THOMAS H. CORMEN
CHARLES E. LEISERSON
RONALD L. RIVEST
CLIFFORD STEIN

Over
1 MILLION
copies sold
worldwide

INTRODUCTION TO ALGORITHMS

FOURTH EDITION

CSCE 500

Design and Analysis of Algorithms

Fall 2024

August 26, 2024

Instructor: Nian-Feng Tzeng
Office: Rm. OLVR 354 (× 2-6304; tzeng@louisiana.edu)
Class meeting: MW 10:00 – 11:15, OLVR 113

Textbook and Supplemental Materials:

1. Introduction to Algorithms, Fourth Edition, by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, The MIT Press, 2022, ISBN: 978–026204630–5.
2. Published articles supplementary to covered topics.

Course Description:

This course provides a comprehensive coverage of modern computer algorithms, aiming at in-depth treatment of algorithmic design and analysis with elementary explanation while keeping mathematical rigor. Based on the textbook of "Introduction to Algorithms", this class covers the topics listed below in sequence.

(1) Foundations.
(2) Data Structures – hash tables, binary search trees, red-black trees, B-trees.
(3) Design and Analysis Techniques – dynamic programming, greedy algorithms.
(4) Graph Algorithms – elementary graph algorithms, algorithms for shortest paths, maximum flows, minimum spanning trees.
(5) Selected Topics – NP-completeness, approximation algorithms, multithreaded algorithms.

Each covered topic starts with the description of pertinent algorithms in English and/or in the pseudocode(s), followed by their careful complexity analyses.

Course Requirements:

1. Homework assignments (2) (10%)
2. Midterm exams (2) (50%)
3. Final exam (comprehensive) (40%)

Table of Contents

Preface *xiii*

I *Foundations*

Introduction *3*

1 The Role of Algorithms in Computing *5*
- 1.1 Algorithms *5*
- 1.2 Algorithms as a technology *12*

2 Getting Started *17*
- 2.1 Insertion sort *17*
- 2.2 Analyzing algorithms *25*
- 2.3 Designing algorithms *34*

3 Characterizing Running Times *49*
- 3.1 O-notation, Ω-notation, and Θ-notation *50*
- 3.2 Asymptotic notation: formal definitions *53*
- 3.3 Standard notations and common functions *63*

4 Divide-and-Conquer *76*
- 4.1 Multiplying square matrices *80*
- 4.2 Strassen's algorithm for matrix multiplication *85*
- 4.3 The substitution method for solving recurrences *90*
- 4.4 The recursion-tree method for solving recurrences *95*
- 4.5 The master method for solving recurrences *101*
- ★ 4.6 Proof of the continuous master theorem *107*
- ★ 4.7 Akra-Bazzi recurrences *115*

Table of Contents

5 Probabilistic Analysis and Randomized Algorithms *126*
- 5.1 The hiring problem *126*
- 5.2 Indicator random variables *130*
- 5.3 Randomized algorithms *134*
- ★ 5.4 Probabilistic analysis and further uses of indicator random variables *140*

II Sorting and Order Statistics

Introduction *157*

6 Heapsort *161*
- 6.1 Heaps *161*
- 6.2 Maintaining the heap property *164*
- 6.3 Building a heap *167*
- 6.4 The heapsort algorithm *170*
- 6.5 Priority queues *172*

7 Quicksort *182*
- 7.1 Description of quicksort *183*
- 7.2 Performance of quicksort *187*
- 7.3 A randomized version of quicksort *191*
- 7.4 Analysis of quicksort *193*

8 Sorting in Linear Time *205*
- 8.1 Lower bounds for sorting *205*
- 8.2 Counting sort *208*
- 8.3 Radix sort *211*
- 8.4 Bucket sort *215*

9 Medians and Order Statistics *227*
- 9.1 Minimum and maximum *228*
- 9.2 Selection in expected linear time *230*
- 9.3 Selection in worst-case linear time *236*

III Data Structures

Introduction *249*

10 Elementary Data Structures *252*
- 10.1 Simple array-based data structures: arrays, matrices, stacks, queues *252*
- 10.2 Linked lists *258*
- 10.3 Representing rooted trees *265*

11 Hash Tables *272*
11.1 Direct-address tables *273*
11.2 Hash tables *275*
11.3 Hash functions *282*
11.4 Open addressing *293*
11.5 Practical considerations *301*

12 Binary Search Trees *312*
12.1 What is a binary search tree? *312*
12.2 Querying a binary search tree *316*
12.3 Insertion and deletion *321*

13 Red-Black Trees *331*
13.1 Properties of red-black trees *331*
13.2 Rotations *335*
13.3 Insertion *338*
13.4 Deletion *346*

IV Advanced Design and Analysis Techniques

Introduction *361*

14 Dynamic Programming *362*
14.1 Rod cutting *363*
14.2 Matrix-chain multiplication *373*
14.3 Elements of dynamic programming *382*
14.4 Longest common subsequence *393*
14.5 Optimal binary search trees *400*

15 Greedy Algorithms *417*
15.1 An activity-selection problem *418*
15.2 Elements of the greedy strategy *426*
15.3 Huffman codes *431*
15.4 Offline caching *440*

16 Amortized Analysis *448*
16.1 Aggregate analysis *449*
16.2 The accounting method *453*
16.3 The potential method *456*
16.4 Dynamic tables *460*

Table of Contents

Table of Contents

V *Advanced Data Structures*

Introduction *477*

17 Augmenting Data Structures ***480***
- 17.1 Dynamic order statistics *480*
- 17.2 How to augment a data structure *486*
- 17.3 Interval trees *489*

18 B-Trees ***497***
- 18.1 Definition of B-trees *501*
- 18.2 Basic operations on B-trees *504*
- 18.3 Deleting a key from a B-tree *513*

19 Data Structures for Disjoint Sets ***520***
- 19.1 Disjoint-set operations *520*
- 19.2 Linked-list representation of disjoint sets *523*
- 19.3 Disjoint-set forests *527*
- ★ 19.4 Analysis of union by rank with path compression *531*

VI *Graph Algorithms*

Introduction *547*

20 Elementary Graph Algorithms ***549***
- 20.1 Representations of graphs *549*
- 20.2 Breadth-first search *554*
- 20.3 Depth-first search *563*
- 20.4 Topological sort *573*
- 20.5 Strongly connected components *576*

21 Minimum Spanning Trees ***585***
- 21.1 Growing a minimum spanning tree *586*
- 21.2 The algorithms of Kruskal and Prim *591*

22 Single-Source Shortest Paths ***604***
- 22.1 The Bellman-Ford algorithm *612*
- 22.2 Single-source shortest paths in directed acyclic graphs *616*
- 22.3 Dijkstra's algorithm *620*
- 22.4 Difference constraints and shortest paths *626*
- 22.5 Proofs of shortest-paths properties *633*

23 All-Pairs Shortest Paths *646*
23.1 Shortest paths and matrix multiplication *648*
23.2 The Floyd-Warshall algorithm *655*
23.3 Johnson's algorithm for sparse graphs *662*

24 Maximum Flow *670*
24.1 Flow networks *671*
24.2 The Ford-Fulkerson method *676*
24.3 Maximum bipartite matching *693*

25 Matchings in Bipartite Graphs *704*
25.1 Maximum bipartite matching (revisited) *705*
25.2 The stable-marriage problem *716*
25.3 The Hungarian algorithm for the assignment problem *723*

VII Selected Topics

Introduction *745*

26 Parallel Algorithms *748*
26.1 The basics of fork-join parallelism *750*
26.2 Parallel matrix multiplication *770*
26.3 Parallel merge sort *775*

27 Online Algorithms *791*
27.1 Waiting for an elevator *792*
27.2 Maintaining a search list *795*
27.3 Online caching *802*

28 Matrix Operations *819*
28.1 Solving systems of linear equations *819*
28.2 Inverting matrices *833*
28.3 Symmetric positive-definite matrices and least-squares approximation *838*

29 Linear Programming *850*
29.1 Linear programming formulations and algorithms *853*
29.2 Formulating problems as linear programs *860*
29.3 Duality *866*

30 Polynomials and the FFT *877*
30.1 Representing polynomials *879*
30.2 The DFT and FFT *885*
30.3 FFT circuits *894*

Table of Contents

31 Number-Theoretic Algorithms *903*
31.1 Elementary number-theoretic notions *904*
31.2 Greatest common divisor *911*
31.3 Modular arithmetic *916*
31.4 Solving modular linear equations *924*
31.5 The Chinese remainder theorem *928*
31.6 Powers of an element *932*
31.7 The RSA public-key cryptosystem *936*
★ 31.8 Primality testing *942*

32 String Matching *957*
32.1 The naive string-matching algorithm *960*
32.2 The Rabin-Karp algorithm *962*
32.3 String matching with finite automata *967*
★ 32.4 The Knuth-Morris-Pratt algorithm *975*
32.5 Suffix arrays *985*

33 Machine-Learning Algorithms *1003*
33.1 Clustering *1005*
33.2 Multiplicative-weights algorithms *1015*
33.3 Gradient descent *1022*

34 NP-Completeness *1042*
34.1 Polynomial time *1048*
34.2 Polynomial-time verification *1056*
34.3 NP-completeness and reducibility *1061*
34.4 NP-completeness proofs *1072*
34.5 NP-complete problems *1080*

35 Approximation Algorithms *1104*
35.1 The vertex-cover problem *1106*
35.2 The traveling-salesperson problem *1109*
35.3 The set-covering problem *1115*
35.4 Randomization and linear programming *1119*
35.5 The subset-sum problem *1124*

VIII Appendix: Mathematical Background

Introduction *1139*

A Summations *1140*
A.1 Summation formulas and properties *1140*
A.2 Bounding summations *1145*

Table of Contents

Analyzing Algorithms

§ **Run Time Analysis**

– Order of growth

– Worst case analysis

– Average case analysis

§ **Insertion Sort for Array A[*i*]**

– idea: insert A[*i*] into sorted subarrays: A[1 : *i*-1]

– repeat insertion until A[*i*] is fully sorted

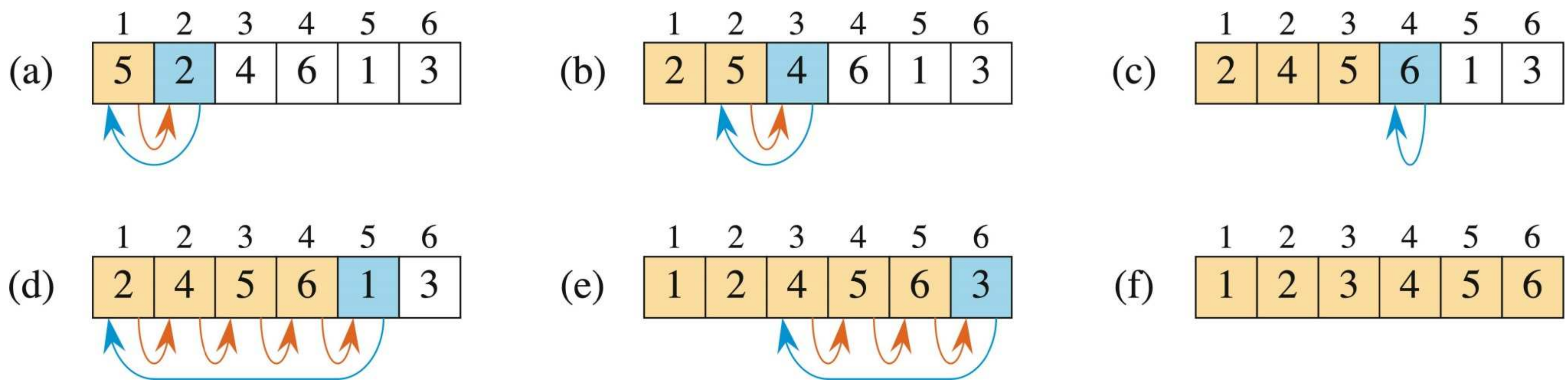
INSERTION-SORT(A, n)	*cost*	*times*
for $i = 2$ **to** n	c_1	n
$\quad key = A[i]$	c_2	$n - 1$
$\quad$ // Insert $A[i]$ into the sorted subarray $A[1 : i - 1]$.	0	$n - 1$
$\quad j = i - 1$	c_4	$n - 1$
$\quad$ **while** $j > 0$ and $A[j] > key$	c_5	$\sum_{i=2}^{n} t_i$
$\qquad A[j + 1] = A[j]$	c_6	$\sum_{i=2}^{n} (t_i - 1)$
$\qquad j = j - 1$	c_7	$\sum_{i=2}^{n} (t_i - 1)$
$\quad A[j + 1] = key$	c_8	$n - 1$

Insert A(*i*) properly

Complexity: $T(n) = \sum_{i=1}^{8} c_i$

Analyzing Algorithms (continued)

- **Insertion Sort for Array A[*i*]**
  - idea: insert A[*i*] into sorted subarrays: A[1 : *i*-1]
  - repeat insertion until A[*i*] is fully sorted



Worst-Case Complexity:

$$T(n) = \sum_{i=1}^{8} c_i = O(n^2)$$

INSERTION-SORT(A, n)	*cost*	*times*
for $i = 2$ **to** n	c_1	n
$key = A[i]$	c_2	$n-1$
// Insert $A[i]$ into the sorted subarray $A[1:i-1]$.	0	$n-1$
$j = i - 1$	c_4	$n-1$
while $j > 0$ and $A[j] > key$	c_5	$\sum_{i=2}^{n} t_i$
$A[j+1] = A[j]$	c_6	$\sum_{i=2}^{n} (t_i - 1)$
$j = j - 1$	c_7	$\sum_{i=2}^{n} (t_i - 1)$
$A[j+1] = key$	c_8	$n-1$

Designing Algorithms

- **Divide-and-Conquer Approaches with Recursive Nature**
  - Divide the problem
  - Conquer subproblems recursively
  - Combine solutions to subproblems

- **Example: Merge Sort**
  - two sorted subarrays: A[*p* .. *q*] & A[*q*+1 .. *r*]
  - merge the two sorted subarrays
  - merging takes Θ(*n*) time

conquer separately

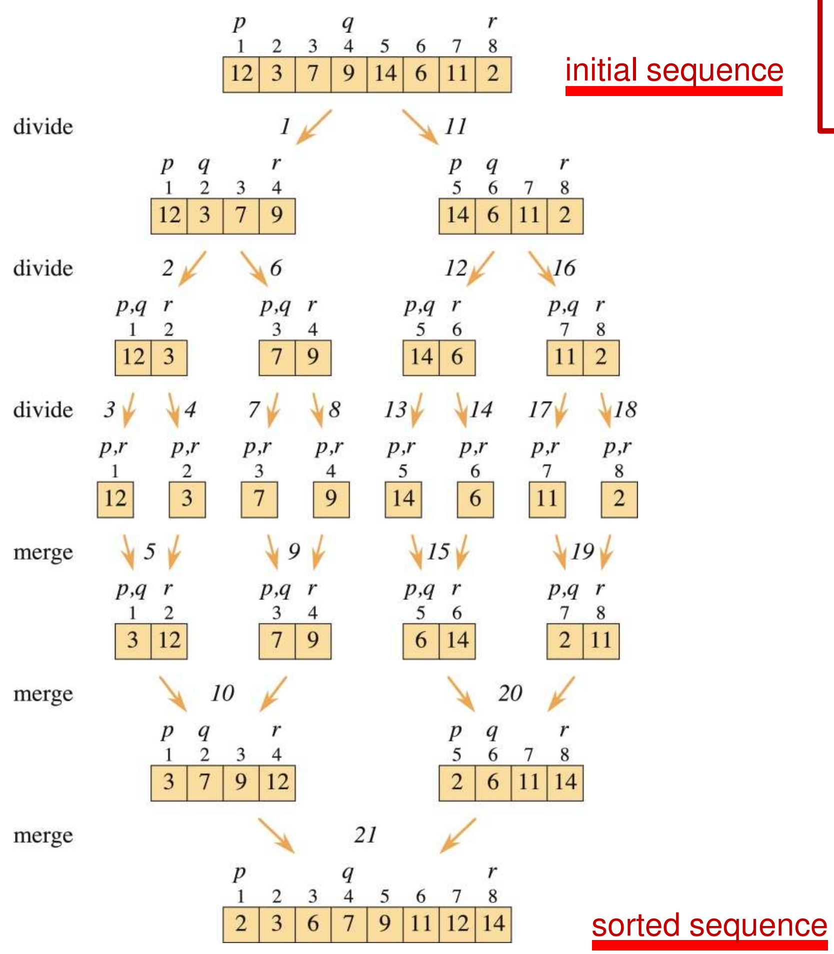
```
MERGE-SORT(A, p, r)
if p ≥ r                      // zero or one element?
    return
q = ⌊(p + r)/2⌋               // midpoint of A[p : r]
MERGE-SORT(A, p, q)           // recursively sort A[p : q]
MERGE-SORT(A, q + 1, r)       // recursively sort A[q + 1 : r]
// Merge A[p : q] and A[q + 1 : r] into A[p : r].
MERGE(A, p, q, r)
```

```
MERGE(A, p, q, r)
n_L = q − p + 1          // length of A[p : q]
n_R = r − q              // length of A[q + 1 : r]
let L[0 : n_L − 1] and R[0 : n_R − 1] be new arrays
for i = 0 to n_L − 1     // copy A[p : q] into L[0 : n_L − 1]
    L[i] = A[p + i]
for j = 0 to n_R − 1     // copy A[q + 1 : r] into R[0 : n_R − 1]
    R[j] = A[q + j + 1]
i = 0                    // i indexes the smallest remaining element in L
j = 0                    // j indexes the smallest remaining element in R
k = p                    // k indexes the location in A to fill
// As long as each of the arrays L and R contains an unmerged element,
//     copy the smallest unmerged element back into A[p : r].
while i < n_L and j < n_R
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else A[k] = R[j]
        j = j + 1
    k = k + 1
// Having gone through one of L and R entirely, copy the
//     remainder of the other to the end of A[p : r].
while i < n_L
    A[k] = L[i]
    i = i + 1
    k = k + 1
while j < n_R
    A[k] = R[j]
    j = j + 1
    k = k + 1
```

Analyzing Algorithms

- **Analysis of Divide-and-Conquer Algorithms**
  - Merge Sort
  - Time complexity:

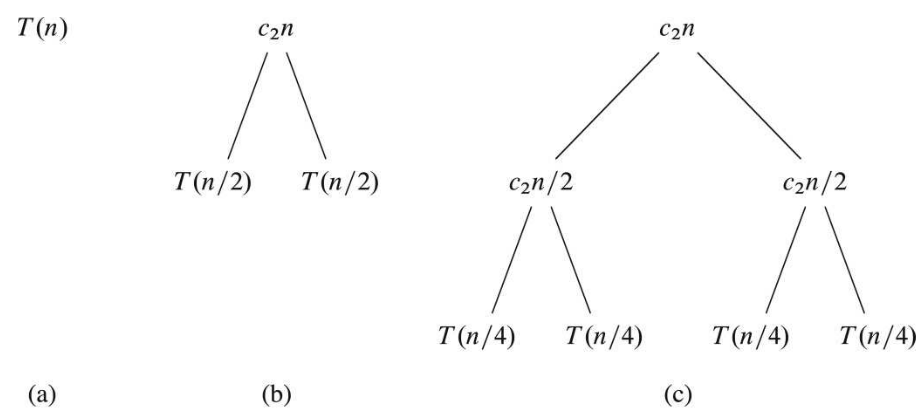
$$T(n) = \begin{cases} c_1 & \text{if } n = 1 \\ 2T(n/2) + c_2 \cdot (n) & \text{if } n > 1 \end{cases}$$




```
MERGE-SORT(A, p, r)
  if p < r                          // check for base case
      q = ⌊(p + r)/2⌋               // divide
      MERGE-SORT(A, p, q)           // conquer
      MERGE-SORT(A, q + 1, r)       // conquer
      MERGE(A, p, q, r)             // combine
```

Analyzing Algorithms (continued)

- **Evaluating: $2T(n/2) + c_2 \cdot (n)$**
  - Recursion tree, shown right, equal to $c_2 n \cdot \lg(n) + c_1 n = \Theta(n \cdot \lg n)$



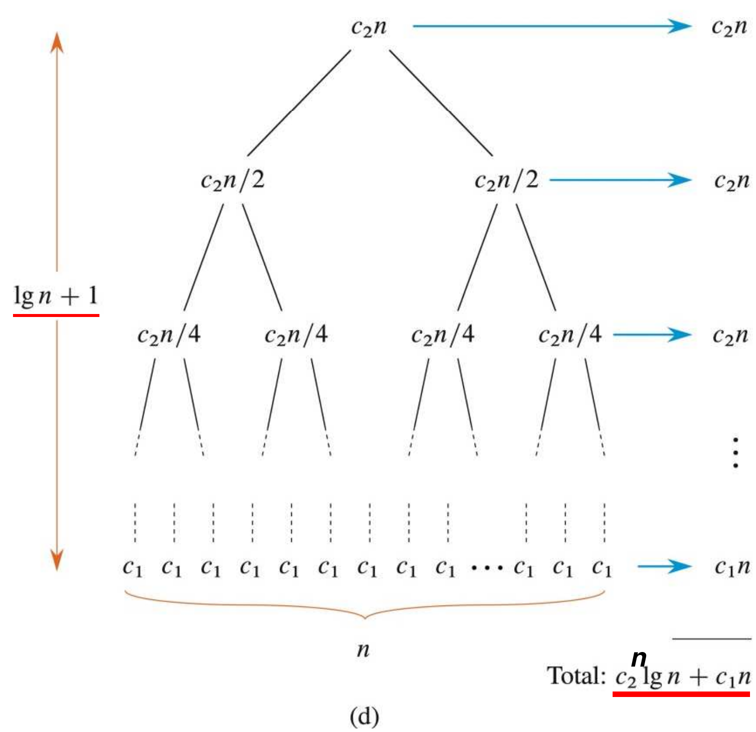
Another approach for solution:

$$T(n) = 2T(n/2) + c_2 \cdot (n)$$
$$= 2(2T(n/4) + c_2 \cdot (n/2)) + c_2 \cdot (n)$$
$$= 2^2 T(n/4) + 2 \cdot c_2 \cdot (n)$$
$$= 2^2(2T(n/8) + c_2 \cdot (n/4)) + 2 \cdot c_2 \cdot (n)$$
$$= 2^3 T(n/8) + 3 \cdot c_2 \cdot (n)$$

....

....

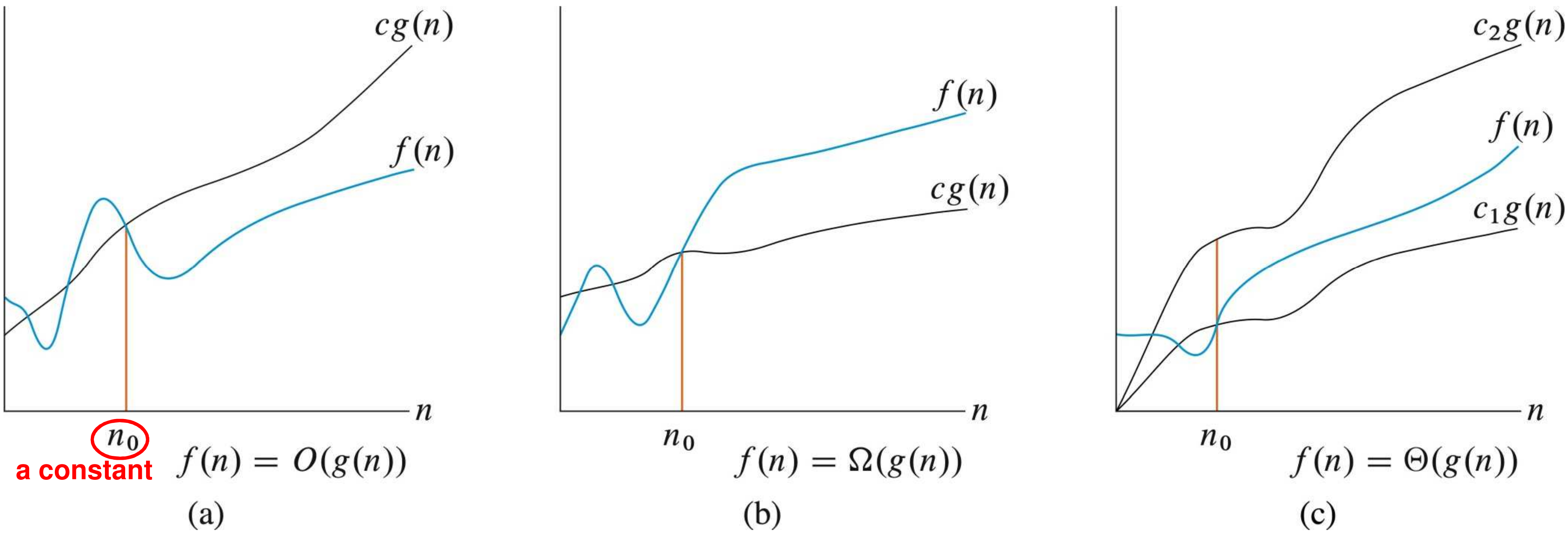
$$= n \cdot T(1) + \lg(n) \cdot c_2 \cdot (n)$$
$$= n \cdot c_1 + c_2 \cdot (n) \lg(n)$$



Growth of Functions

§ Asymptotic Notations of Running Times

- *O*-notation: upper-bounding a function to within a constant factor
- Ω-notation: lower-bounding a function to within a constant factor
- Θ-notation: bounding a function to within constant factors



g(*n*) is an asymptotical upper bound for *f*(*n*)
(may or may not be tight)

g(*n*) is an asymptotical lower bound for *f*(*n*)
(may or may not be tight)

g(*n*) is an asymptotically tight bound for *f*(*n*)

There exist positive constants c_1, c_2, and n_0 s.t. $0 \le c_1 \cdot g(n) \le f(n) \le c_2 \cdot g(n)$ for all $n \ge n_0$

Growth of Functions (continued)

- *o*-notation: not asymptotically-tight upper-bound

o-notation

$o(g(n)) = \{f(n) :$ for any constant $c > 0$, there exists a constant $n_0 > 0$ such that $0 \le f(n) < cg(n)$ for all $n \ge n_0\}$.

Another view, probably easier to use: $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0.$

$n^{1.9999} = o(n^2)$
$n^2 / \lg n = o(n^2)$
$n^2 \ne o(n^2)$ (just like $2 \not< 2$)
$n^2/1000 \ne o(n^2)$

- ω-notation: not asymptotically-tight lower-bound

ω-notation

$\omega(g(n)) = \{f(n) :$ for any constant $c > 0$, there exists a constant $n_0 > 0$ such that $0 \le cg(n) < f(n)$ for all $n \ge n_0\}$

Another view, again, probably easier to use: $\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty.$

$n^{2.0001} = \omega(n^2)$
$n^2 \lg n = \omega(n^2)$
$n^2 \ne \omega(n^2)$

Solutions after Divide-and-Conquer

§ **Divide-and-Conquer**

– leads to recurrences in various forms

– there are 3 kinds of methods for solving recurrences

+ **substitution** methods

+ **recursion-tree** methods

+ **master methods** to find bounds for recurrences of $T(n) = a \cdot T(n/b) + f(n)$, with $a > 0$ and $b > 1$

§ **Example: Multiplying two square matrices sized $n \times n$**

– $C = A \cdot B$

– divide each $n \times n$ matrix into four $n/2 \times n/2$ submatrices for multiplying:

$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$ and $B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$ to get $C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$, with

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

$$= \begin{pmatrix} A_{11}.B_{11} + A_{12}.B_{21} & A_{11}.B_{12} + A_{12}.B_{22} \\ A_{21}.B_{11} + A_{22}.B_{21} & A_{21}.B_{12} + A_{22}.B_{22} \end{pmatrix}.$$

Hence, we have

$$C_{11} = A_{11}.B_{11} + A_{12}.B_{21}; \qquad C_{12} = A_{11}.B_{12} + A_{12}.B_{22}$$

$$C_{21} = A_{21}.B_{11} + A_{22}.B_{21}; \qquad C_{22} = A_{21}.B_{12} + A_{22}.B_{22}$$

Divide-and-Conquer (continued)

§ Multiplying two square matrices

– solution involves eight **MATRIX-MULTIPLY RECURSIVE** calls

```
MATRIX-MULTIPLY-RECURSIVE(A, B, C, n)
if n == 1
// Base case.
    c11 = c11 + a11 · b11
    return
// Divide.
partition A, B, and C into n/2 × n/2 submatrices
    A11, A12, A21, A22; B11, B12, B21, B22;
    and C11, C12, C21, C22; respectively
// Conquer.
MATRIX-MULTIPLY-RECURSIVE(A11, B11, C11, n/2)
MATRIX-MULTIPLY-RECURSIVE(A11, B12, C12, n/2)
MATRIX-MULTIPLY-RECURSIVE(A21, B11, C21, n/2)
MATRIX-MULTIPLY-RECURSIVE(A21, B12, C22, n/2)
MATRIX-MULTIPLY-RECURSIVE(A12, B21, C11, n/2)
MATRIX-MULTIPLY-RECURSIVE(A12, B22, C12, n/2)
MATRIX-MULTIPLY-RECURSIVE(A22, B21, C21, n/2)
MATRIX-MULTIPLY-RECURSIVE(A22, B22, C22, n/2)
```

// C is initialized with nil.

The time complexity of this procedure:

$$T(n) = D(n) + 8T(n/2) + \Theta(1)$$
$$= \Theta(n^3).$$

Divide-and-Conquer (continued)

§ **Substitution Methods**: two steps involved
– guess the solution form
– mathematic induction to validate the constants

Substitution method for proving an upper bound on the recurrence of
$T(n) = 2T(\lfloor n/2 \rfloor) + \Theta(\underline{n})$ being $\underline{T(n) \leq c \cdot n \cdot \lg n}$ for a constant $c > 0$.

This is due to composing the full solution.

This is obtained by guessing its solution to be $T(n) = O(n \cdot \lg n)$ and then substituting $T(\lfloor n/2 \rfloor) \leq c \cdot \lfloor n/2 \rfloor \cdot \lg(\lfloor n/2 \rfloor)$ into the recurrence:

$$\begin{aligned} T(n) &\leq 2(c \cdot \lfloor n/2 \rfloor \cdot \lg(\lfloor n/2 \rfloor)) + \Theta(n) \\ &\leq c \cdot n \cdot \lg(\lfloor n/2 \rfloor) + \Theta(n) \\ &\leq c \cdot n \cdot \lg(n) - c \cdot n \cdot \lg(2) + \Theta(n) \\ &\leq c \cdot n \cdot \lg n, \quad \text{for } c \geq 1 \end{aligned}$$

Similar substitution method for proving an upper bound on recurrence
$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(\underline{1})$,
which equals $O(n)$.

This is due to composing the full solution.

Divide-and-Conquer (continued)

§ Substitution Methods

– changing variables and/or function renaming

Substitution method for proving: $T(n) = 2T(\lfloor\sqrt{n}\rfloor) + \lg n$

Rename $m = \lg n$ (and ignore rounding) to get $T(n)$ (after parameter renaming):

$$T(n) = T(2^m) = 2T(2^{m/2}) + m$$

Further renaming $T(2^m)$ as $S(m)$, we have (function renaming)

$S(m) = 2S(m/2) + m$, which has the solution of

$S(m) = O(m \cdot \lg m)$ ← via the prior result

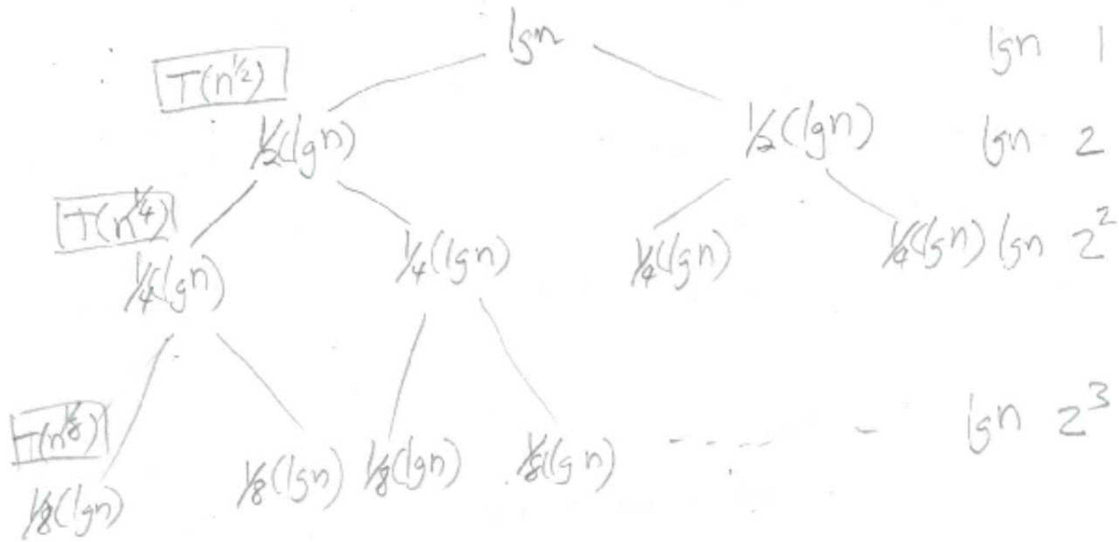
We thus have $T(n) = T(2^m) = S(m) = O(m \cdot \lg m)$

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

Note: this problem can also be solved by the recursion-tree method, described next.

Recursion tree approach for solving

$T(n) = 2T(\sqrt{n}) + \lg n$



total leaf count $= 2^{\lg\lg} = \underline{\lg n}$

Find out:

$n^{1/2^k}$ = a constant, say, $\underline{2}$

$1/2^k (\lg n) = 1$,

$2^k = \lg n \Rightarrow \underline{k = \lg\lg n}$, so the tree height $= 1 + \lg\lg n$

So, the solution to $T(n)$

$\leq c(\lg n * \lg\lg n)$

You try to solve $T(n) = T(n/3) + T(2n/3) + cn$

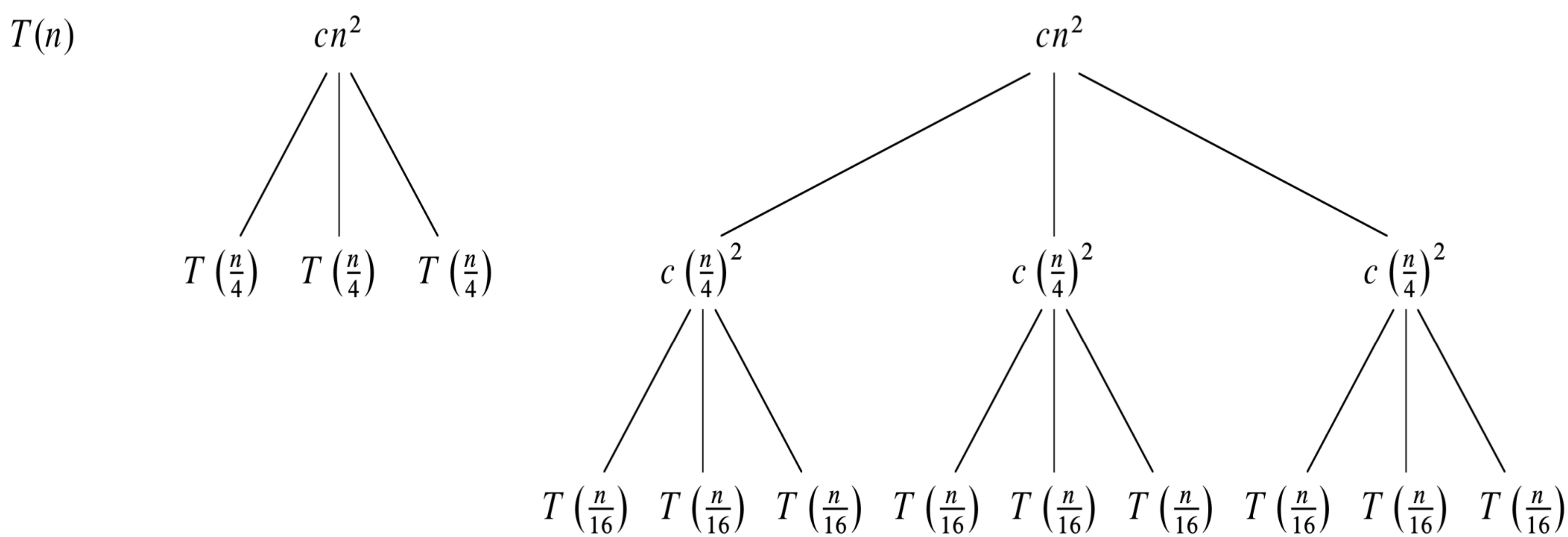
by the substitution method

Divide-and-Conquer (continued)

§ Recursion-Tree Methods

– best for generating good complexity bounds in general
– two examples given below

For recurrence: $T(n) = 3T(n/4) + \Theta(n^2)$

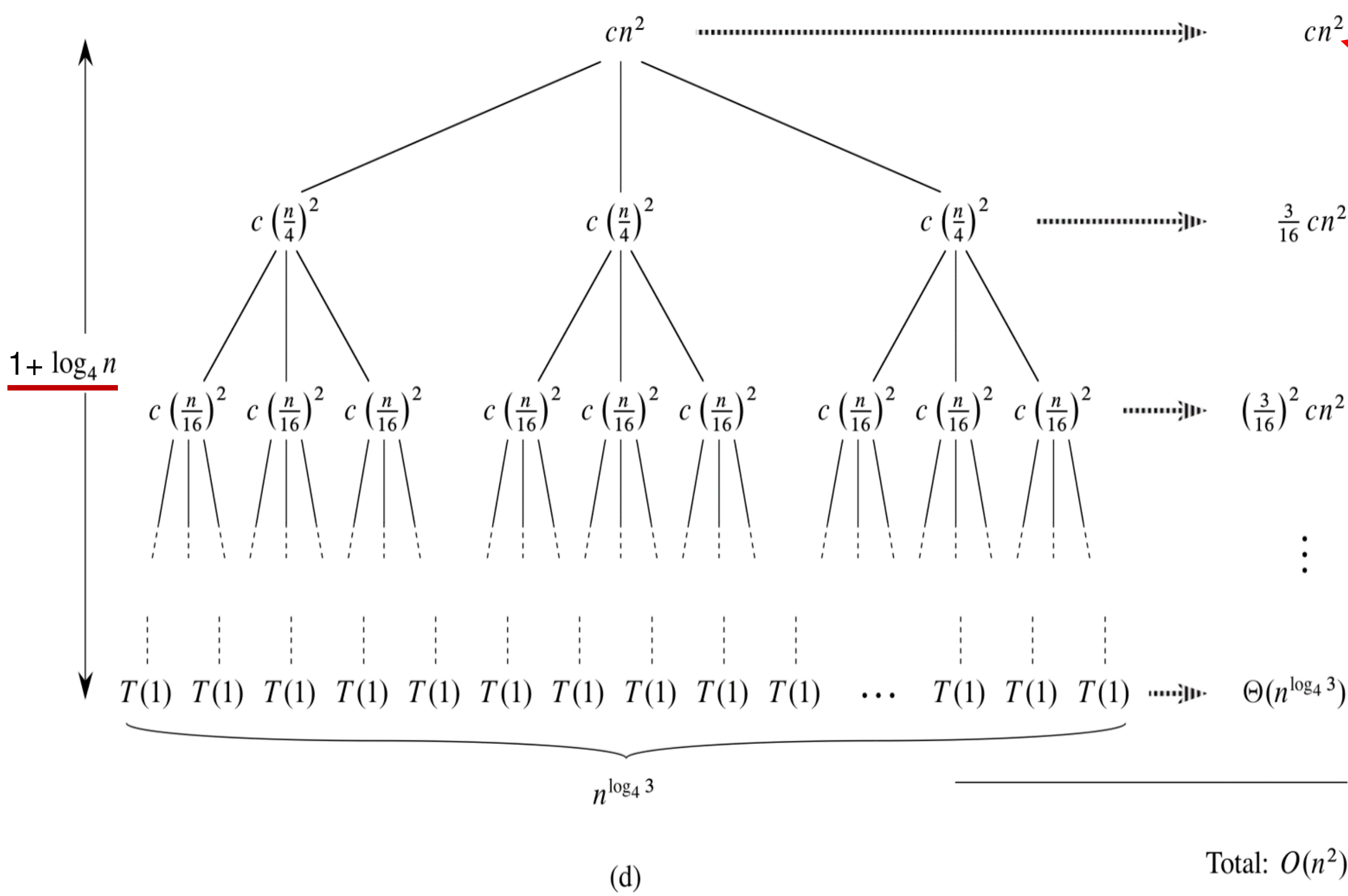


(a) (b) (c)

Divide-and-Conquer (continued)

§ **Recursion-Tree Methods**

For recurrence: $T(n) = 3T(n/4) + \Theta(n^2)$



$T(n)$ satisfies $\Omega(n^2)$.

3 elements

3^2 elements

$\boxed{3^{\log_4 n} \text{ elements}}$

$$T(n) = \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3})$$

$$< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) = \frac{1}{1-\left(\frac{3}{16}\right)} cn^2 + \Theta(n^{\log_4 3}) = O(n^2).$$

Use the property of: $v^{a \bullet b} = (v^a)^b = (v^b)^a$.

Let $3^{\log_4 n} = x$.

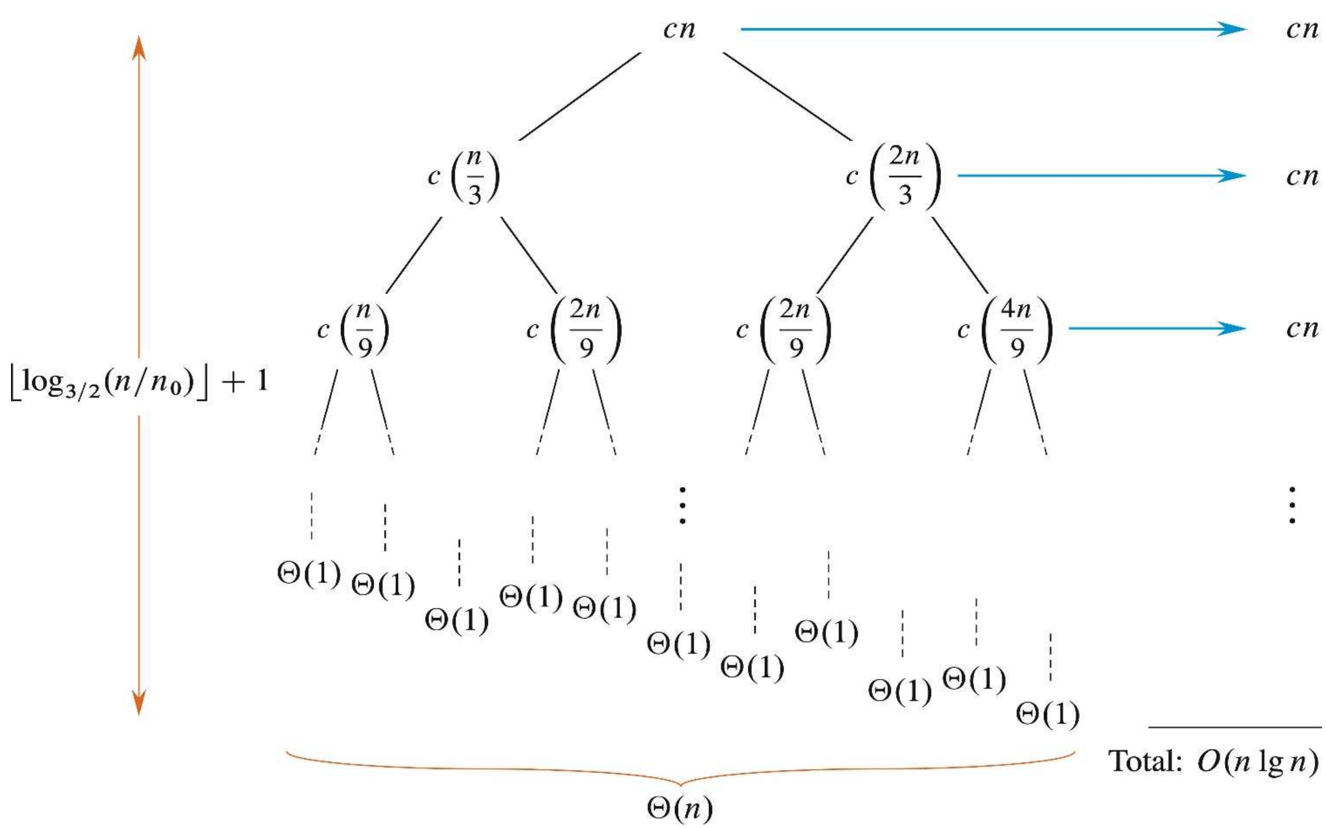
After taking $\log_4$ on both sides, we have: $(\log_4 n) \cdot (\log_4 3) = \log_4 x$.

We then take a power of 4 on both sides to yield: $4^{\log_4 n \bullet \log_4 3} = x$, which becomes:

$[4^{\log_4 n}]^{\log_4 3} = [n]^{\log_4 3} = x$. Hence, $3^{\log_4 n} = n^{\log_4 3}$.

Divide-and-Conquer (continued)

Another recurrence: $T(n) = T(n/3) + T(2n/3) + \Theta(n)$



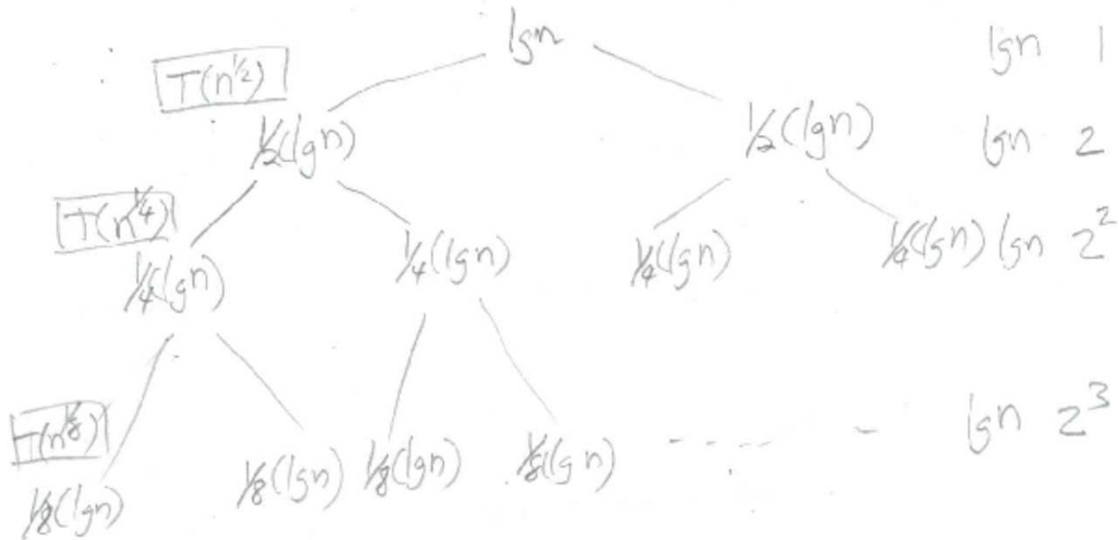
Longest path: $cn \to c(\frac{2}{3})n \to c(\frac{2}{3})^2 n \to c(\frac{2}{3})^3 n \to \cdots \to 1$, we have: $k = \log_{3/2} n$, as $(\frac{2}{3})^k n = 1$

Shortest path: $cn \to c(\frac{1}{3})n \to c(\frac{1}{3})^2 n \to c(\frac{1}{3})^3 n \to \cdots \to 1$ to get $k = \log_3 n$, ~ $(\log_{3/2} n)/2.7$

Similarly, one may show $T(n)$ upper bounded by $O(n \cdot \lg n)$ via substitution method.

Recursion tree approach for solving

$$T(n) = 2T(\sqrt{n}) + \lg n$$



total leaf count $= 2^{\lg\lg} = \lg n$

find out:

$n^{1/2^k}$ = a constant, say, 2

$\frac{1}{2^k}(\lg n) = 1$,

$2^k = \lg n \Rightarrow k = \lg\lg n$, so the tree height $= 1 + \lg\lg n$

So, the solution to $T(n)$

$\leq c(\lg n * \lg\lg n)$

same as that obtained previously by he substitution method

Master Method for Solving Recurrences

Recurrences of $T(n) = a \cdot T(n/b) + f(n)$ with constants $a > 0$ and $b > 1$
and $f(n)$ nonnegative function that covers work on
dividing the problem and on combining subproblems' results

Theorem 4.1

$T(n) = a \cdot T(n/b) + f(n)$ has following asymptotical bounds:

1. for $f(n) = O(n^{\log_b a - \epsilon})$ with constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$
2. for $f(n) = \Theta(n^{\log_b a} \cdot \lg^k n)$ with constant $k \geq 0$, then $T(n) = \Theta(n^{\log_b a} \cdot \lg^{k+1} n)$
3. for $f(n) = \Omega(n^{\log_b a + \epsilon})$ with constant $\epsilon > 0$ and $a \cdot f(n/b) \leq c \cdot f(n)$, then $T(n) = \Theta(f(n))$

In the recursion-tree, 2[nd] level sums to **no more than** 1[st] level & $f(n)$ is *polynomially larger*

Note: bound is the **larger** of the two: $f(n)$ and $n^{\log_b a}$

In Case 1, $n^{\log_b a}$ is *polynomially larger* than $f(n)$

In Case 2 with $k = 0$ most commonly, $n^{\log_b a}$ and $f(n)$ are of the same size

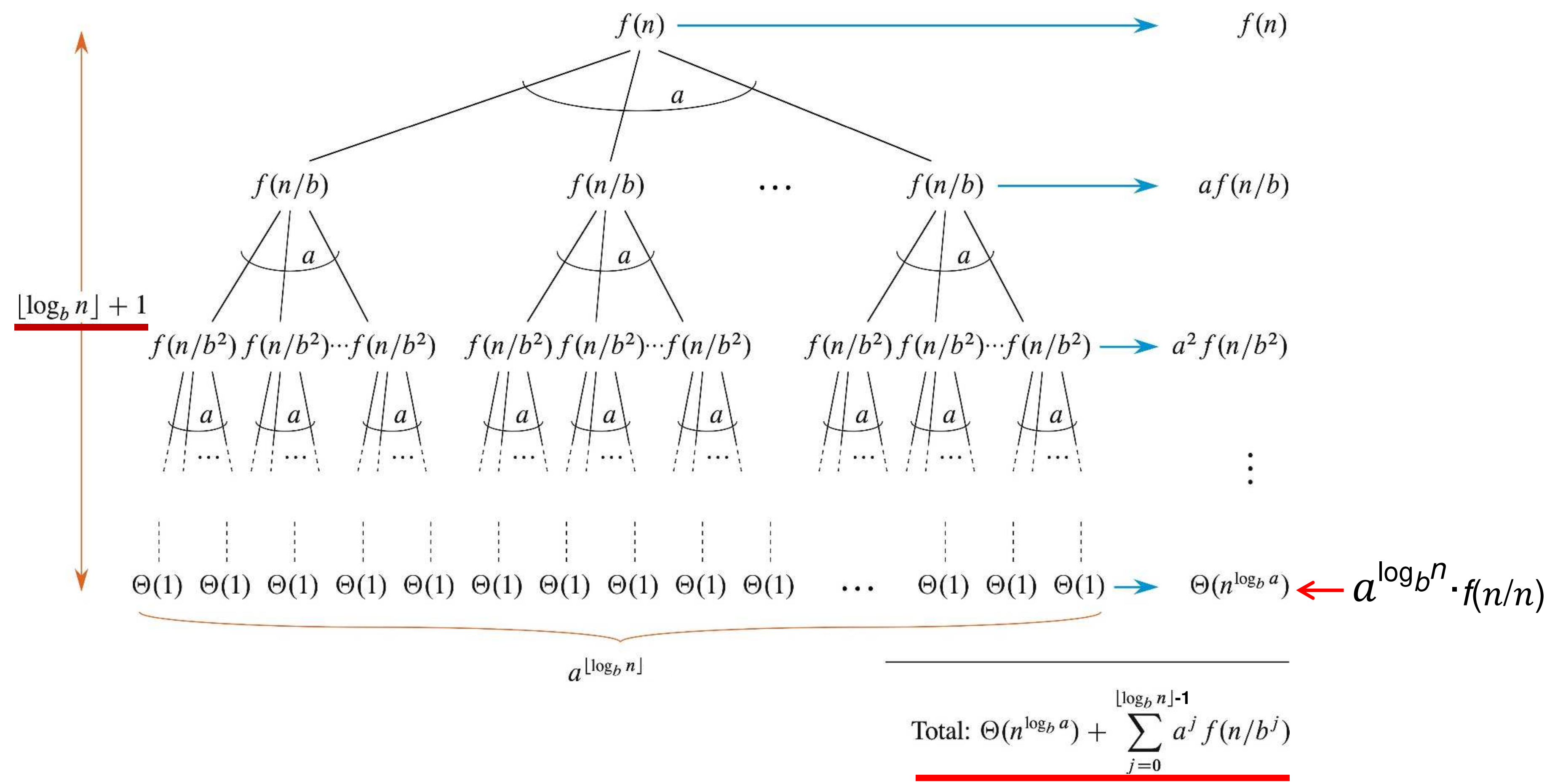
In Case 3, $f(n)$ is *polynomially larger* than $n^{\log_b a}$

Master Method (continued)

Let $T(n) = a \cdot T(n/b) + f(n)$ with constant $a > 0$, $b > 1$, and $n \geq 1$.   We have:

Lemma 4.2

$T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\lfloor \log_b n \rfloor - 1} a^j \cdot f(n/b^j)$



Master Method (continued)

Example Recurrences $T(n) = a \cdot T(n/b) + f(n)$ solved by the master method:

1. for $f(n) = O(n^{\log_b a - \epsilon})$ with constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$
2. for $f(n) = \Theta(n^{\log_b a} \cdot \lg^k n)$, then $T(n) = \Theta(n^{\log_b a} \cdot \lg^{k+1} n)$, with **$k$ = 0 most commonly**
3. for $f(n) = \Omega(n^{\log_b a + \epsilon})$ with constant $\epsilon > 0$ and $a \cdot f(n/b) \le c \cdot f(n)$, then $T(n) = \Theta(f(n))$

$T(n) = 9T(n/3) + n$

Here, $a = 9$, $b = 3$, and $f(n) = n$

From $n^{\log_3 9} = n^2$, we have $f(n) = n = O(n^{\log_3 9 - 1})$ to get $T(n) = \Theta(n^2)$

$T(n) = T(2n/3) + 1$

Here, $a = 1$, $b = 3/2$, and $f(n) = 1$

From $n^{\log_{3/2} 1} = n^0$, we have $f(n) = 1 = O(n^{\log_{3/2} 1})$ to get $T(n) = \Theta(\lg n)$

$T(n) = 3T(n/4) + n \lg n$

Here, $a = 3$, $b = 4$, and $f(n) = n \lg n$

From $n^{\log_4 3} = O(n^{0.793})$, we have $f(n) = n \lg n = \Omega(n^{\log_4 3 + \epsilon})$ to get $T(n) = \Theta(n \lg n)$

Master Method (continued)

Example Recurrences $T(n) = a \cdot T(n/b) + f(n)$:

1. for $f(n) = O(n^{\log_b a - \epsilon})$ with constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$
2. for $f(n) = \Theta(n^{\log_b a} \cdot \lg^k n)$, then $T(n) = \Theta(n^{\log_b a} \cdot \lg^{k+1} n)$, with $k = 0$ most commonly
3. for $f(n) = \Omega(n^{\log_b a + \epsilon})$ with constant $\epsilon > 0$ and $a \cdot f(n/b) \leq c \cdot f(n)$, then $T(n) = \Theta(f(n))$

f(n)* is *polynomially larger

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

Here, $a = 2$, $b = 2$, and $f(n) = n \lg n$

From $n^{\log_2 2} = n$, we have $f(n) = n \lg n > n^{\log_2 2}$ but ***not polynomially*** **>** $n^{\log_2 2}$

(use substitution or recursion-tree to solve this)

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

Here, $a = 7$, $b = 2$, and $f(n) = \Theta(n^2)$

From $n^{\log_2 7} = n^{2.8^+}$, we have $f(n) = O(n^{\log_2 7 - \epsilon})$ to get $T(n) = \Theta(n^{\log_2 7})$

Master Method (continued)

Lemma 4.3

Given $g(n) = \sum_{j=0}^{\lfloor \log_b n \rfloor} a^j \cdot f(n/b^j)$ for $a > 0$ and n an exact power of b (> 1), we have:

1. for $f(n) = O(n^{\log_b a - \epsilon})$ with constant $\epsilon > 0$, then $g(n) = \Theta(n^{\log_b a})$
2. for $f(n) = \Theta(n^{\log_b a} \cdot \lg^k n)$, then $g(n) = \Theta(n^{\log_b a} \cdot \lg^{k+1} n)$, with $k = 0$ most commonly
3. if $a \cdot f(n/b) \le c \cdot f(n)$ for constant $c < 1$ and for sufficiently large n, $g(n) = \Theta(f(n))$