

“Asymptotics, Summation, Divide & Conquer Algorithms, Recurrences and Master Theorem”

□ Lecture Topics and Readings

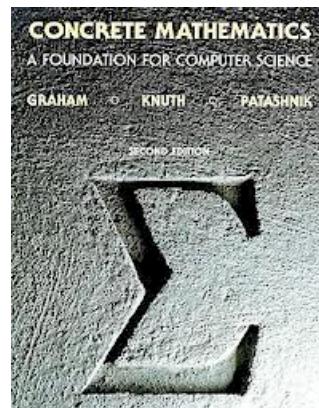
- ❖ Asymptotic, Summation [CLRS]-C3
- ❖ Divide and Conquer Algorithms [CLRS]-C2
- ❖ Recurrences, Master Theorem [CLRS]-C4

*Algorithms & Discrete Math
have very happy, stable marriage!*

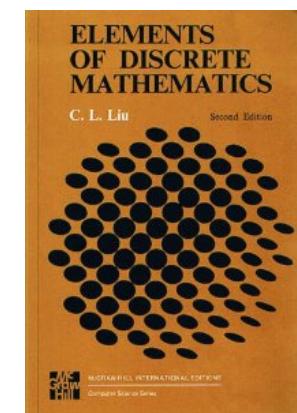
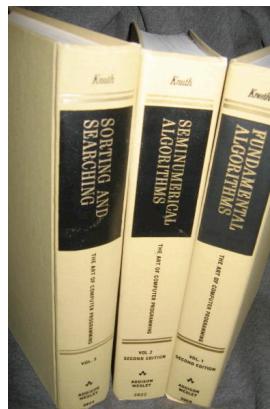
Early pioneers of Math for CS



Don Knuth
Stanford



C. L. Liu,
MIT, UIUC,
NTHU (tw)



Elements of Discrete Math, 1977

“Elements of Discrete Mathematics,” C. L. Liu, McGraw-Hill, (1977),
has the most important topics in DM for CS.

Relations & Functions

Graphs & Planar Graphs

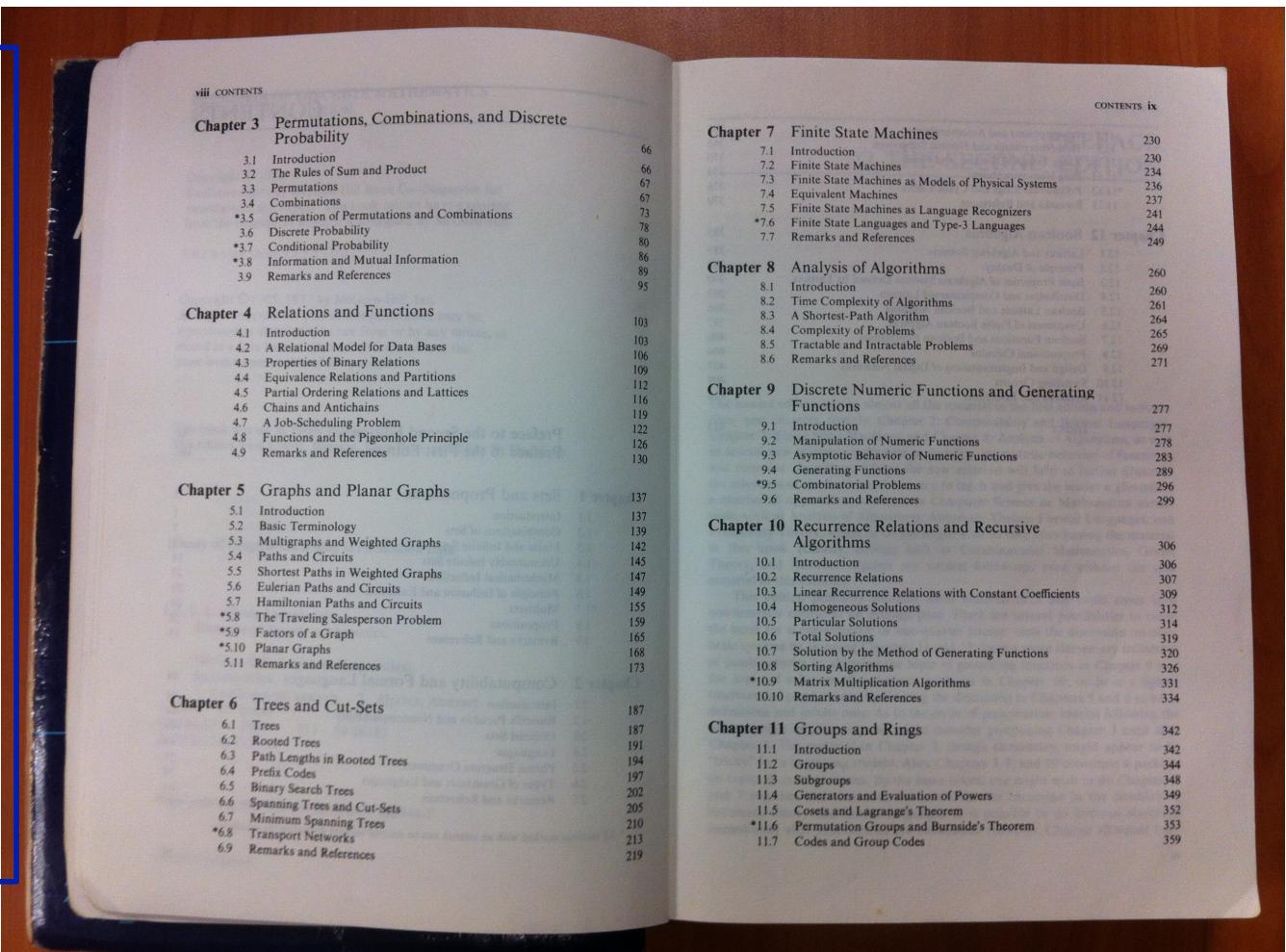
Trees and Cut-Sets

Finite State Machines

Analysis of Algorithms

Recurrence Relations

...



The image shows an open book with the title "Elements of Discrete Mathematics" by C. L. Liu, McGraw-Hill, (1977). The left page (viii) contains the Table of Contents, and the right page (ix) contains the chapter titles and their corresponding page numbers.

Chapter	Title	Page Number
Chapter 3	Permutations, Combinations, and Discrete Probability	66
3.1	Introduction	66
3.2	The Rules of Sum and Product	66
3.3	Permutations	67
3.4	Combinations	67
*3.5	Generation of Permutations and Combinations	73
3.6	Discrete Probability	78
*3.7	Conditional Probability	80
*3.8	Information and Mutual Information	86
3.9	Remarks and References	95
Chapter 4	Relations and Functions	103
4.1	Introduction	103
4.2	A Relational Model for Data Bases	103
4.3	Properties of Binary Relations	106
4.4	Equivalence Relations and Partitions	109
4.5	Partial Ordering Relations and Lattices	112
4.6	Chains and Antichains	116
4.7	A Job-Scheduling Problem	119
4.8	Functions and the Pigeonhole Principle	122
4.9	Remarks and References	126
Chapter 5	Graphs and Planar Graphs	137
5.1	Introduction	137
5.2	Basic Terminology	139
5.3	Multigraphs and Weighted Graphs	142
5.4	Paths and Circuits	145
5.5	Shortest Paths in Weighted Graphs	147
5.6	Eulerian Paths and Circuits	149
5.7	Hamiltonian Paths and Circuits	155
*5.8	The Traveling Salesperson Problem	159
*5.9	Factors of a Graph	165
*5.10	Planar Graphs	168
5.11	Remarks and References	173
Chapter 6	Trees and Cut-Sets	187
6.1	Trees	187
6.2	Rooted Trees	191
6.3	Path Lengths in Rooted Trees	194
6.4	Prefix Codes	197
6.5	Binary Search Trees	202
6.6	Spanning Trees and Cut-Sets	205
6.7	Minimum Spanning Trees	210
*6.8	Transport Networks	213
6.9	Remarks and References	219
Chapter 7	Finite State Machines	230
7.1	Introduction	230
7.2	Finite State Machines	234
7.3	Finite State Machines as Models of Physical Systems	236
7.4	Equivalent Machines	237
7.5	Finite State Machines as Language Recognizers	241
*7.6	Finite State Languages and Type-3 Languages	244
7.7	Remarks and References	249
Chapter 8	Analysis of Algorithms	260
8.1	Introduction	260
8.2	Time Complexity of Algorithms	261
8.3	A Shortest-Path Algorithm	264
8.4	Complexity of Problems	265
8.5	Tractable and Intractable Problems	269
8.6	Remarks and References	271
Chapter 9	Discrete Numeric Functions and Generating Functions	277
9.1	Introduction	277
9.2	Manipulation of Numeric Functions	278
9.3	Asymptotic Behavior of Numeric Functions	283
9.4	Generating Functions	289
*9.5	Combinatorial Problems	296
9.6	Remarks and References	299
Chapter 10	Recurrence Relations and Recursive Algorithms	306
10.1	Introduction	306
10.2	Recurrence Relations	307
10.3	Linear Recurrence Relations with Constant Coefficients	309
10.4	Homogeneous Solutions	312
10.5	Particular Solutions	314
10.6	Total Solutions	319
10.7	Solution by the Method of Generating Functions	320
10.8	Sorting Algorithms	326
*10.9	Matrix Multiplication Algorithms	331
10.10	Remarks and References	334
Chapter 11	Groups and Rings	342
11.1	Introduction	342
11.2	Groups	344
11.3	Subgroups	348
11.4	Generators and Evaluation of Powers	349
11.5	Cosets and Lagrange's Theorem	352
*11.6	Permutation Groups and Burnside's Theorem	353
11.7	Codes and Group Codes	359

David Plaisted



David Plaisted
UIUC 1978-84
UNC, 1985--

Spring 1980, LeongHW took CS373 by Plaisted;
Summer 1984, Plaisted: UIUC ----> UNC
Spring 1985 @UIUC: CS373 by Plaisted
Dave Liu recommend PhD student LeongHW
to the Dept Head for CS373



Herbrand Award, 2010
(for distinguished contribution
to Automated Reasoning)

<http://www.cs.miami.edu/~geoff/Conferences/CADE/HerbrandAward.html>

Volker Strassen



Volker Strassen

Knuth Prize, 2009

“seminal and influential contributions
To the design and analysis
of efficient algorithms”

http://en.wikipedia.org/wiki/Volker_Strassen

Thank you.

Q & A



School *of* Computing

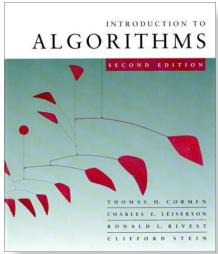
“Asymptotics, Summation, Divide & Conquer Algorithms, Recurrences and Master Theorem”

□ Lecture Topics and Readings

- ❖ Asymptotics, Summation [CLRS]-C3
- ❖ Divide and Conquer Algorithms [CLRS]-C2
- ❖ Recurrences, Master Theorem [CLRS]-C4

*Algorithms & Discrete Math
have very happy, stable marriage!*



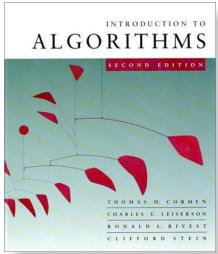


Merge sort

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



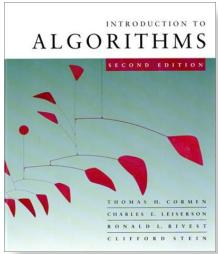
Merging two sorted arrays

20 12

13 11

7 9

2 1



Merging two sorted arrays

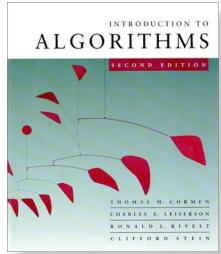
20 12

13 11

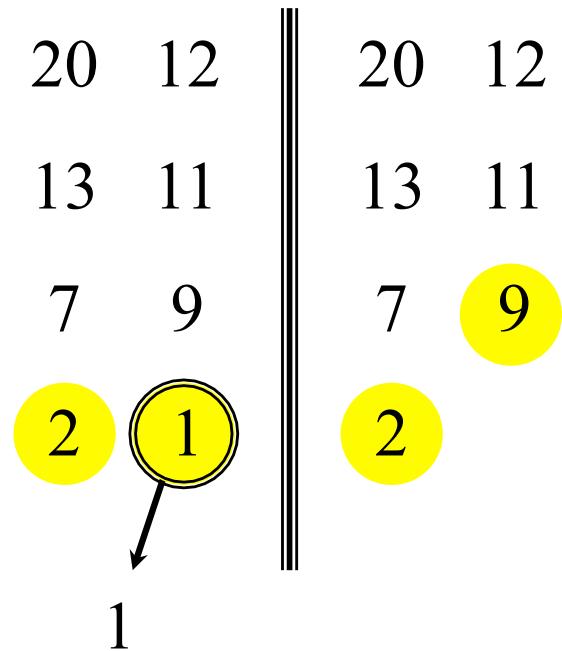
7 9

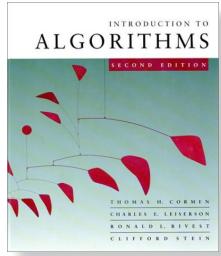
2 1

1

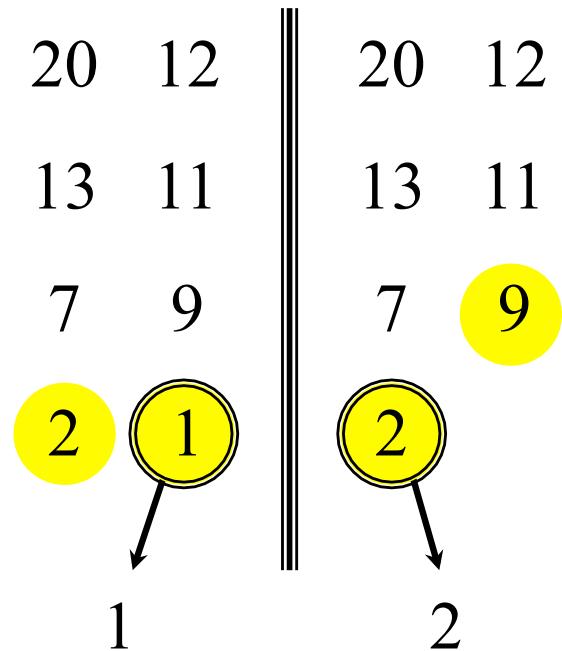


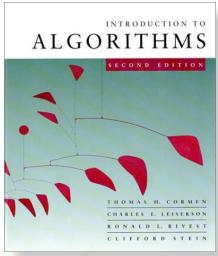
Merging two sorted arrays



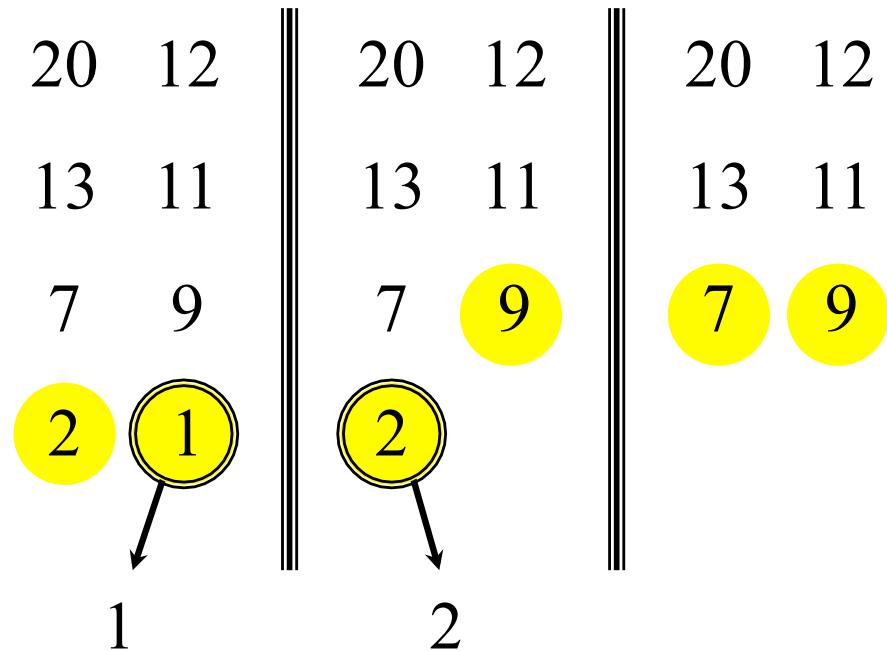


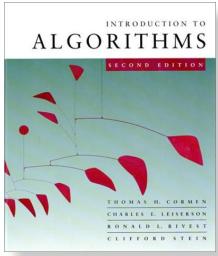
Merging two sorted arrays



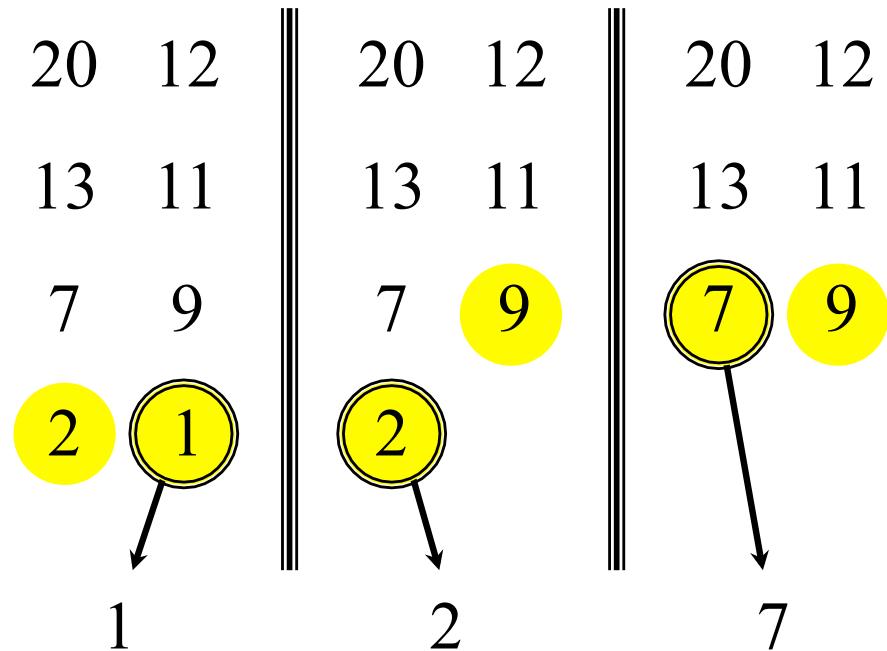


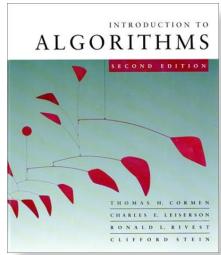
Merging two sorted arrays



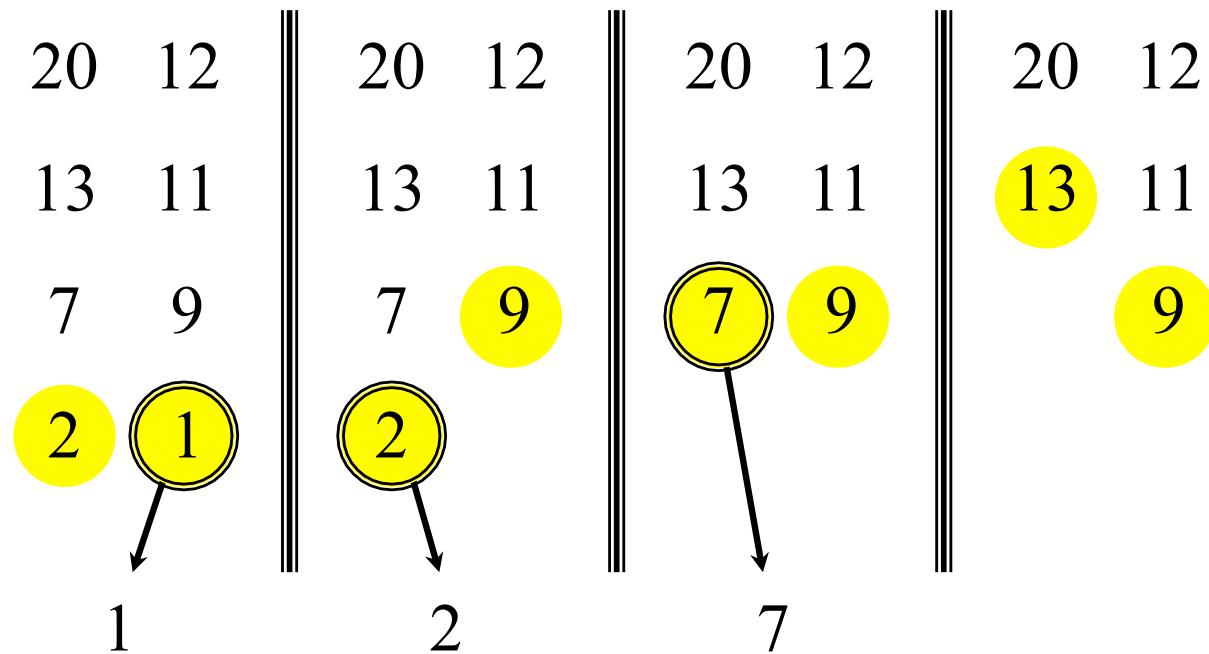


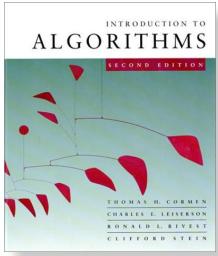
Merging two sorted arrays



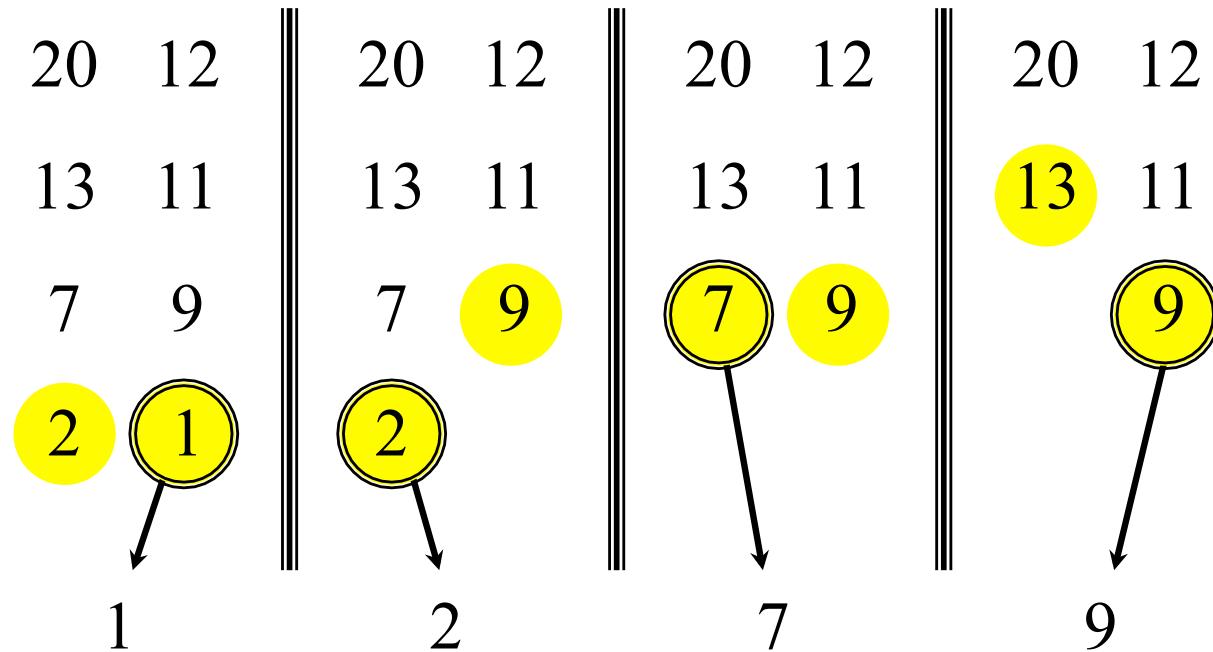


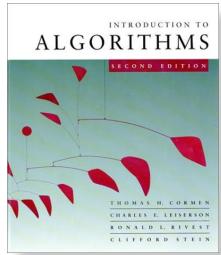
Merging two sorted arrays



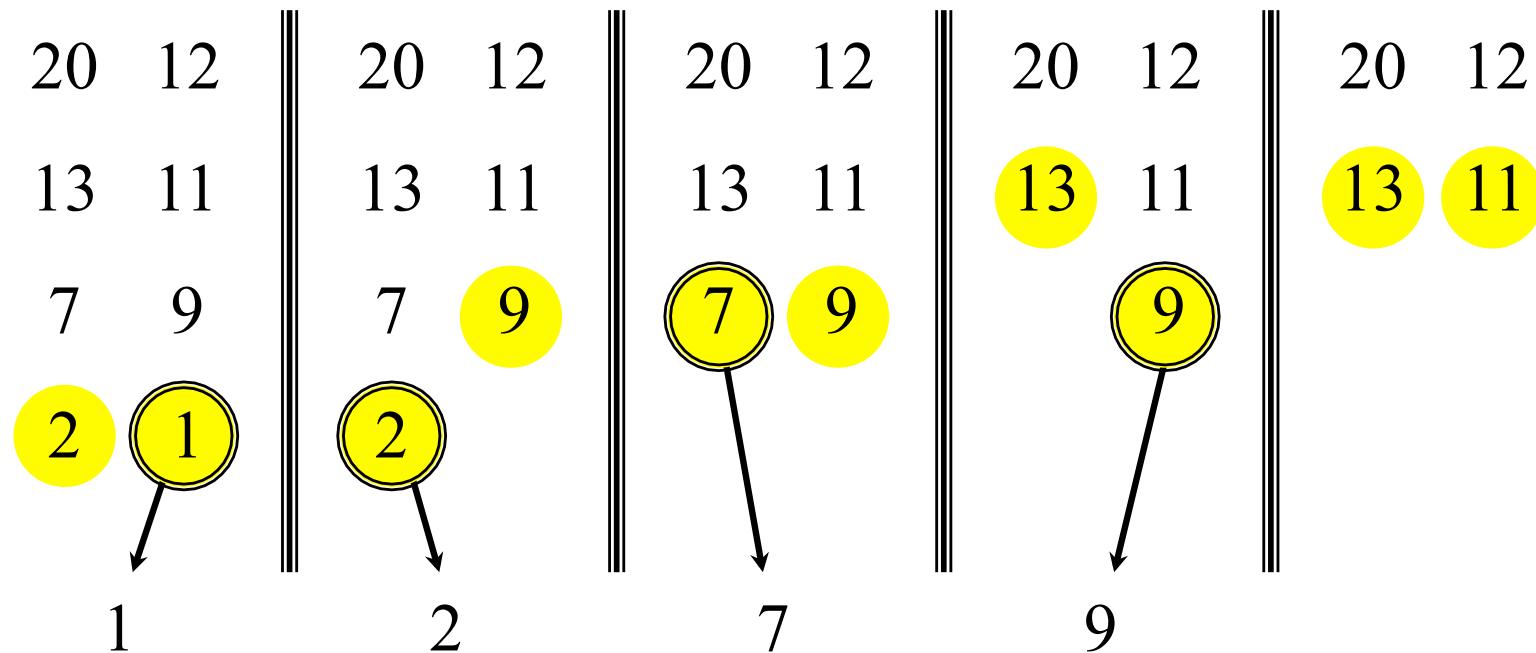


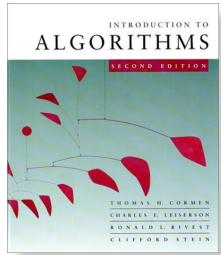
Merging two sorted arrays



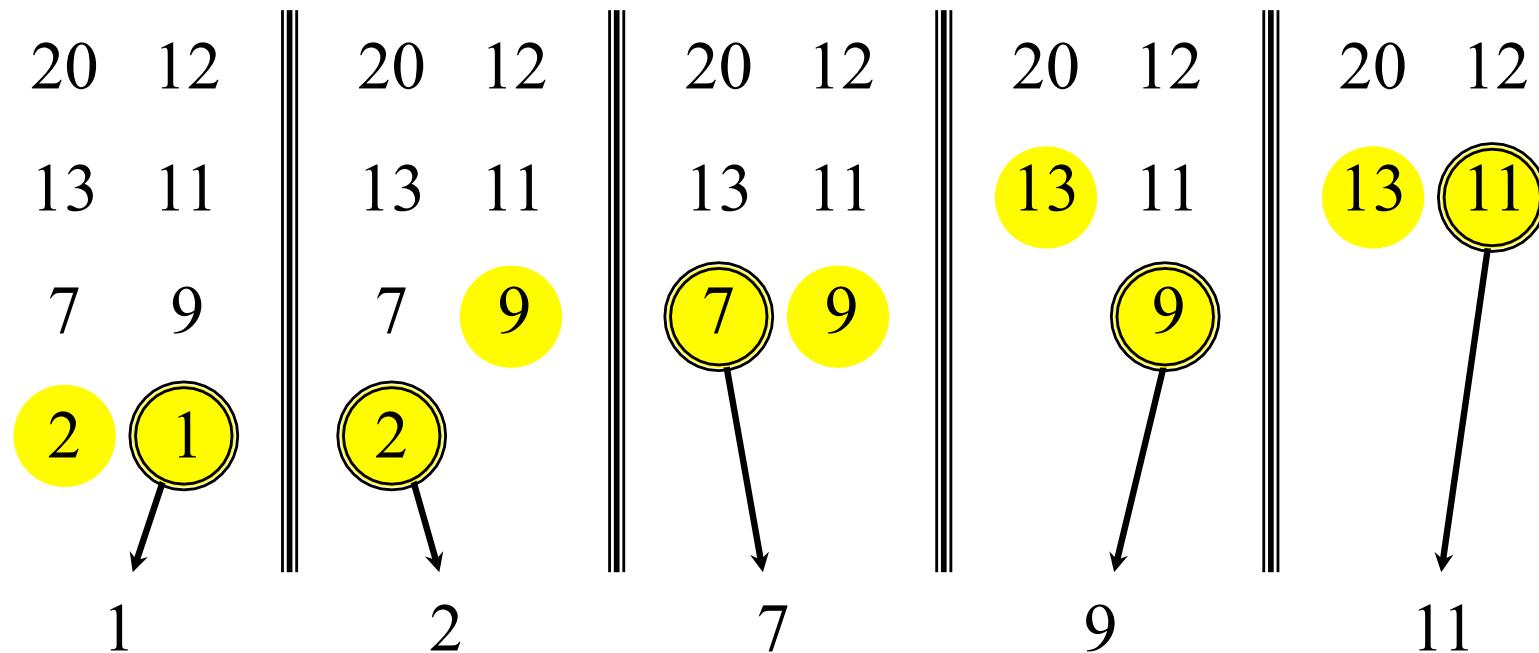


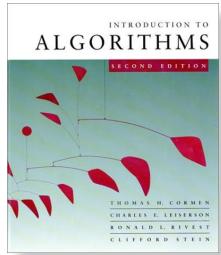
Merging two sorted arrays



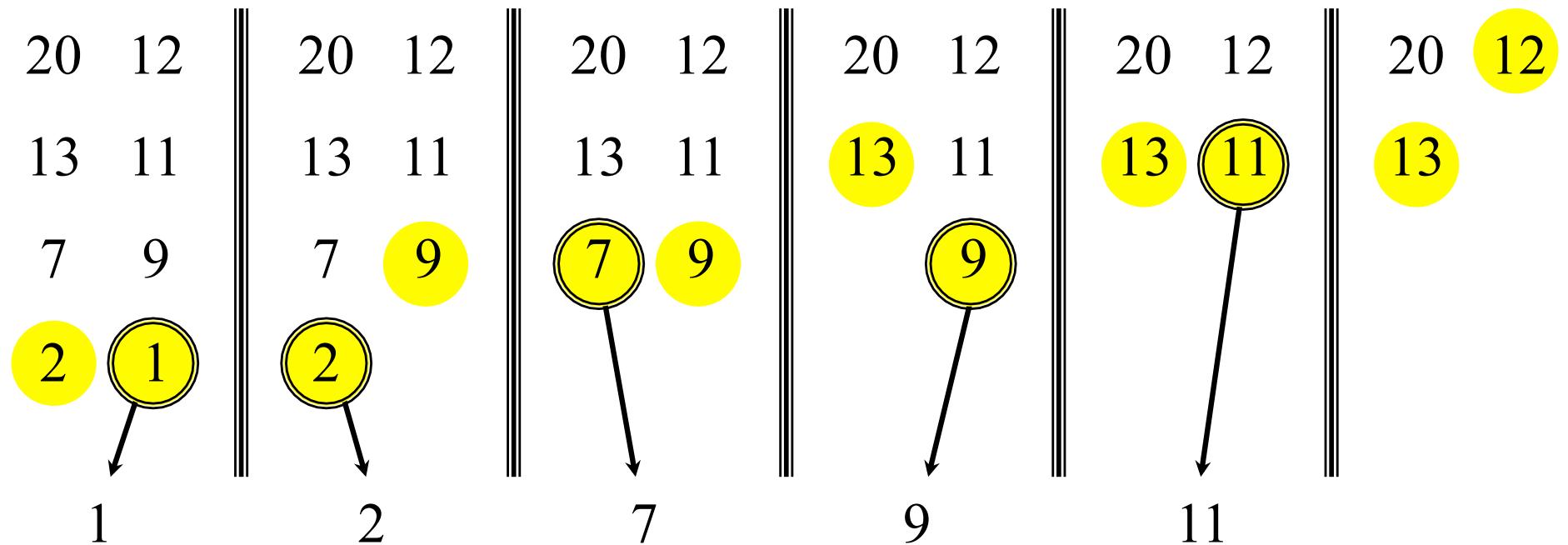


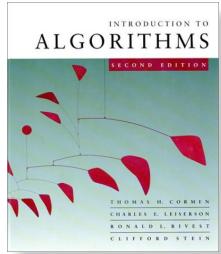
Merging two sorted arrays



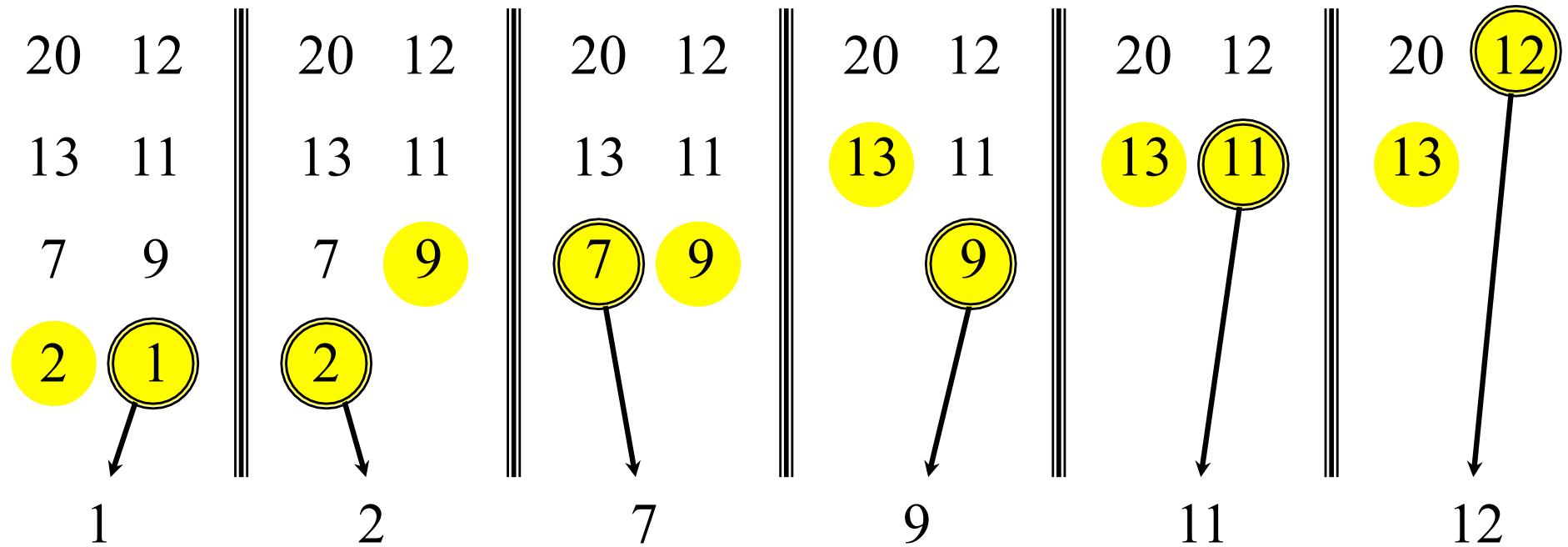


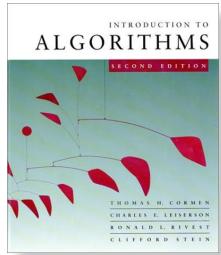
Merging two sorted arrays



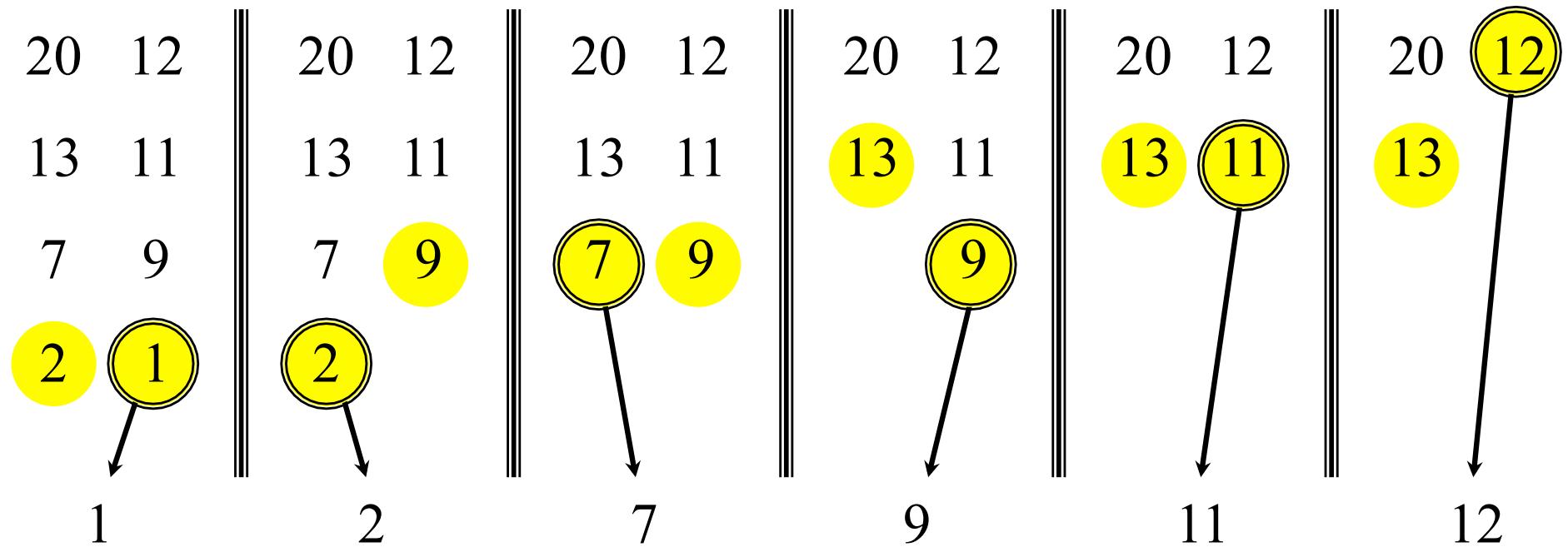


Merging two sorted arrays

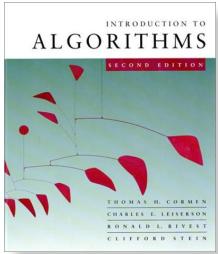




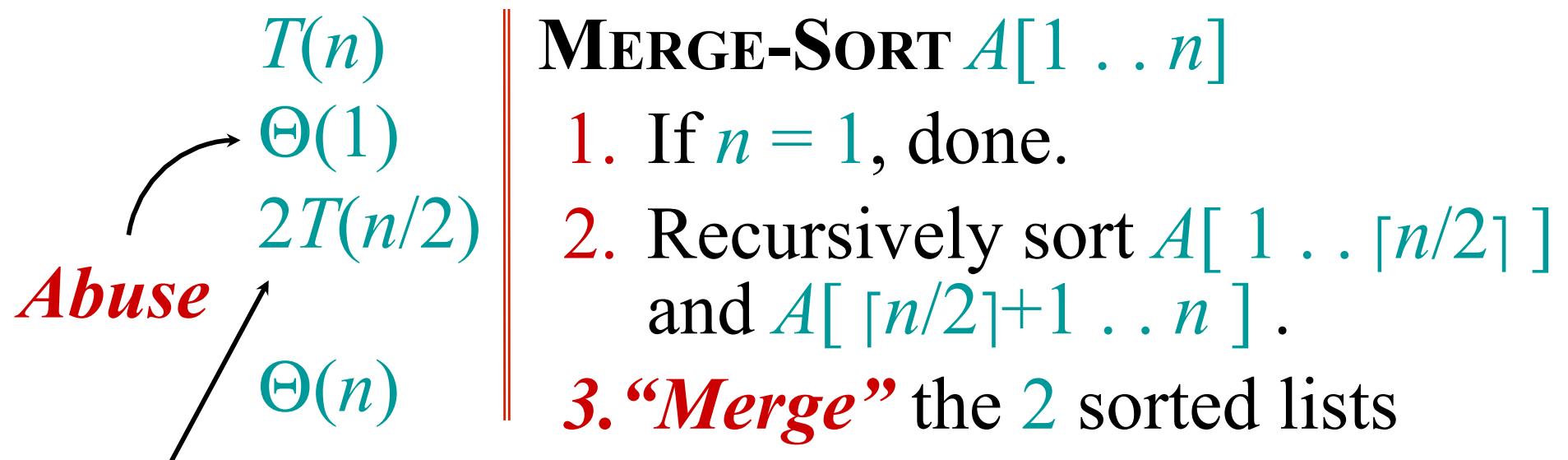
Merging two sorted arrays



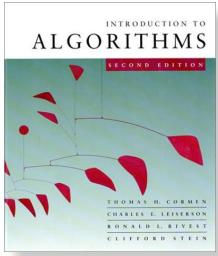
Time = $\Theta(n)$ to merge a total
of n elements (linear time).



Analyzing merge sort



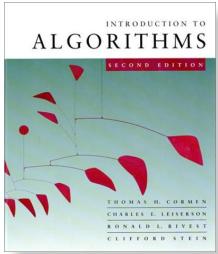
Sloppiness: Should be $T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor)$,
but it turns out not to matter asymptotically.



Recurrence for merge sort

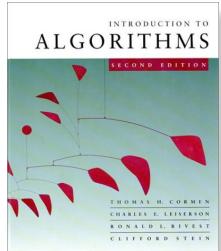
$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1; \\ 2T(n/2) + \Theta(n) & \text{if } n > 1. \end{cases}$$

- 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.



Solving recurrences

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



Substitution method

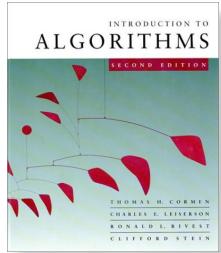
Recursion Tree method

basis of the Master Theorem

The most general method:

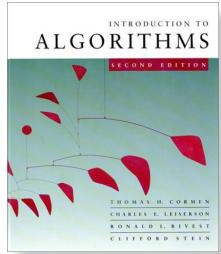
- 1. Guess** the form of the solution.
- 2. Verify** by induction.
- 3. Solve** for constants.

***Not covered in CS3230; those interested
see example and details in CLRS***



Recursion tree

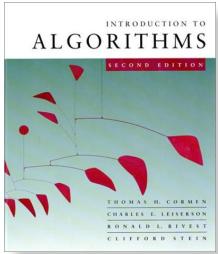
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



Recursion tree

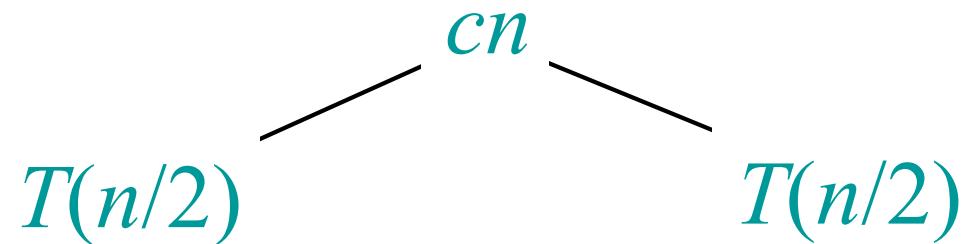
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

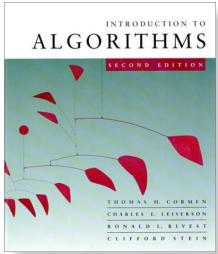
$$T(n)$$



Recursion tree

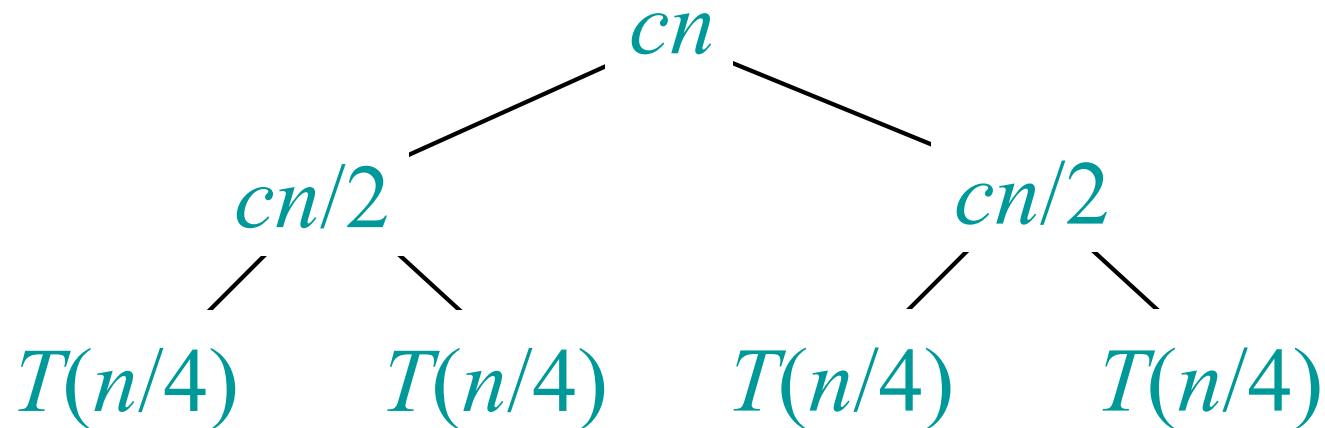
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

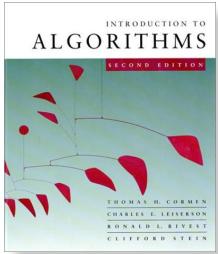




Recursion tree

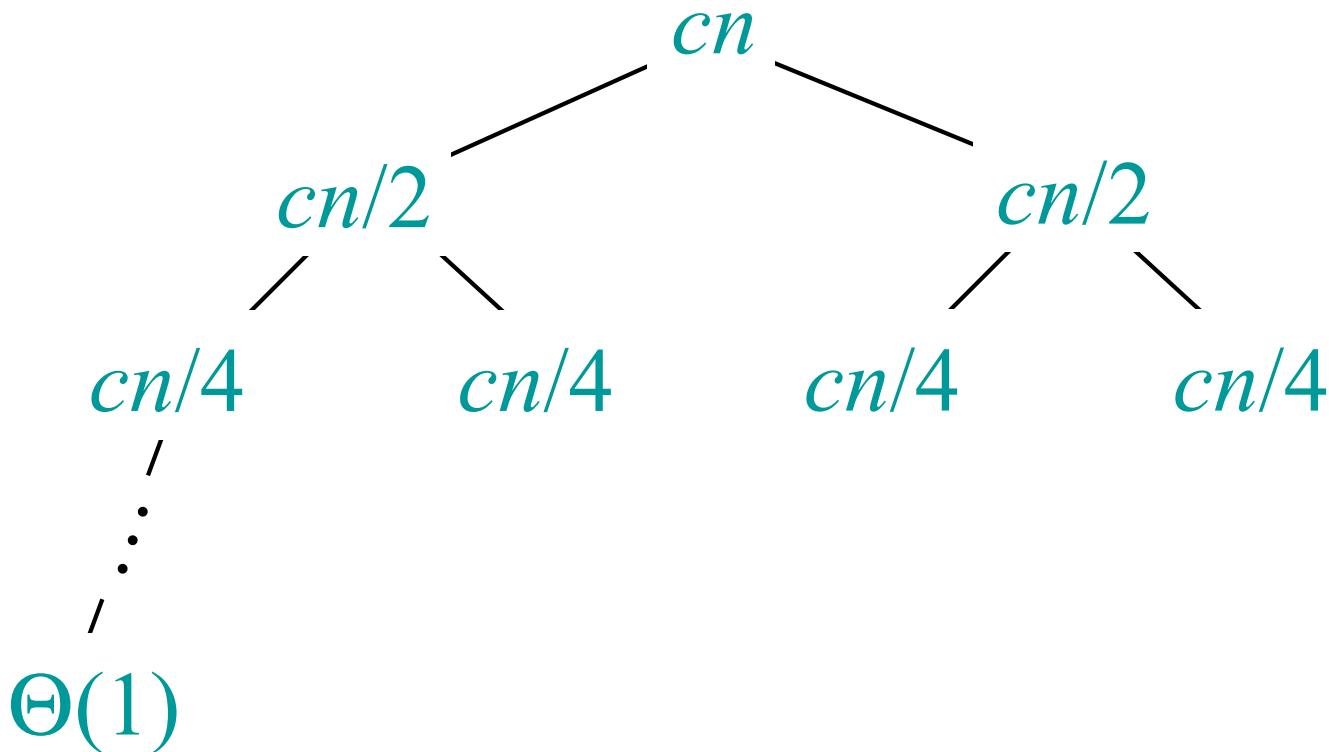
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

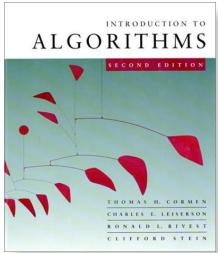




Recursion tree

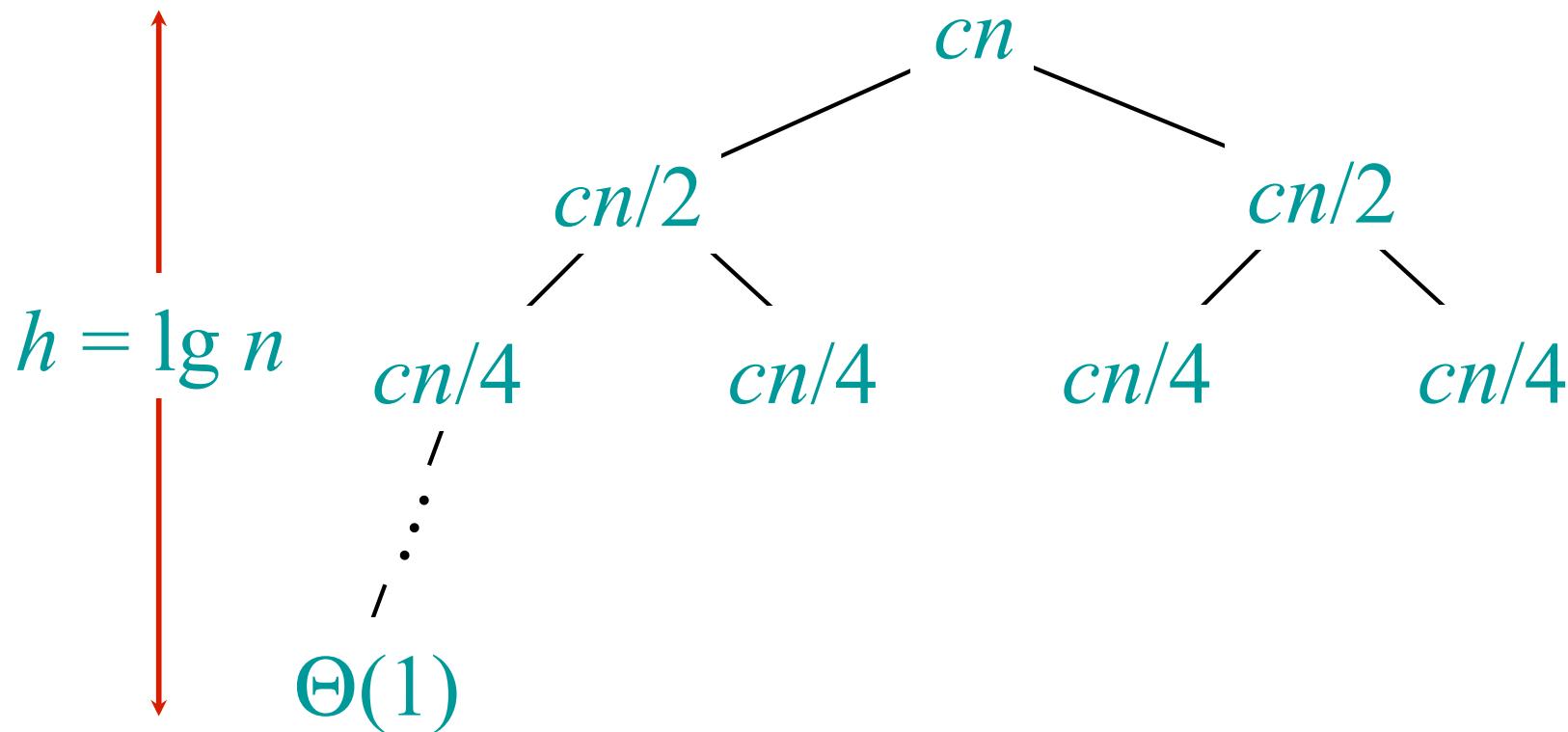
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

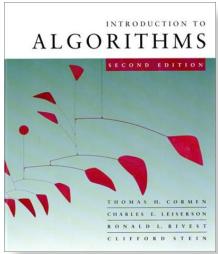




Recursion tree

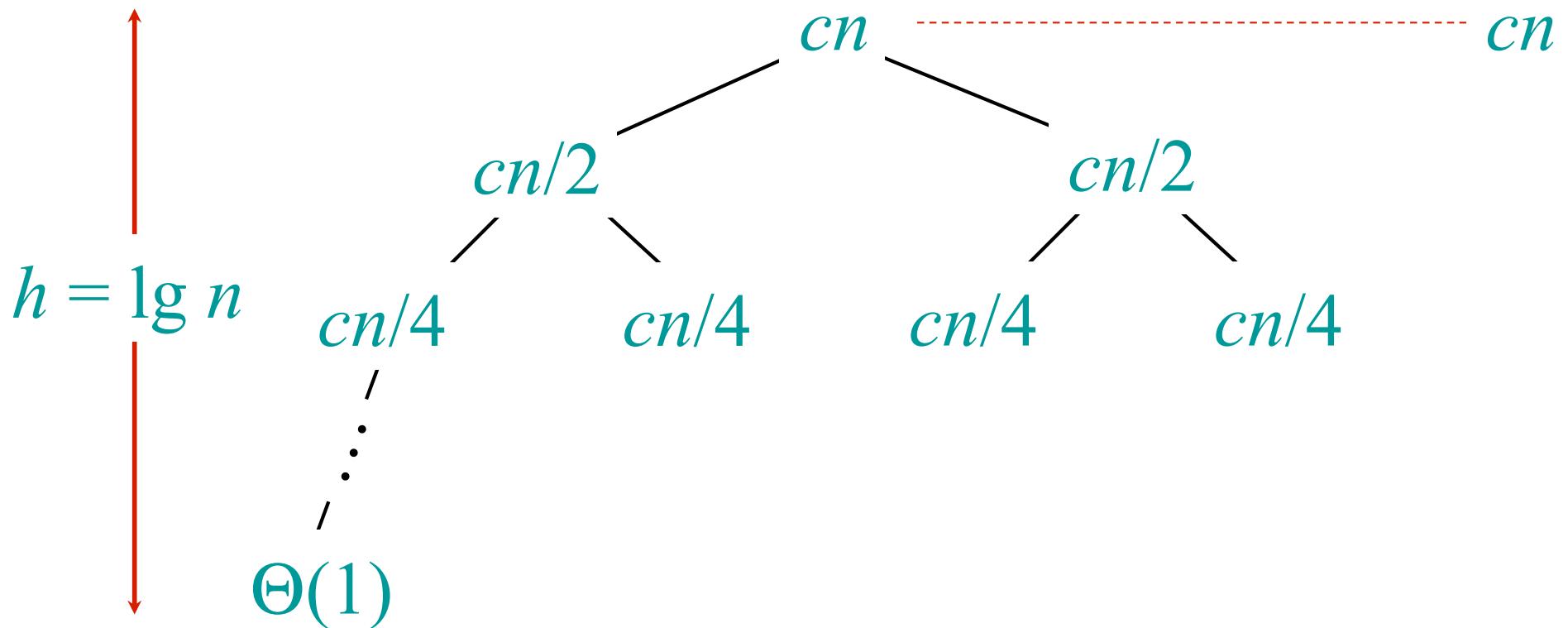
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

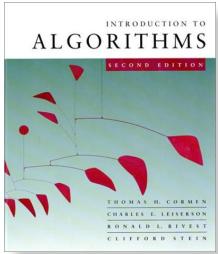




Recursion tree

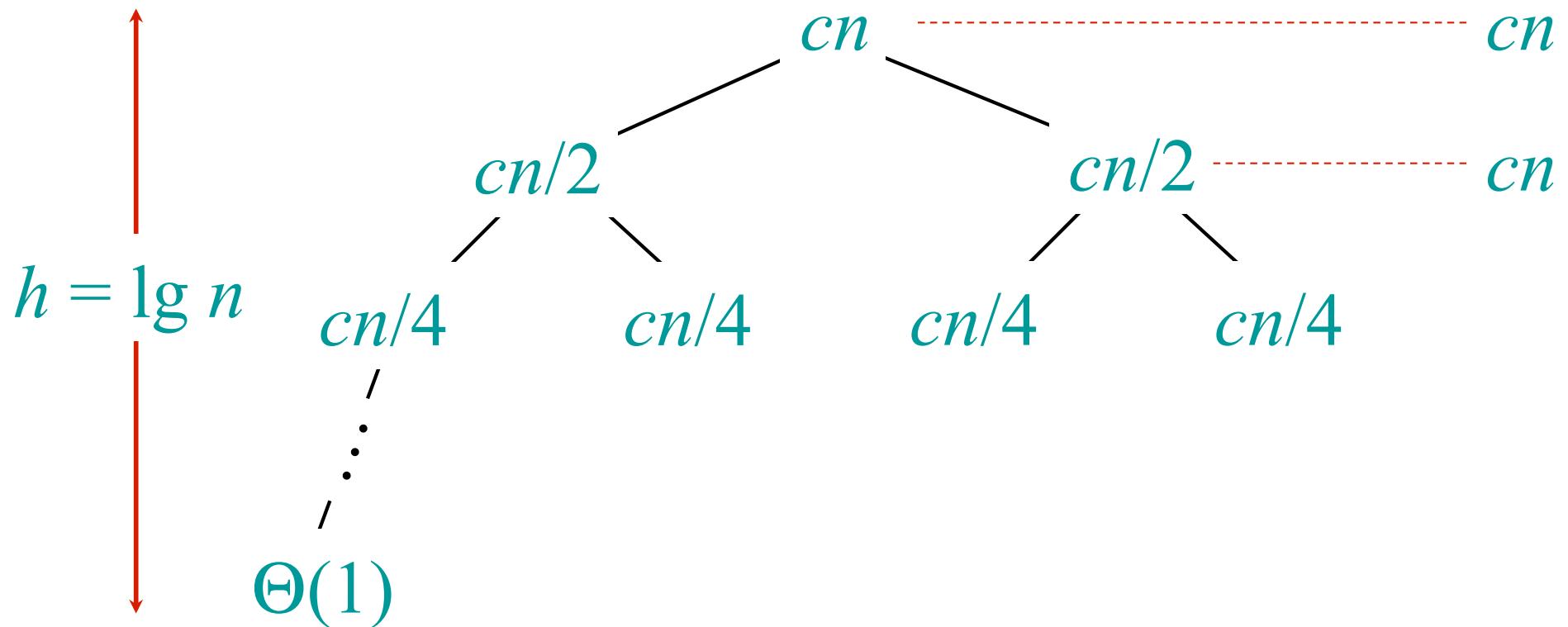
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

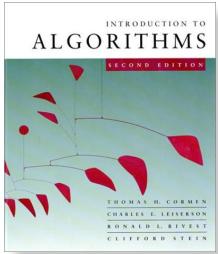




Recursion tree

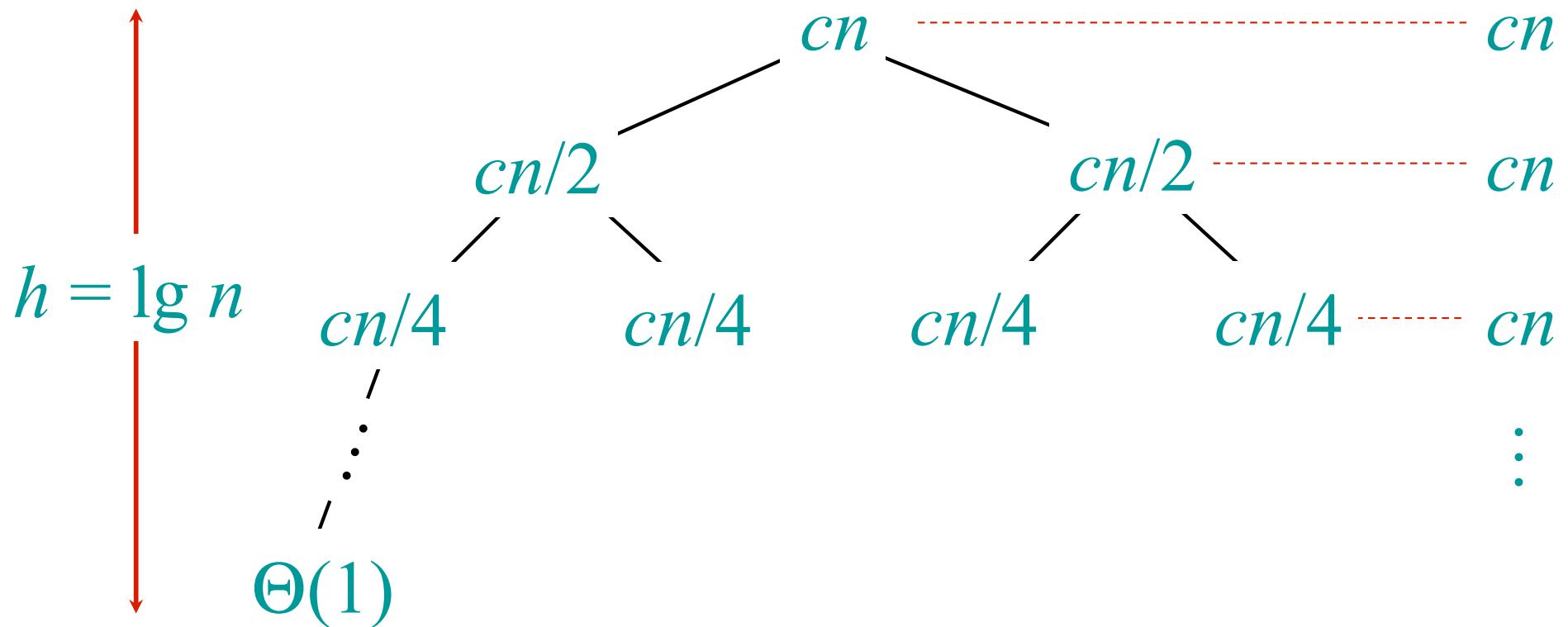
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

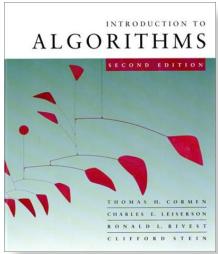




Recursion tree

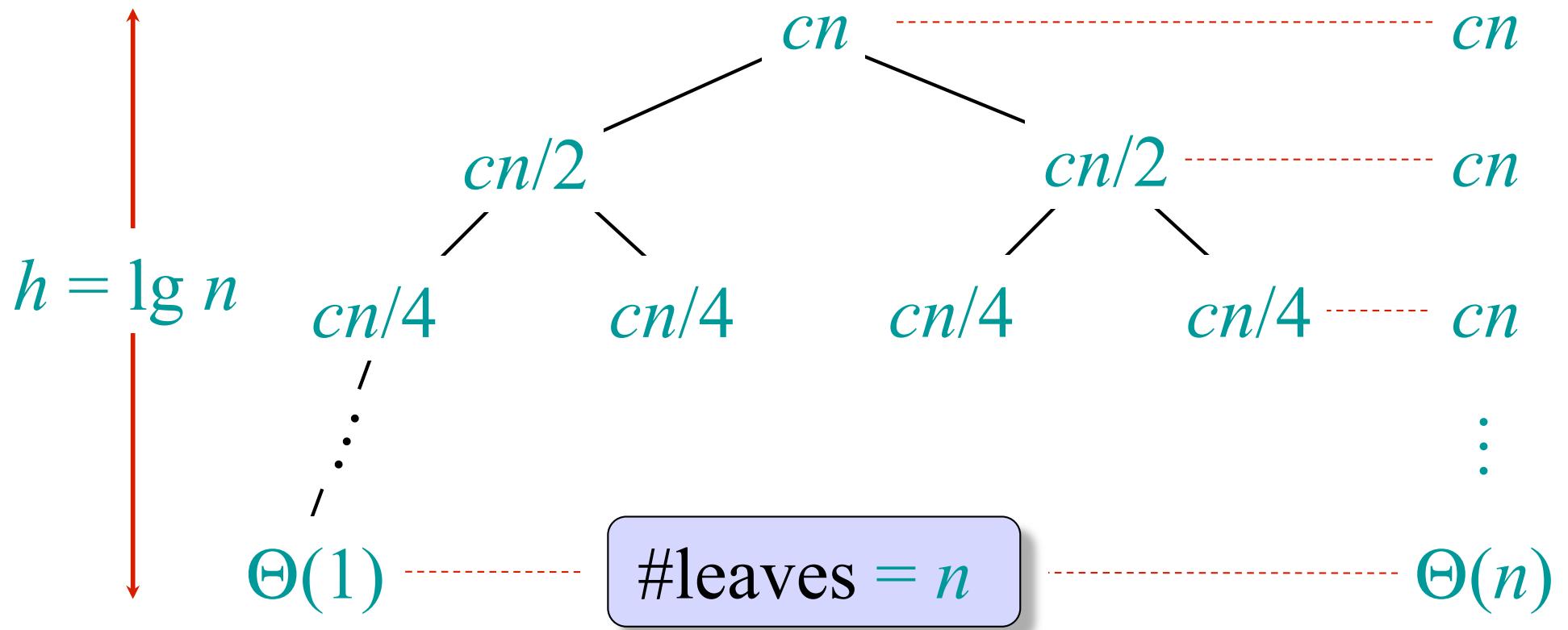
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

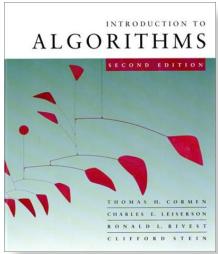




Recursion tree

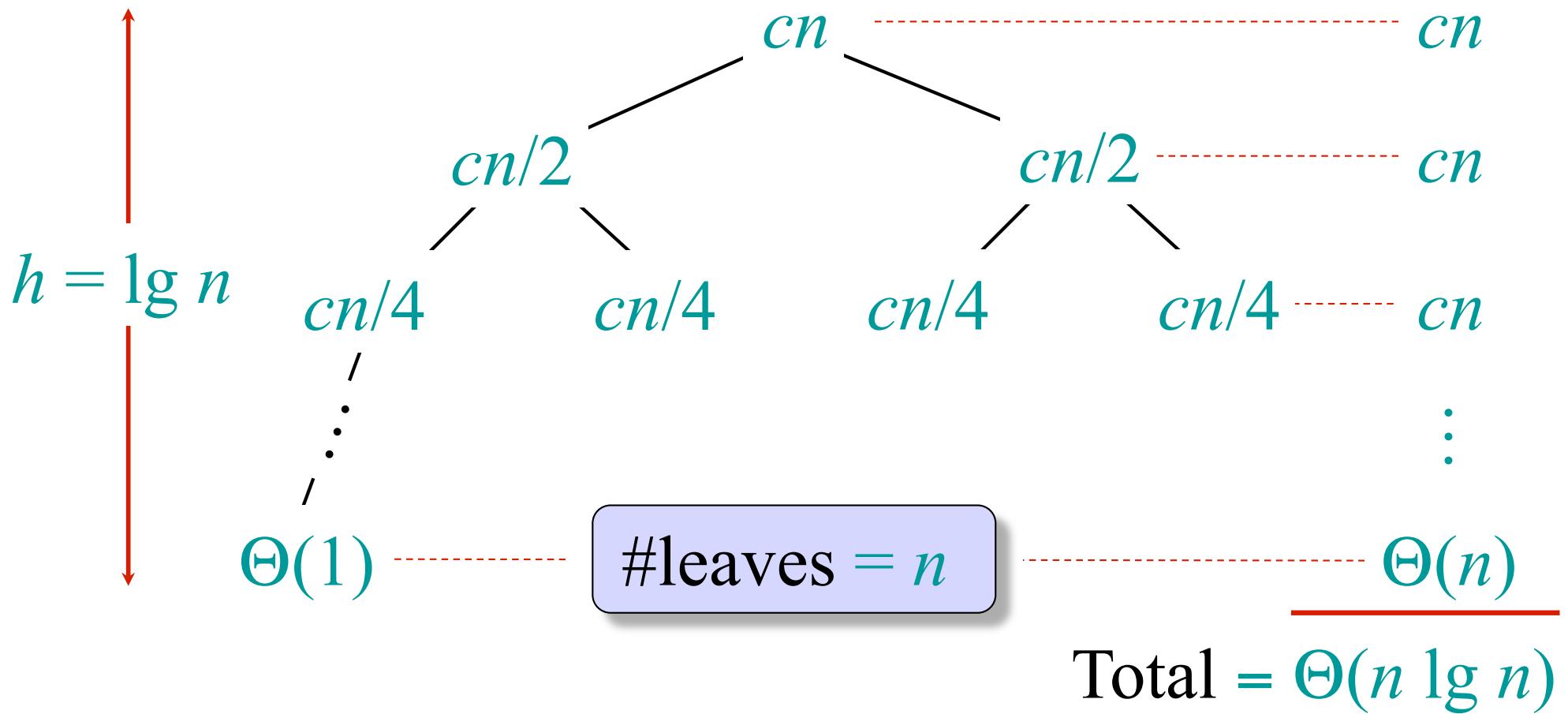
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

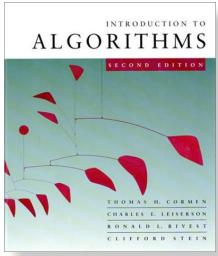




Recursion tree

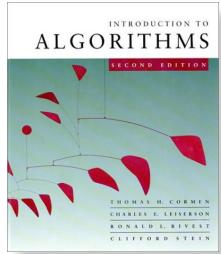
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.





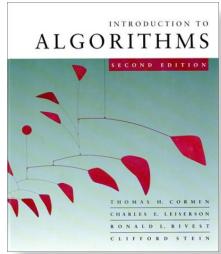
Recursion-tree method

- 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

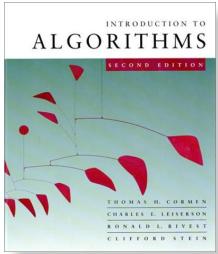
Solve $T(n) = T(n/4) + T(n/2) + n^2$:



Example of recursion tree

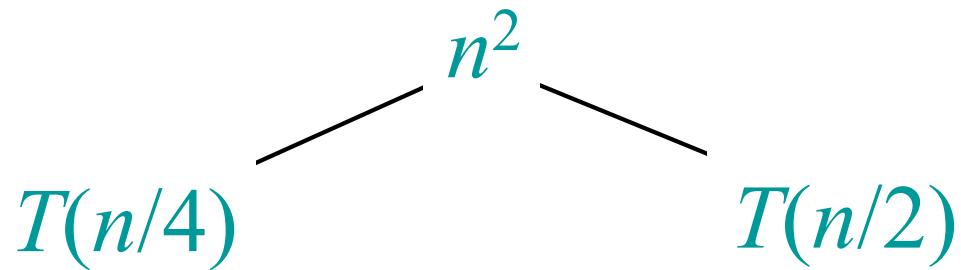
Solve $T(n) = T(n/4) + T(n/2) + n^2$:

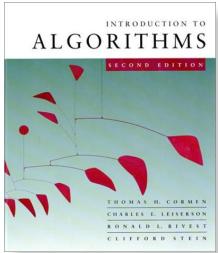
$$T(n)$$



Example of recursion tree

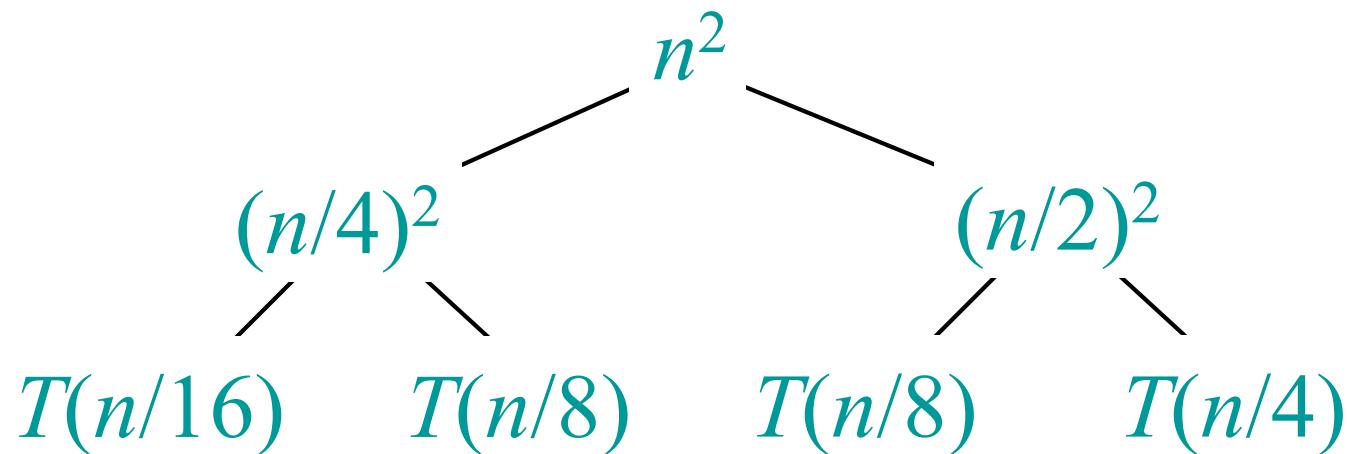
Solve $T(n) = T(n/4) + T(n/2) + n^2$:

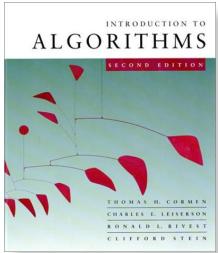




Example of recursion tree

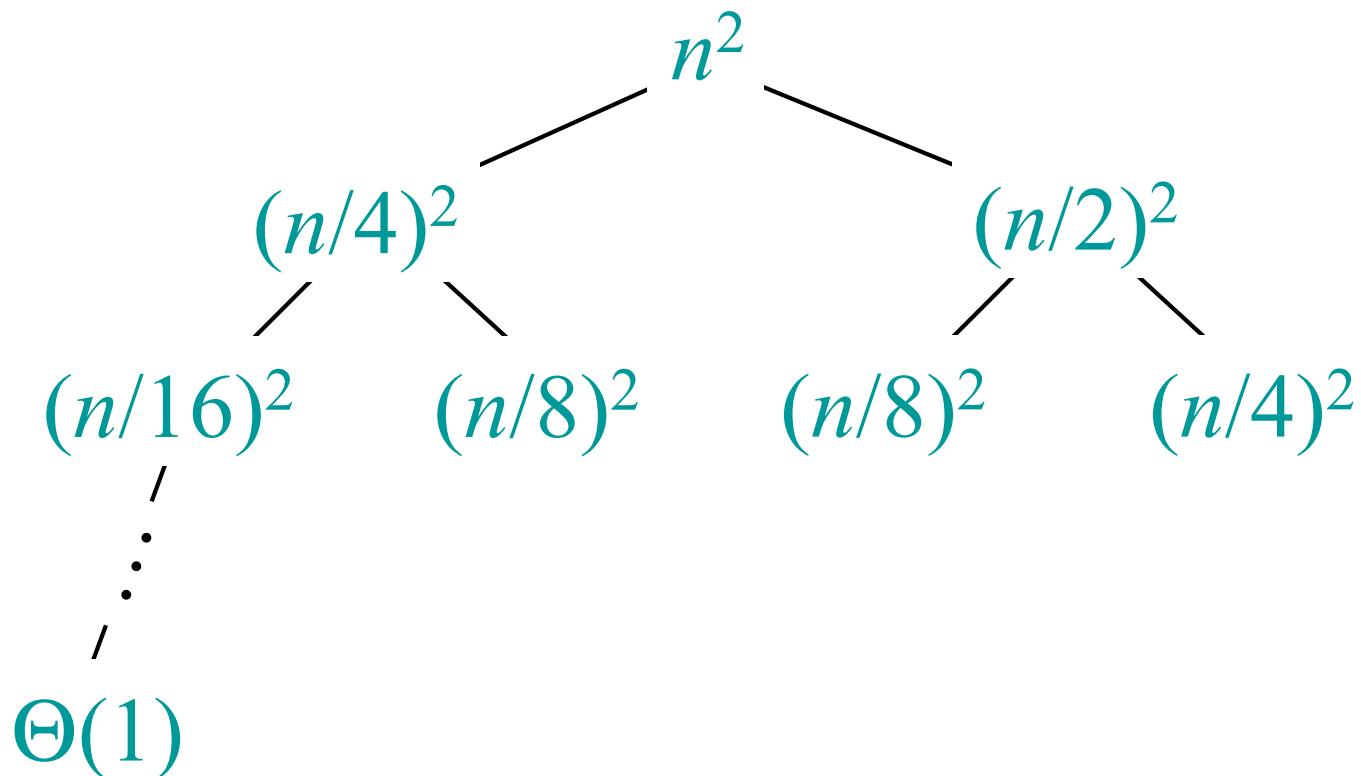
Solve $T(n) = T(n/4) + T(n/2) + n^2$:

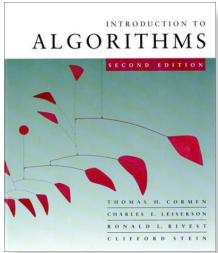




Example of recursion tree

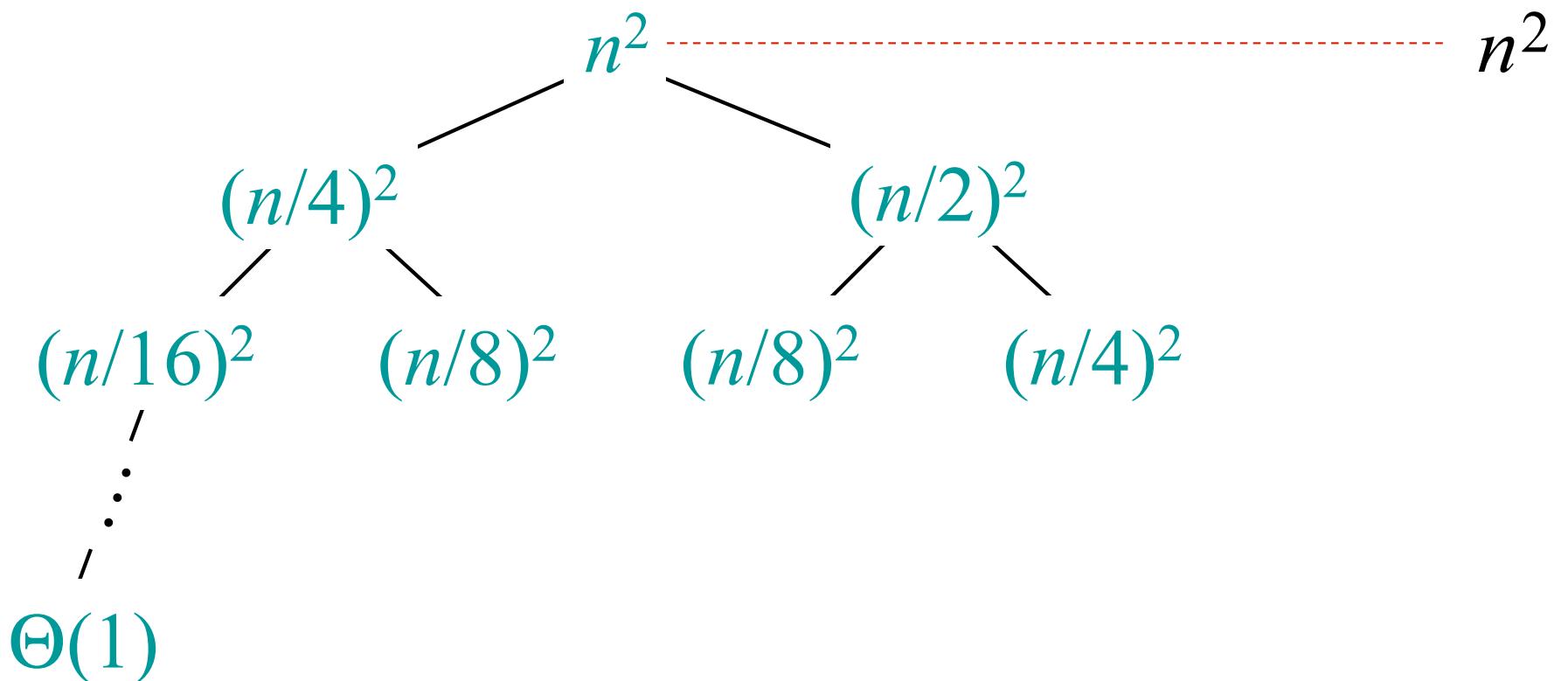
Solve $T(n) = T(n/4) + T(n/2) + n^2$:

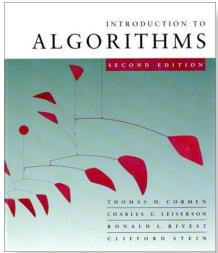




Example of recursion tree

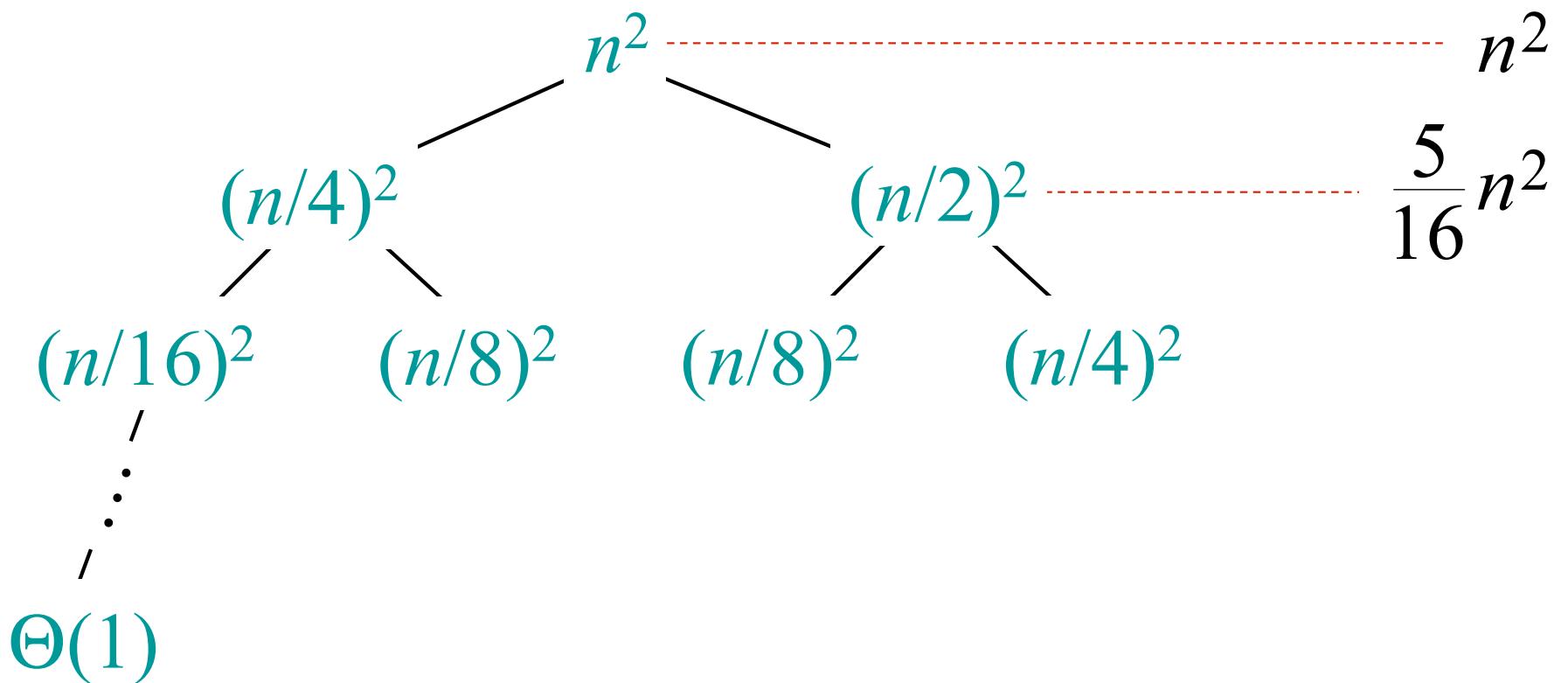
Solve $T(n) = T(n/4) + T(n/2) + n^2$:

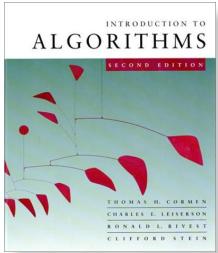




Example of recursion tree

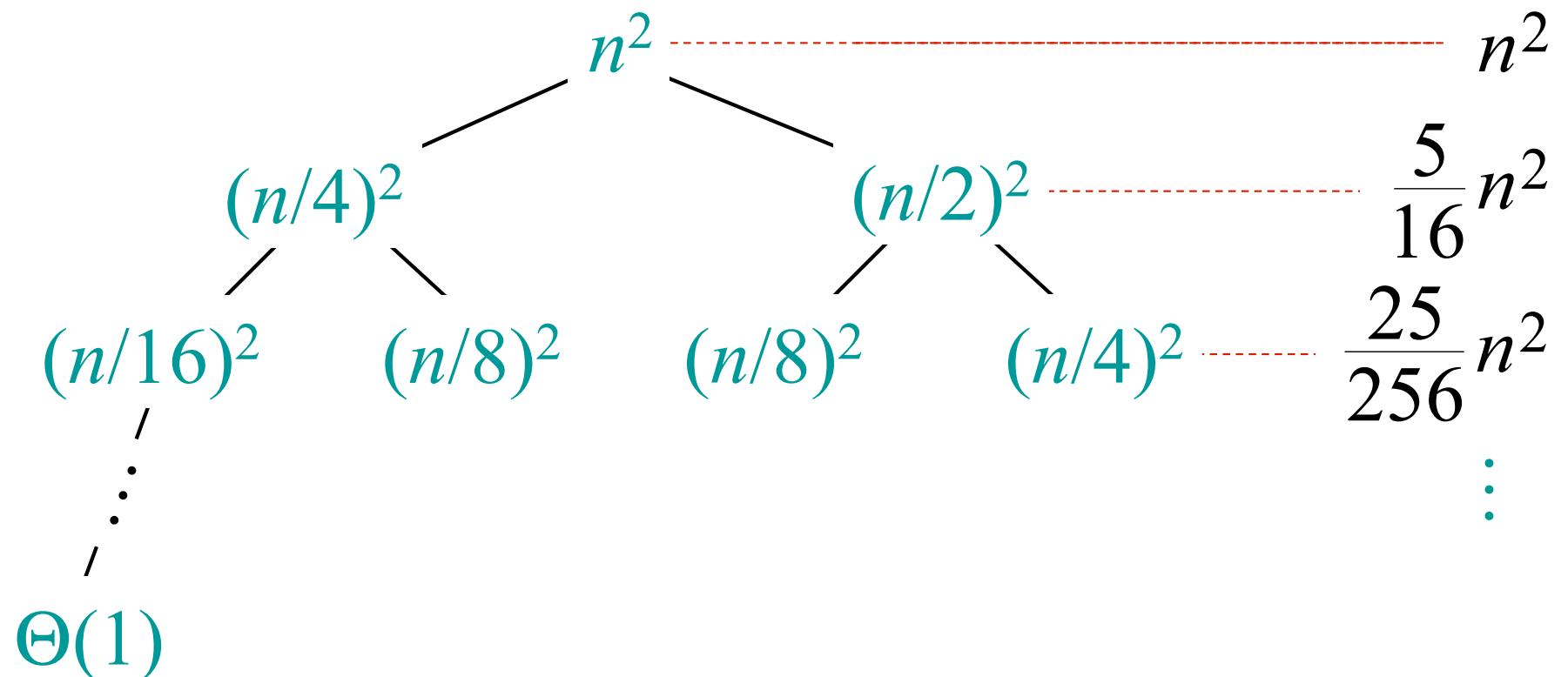
Solve $T(n) = T(n/4) + T(n/2) + n^2$:

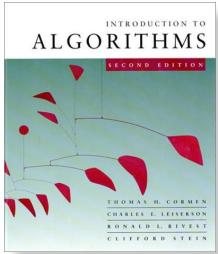




Example of recursion tree

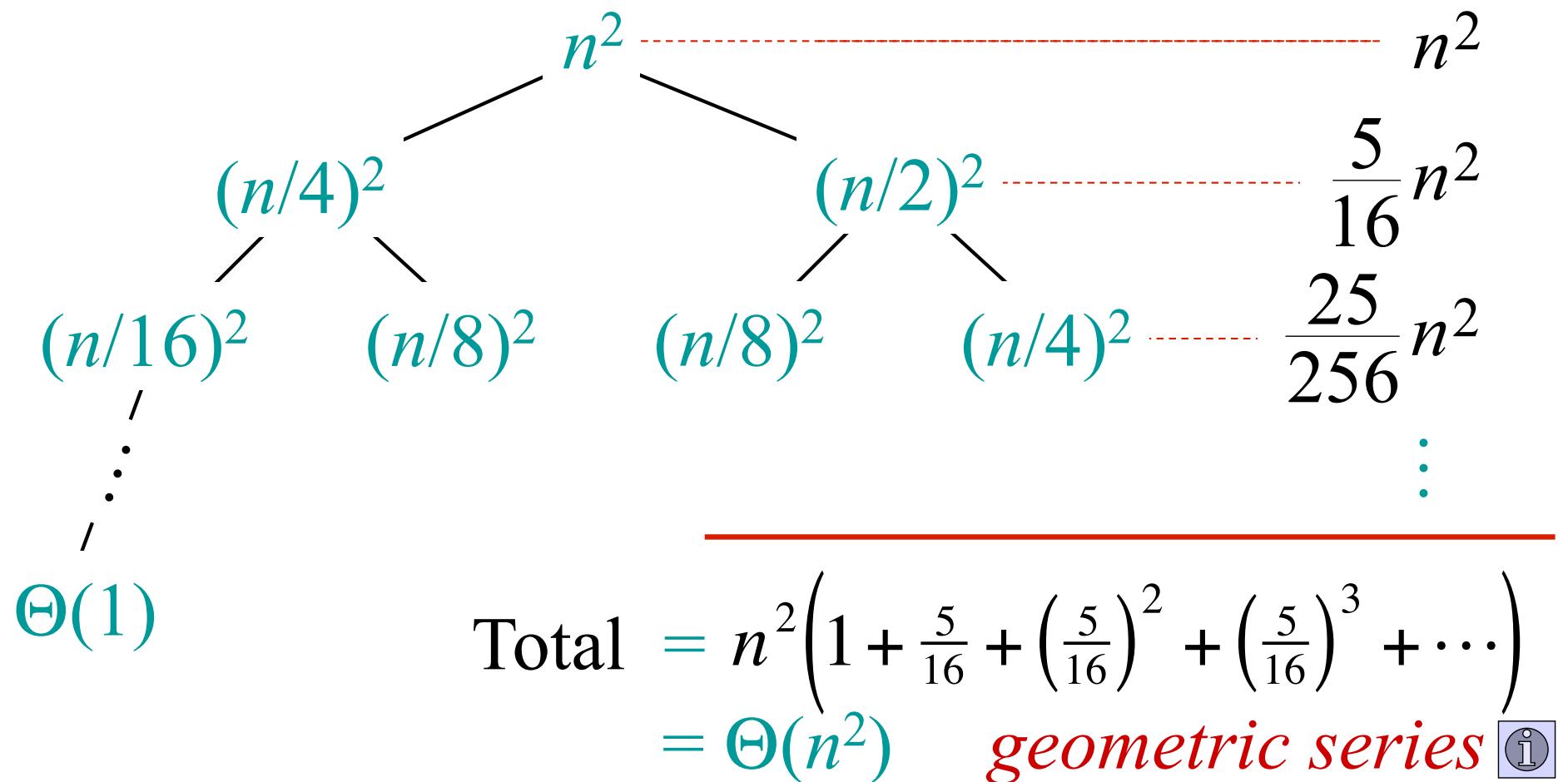
Solve $T(n) = T(n/4) + T(n/2) + n^2$:

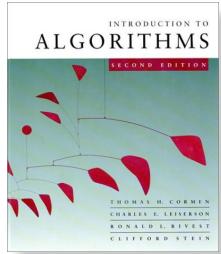




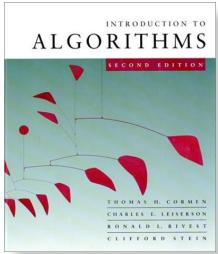
Example of recursion tree

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





The Master Theorem

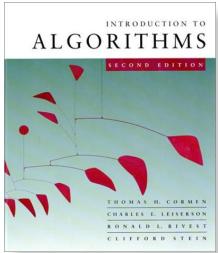


The master method

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.

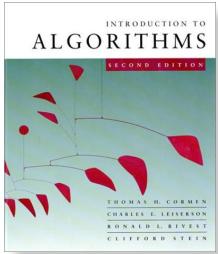


Three common cases

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^ε factor).

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



Three common cases

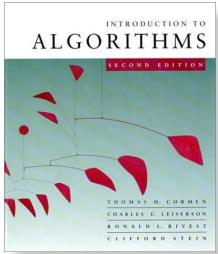
Compare $f(n)$ with $n^{\log_b a}$:

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



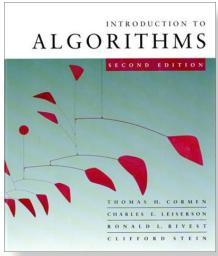
Three common cases (cont.)

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^ε factor),

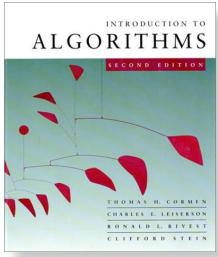
and $f(n)$ satisfies the ***regularity condition*** that $af(n/b) \leq cf(n)$ for some constant $c < 1$.

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 $\varepsilon = 1$.

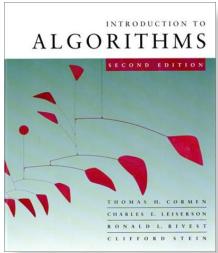
$\therefore T(n) = \Theta(n^2).$

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

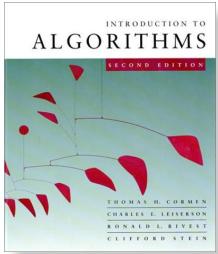
Ex. $T(n) = 4T(n/2) + n^3$

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

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 $\varepsilon = 1$

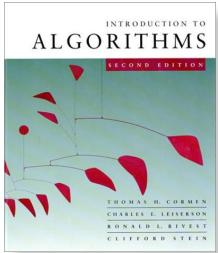
and $4(n/2)^3 \leq cn^3$ (reg. cond.) for $c = 1/2$.

$\therefore T(n) = \Theta(n^3).$

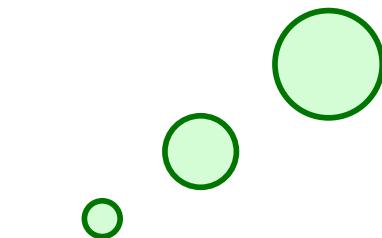
Ex. $T(n) = 4T(n/2) + n^2/\lg n$

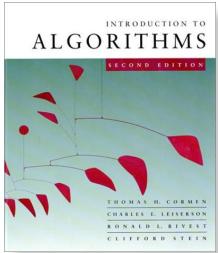
$a = 4, b = 2 \Rightarrow n^{\log_b a} = n^2; f(n) = n^2/\lg n.$

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



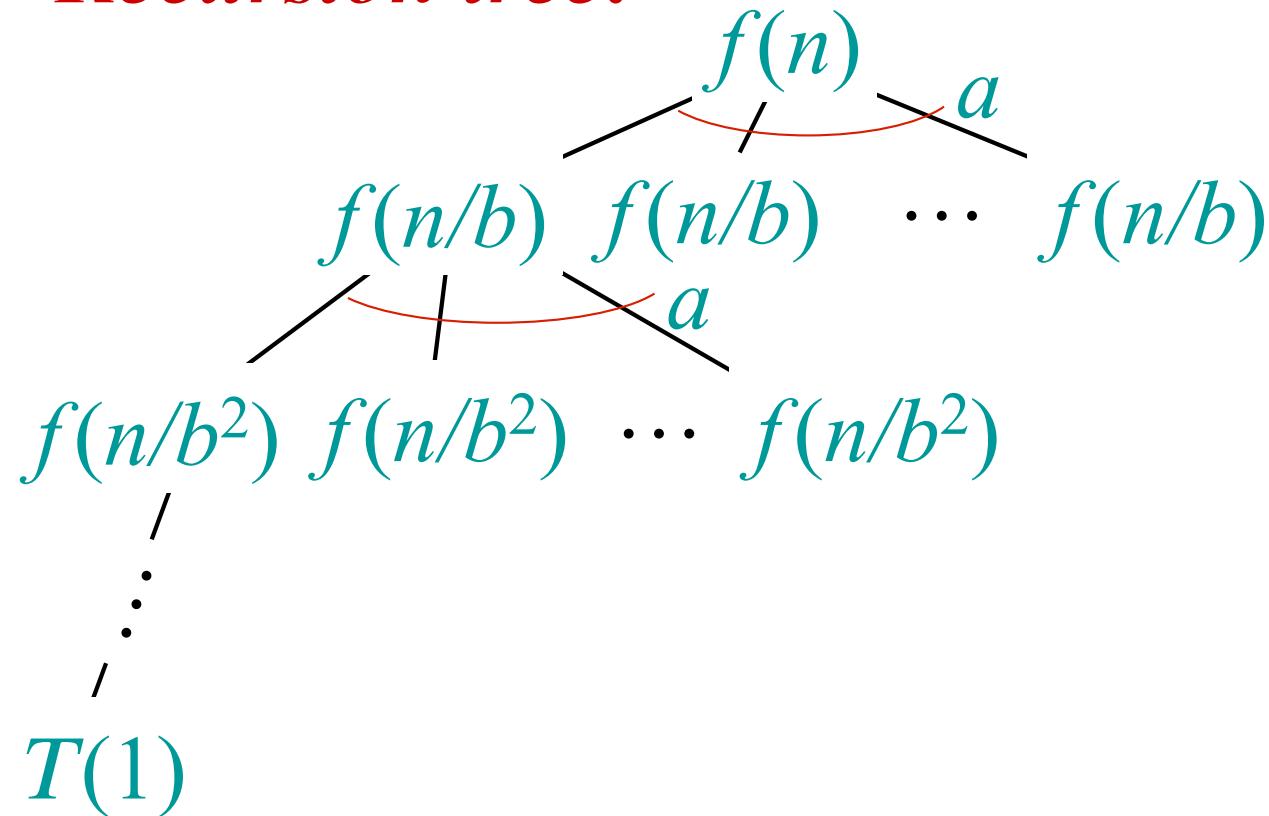
*Idea behind the proof
of the Master Theorem*

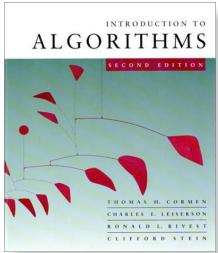




Idea of master theorem

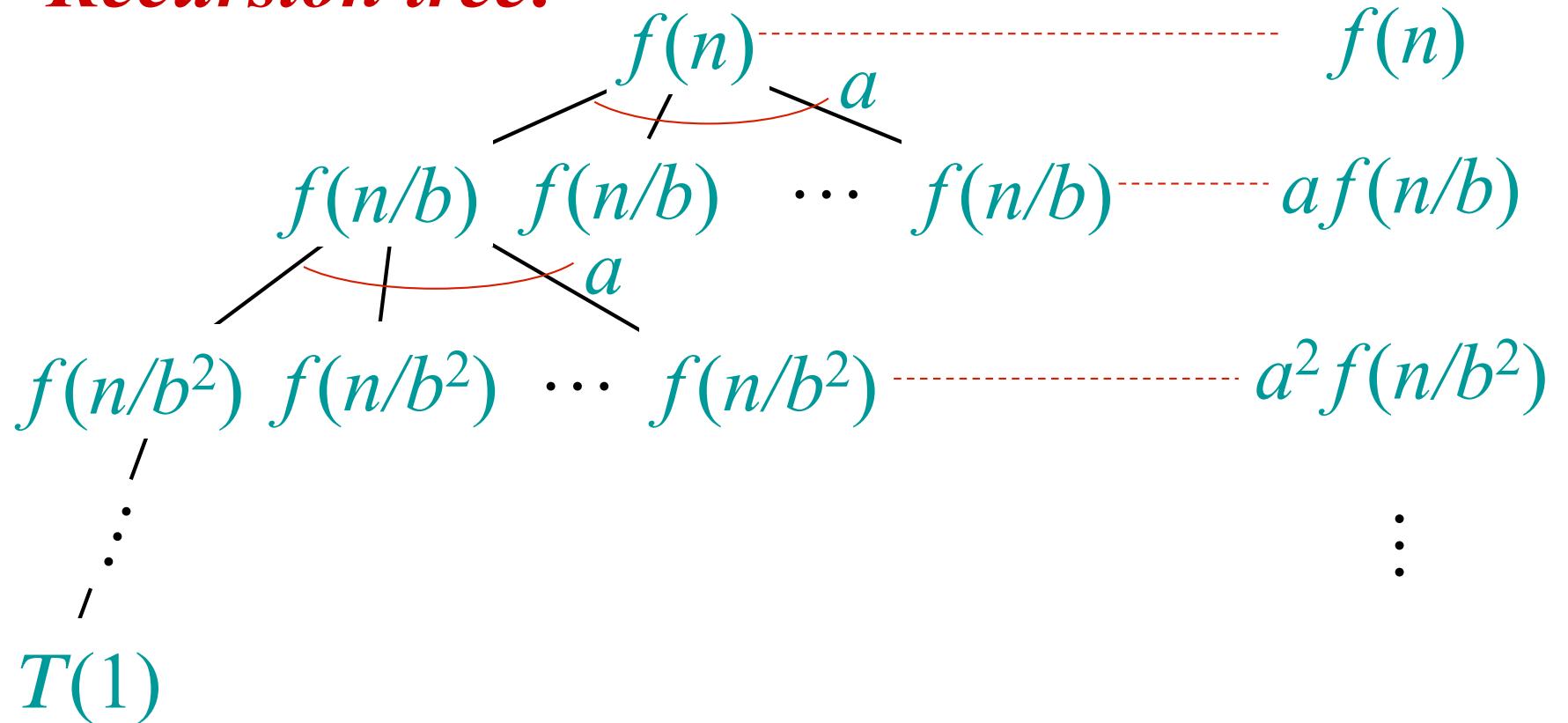
Recursion tree:

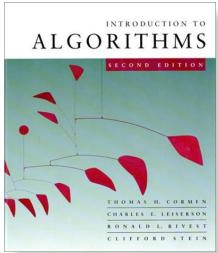




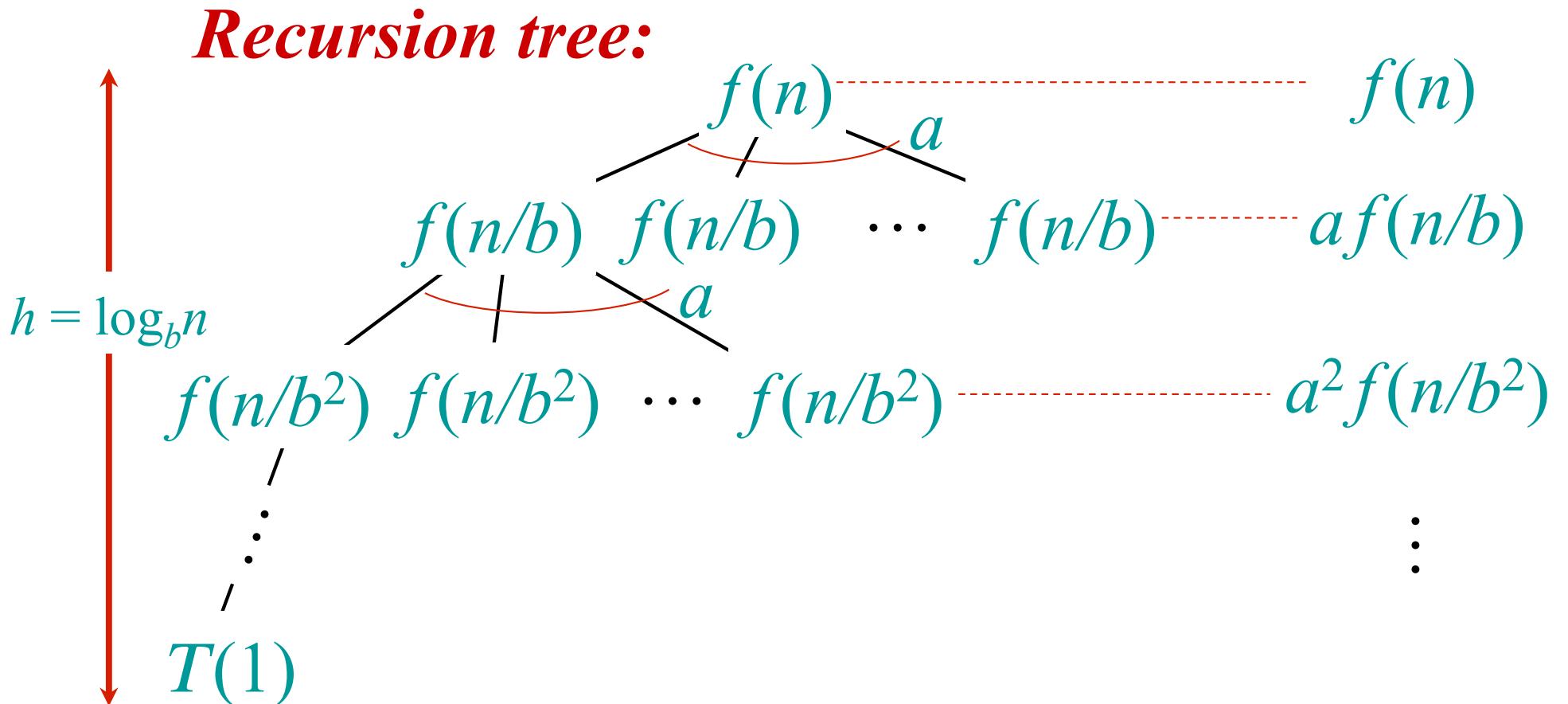
Idea of master theorem

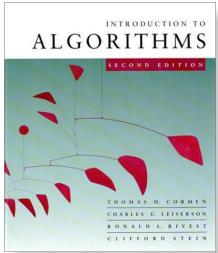
Recursion tree:



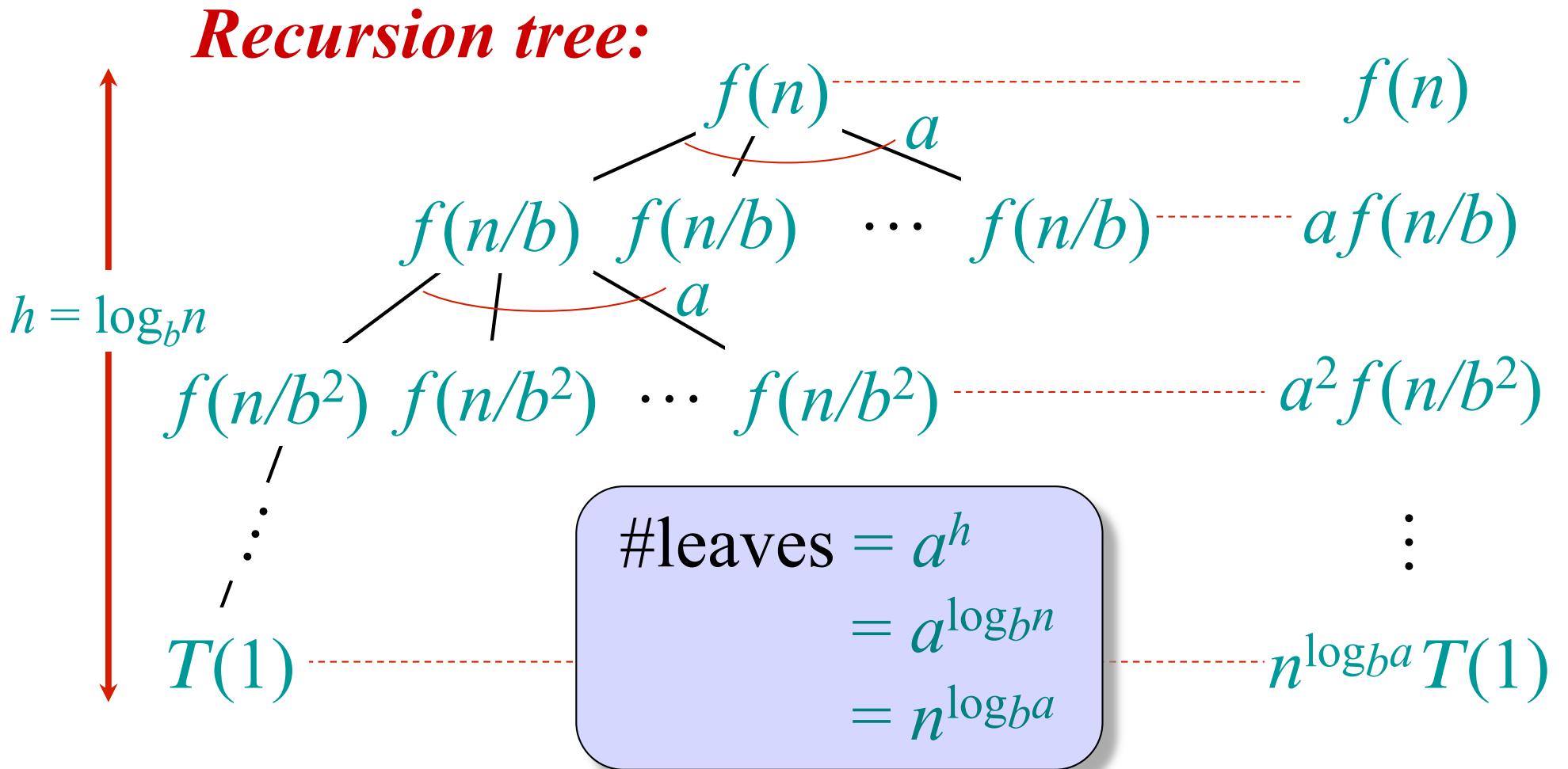


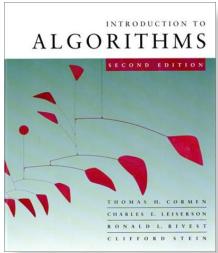
Idea of master theorem



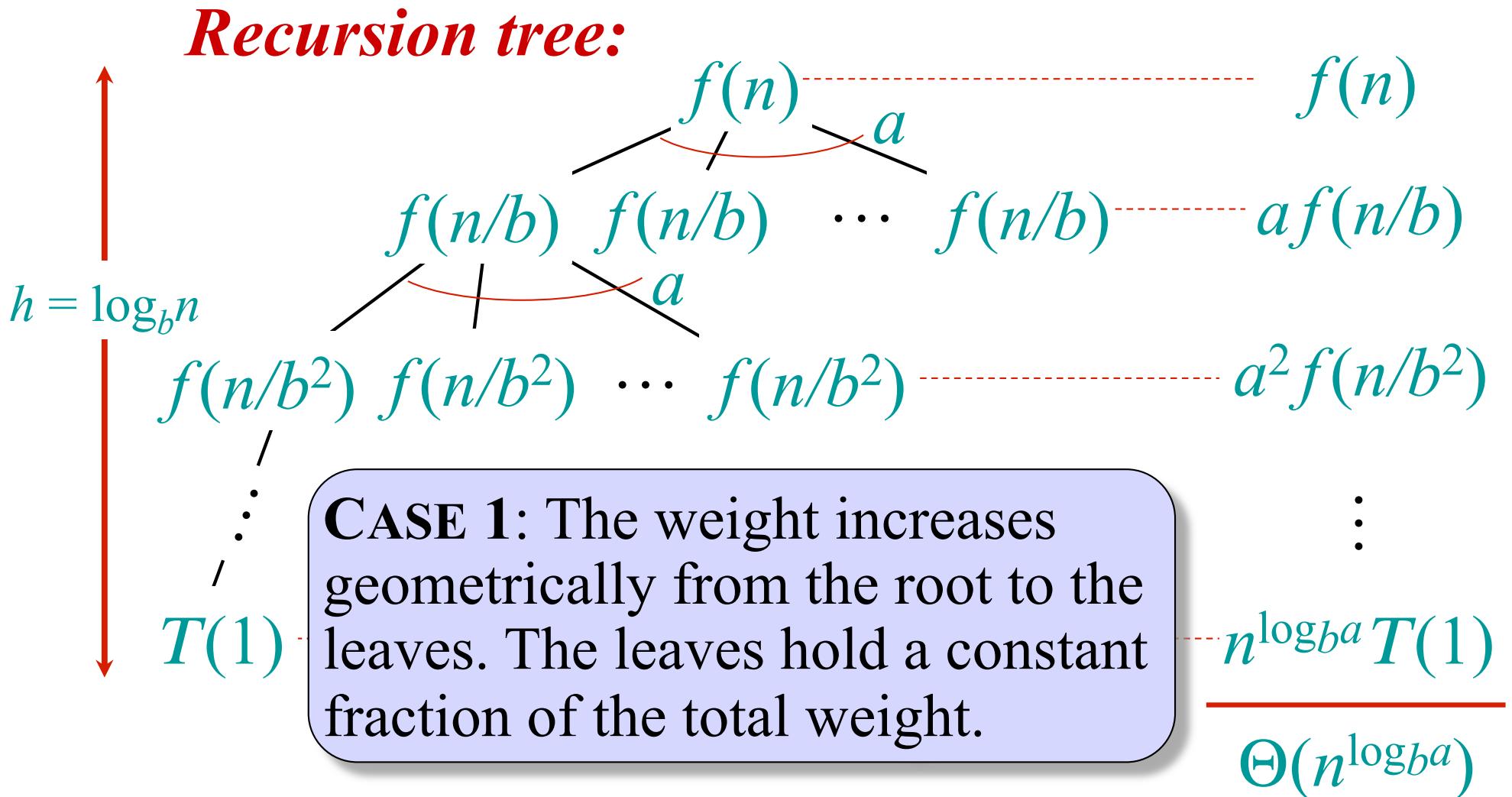


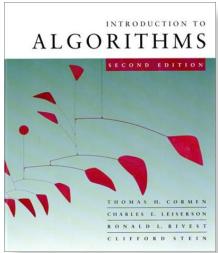
Idea of master theorem





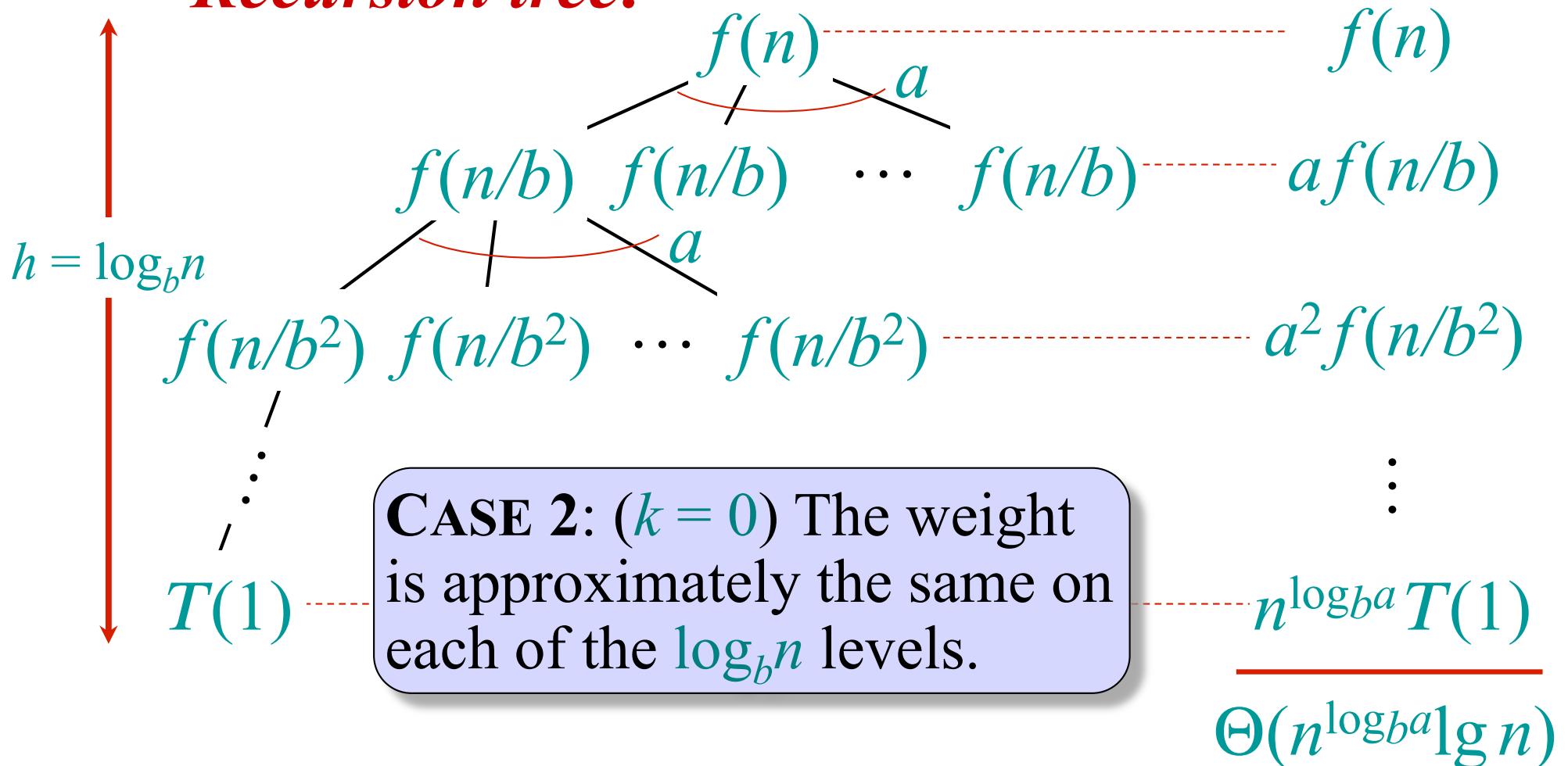
Idea of master theorem

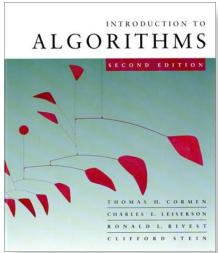




Idea of master theorem

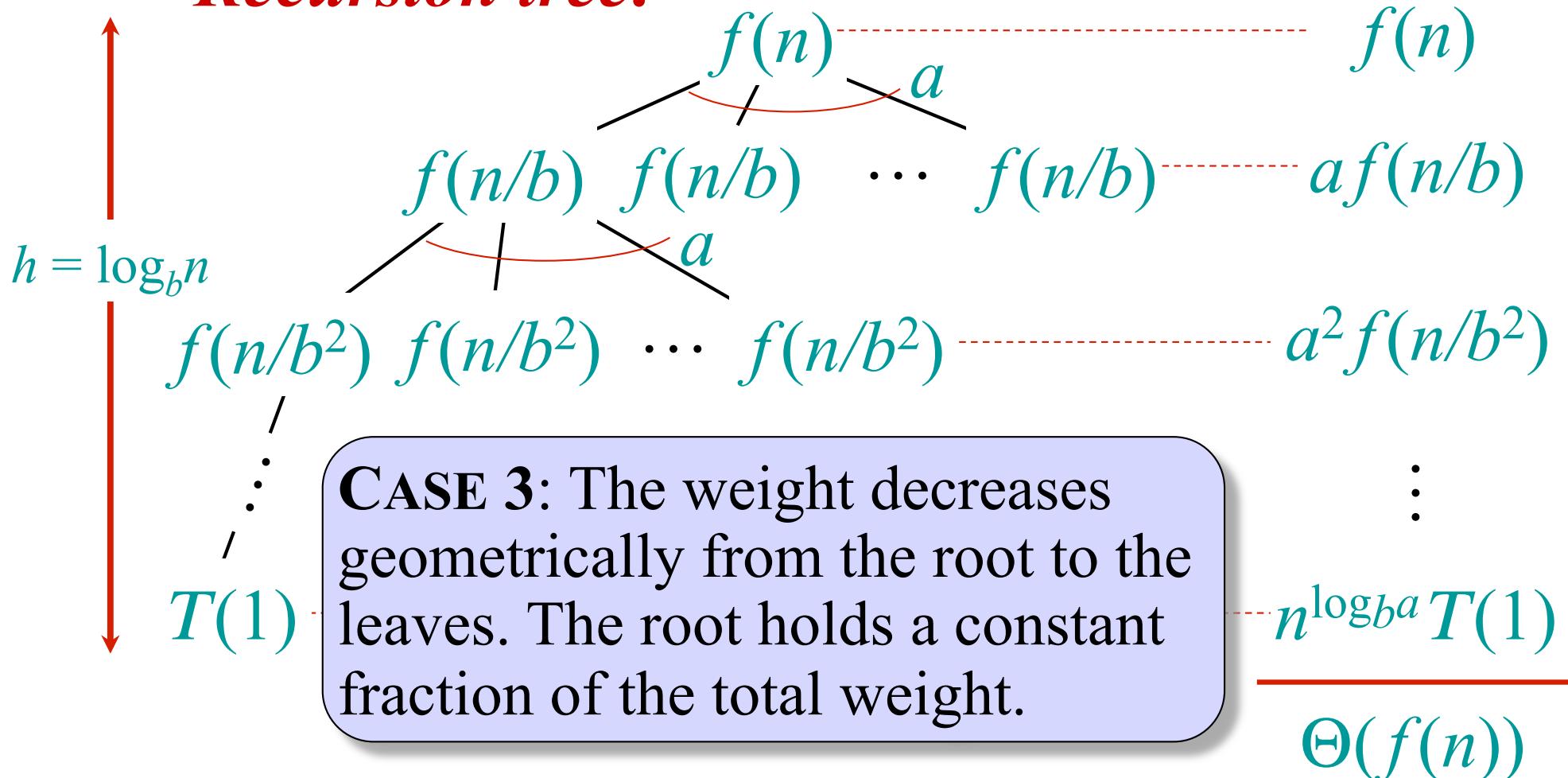
Recursion tree:

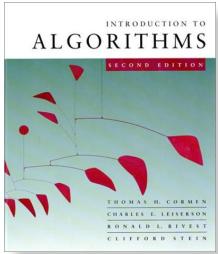




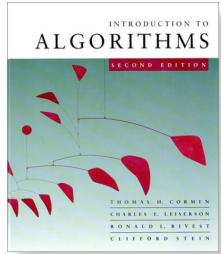
Idea of master theorem

Recursion tree:



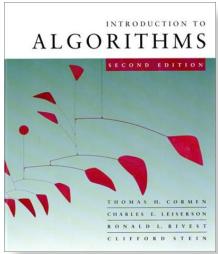


*Revise many
Divide and Conquer
Algorithms*



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.

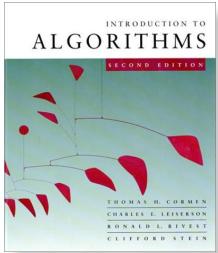


Merge sort

1. Divide: Trivial.

2. Conquer: Recursively sort 2 subarrays.

3. Combine: Linear-time merge.



Merge sort

1. Divide: Trivial.

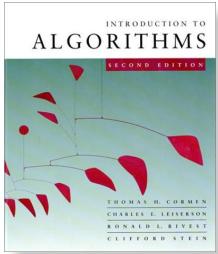
2. Conquer: Recursively sort 2 subarrays.

3. Combine: Linear-time merge.

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

subproblems ↗
subproblem size ↗
work dividing
and combining

The diagram illustrates the recurrence relation for Merge Sort. The equation $T(n) = 2T(n/2) + \Theta(n)$ is shown with three annotations: "# subproblems" points to the coefficient 2, "subproblem size" points to the argument $n/2$, and "work dividing and combining" points to the term $\Theta(n)$.



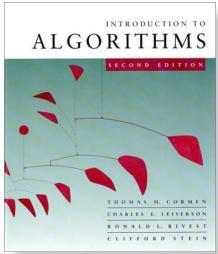
Master theorem (Mergesort)

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

subproblems ↗
 ↓
 subproblem size

work dividing
and combining ↙

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



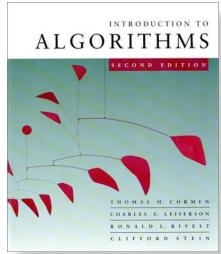
Binary search

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:

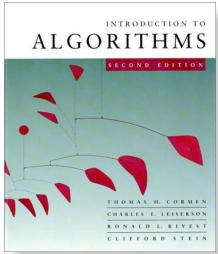
1. Divide: Check middle element.

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:

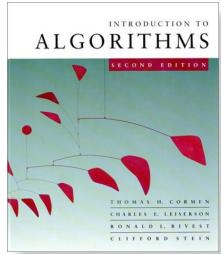
1. Divide: Check middle element.

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:

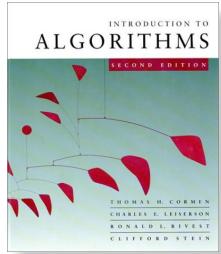
1. Divide: Check middle element.

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:

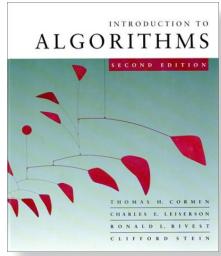
1. Divide: Check middle element.

2. Conquer: Recursively search 1 subarray.

3. Combine: Trivial.

Example: Find 9





Binary search

Find an element in a sorted array:

1. Divide: Check middle element.

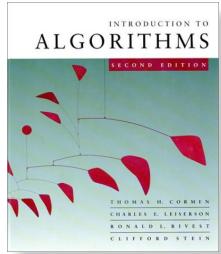
2. Conquer: Recursively search 1 subarray.

3. Combine: Trivial.

Example: Find 9

3 5 7 8 9 12 15





Binary search

Find an element in a sorted array:

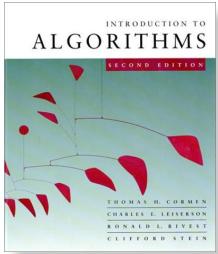
1. Divide: Check middle element.

2. Conquer: Recursively search 1 subarray.

3. Combine: Trivial.

Example: Find 9



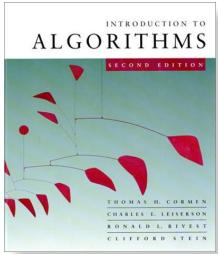


Recurrence for binary search

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

subproblems ↗
subproblem size ↗
work dividing
and combining

The equation $T(n) = 1T(n/2) + \Theta(1)$ is displayed with two yellow circles highlighting the terms $1T(n/2)$ and $\Theta(1)$. Three arrows point from the text below to these highlighted terms: one arrow from "# subproblems" points to the first $T(n/2)$, another from "subproblem size" points to the same term, and a third from "work dividing and combining" points to the $\Theta(1)$.

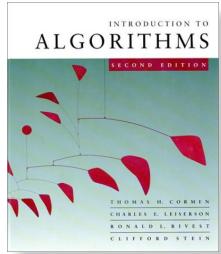


Recurrence for binary search

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

subproblems ↗
subproblem size ↗
work dividing
and combining

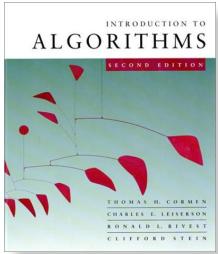
$$\begin{aligned} n^{\log_b a} &= n^{\log_2 1} = n^0 = 1 \Rightarrow \text{CASE 2 } (k = 0) \\ \Rightarrow T(n) &= \Theta(\lg n) . \end{aligned}$$



Powering a number

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

Naive algorithm: $\Theta(n)$.



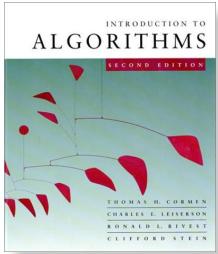
Powering a number

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

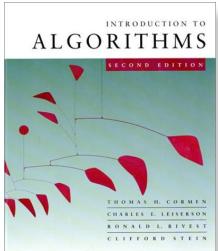
Problem: Compute a^n , where $n \in \mathbb{N}$.

Naive algorithm: $\Theta(n)$.

Divide-and-conquer algorithm:

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

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

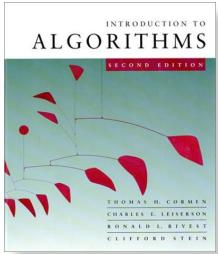


Matrix multiplication

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

$$c_{11} = (a_{11} \cdot b_{11} + a_{12} \cdot b_{21} + a_{13} \cdot b_{31} + \dots + a_{1n} \cdot b_{n1})$$

$$c_{ij} = (a_{i1} \cdot b_{1j} + a_{i2} \cdot b_{2j} + a_{i3} \cdot b_{3j} + \dots + a_{in} \cdot b_{nj})$$

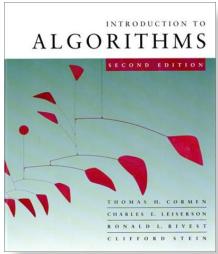


Matrix multiplication

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

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

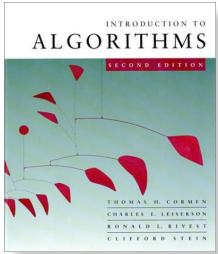
$$c_{ij} = \sum_{k=1}^n a_{ik} \cdot b_{kj}$$



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  $c_{ij} \leftarrow c_{ij} + a_{ik} \cdot b_{kj}$ 
```

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



Divide-and-conquer algorithm

IDEA:

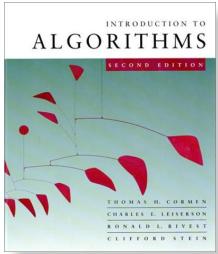
$n \times n$ matrix = 2×2 matrix of $(n/2) \times (n/2)$ submatrices:

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

$$C = A \cdot B$$

$$\left. \begin{array}{l} r = ae + bg \\ s = af + bh \\ t = ce + dg \\ u = cf + dh \end{array} \right\}$$

8 mults of $(n/2) \times (n/2)$ submatrices
4 adds of $(n/2) \times (n/2)$ submatrices



Divide-and-conquer algorithm

IDEA:

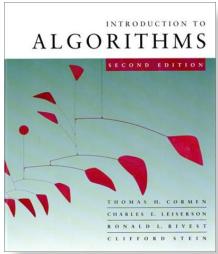
$n \times n$ matrix = 2×2 matrix of $(n/2) \times (n/2)$ submatrices:

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

$$C = A \cdot B$$

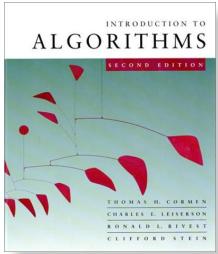
$$\left. \begin{array}{l} r = ae + bg \\ s = af + bh \\ t = ce + dh \\ u = cf + dg \end{array} \right\} \text{recursive}$$

$\underbrace{8}_{4}$ mults of $(n/2) \times (n/2)$ submatrices
4 adds of $(n/2) \times (n/2)$ submatrices



Standard algorithm

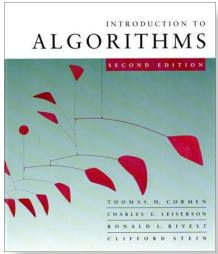
```
for  $i \leftarrow 1$  to  $n$ 
    do for  $j \leftarrow 1$  to  $n$ 
        do  $c_{ij} \leftarrow 0$ 
        for  $k \leftarrow 1$  to  $n$ 
            do  $c_{ij} \leftarrow c_{ij} + a_{ik} \cdot b_{kj}$ 
```



Analysis of D&C algorithm

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

submatrices ↗ work adding
 ↓ submatrices

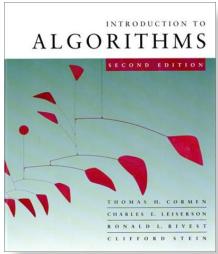


Analysis of D&C algorithm

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

submatrices ↗
submatrix size ↗
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

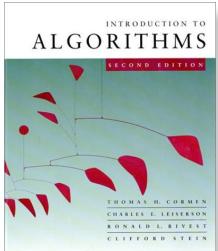
$$T(n) = 8T(n/2) + \Theta(n^2)$$

submatrices ↗
 ↓
 submatrix size
 ↗
 work adding
 submatrices

The equation $T(n) = 8T(n/2) + \Theta(n^2)$ is displayed in the center. Two arrows point from the text "submatrix size" to the term $n/2$ in the recurrence relation. One arrow points from "# submatrices" to the coefficient 8, and another arrow points from "work adding submatrices" to the term $\Theta(n^2)$.

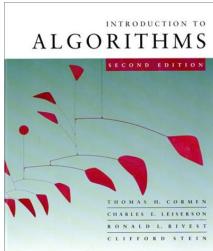
$$n^{\log_b a} = n^{\log_2 8} = n^3 \Rightarrow \text{CASE 1} \Rightarrow T(n) = \Theta(n^3).$$

No better than the ordinary algorithm.



Strassen's idea

- Multiply 2×2 matrices with only 7 recursive mults.



Strassen's idea

- Multiply 2×2 matrices with only 7 recursive mults.

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

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

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

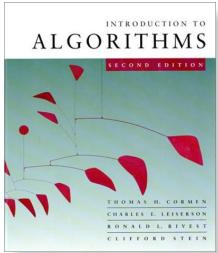
$$P_4 = d \cdot (g - e)$$

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

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

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





Strassen's idea

- Multiply 2×2 matrices with only 7 recursive mults.

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

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

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

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

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

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

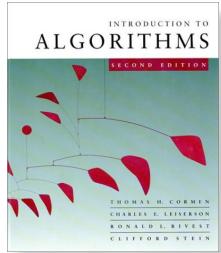
$$P_7 = (a - c) \cdot (e + f)$$

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

$$s = P_1 + P_2$$

$$t = P_3 + P_4$$

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



Strassen's idea

- Multiply 2×2 matrices with only 7 recursive mults.

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

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

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

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

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

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

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

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

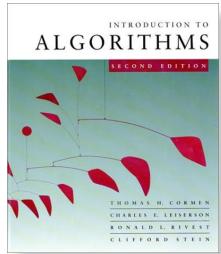
$$s = P_1 + P_2$$

$$t = P_3 + P_4$$

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

7 mults, 18 adds/subs.

Note: No reliance on commutativity of mult!



Strassen's idea

- Multiply 2×2 matrices with only 7 recursive mults.

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

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

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

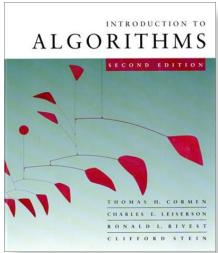
$$P_4 = d \cdot (g - e)$$

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

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

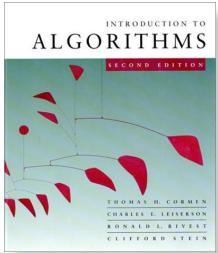
$$P_7 = (a - c) \cdot (e + f)$$

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



Strassen's algorithm

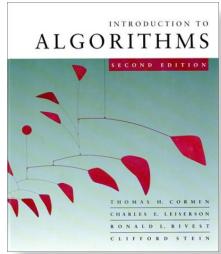
- 1. Divide:** Partition A and B into $(n/2) \times (n/2)$ submatrices. Form terms to be multiplied using $+$ and $-$.
- 2. Conquer:** Perform 7 multiplications of $(n/2) \times (n/2)$ submatrices recursively.
- 3. Combine:** Form C using $+$ and $-$ on $(n/2) \times (n/2)$ submatrices.



Strassen's algorithm

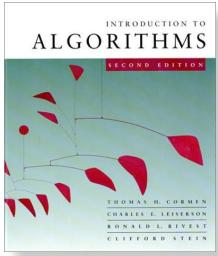
- 1. Divide:** Partition A and B into $(n/2) \times (n/2)$ submatrices. Form terms to be multiplied using $+$ and $-$.
- 2. Conquer:** Perform 7 multiplications of $(n/2) \times (n/2)$ submatrices recursively.
- 3. Combine:** Form C using $+$ and $-$ on $(n/2) \times (n/2)$ submatrices.

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



Analysis of Strassen

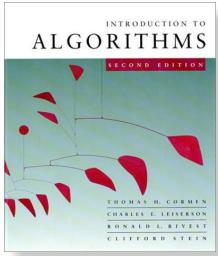
$$T(n) = 7 T(n/2) + \Theta(n^2)$$



Analysis of Strassen

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

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

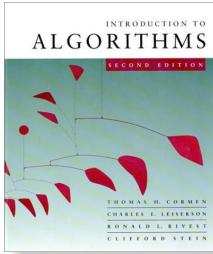


Analysis of Strassen

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

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

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.



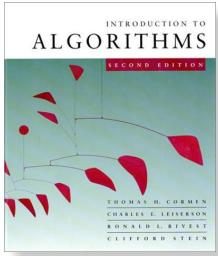
Analysis of Strassen

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

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

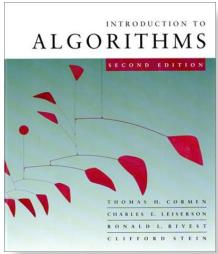
The number 2.81 may not seem much smaller than 3 , but because the difference is in the exponent, the impact on running time is significant. In fact, Strassen's algorithm beats the ordinary algorithm on today's machines for $n \geq 32$ or so.

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



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.



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!

Thank you.

Q & A



School *of* Computing