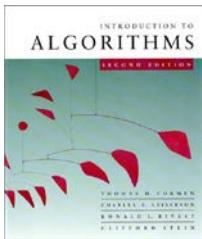


# Application to VLSI layout compaction

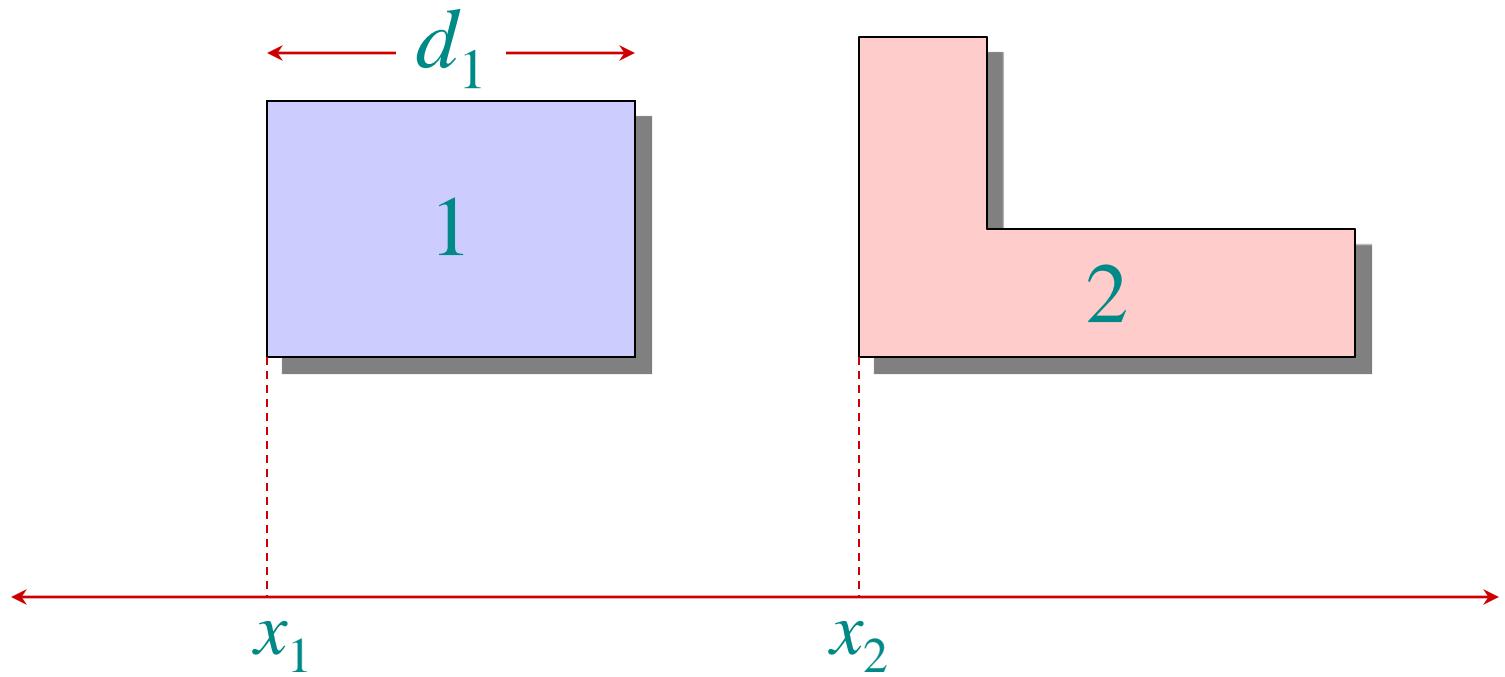
*Integrated  
-circuit  
features:*



**Problem:** Compact (in one dimension) the space between the features of a VLSI layout without bringing any features too close together.



# VLSI layout compaction

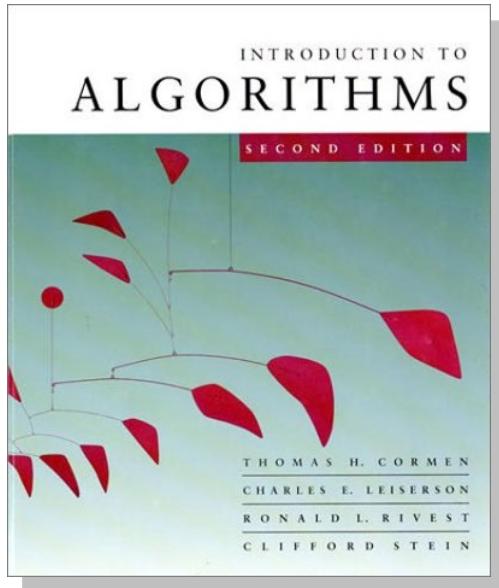


**Constraint:**  $x_2 - x_1 \geq d_1 + \lambda$

Bellman-Ford minimizes  $\max_i\{x_i\} - \min_i\{x_i\}$ , which compacts the layout in the  $x$ -dimension.

# *Introduction to Algorithms*

## 6.046J/18.401J

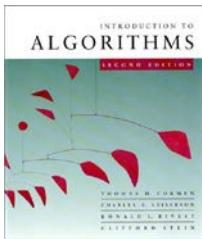


### LECTURE 16

#### Shortest Paths III

- All-pairs shortest paths
- Matrix-multiplication algorithm
- Floyd-Warshall algorithm
- Johnson's algorithm

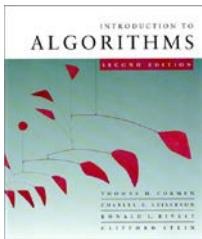
Prof. Erik D. Demaine



# Shortest paths

## Single-source shortest paths

- Nonnegative edge weights
  - ◆ Dijkstra's algorithm:  $O(E + V \lg V)$
- General
  - ◆ Bellman-Ford algorithm:  $O(VE)$
- DAG
  - ◆ One pass of Bellman-Ford:  $O(V + E)$



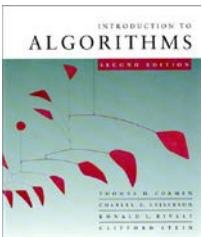
# Shortest paths

## Single-source shortest paths

- Nonnegative edge weights
  - ◆ Dijkstra's algorithm:  $O(E + V \lg V)$
- General
  - ◆ Bellman-Ford algorithm:  $O(VE)$
- DAG
  - ◆ One pass of Bellman-Ford:  $O(V + E)$

## All-pairs shortest paths

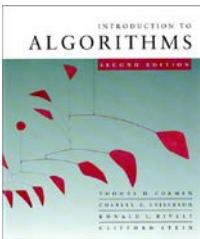
- Nonnegative edge weights
  - ◆ Dijkstra's algorithm  $|V|$  times:  $O(VE + V^2 \lg V)$
- General
  - ◆ Three algorithms today.



# All-pairs shortest paths

**Input:** Digraph  $G = (V, E)$ , where  $V = \{1, 2, \dots, n\}$ , with edge-weight function  $w : E \rightarrow \mathbb{R}$ .

**Output:**  $n \times n$  matrix of shortest-path lengths  $\delta(i, j)$  for all  $i, j \in V$ .



# All-pairs shortest paths

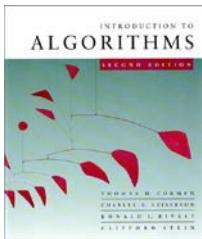
**Input:** Digraph  $G = (V, E)$ , where  $V = \{1, 2, \dots, n\}$ , with edge-weight function  $w : E \rightarrow \mathbb{R}$ .

**Output:**  $n \times n$  matrix of shortest-path lengths  $\delta(i, j)$  for all  $i, j \in V$ .

## IDEA:

- Run Bellman-Ford once from each vertex.
- Time =  $O(V^2E)$ .
- Dense graph ( $\Theta(n^2)$  edges)  $\Rightarrow \Theta(n^4)$  time in the worst case.

*Good first try!*



# Dynamic programming

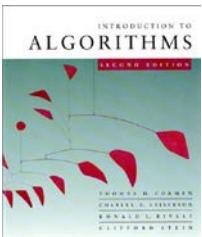
Consider the  $n \times n$  weighted adjacency matrix  $A = (a_{ij})$ , where  $a_{ij} = w(i, j)$  or  $\infty$ , and define  $d_{ij}^{(m)}$  = weight of a shortest path from  $i$  to  $j$  that uses at most  $m$  edges.

**Claim:** We have

$$d_{ij}^{(0)} = \begin{cases} 0 & \text{if } i = j, \\ \infty & \text{if } i \neq j; \end{cases}$$

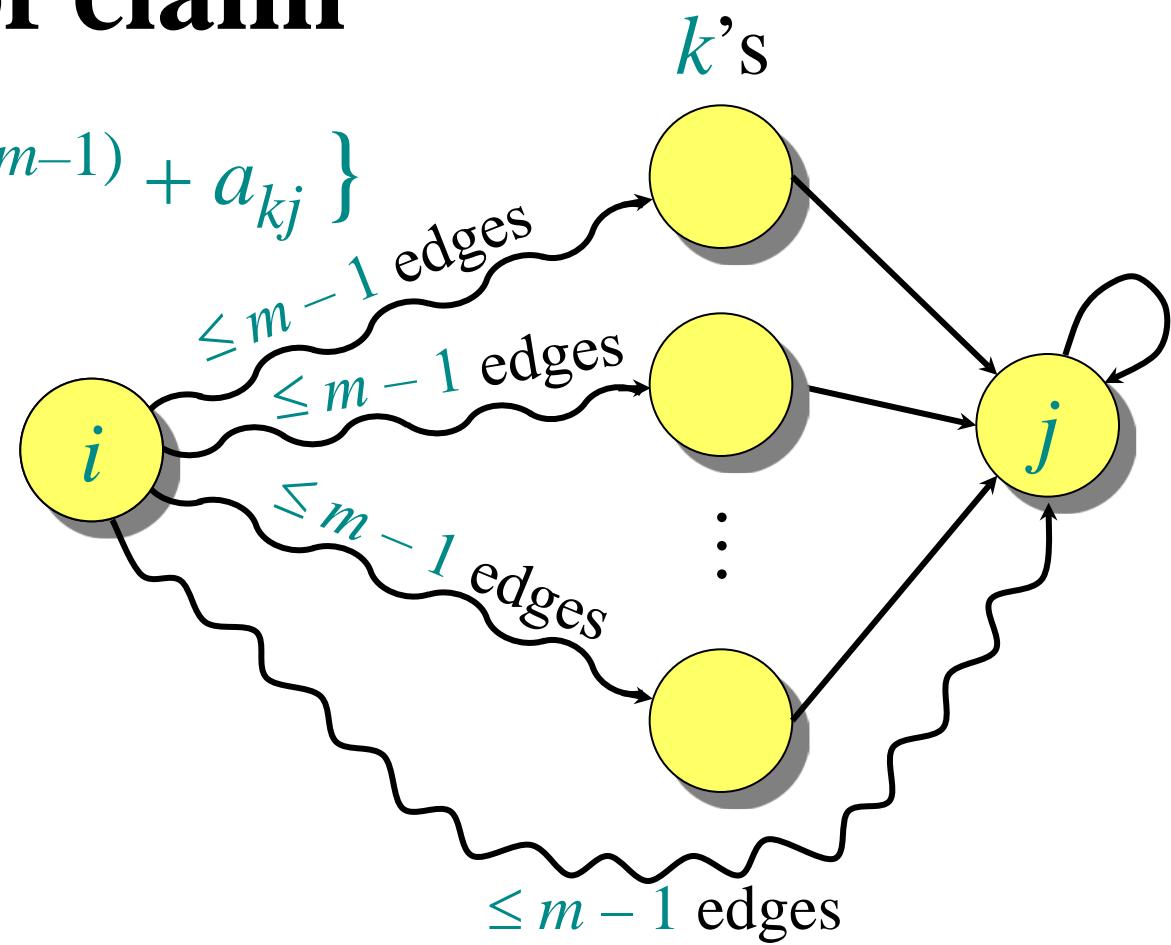
and for  $m = 1, 2, \dots, n - 1$ ,

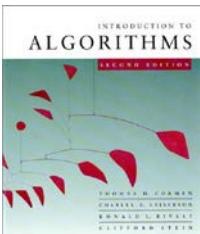
$$d_{ij}^{(m)} = \min_k \{ d_{ik}^{(m-1)} + a_{kj} \}.$$



# Proof of claim

$$d_{ij}^{(m)} = \min_k \{ d_{ik}^{(m-1)} + a_{kj} \}$$





# Proof of claim

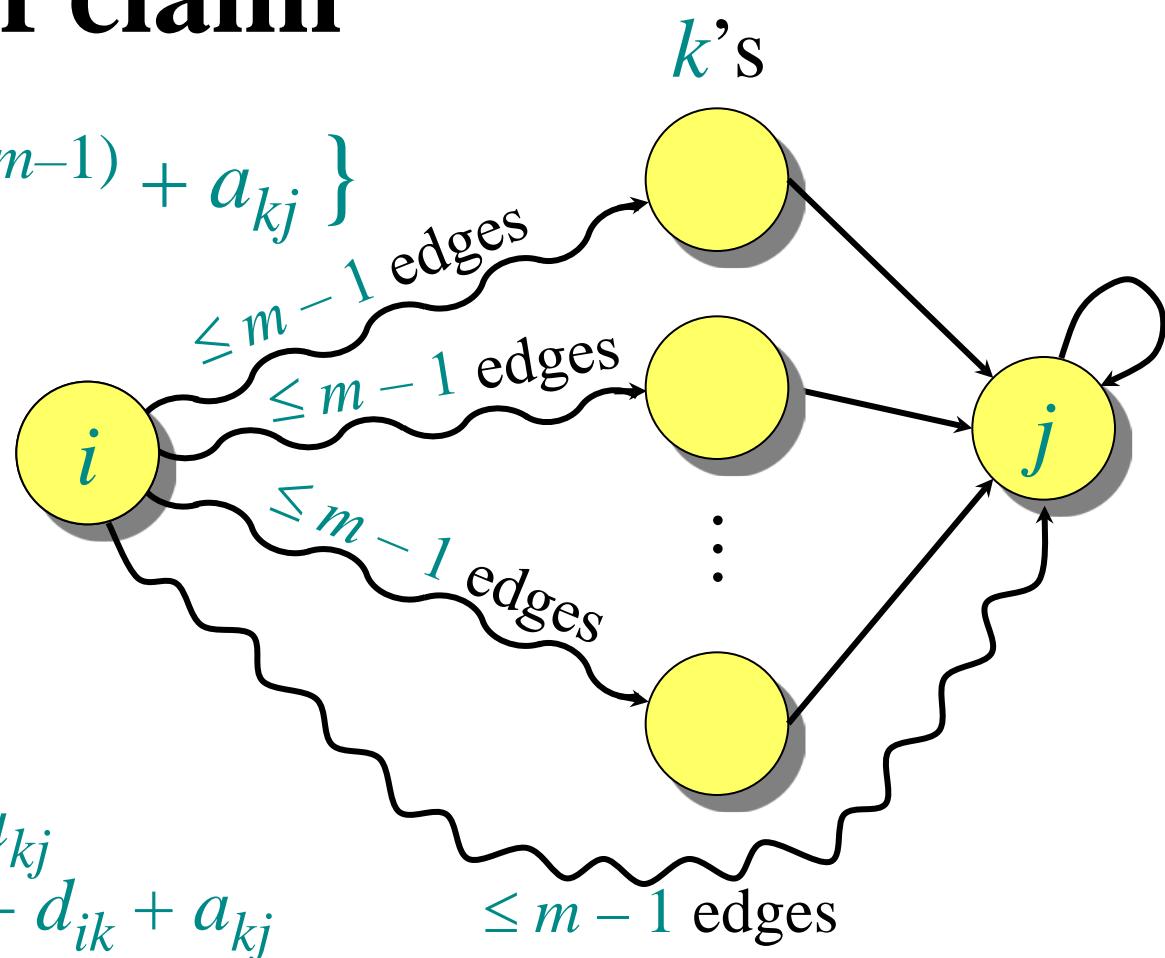
$$d_{ij}^{(m)} = \min_k \{ d_{ik}^{(m-1)} + a_{kj} \}$$

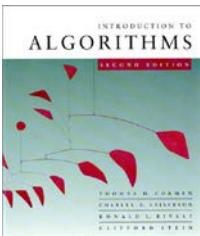
Relaxation!

for  $k \leftarrow 1$  to  $n$

do if  $d_{ij} > d_{ik} + a_{kj}$

then  $d_{ij} \leftarrow d_{ik} + a_{kj}$





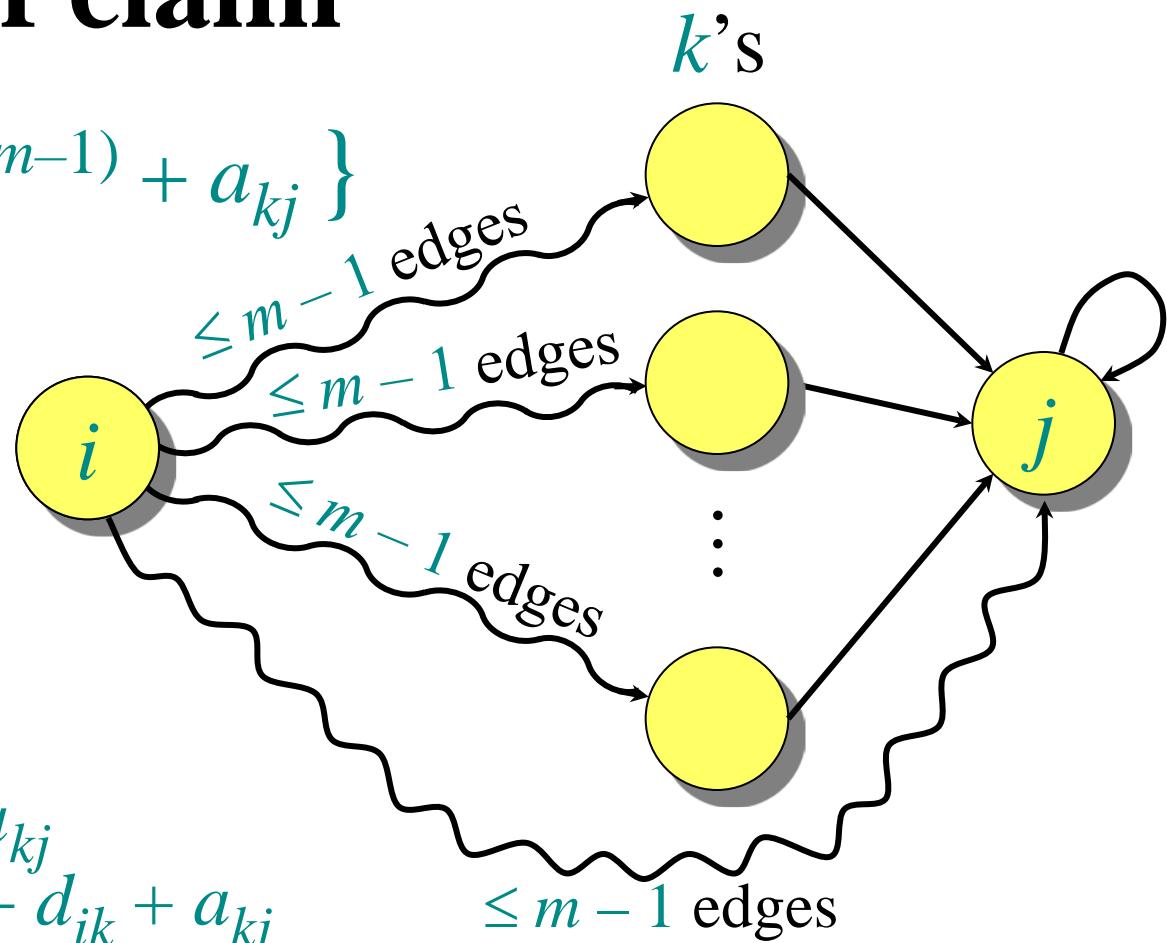
# Proof of claim

$$d_{ij}^{(m)} = \min_k \{ d_{ik}^{(m-1)} + a_{kj} \}$$

Relaxation!

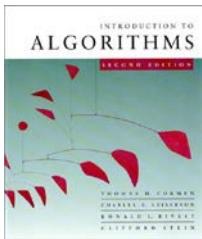
for  $k \leftarrow 1$  to  $n$

do if  $d_{ij} > d_{ik} + a_{kj}$   
then  $d_{ij} \leftarrow d_{ik} + a_{kj}$



**Note:** No negative-weight cycles implies

$$\delta(i, j) = d_{ij}^{(n-1)} = d_{ij}^{(n)} = d_{ij}^{(n+1)} = \dots$$

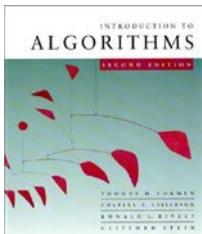


# Matrix multiplication

Compute  $C = A \cdot B$ , where  $C$ ,  $A$ , and  $B$  are  $n \times n$  matrices:

$$c_{ij} = \sum_{k=1}^n a_{ik} b_{kj}.$$

Time =  $\Theta(n^3)$  using the standard algorithm.



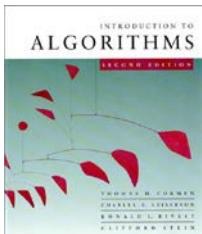
# Matrix multiplication

Compute  $C = A \cdot B$ , where  $C, A$ , and  $B$  are  $n \times n$  matrices:

$$c_{ij} = \sum_{k=1}^n a_{ik} b_{kj}.$$

Time =  $\Theta(n^3)$  using the standard algorithm.

What if we map “+”  $\rightarrow$  “min” and “.”  $\rightarrow$  “+”?



# Matrix multiplication

Compute  $C = A \cdot B$ , where  $C, A$ , and  $B$  are  $n \times n$  matrices:

$$c_{ij} = \sum_{k=1}^n a_{ik} b_{kj}.$$

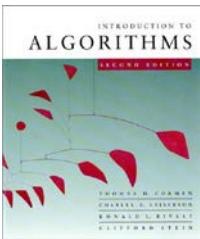
Time =  $\Theta(n^3)$  using the standard algorithm.

What if we map “+”  $\rightarrow$  “min” and “.”  $\rightarrow$  “+”?

$$c_{ij} = \min_k \{a_{ik} + b_{kj}\}.$$

Thus,  $D^{(m)} = D^{(m-1)} \times A$ .

Identity matrix =  $I = \begin{pmatrix} 0 & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty \\ \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & 0 \end{pmatrix} = D^0 = (d_{ij}^{(0)}).$



# Matrix multiplication (continued)

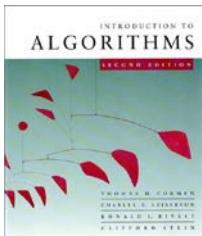
The  $(\min, +)$  multiplication is *associative*, and with the real numbers, it forms an algebraic structure called a *closed semiring*.

Consequently, we can compute

$$\begin{aligned} D^{(1)} &= D^{(0)} \cdot A = A^1 \\ D^{(2)} &= D^{(1)} \cdot A = A^2 \\ &\vdots && \vdots \\ D^{(n-1)} &= D^{(n-2)} \cdot A = A^{n-1}, \end{aligned}$$

yielding  $D^{(n-1)} = (\delta(i, j))$ .

Time =  $\Theta(n \cdot n^3) = \Theta(n^4)$ . No better than  $n \times$  B-F.



# Improved matrix multiplication algorithm

**Repeated squaring:**  $A^{2k} = A^k \times A^k$ .

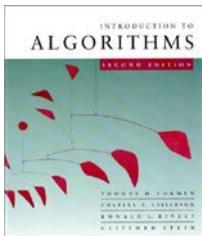
Compute  $\underbrace{A^2, A^4, \dots, A^{2^{\lceil \lg(n-1) \rceil}}}$ .

$O(\lg n)$  squarings

**Note:**  $A^{n-1} = A^n = A^{n+1} = \dots$ .

Time =  $\Theta(n^3 \lg n)$ .

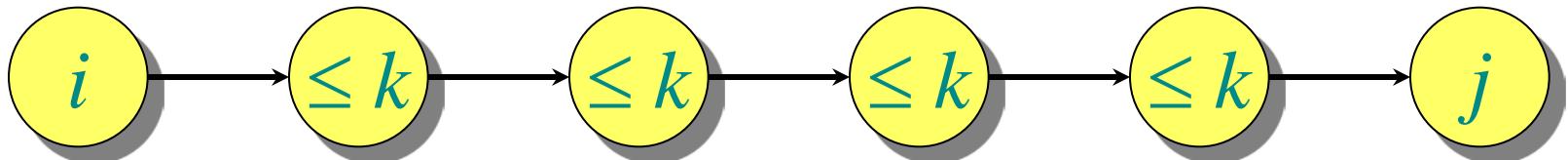
To detect negative-weight cycles, check the diagonal for negative values in  $O(n)$  additional time.



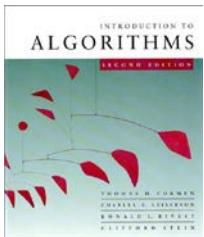
# Floyd-Warshall algorithm

*Also dynamic programming, but faster!*

Define  $c_{ij}^{(k)}$  = weight of a shortest path from  $i$  to  $j$  with intermediate vertices belonging to the set  $\{1, 2, \dots, k\}$ .

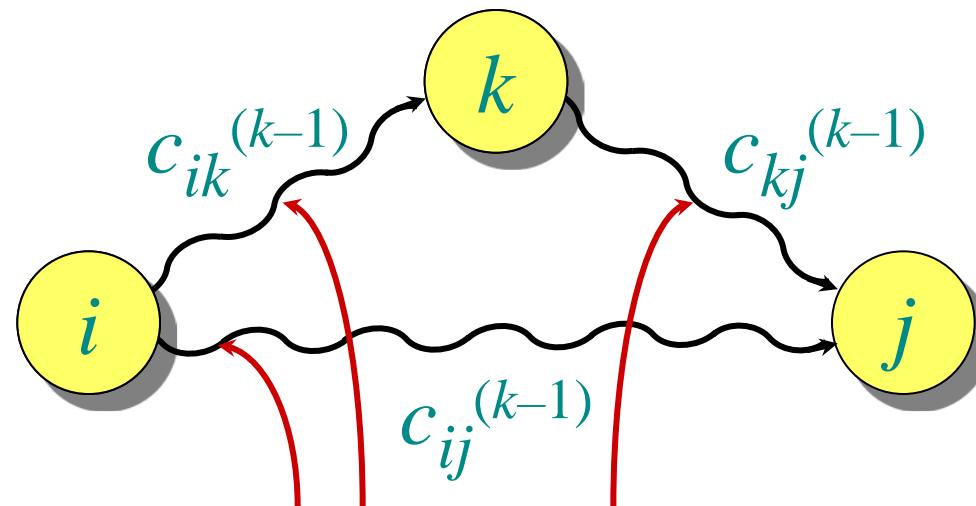


Thus,  $\delta(i, j) = c_{ij}^{(n)}$ . Also,  $c_{ij}^{(0)} = a_{ij}$ .

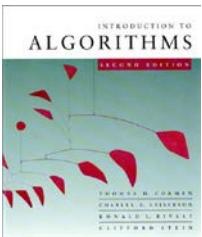


# Floyd-Warshall recurrence

$$c_{ij}^{(k)} = \min \{ c_{ij}^{(k-1)}, c_{ik}^{(k-1)} + c_{kj}^{(k-1)} \}$$



intermediate vertices in  $\{1, 2, \dots, k - 1\}$

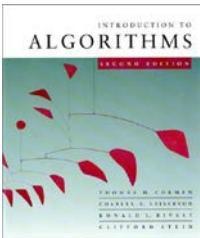


# Pseudocode for Floyd-Warshall

```
for  $k \leftarrow 1$  to  $n$ 
    do for  $i \leftarrow 1$  to  $n$ 
        do for  $j \leftarrow 1$  to  $n$ 
            do if  $c_{ij} > c_{ik} + c_{kj}$ 
                then  $c_{ij} \leftarrow c_{ik} + c_{kj}$  } relaxation
```

## Notes:

- Okay to omit superscripts, since extra relaxations can't hurt.
- Runs in  $\Theta(n^3)$  time.
- Simple to code.
- Efficient in practice.



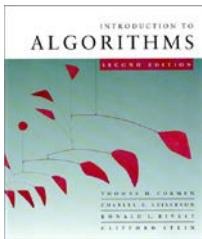
# Transitive closure of a directed graph

Compute  $t_{ij} = \begin{cases} 1 & \text{if there exists a path from } i \text{ to } j, \\ 0 & \text{otherwise.} \end{cases}$

**IDEA:** Use Floyd-Warshall, but with  $(\vee, \wedge)$  instead of  $(\min, +)$ :

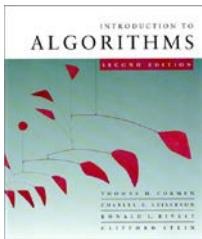
$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}).$$

Time =  $\Theta(n^3)$ .



# Graph reweighting

**Theorem.** Given a function  $h : V \rightarrow \mathbb{R}$ , *reweight* each edge  $(u, v) \in E$  by  $w_h(u, v) = w(u, v) + h(u) - h(v)$ . Then, for any two vertices, all paths between them are reweighted by the same amount.



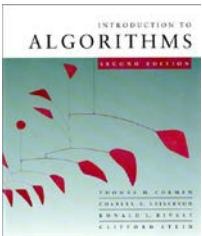
# Graph reweighting

**Theorem.** Given a function  $h : V \rightarrow \mathbb{R}$ , *reweight* each edge  $(u, v) \in E$  by  $w_h(u, v) = w(u, v) + h(u) - h(v)$ . Then, for any two vertices, all paths between them are reweighted by the same amount.

**Proof.** Let  $p = v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_k$  be a path in  $G$ . We have

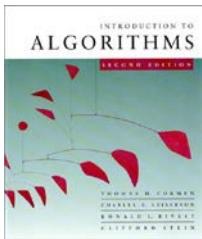
$$\begin{aligned} w_h(p) &= \sum_{i=1}^{k-1} w_h(v_i, v_{i+1}) \\ &= \sum_{i=1}^{k-1} (w(v_i, v_{i+1}) + h(v_i) - h(v_{i+1})) \\ &= \sum_{i=1}^{k-1} w(v_i, v_{i+1}) + h(v_1) - h(v_k) \\ &= w(p) + h(v_1) - h(v_k). \end{aligned}$$

*Same amount!*



# Shortest paths in reweighted graphs

**Corollary.**  $\delta_h(u, v) = \delta(u, v) + h(u) - h(v)$ . □

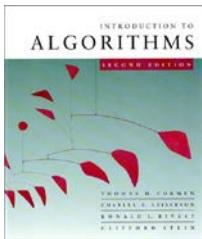


# Shortest paths in reweighted graphs

**Corollary.**  $\delta_h(u, v) = \delta(u, v) + h(u) - h(v)$ . □

**IDEA:** Find a function  $h : V \rightarrow \mathbb{R}$  such that  $w_h(u, v) \geq 0$  for all  $(u, v) \in E$ . Then, run Dijkstra's algorithm from each vertex on the reweighted graph.

**NOTE:**  $w_h(u, v) \geq 0$  iff  $h(v) - h(u) \leq w(u, v)$ .



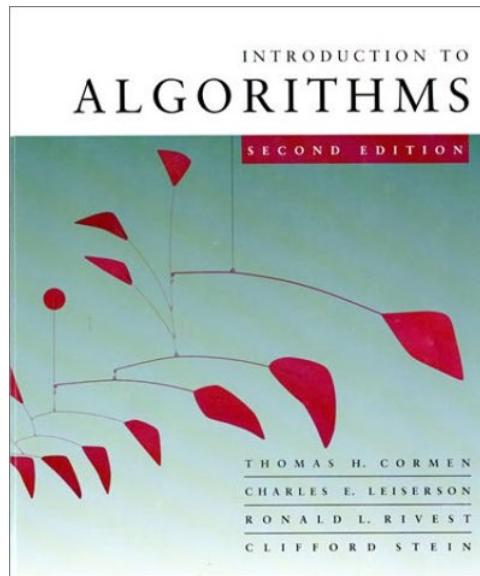
# Johnson's algorithm

1. Find a function  $h : V \rightarrow \mathbb{R}$  such that  $w_h(u, v) \geq 0$  for all  $(u, v) \in E$  by using Bellman-Ford to solve the difference constraints  $h(v) - h(u) \leq w(u, v)$ , or determine that a negative-weight cycle exists.
  - Time =  $O(VE)$ .
2. Run Dijkstra's algorithm using  $w_h$  from each vertex  $u \in V$  to compute  $\delta_h(u, v)$  for all  $v \in V$ .
  - Time =  $O(VE + V^2 \lg V)$ .
3. For each  $(u, v) \in V \times V$ , compute
$$\delta(u, v) = \delta_h(u, v) - h(u) + h(v) .$$
  - Time =  $O(V^2)$ .

Total time =  $O(VE + V^2 \lg V)$ .

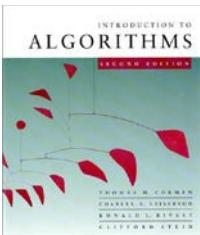
# *Introduction to Algorithms*

## 6.046J/18.401J/SMA5503



## *Lecture 12*

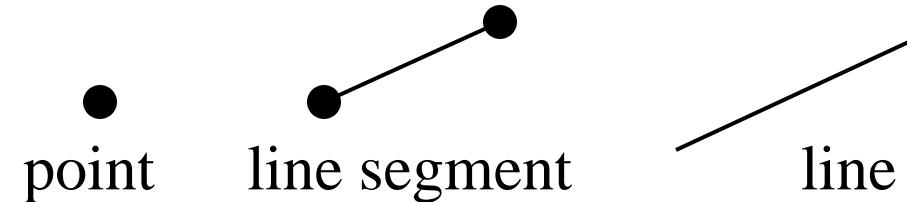
Prof. Erik Demaine



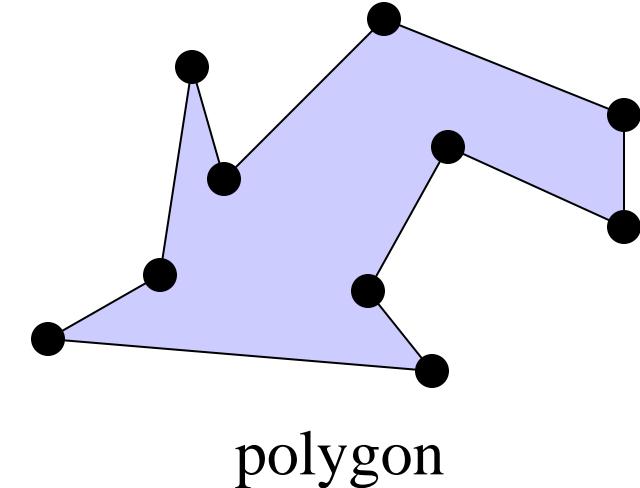
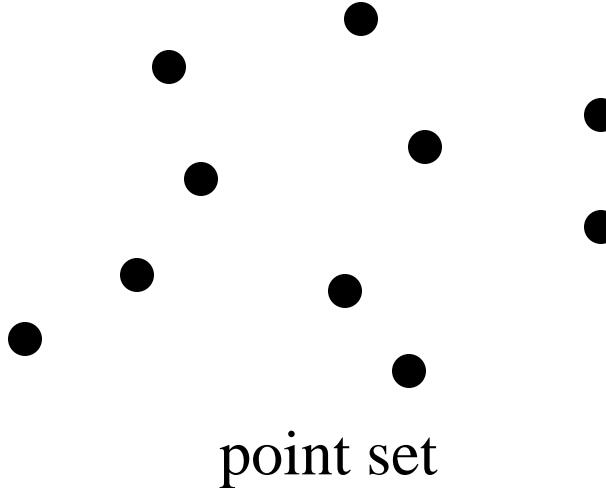
# Computational geometry

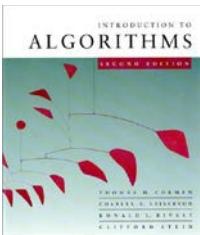
Algorithms for solving “geometric problems”  
in 2D and higher.

Fundamental objects:



Basic structures:

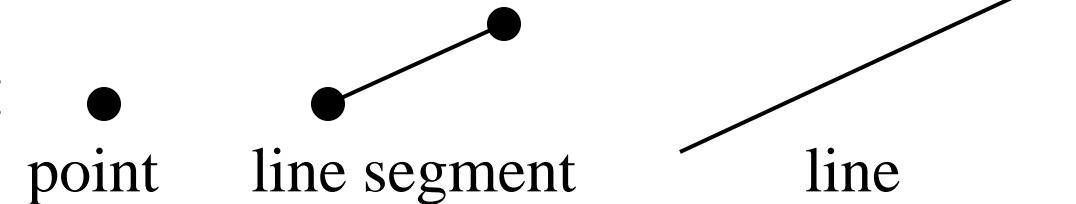




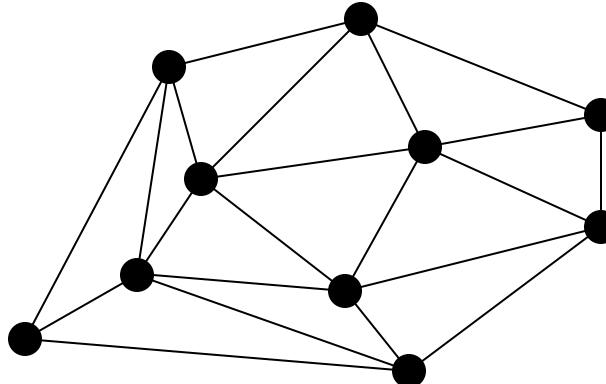
# Computational geometry

Algorithms for solving “geometric problems”  
in 2D and higher.

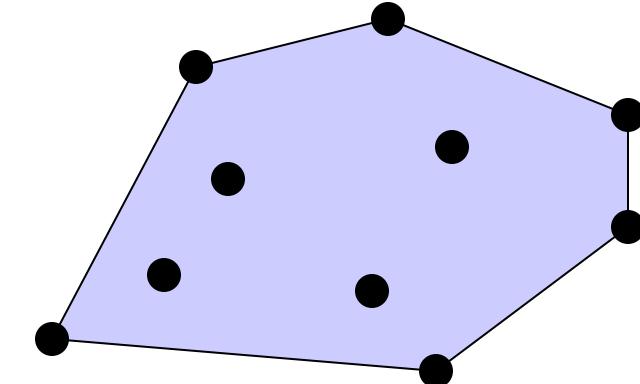
Fundamental objects:



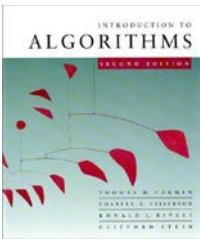
Basic structures:



triangulation



convex hull



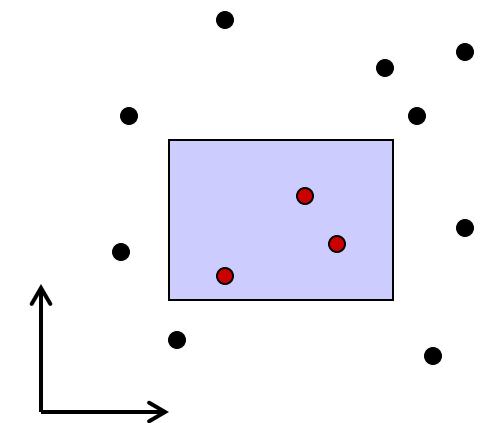
# Orthogonal range searching

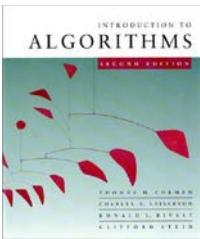
**Input:**  $n$  points in  $d$  dimensions

- E.g., representing a database of  $n$  records each with  $d$  numeric fields

**Query:** Axis-aligned *box* (in 2D, a rectangle)

- Report on the points inside the box:
  - Are there any points?
  - How many are there?
  - List the points.





# Orthogonal range searching

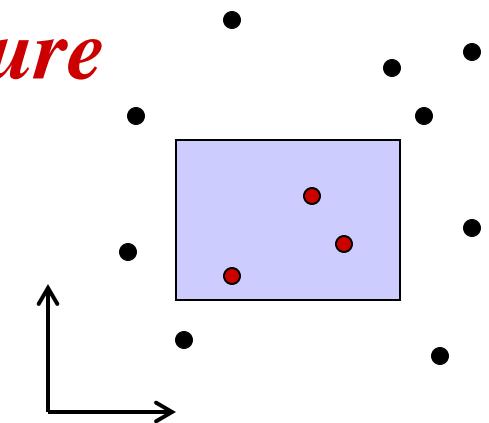
**Input:**  $n$  points in  $d$  dimensions

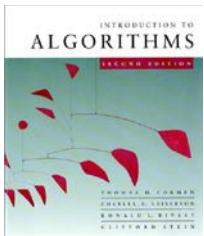
**Query:** Axis-aligned *box* (in 2D, a rectangle)

- Report on the points inside the box

**Goal:** Preprocess points into a data structure  
to support fast queries

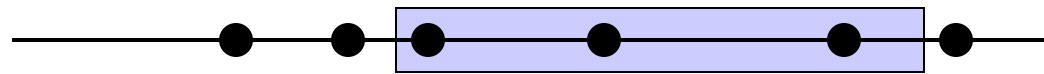
- Primary goal: *Static data structure*
- In 1D, we will also obtain a  
dynamic data structure  
supporting insert and delete





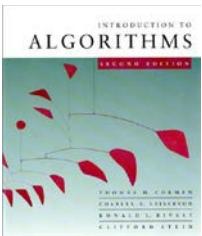
# 1D range searching

In 1D, the query is an interval:



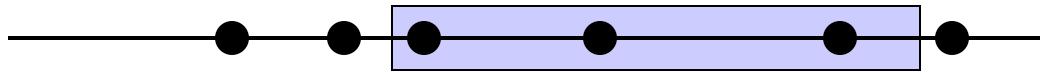
First solution using ideas we know:

- Interval trees
  - Represent each point  $x$  by the interval  $[x, x]$ .
  - Obtain a dynamic structure that can list  $k$  answers in a query in  $\mathcal{O}(k \lg n)$  time.



# 1D range searching

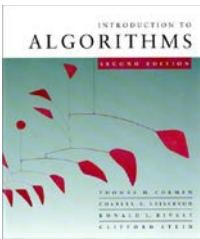
In 1D, the query is an interval:



Second solution using ideas we know:

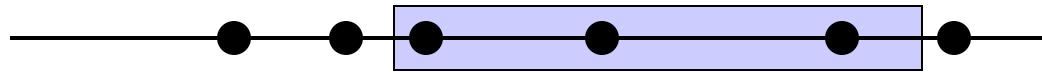
- Sort the points and store them in an array
  - Solve query by binary search on endpoints.
  - Obtain a static structure that can list  $k$  answers in a query in  $O(k + \lg n)$  time.

**Goal:** Obtain a dynamic structure that can list  $k$  answers in a query in  $O(k + \lg n)$  time.



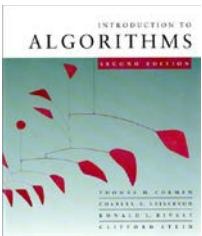
# 1D range searching

In 1D, the query is an interval:

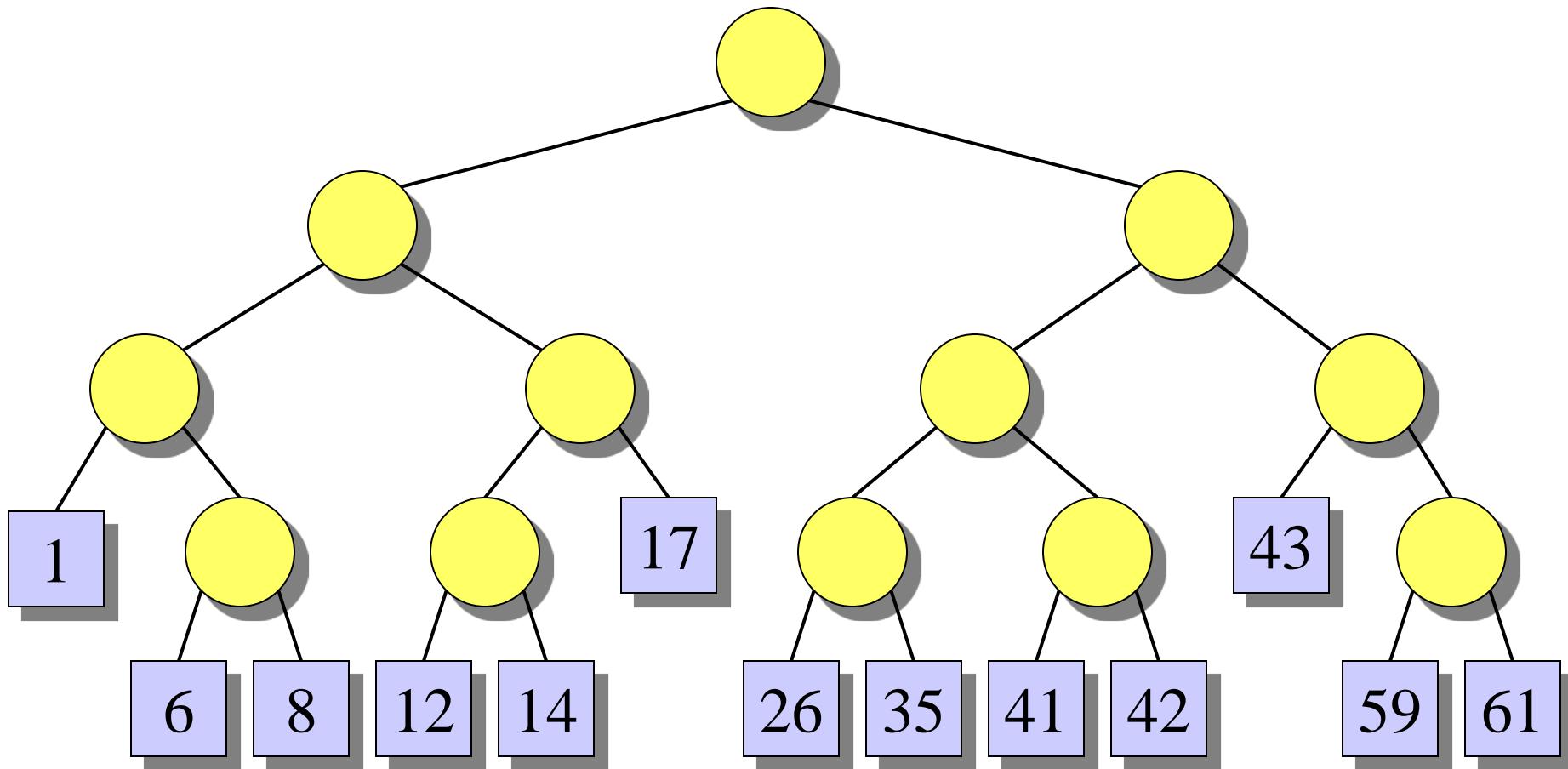


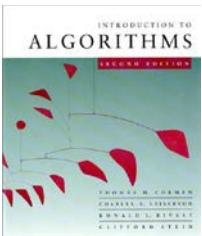
New solution that extends to higher dimensions:

- Balanced binary search tree
  - New organization principle:  
Store points in the *leaves* of the tree.
  - Internal nodes store copies of the leaves to satisfy binary search property:
    - Node  $x$  stores in  $\text{key}[x]$  the maximum key of any leaf in the left subtree of  $x$ .

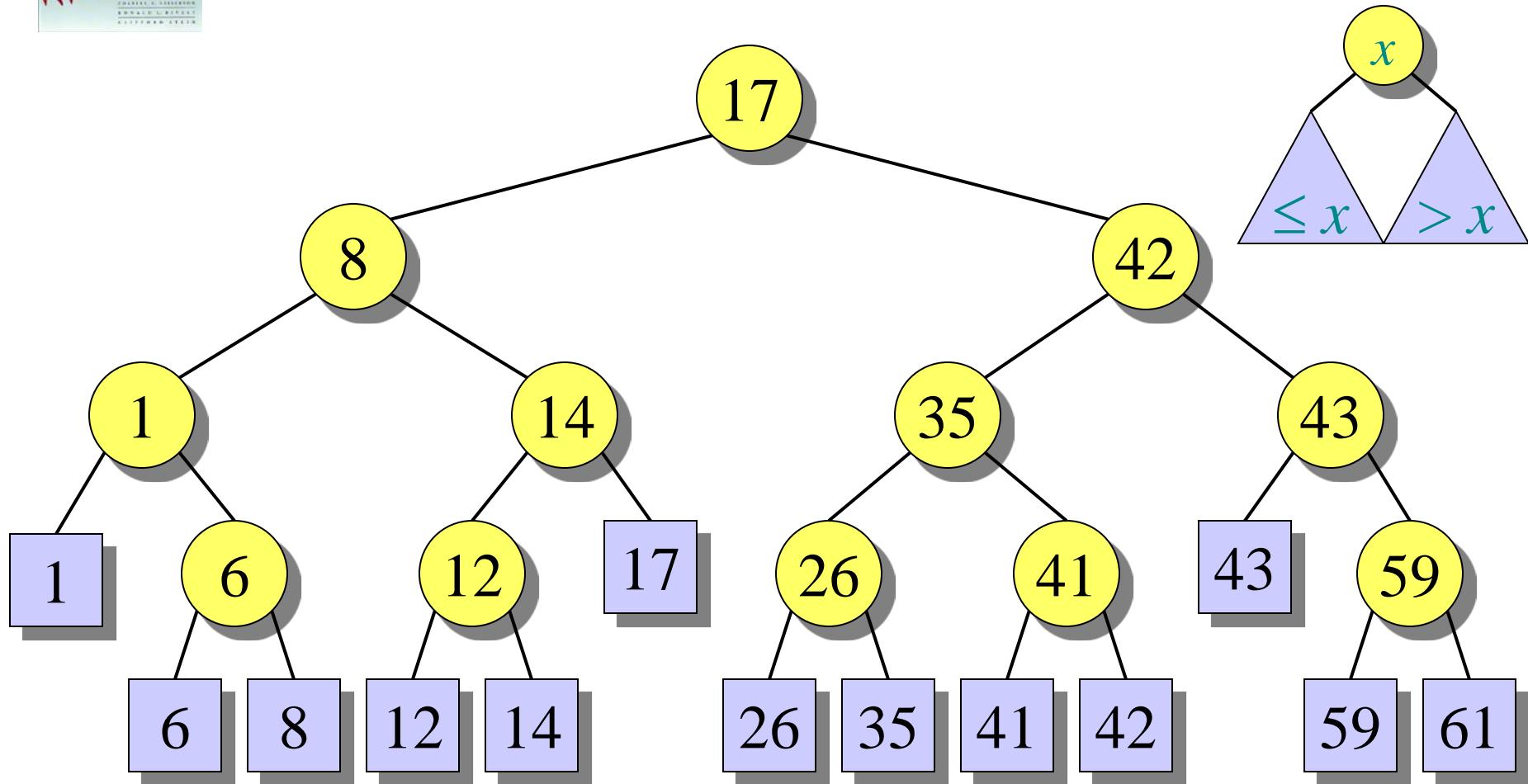


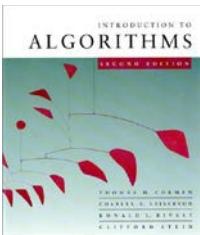
# Example of a 1D range tree



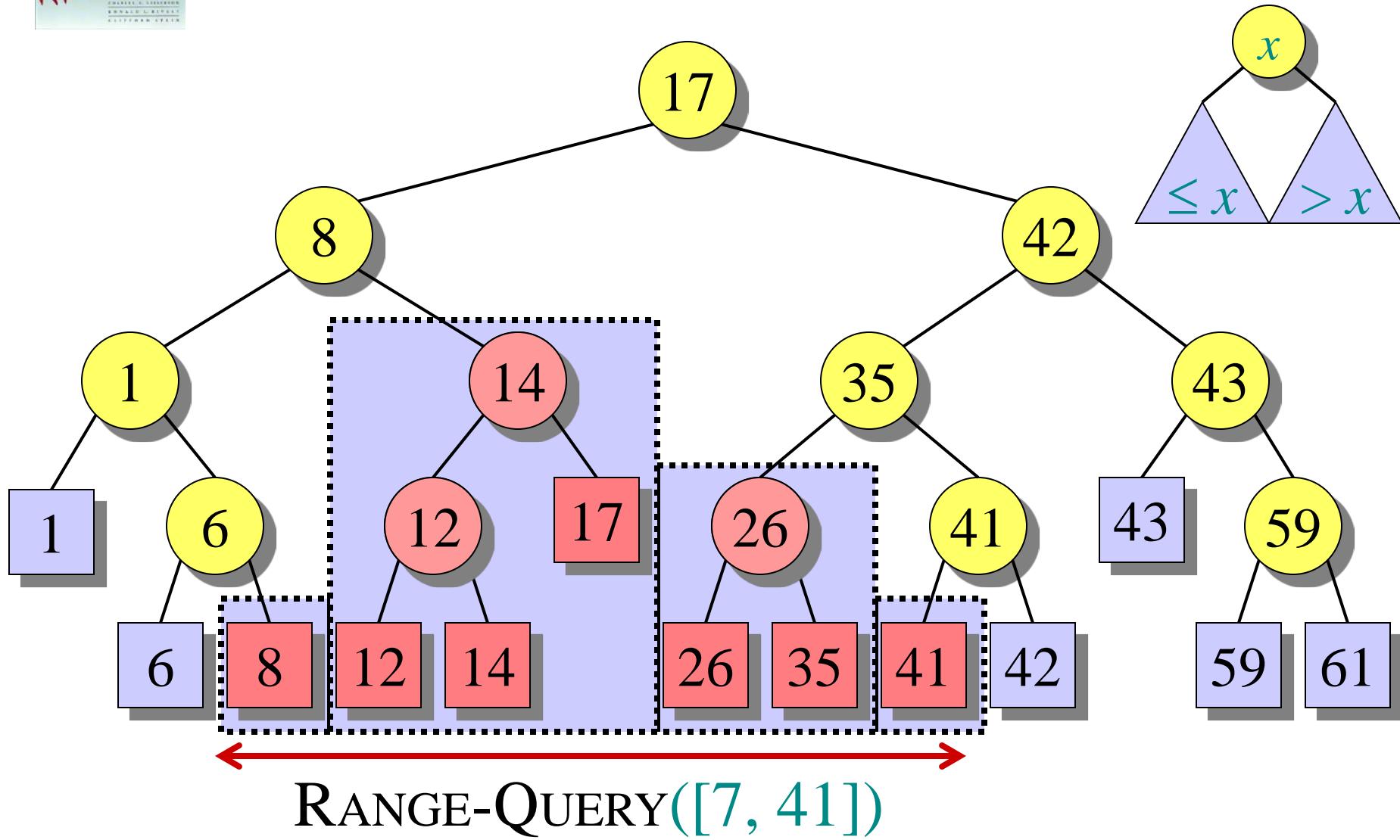


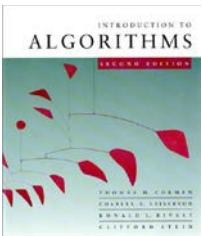
# Example of a 1D range tree



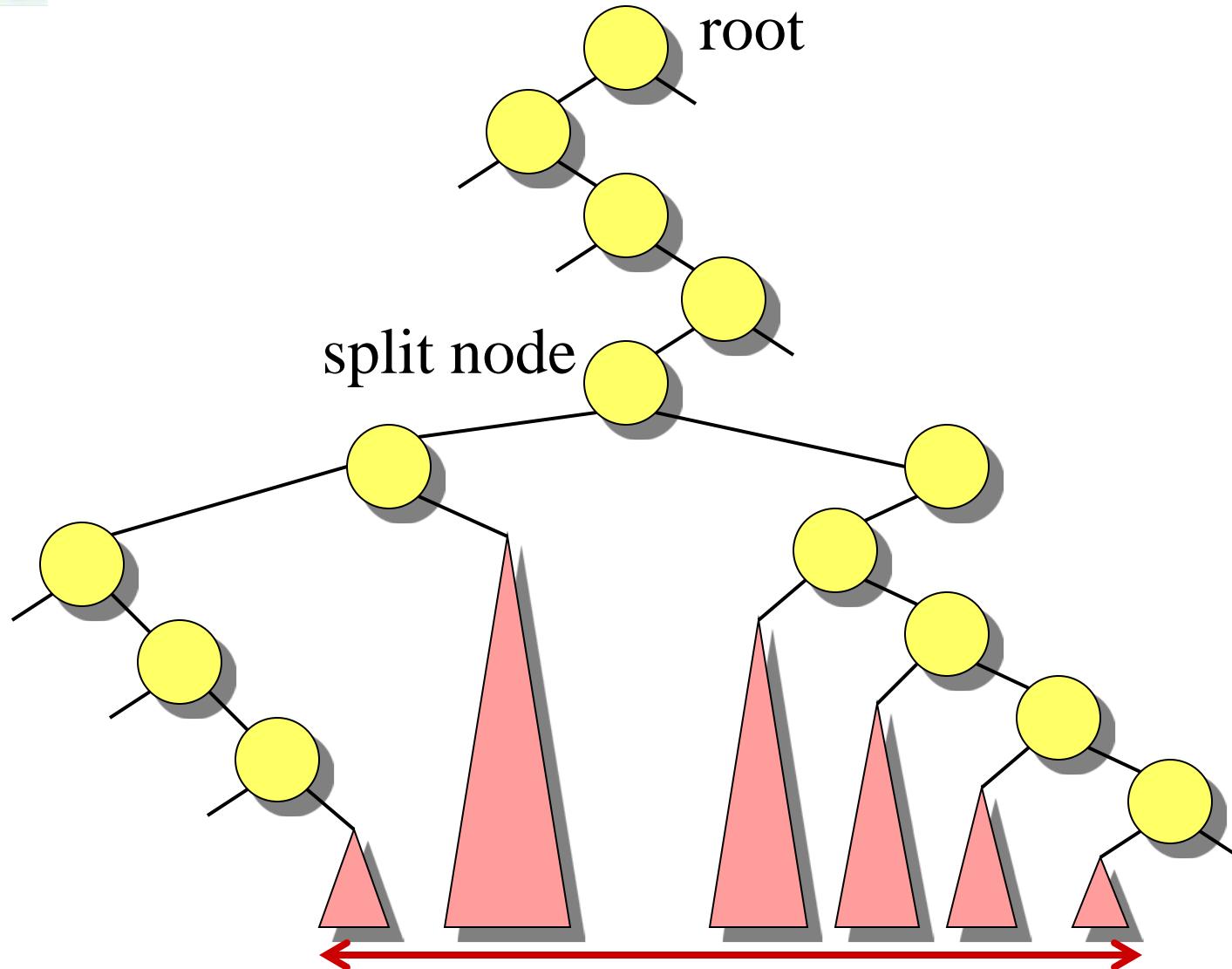


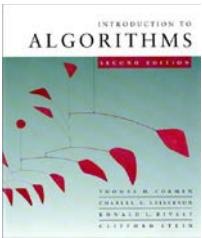
# Example of a 1D range query





# General 1D range query





# Pseudocode, part 1: Find the split node

1D-RANGE-QUERY( $T, [x_1, x_2]$ )

$w \leftarrow \text{root}[T]$

**while**  $w$  is not a leaf and  $(x_2 \leq \text{key}[w] \text{ or } \text{key}[w] < x_1)$

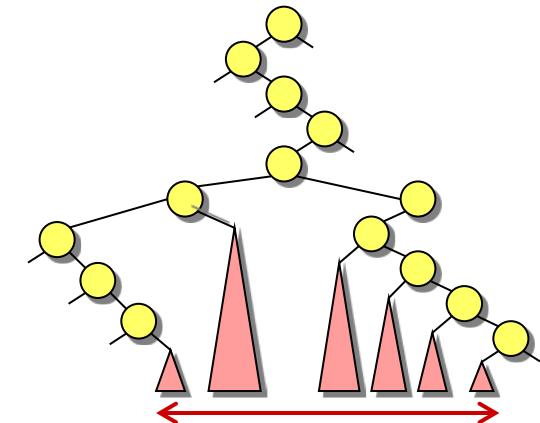
**do if**  $x_2 \leq \text{key}[w]$

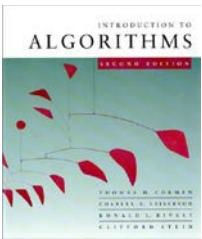
**then**  $w \leftarrow \text{left}[w]$

**else**  $w \leftarrow \text{right}[w]$

  ▷  $w$  is now the split node

*[traverse left and right from  $w$  and report relevant subtrees]*





# Pseudocode, part 2: Traverse left and right from split node

1D-RANGE-QUERY( $T, [x_1, x_2]$ )

*[find the split node]*

▷  $w$  is now the split node

if  $w$  is a leaf

then output the leaf  $w$  if  $x_1 \leq \text{key}[w] \leq x_2$

else  $v \leftarrow \text{left}[w]$

▷ Left traversal

while  $v$  is not a leaf

do if  $x_1 \leq \text{key}[v]$

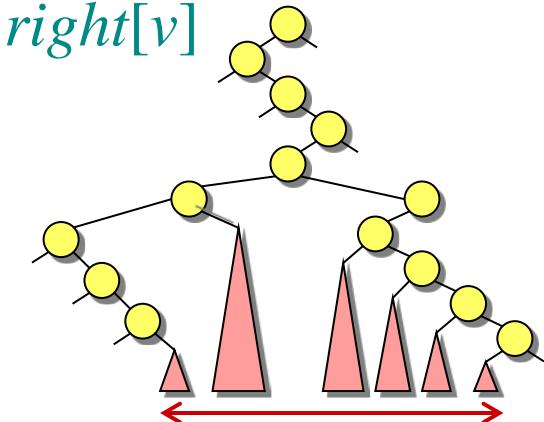
then output the subtree rooted at  $\text{right}[v]$

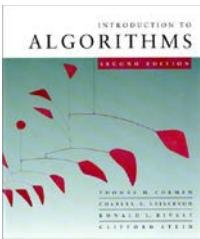
$v \leftarrow \text{left}[v]$

else  $v \leftarrow \text{right}[v]$

output the leaf  $v$  if  $x_1 \leq \text{key}[v] \leq x_2$

*[symmetrically for right traversal]*





# Analysis of 1D-RANGE-QUERY

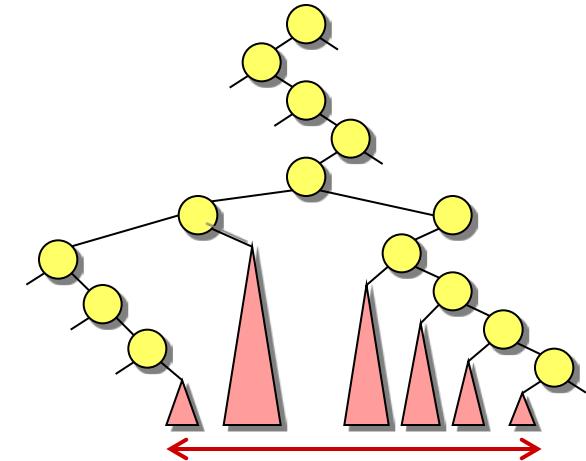
**Query time:** Answer to range query represented by  $O(\lg n)$  subtrees found in  $O(\lg n)$  time.

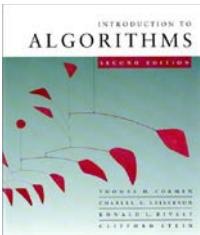
Thus:

- Can test for points in interval in  $O(\lg n)$  time.
- Can count points in interval in  $O(\lg n)$  time if we augment the tree with subtree sizes.
- Can report the first  $k$  points in interval in  $O(k + \lg n)$  time.

**Space:**  $O(n)$

**Preprocessing time:**  $O(n \lg n)$



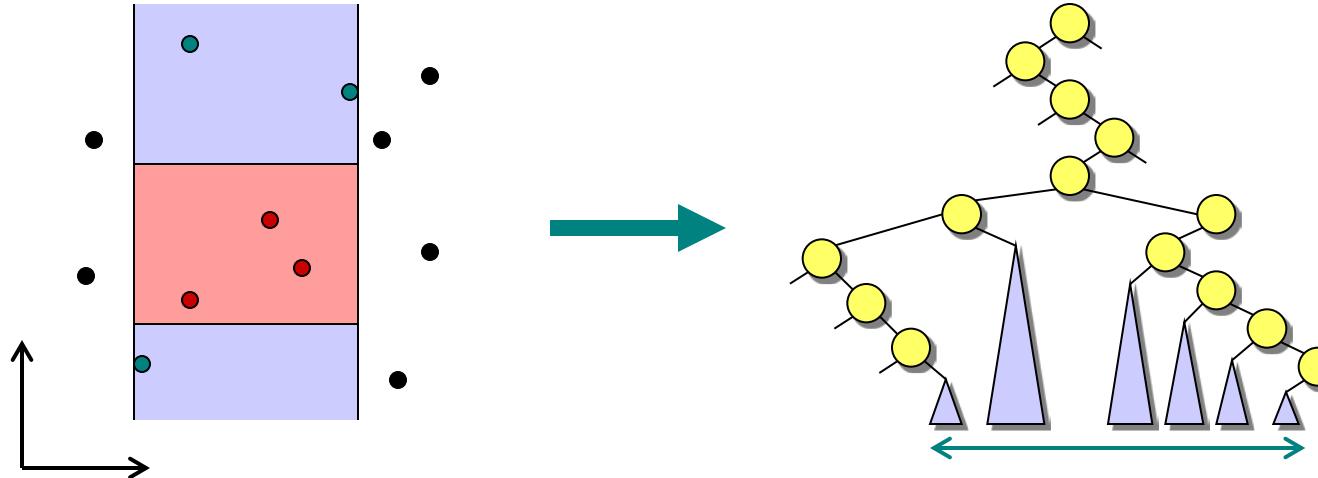


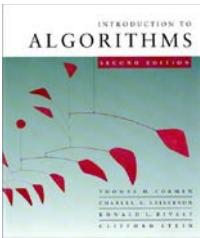
# 2D range trees

Store a *primary* 1D range tree for all the points based on  $x$ -coordinate.

Thus in  $O(\lg n)$  time we can find  $O(\lg n)$  subtrees representing the points with proper  $x$ -coordinate.

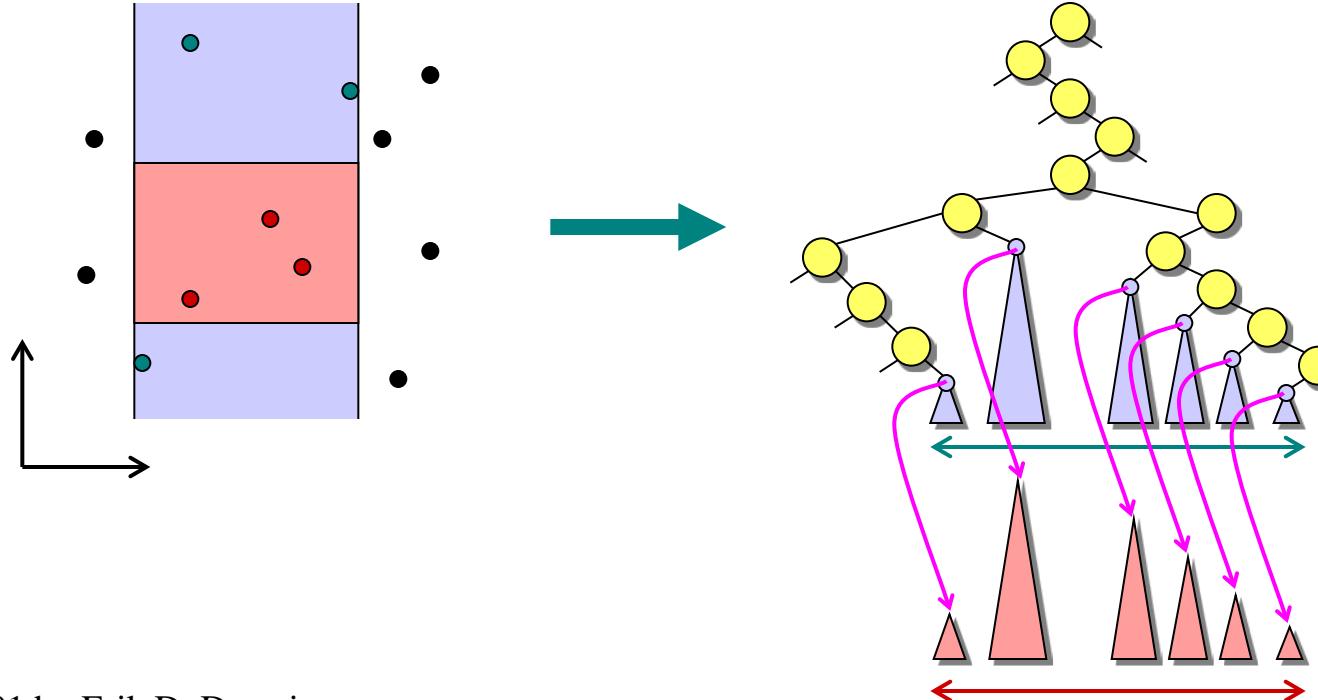
How to restrict to points with proper  $y$ -coordinate?

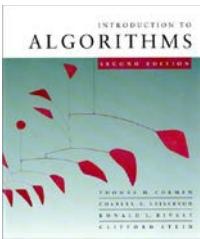




# 2D range trees

**Idea:** In primary 1D range tree of  $x$ -coordinate, every node stores a *secondary* 1D range tree based on  $y$ -coordinate for all points in the subtree of the node. Recursively search within each.



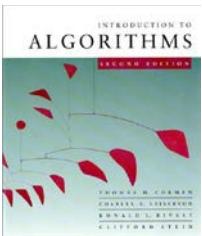


# Analysis of 2D range trees

**Query time:** In  $O((\lg n)^2)$  time, we can represent the answer to range query by  $O((\lg n)^2)$  subtrees. Total cost for reporting  $k$  points:  $O(k + (\lg n)^2)$ .

**Space:** The secondary trees at each level of the primary tree together store a copy of the points. Also, each point is present in each secondary tree along the path from the leaf to the root. Either way, we obtain that the space is  $O(n \lg n)$ .

**Preprocessing time:**  $O(n \lg n)$



# $d$ -dimensional range trees ( $d \geq 2$ )

Each node of the secondary  $y$ -structure stores a tertiary  $z$ -structure representing the points in the subtree rooted at the node, etc.

**Query time:**  $O(k + (\lg n)^d)$  to report  $k$  points.

**Space:**  $O(n (\lg n)^{d-1})$

**Preprocessing time:**  $O(n (\lg n)^{d-1})$

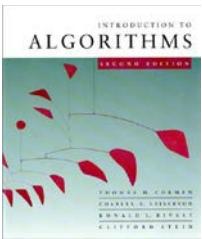
---

## Best data structure to date:

**Query time:**  $O(k + (\lg n)^{d-1})$  to report  $k$  points.

**Space:**  $O(n (\lg n / \lg \lg n)^{d-1})$

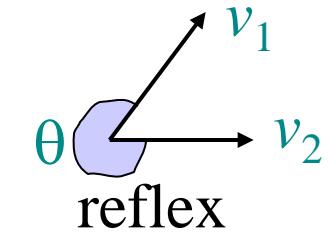
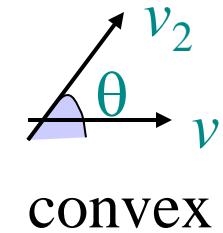
**Preprocessing time:**  $O(n (\lg n)^{d-1})$



# Primitive operations: Crossproduct

Given two vectors  $v_1 = (x_1, y_1)$  and  $v_2 = (x_2, y_2)$ ,  
is their counterclockwise angle  $\theta$

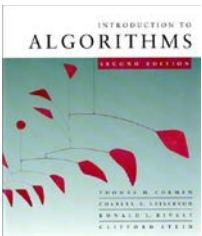
- **convex** ( $< 180^\circ$ ),
- **reflex** ( $> 180^\circ$ ), or
- borderline ( $0$  or  $180^\circ$ )?



**Crossproduct**  $v_1 \times v_2 = x_1 x_2 - y_1 y_2$   
 $= |v_1| |v_2| \sin \theta .$

Thus,  $\text{sign}(v_1 \times v_2) = \text{sign}(\sin \theta)$

- $> 0$  if  $\theta$  convex,
- $< 0$  if  $\theta$  reflex,
- $= 0$  if  $\theta$  borderline.



# Primitive operations: Orientation test

Given three points  $p_1, p_2, p_3$  are they

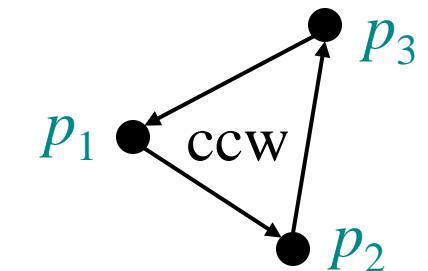
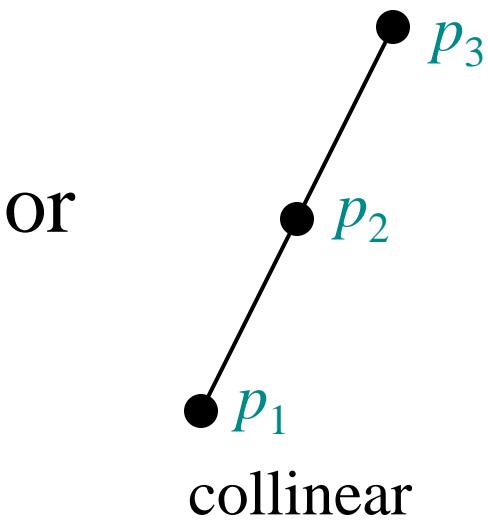
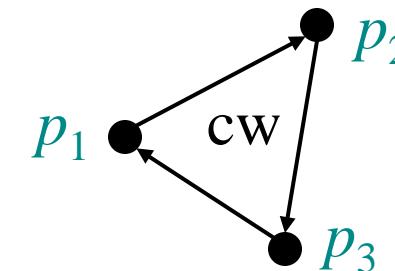
- in *clockwise (cw) order*,
- in *counterclockwise (ccw) order*, or
- *collinear*?

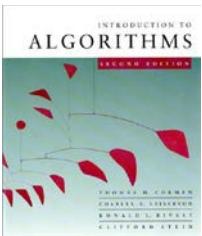
$$(p_2 - p_1) \times (p_3 - p_1)$$

$> 0$  if ccw

$< 0$  if cw

$= 0$  if collinear





# Primitive operations: Sidedness test

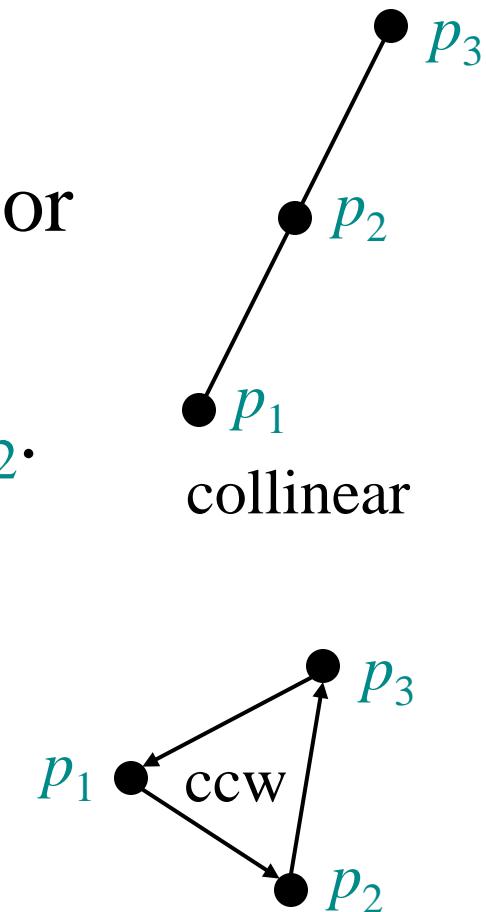
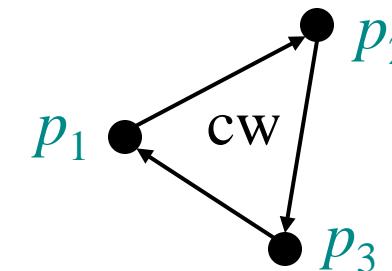
Given three points  $p_1, p_2, p_3$  are they

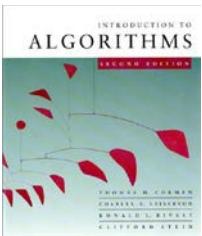
- in *clockwise (cw) order*,
- in *counterclockwise (ccw) order*, or
- *collinear*?

Let  $L$  be the oriented line from  $p_1$  to  $p_2$ .

Equivalently, is the point  $p_3$

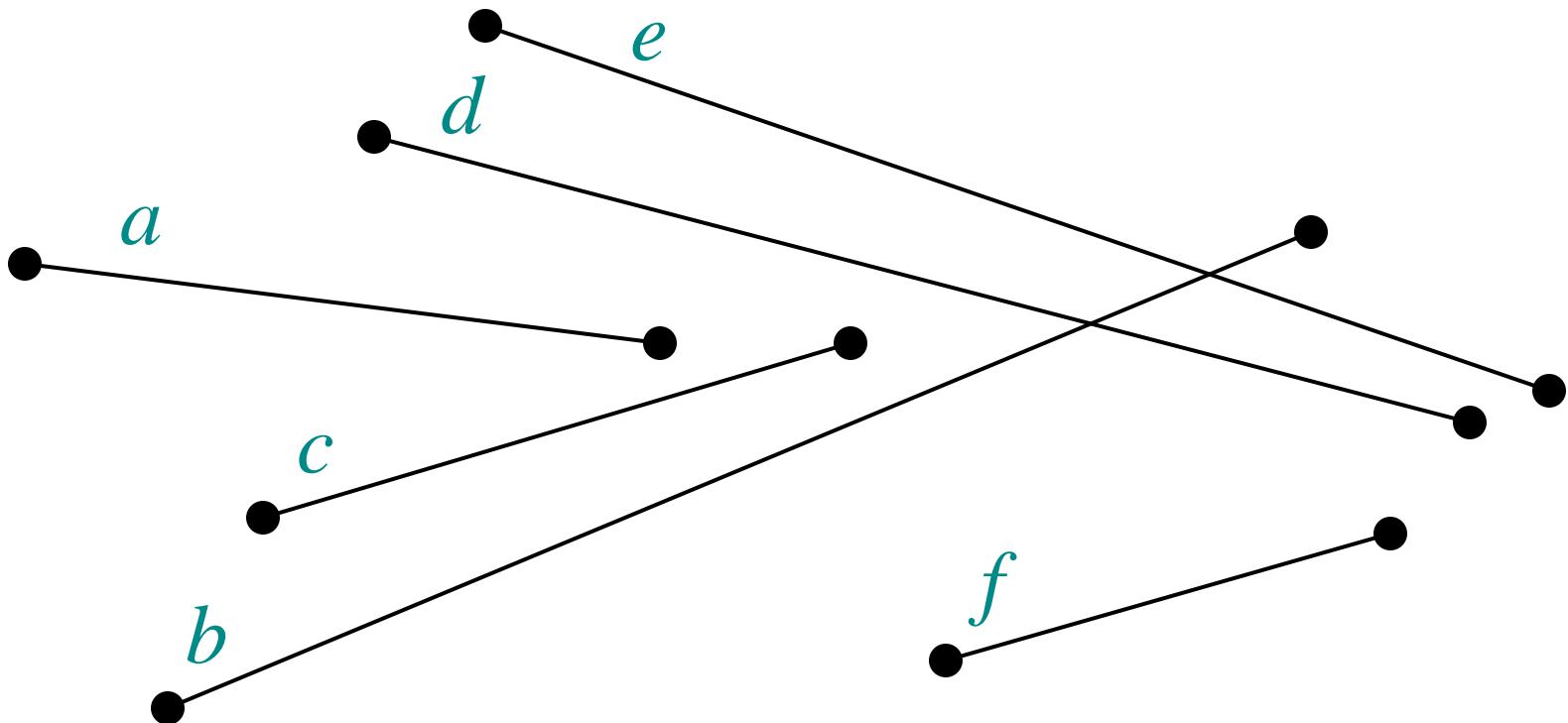
- *right* of  $L$ ,
- *left* of  $L$ , or
- *on*  $L$ ?

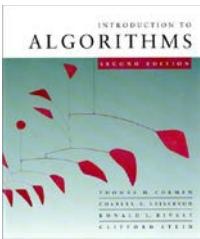




# Line-segment intersection

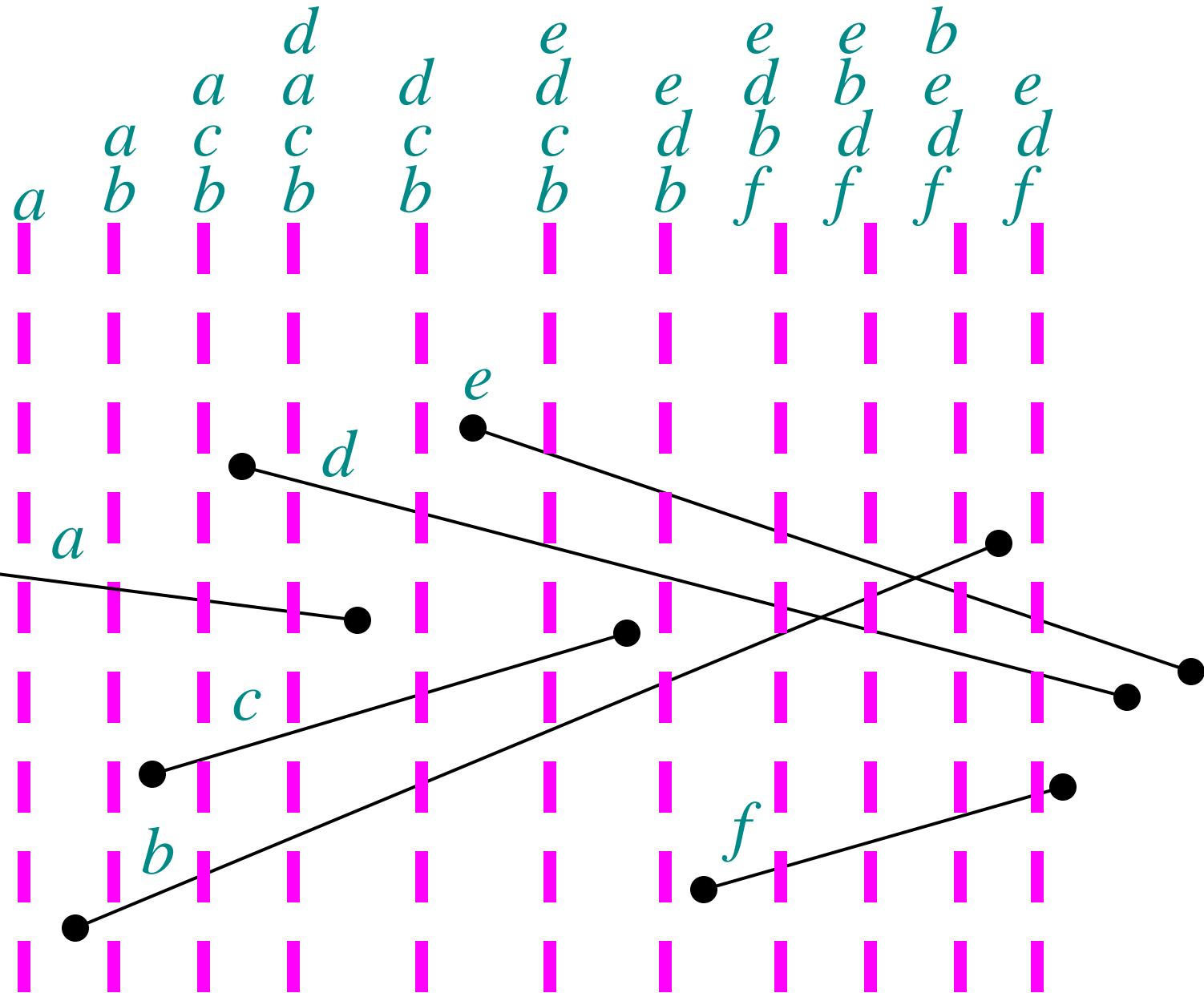
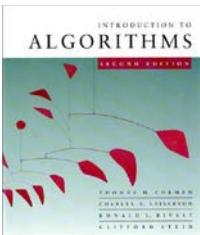
Given  $n$  line segments, does any pair intersect?  
Obvious algorithm:  $O(n^2)$ .

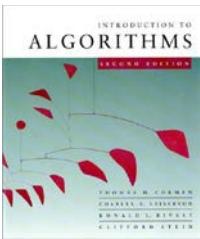




# Sweep-line algorithm

- Sweep a vertical line from left to right (conceptually replacing  $x$ -coordinate with time).
- Maintain dynamic set  $S$  of segments that intersect the sweep line, ordered (tentatively) by  $y$ -coordinate of intersection.
- Order changes when
  - new segment is encountered,
  - existing segment finishes, or
  - two segments cross
- Key *event points* are therefore segment endpoints.

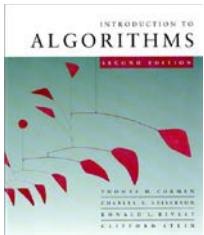




# Sweep-line algorithm

Process event points in order by sorting segment endpoints by  $x$ -coordinate and looping through:

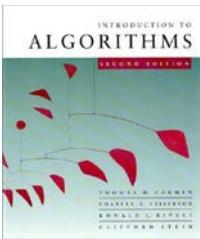
- For a left endpoint of segment  $s$ :
  - Add segment  $s$  to dynamic set  $S$ .
  - Check for intersection between  $s$  and its neighbors in  $S$ .
- For a right endpoint of segment  $s$ :
  - Remove segment  $s$  from dynamic set  $S$ .
  - Check for intersection between the neighbors of  $s$  in  $S$ .



# Analysis

Use red-black tree to store dynamic set  $S$ .

Total running time:  $O(n \lg n)$ .



# Correctness

**Theorem:** If there is an intersection, the algorithm finds it.

*Proof:* Let  $X$  be the leftmost intersection point. Assume for simplicity that

- only two segments  $s_1, s_2$  pass through  $X$ , and
- no two points have the same  $x$ -coordinate.

At some point before we reach  $X$ ,

$s_1$  and  $s_2$  become consecutive in the order of  $S$ .

Either initially consecutive when  $s_1$  or  $s_2$  inserted, or became consecutive when another deleted. □