Lecture 4

Divide-and-Conquer

Slides are based on the textbook and its notes

Overview

- Recall the divide-and-conquer approach, which we used for merge sort.
- **Divide** the problem into a number of subproblems that are smaller instances of the same problem.
- **Conquer** the subproblems by solving them recursively.
  - ***Base case***: If the subproblems are small enough, just solve them by brute force.
- **Combine** the subproblem solutions to give a solution to the original problem.

- We look at two algorithms for multiplying square matrices, based on divide-and-conquer.

Recurrences

- A ***recurrence*** is an equation that describes a function in terms of its value on other, typically smaller arguments.

- Recurrences go hand in hand with the divide-and-conquer method because they give us a natural way to characterize the running times of recursive algorithms mathematically.

- There may be zero, one, or many functions that satisfy the statement of the recurrence. The recurrence is ***well defined*** if there is at least one function satisfies it, and ***ill defined*** otherwise.

Algorithmic recurrences

- The general form of a recurrence is an equation or inequality that describes a function over the integers or reals using the function itself. It contains two or more cases, depending on the argument. If a case involves the recursive invocation of the function on different (usually smaller) inputs, it is a ***recursive case***. If a case does not involve a recursive invocation, it is a ***base case***.

- A recurrence $T(n)$ is ***algorithmic*** if, for every sufficiently large ***threshold*** constant $n_0 > 0$, the following two properties hold:
  - For all $n < n_0$, we have $T(n) = \Theta(1)$.
  - For all $n \geq n_0$, every path of recursion terminates in a defined base case within a finite number of recursive invocations.

Conventions for recurrences

- Will often state recurrences without base cases. When analyzing algorithms, assume that if no base case is given, the recurrence is algorithmic. Allows us to pick any sufficiently large threshold constant n_0 without changing the asymptotic behavior of the solution.
- Floors and ceilings in divide-and-conquer recurrences do not change the asymptotic solution => often state algorithmic recurrences without floors and ceilings. Doing so generally simplifies the statement of the recurrences, as well as any math that we do with them.
- Some recurrences are inequalities rather than equations.
  - *Example*: $T(n) \le 2T(n/2) + \Theta(n)$ gives only an upper bound on $T(n)$, so state the solution using O-notation rather than Θ-notation.

Examples of divide-and-conquer recurrences

- $n \times n$ matrix multiplication by breaking into 8 subproblems of size $n/2 \times n/2$: $T(n) = 8T(n/2) + \Theta(1)$. Solution: $T(n) = \Theta(n^3)$.
- Strassen's algorithm for $n \times n$ matrix multiplication by breaking into 7 subproblems of size $n/2 \times n/2$: $T(n) = 7T(n/2) + \Theta(n^2)$.

  Solution: $T(n) = \Theta(n^{\lg 7}) = \Theta(n^{2.81})$.
- An algorithm that breaks a problem of size n into one subproblem of size $n/3$ and another of size $2n/3$, taking $\Theta(n)$ time to divide and combine:

  $T(n) = T(n/3) + T(2n/3) + \Theta(n)$. Solution: $T(n) = \Theta(n \lg n)$.
- An algorithm that breaks a problem of size n into one subproblem of size $n/5$ and another of size $7n/10$, taking $\Theta(n)$ time to divide and combine:

  $T(n) = T(n/5) + T(7n/10) + \Theta(n)$. Solution: $T(n) = \Theta(n)$.
- Subproblems do not always have to be a constant fraction of the original problem size. *Example*: recursive linear search creates one subproblem and it has one element less than the original problem. Time to divide and combine is $\Theta(1)$, giving $T(n) = T(n - 1) + \Theta(1)$. Solution: $T(n) = \Theta(n)$.

Multiplying square matrices

Input: Three $n \times n$ (square) matrices, $A = (a_{ij})$, $B = (b_{ij})$, and $C = (c_{ij})$.

Result: The matrix product $A \cdot B$ is added into C, so that

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

for $i, j = 1, 2, \ldots, n$.

If only the product $A \cdot B$ is needed, then zero out all entries of C beforehand.

Straightforward method

MATRIX-MULTIPLY(A, B, C, n)

```
for i = 1 to n                          // compute entries in each of n rows
    for j = 1 to n                      // compute n entries in row i
        for k = 1 to n
            c_ij = c_ij + a_ik · b_kj   // add in another term
```

Time: $\Theta(n^3)$ because of triply nested loops.

Simple divide-and-conquer algorithm

- For simplicity, assume that C is initialized to 0, so computing $C = A \cdot B$.
- If $n > 1$, partition each of A, B, C, into four $n/2 \times n/2$ matrices:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}.$$

- Rewrite $C = A \cdot B$ as

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

giving the four equations

$$\begin{aligned} C_{11} &= A_{11} \cdot B_{11} + A_{12} \cdot B_{21}, \\ C_{12} &= A_{11} \cdot B_{12} + A_{12} \cdot B_{22}, \\ C_{21} &= A_{21} \cdot B_{11} + A_{22} \cdot B_{21}, \\ C_{22} &= A_{21} \cdot B_{12} + A_{22} \cdot B_{22}. \end{aligned}$$

- Each of these equations multiples two $n/2 \times n/2$ matrices and then adds their $n/2 \times n/2$ products. Assume that n is an exact power of 2, so that submatrix dimensions are always integer.

Simple divide-and-conquer algorithm

- Use these equations to get a divide-and-conquer algorithm:

MATRIX-MULTIPLY-RECURSIVE(A, B, C, n)

1 **if** $n == 1$
2 // Base case.
3 $c_{11} = c_{11} + a_{11} \cdot b_{11}$
4 **return**
5 // Divide.
6 partition A, B, and C into $n/2 \times n/2$ submatrices
$A_{11}, A_{12}, A_{21}, A_{22}$; $B_{11}, B_{12}, B_{21}, B_{22}$;
and $C_{11}, C_{12}, C_{21}, C_{22}$; respectively
7 // Conquer.
8 MATRIX-MULTIPLY-RECURSIVE$(A_{11}, B_{11}, C_{11}, n/2)$
9 MATRIX-MULTIPLY-RECURSIVE$(A_{11}, B_{12}, C_{12}, n/2)$
10 MATRIX-MULTIPLY-RECURSIVE$(A_{21}, B_{11}, C_{21}, n/2)$
11 MATRIX-MULTIPLY-RECURSIVE$(A_{21}, B_{12}, C_{22}, n/2)$
12 MATRIX-MULTIPLY-RECURSIVE$(A_{12}, B_{21}, C_{11}, n/2)$
13 MATRIX-MULTIPLY-RECURSIVE$(A_{12}, B_{22}, C_{12}, n/2)$
14 MATRIX-MULTIPLY-RECURSIVE$(A_{22}, B_{21}, C_{21}, n/2)$
15 MATRIX-MULTIPLY-RECURSIVE$(A_{22}, B_{22}, C_{22}, n/2)$

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ 8T(n/2) + \Theta(1) & n > 1 \end{cases} = \begin{cases} c_0 & n = 1 \\ 8T(n/2) + c_1 & n > 1 \end{cases}$$

when $n > 1$

$$T(n) = c_1 + \sum_{i=1}^{\log_2 n - 1} 8^i \cdot \frac{c_1}{2} + 8^{\log_2 n} \cdot c_0$$

$$= c_1 + \frac{c_1}{2} \sum_{i=1}^{\log_2 n - 1} 8^i + c_0 n^3$$

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

Simple divide-and-conquer algorithm: Analysis

- Let $T(n)$ be the time to multiply two $n \times n$ matrices.
  - **Base case**: $n = 1$. Perform one scalar multiplication: $\Theta(1)$.
  - **Recursive case**: $n > 1$.
    - Dividing takes $\Theta(1)$ time, using index calculations. (Otherwise, $\Theta(n^2)$ time, since $3n^2$ elements are copied.)
    - Conquering makes 8 recursive calls, each multiplying $n/2 \times n/2$ matrices => $8T(n/2)$.
    - No combine step, because C is updated in place.
  - Recurrence (omitting the base case) is

    $T(n) = 8T(n/2) + \Theta(1)$. Can use master method to show that it has solution $T(n) = \Theta(n^3)$. Asymptotically, no better than the obvious method.

Strassen's algorithm

- ***Idea***: Make the recursion tree less bushy. Perform only 7 recursive multiplications of $n/2 \times n/2$ matrices, rather than 8. Will cost several additions/subtractions of $n/2 \times n/2$ matrices.
- Since a subtraction is a "negative addition", just refer to all additions and subtractions as additions.
- ***Example of reducing multiplications***: Given x and y, compute $x^2 - y^2$. Obvious way uses 2 multiplications and one subtraction. But observe:

$$\begin{aligned} x^2 - y^2 &= x^2 - xy + xy - y^2 \\ &= x(x - y) + y(x - y) \\ &= (x + y)(x - y) \end{aligned}$$

so at the expense of one extra addition, can get by with only 1 multiplication. Not a big deal if x, y are scalars, but can make a difference if they are matrices.

Strassen's algorithm

- The algorithm:

1. Same base case as before, when $n = 1$.

2. When $n > 1$, then as in the recursive method, partition each of the matrices into four $n/2 \times n/2$ submatrices. Time: $\Theta(1)$, using index calculations.

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}.$$

3. Create 10 matrices $S_1, S_2, \ldots, S_{10}$. Each is $n/2 \times n/2$ and is the sum or difference of two matrices created in previous step. Time: $\Theta(n^2)$ to create all 10 matrices.

4. Create and zero the entries of 7 matrices $P_1, P_2, \ldots, P_7$, each $n/2 \times n/2$. Time: $\Theta(n^2)$.

5. Using the submatrices of A and B and the matrices $S_1, S_2, \ldots, S_{10}$, recursively compute $P_1, P_2, \ldots, P_7$. Time: $7T(n/2)$.

6. Update the four $n/2 \times n/2$ submatrices $C_{11}, C_{12}, C_{21}, C_{22}$ of C by adding and$_{12}$ subtracting various combinations of the P_i. Time: $\Theta(n^2)$.

Strassen's algorithm

- In ***Step 3***, we create the following 10 matrices:

$$
\begin{aligned}
S_1 &= B_{12} - B_{22} , \\
S_2 &= A_{11} + A_{12} , \\
S_3 &= A_{21} + A_{22} , \\
S_4 &= B_{21} - B_{11} , \\
S_5 &= A_{11} + A_{22} , \\
S_6 &= B_{11} + B_{22} , \\
S_7 &= A_{12} - A_{22} , \\
S_8 &= B_{21} + B_{22} , \\
S_9 &= A_{11} - A_{21} , \\
S_{10} &= B_{11} + B_{12} .
\end{aligned}
$$

Add or subtract $n/2 \times n/2$ matrices 10 times $\Rightarrow$ time is $\Theta(n^2)$.

Strassen's algorithm

- In ***Step 5***, we recursively multiply $n/2 \times n/2$ matrices 7 times to compute the following $n/2 \times n/2$ matrices, each of which is the sum or difference of products of A and B submatrices:

$$
\begin{aligned}
P_1 &= A_{11} \cdot S_1 = A_{11} \cdot B_{12} - A_{11} \cdot B_{22} , \\
P_2 &= S_2 \cdot B_{22} = A_{11} \cdot B_{22} + A_{12} \cdot B_{22} , \\
P_3 &= S_3 \cdot B_{11} = A_{21} \cdot B_{11} + A_{22} \cdot B_{11} , \\
P_4 &= A_{22} \cdot S_4 = A_{22} \cdot B_{21} - A_{22} \cdot B_{11} , \\
P_5 &= S_5 \cdot S_6 = A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22} , \\
P_6 &= S_7 \cdot S_8 = A_{12} \cdot B_{21} + A_{12} \cdot B_{22} - A_{22} \cdot B_{21} - A_{22} \cdot B_{22} , \\
P_7 &= S_9 \cdot S_{10} = A_{11} \cdot B_{11} + A_{11} \cdot B_{12} - A_{21} \cdot B_{11} - A_{21} \cdot B_{12} .
\end{aligned}
$$

- The only multiplications needed are in the middle column; right-hand column just shows the products in terms of the original submatrices of A and B.

Strassen's algorithm

- In ***Step 6***, we add and subtract the P_i to construct submatrices of C:

$$
\begin{aligned}
C_{11} &= P_5 + P_4 - P_2 + P_6 \,, \\
C_{12} &= P_1 + P_2 \,, \\
C_{21} &= P_3 + P_4 \,, \\
C_{22} &= P_5 + P_1 - P_3 - P_7 \,.
\end{aligned}
$$

Strassen's algorithm

- To see how these computations work, expand each right-hand side, replacing each P_i with the submatrices of A and B that form it, and cancel terms. As $C_{11} = P_5 + P_4 - P_2 + P_6$, we see that C_{11} equals:

$$
\begin{array}{l}
A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22} \\
\qquad - A_{22} \cdot B_{11} \qquad\qquad + A_{22} \cdot B_{21} \\
\quad - A_{11} \cdot B_{22} \qquad\qquad\qquad - A_{12} \cdot B_{22} \\
\qquad\qquad - A_{22} \cdot B_{22} - A_{22} \cdot B_{21} + A_{12} \cdot B_{22} + A_{12} \cdot B_{21} \\
\hline
A_{11} \cdot B_{11} \qquad\qquad\qquad\qquad + A_{12} \cdot B_{21}
\end{array}
$$

which corresponds to $C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}$.

Strassen's algorithm

- Similarly, $C_{12} = P_1 + P_2$ and so C_{12} equals:

$$\begin{array}{l} A_{11} \cdot B_{12} - A_{11} \cdot B_{22} \\ \qquad + A_{11} \cdot B_{22} + A_{12} \cdot B_{22} \\ \hline A_{11} \cdot B_{12} \qquad\qquad + A_{12} \cdot B_{22} \end{array}$$

which corresponds to $C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22}$.

Strassen's algorithm

- $C_{21} = P_3 + P_4$ and makes C_{21} equal:

$$\begin{array}{l} A_{21} \cdot B_{11} + A_{22} \cdot B_{11} \\ \qquad - A_{22} \cdot B_{11} + A_{22} \cdot B_{21} \\ \hline A_{21} \cdot B_{11} \qquad\qquad + A_{22} \cdot B_{21} \end{array}$$

which corresponds to $C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21}$.

Strassen's algorithm

- Finally, $C_{22} = P_5 + P_1 - P_3 - P_7$ and makes C_{22} equal:

$$
\begin{array}{llll}
A_{11}\cdot B_{11} + A_{11}\cdot B_{22} + A_{22}\cdot B_{11} + A_{22}\cdot B_{22} & & & \\
\quad - A_{11}\cdot B_{22} & & + A_{11}\cdot B_{12} & \\
\quad - A_{22}\cdot B_{11} & & - A_{21}\cdot B_{11} & \\
- A_{11}\cdot B_{11} & & - A_{11}\cdot B_{12} + A_{21}\cdot B_{11} + A_{21}\cdot B_{12} & \\
\hline
 & A_{22}\cdot B_{22} & & + A_{21}\cdot B_{12}
\end{array}
$$

which corresponds to $C_{22} = A_{21}\cdot B_{12} + A_{22}\cdot B_{22}$.

Strassen's algorithm: Analysis and notes

- Recurrence will be $T(n) = 7T(n/2) + \Theta(n^2)$. By the master method, solution is $T(n) = \Theta(n^{\lg 7})$. Since $\lg 7 < 2.81$, the running time is $O(n^{2.81})$, beating the $\Theta(n^3)$-time methods.
- Strassen's algorithm was the first to beat $\Theta(n^3)$ time, but it is not the asymptotically fastest known. A method by Coppersmith and Winograd runs in $O(n^{2.376})$ time.
- Practical issues against Strassen's algorithm:
  - Higher constant factor than the obvious $\Theta(n^3)$-time method.
  - Not good for sparse matrices.
  - Not numerically stable: larger error accumulate than in the obvious method.
  - Submatrices consume space, especially if copying.

Methods for solving recurrences

- This lecture covers three methods for solving recurrences. Each gives asymptotic bounds.
  - **Substitution method**: Guess the solution, then use induction to prove that it is correct.
  - **Recursion-tree method**: Draw out a recursion tree, determine the costs at each level, and sum them up. Useful for coming up with a guess for the substitution method.
  - **Master method**: A cookbook method for recurrences of the form $T(n) = aT(n/b) + f(n)$, where $a > 0$ and $b > 1$ are constants, subject to certain conditions. Requires memorizing three cases, but applies to many divide-and-conquer algorithms.

Substitution method for solving recurrences

1. Guess the solution.
2. Use induction to find the constants and show that the solution works.

Usually use the substitution method to establish either an upper bound (O-bound) or a lower bound (Ω-bound).

- ***Example***:

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

1. Guess: $T(n) = n \lg n + n$.
2. Induction.

Substitution method for solving recurrences

2. Induction:

Base case is when $n = 1$, and we have $n \lg n + n = 1 \lg 1 + 1 = 1$

For the ***inductive step***, our ***inductive hypothesis*** is that $T(n/2) = (n/2) \lg (n/2) + n/2$. Then we have

$$
\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + n \\
&= 2\left(\frac{n}{2}\lg\frac{n}{2} + \frac{n}{2}\right) + n \qquad \text{(by inductive hypothesis)} \\
&= n\lg\frac{n}{2} + n + n \\
&= n(\lg n - \lg 2) + n + n \\
&= n\lg n - n + n + n \\
&= n\lg n + n\ .
\end{aligned}
$$

Substitution method for solving recurrences

- When the additive term uses asymptotic notation
  - Name the constant in the additive term.
  - Show the upper (O) and lower (Ω) bounds separately. Might need to use different constants for each.

- ***Example***

$T(n) = 2T(n/2) + \Theta(n)$. If we want to show an upper bound of $T(n) = 2T(n/2) + O(n)$, we write $T(n) \leq 2T(n/2) + cn$ for some positive constant c.

- We get to name the constant hidden in the asymptotic notation (c in this case), but we do not get to choose it, other than assume that it is enough to handle the base case of the recursion.

Substitution method for solving recurrences

1. ***Upper bound***:

   Guess: $T(n) \le dn \lg n$ for some positive constant d. This is the inductive hypothesis.

- We get to both name and choose the constant in the inductive hypothesis (d in this case). It is OK for the constant in the inductive hypothesis (d) to depend on the constant hidden in the asymptotic notation (c).

Substitution:

$$\begin{aligned} T(n) &\le 2T(n/2) + cn \\ &= 2\left(d\frac{n}{2}\lg\frac{n}{2}\right) + cn \\ &= dn\lg\frac{n}{2} + cn \\ &= dn\lg n - dn + cn \\ &\le dn\lg n \qquad \text{if } -dn + cn \le 0, \\ &\qquad\qquad\qquad\quad d \ge c \end{aligned}$$

Therefore, $T(n) = O(n \lg n)$.

Substitution method for solving recurrences

2. ***Lower bound***: Write $T(n) \geq 2T(n/2) + cn$ for some positive constant c.

   Guess: $T(n) \geq dn \lg n$ for some positive constant d.

   Substitution:

$$\begin{aligned} T(n) &\geq 2T(n/2) + cn \\ &= 2\left(d\frac{n}{2}\lg\frac{n}{2}\right) + cn \\ &= dn\lg\frac{n}{2} + cn \\ &= dn\lg n - dn + cn \\ &\geq dn\lg n \qquad \text{if } -dn + cn \geq 0, \\ &\qquad\qquad\qquad\quad d \leq c \end{aligned}$$

   Therefore, $T(n) = \Omega(n \lg n)$.

Therefore, $T(n) = \Theta(n \lg n)$.

$\geq dn\log_2 n + (c_1 - d)n$

$T(n) = 2T(\frac{n}{2}) + \Omega(n) \Leftrightarrow T(n) \geq 2T(\frac{n}{2}) + c_1 n$

$2T(\frac{n}{2}) + c_1 n \geq 2 \cdot d\frac{n}{2}\log_2\frac{n}{2} + c_1 n$

$= dn(\log_2 n - 1) + c_1 n$

$= dn\log_2 n + (c_1 - d)n$

$\geq dn\log_2 n \quad (\exists c_1 \geq d)$

$\Rightarrow T(n) = \Omega(n\log_2 n)$

$\Rightarrow T(n) = \Theta(n\log_2 n)$

Substitution method for solving recurrences

- **Subtracting a low-order term**
  - Might guess the right asymptotic bound, but the math does not go through in the proof. Resolve by subtracting a lower-order term.

let $T(n) = \begin{cases} c_0 & n = 0 \\ 2T(\frac{n}{2}) + c_1 & n \geq 1 \end{cases}$

$\Rightarrow T(n) = \sum_{i=0}^{\log_2 n - 1} 2^i c_1 + n c_0 = (c_0 + c_1) n - c_1 = O(n)$

- ***Example***

$T(n) = 2T(n/2) + \Theta(1)$. Guess that $T(n) = O(n)$, and try to show $T(n) \leq cn$ for $n \geq n_0$, where we choose c, n_0:

$(c_0 + c_1) n - c_1 \leq cn$ (for $n \geq n_0$)

$\Rightarrow c \geq c_0 + c_1 - \frac{c_1}{n}$

$T(n) \leq 2(c(n/2)) + \Theta(1)$

$= cn + \Theta(1)$

$\not\leq cn$ does not work

But this does not say that $T(n) \leq cn$ for *any* choice of c.

- Could try a larger guess, such as $T(n) = O(n^2)$, but not necessary. We are off only by $\Theta(1)$, a lower-order term.

Substitution method for solving recurrences

- Try subtracting a lower-order term in the guess:

  $T(n) \le cn - d$, where $d \ge 0$ is a constant:

  $$\begin{aligned} T(n) &\le 2(c(n/2) - d) + \Theta(1) \\ &= cn - 2d + \Theta(1) \\ &\le cn - d - (d - \Theta(1)) \\ &\le cn - d \end{aligned}$$

  as long as d is larger than the constant in $\Theta(1)$.

- ***Why subtract off a lower-order term, rather than add it?***

  Notice that it's subtracted twice. Adding a lower-order term twice would take us further away from the inductive hypothesis. Subtracting it twice gives us $T(n) \le cn - d - (d - \Theta(1))$, and it is easy to choose d to make that inequality hold.

- Once again, we get to name and choose the constant c in the inductive hypothesis. And we also get to name and choose the constant d that we subtract off.

Substitution method for solving recurrences

- **Be careful when using asymptotic notation**

A false proof for the recurrence $T(n) = 2T(\lfloor n/2 \rfloor) + \Theta(n)$, that $T(n) = O(n)$:

$$
\begin{aligned}
T(n) &\le 2 \cdot O(\lfloor n/2 \rfloor) + \Theta(n) \\
&= 2 \cdot O(n) + \Theta(n) \\
&= O(n) . \qquad \Longleftarrow \textit{wrong!}
\end{aligned}
$$

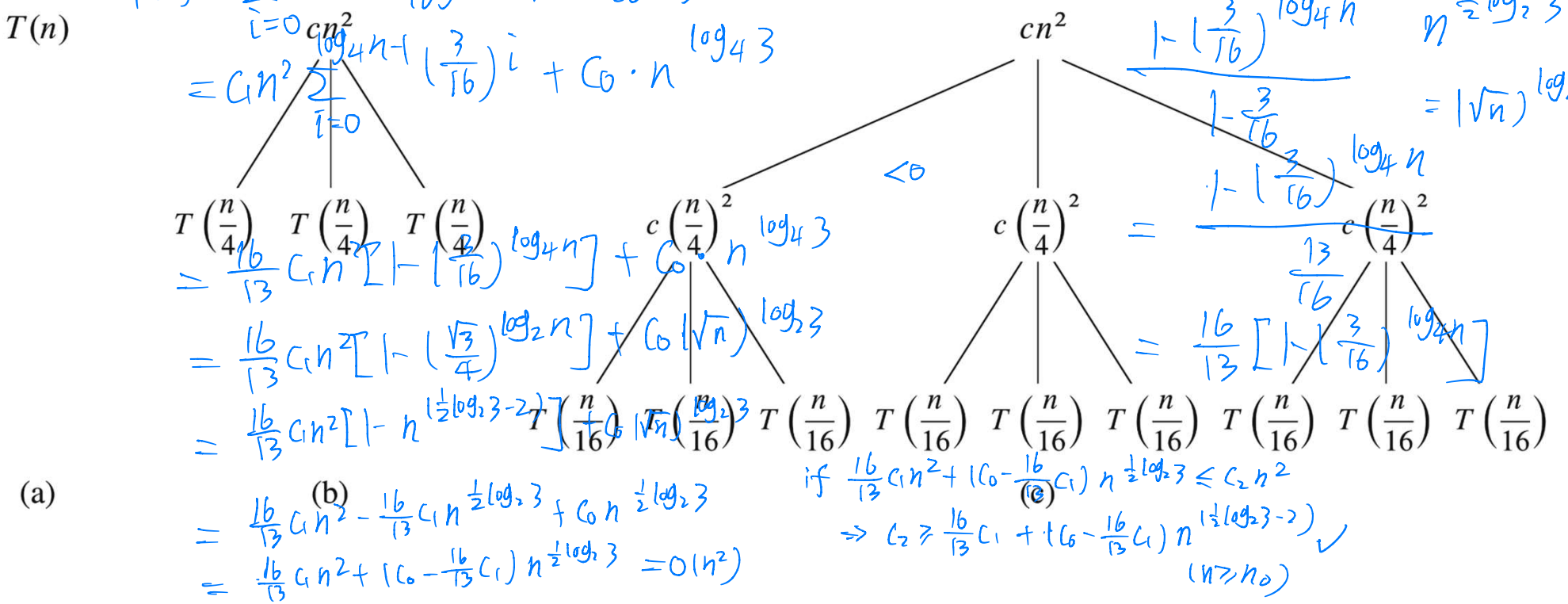
This "proof" changes the constant in the Θ-notation. Can see this by using an explicit constant. Assume $T(n) \le cn$ for all $n \ge n_0$:

$$
\begin{aligned}
T(n) &\le 2(c \lfloor n/2 \rfloor) + \Theta(n) \\
&\le cn + \Theta(n) ,
\end{aligned}
$$

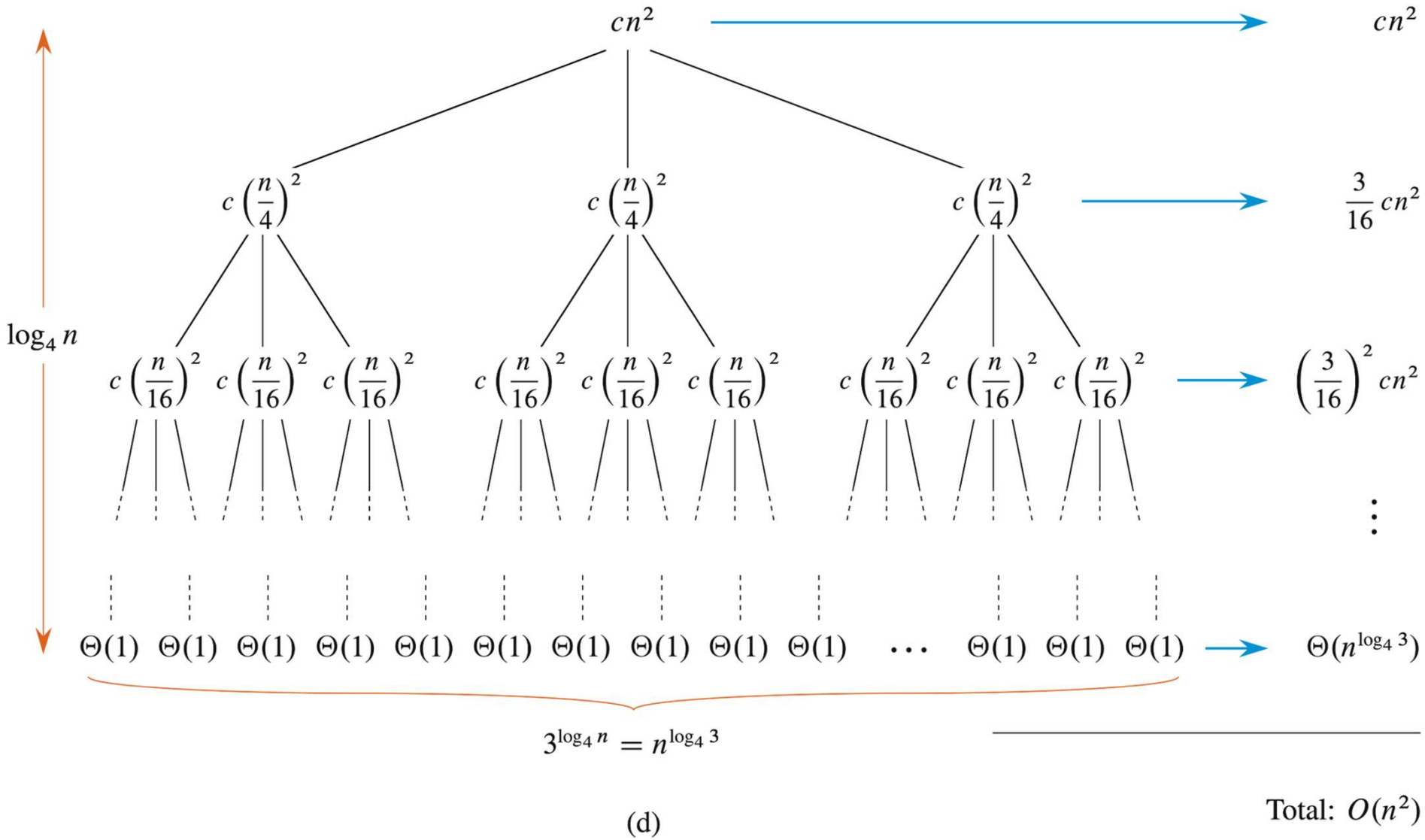
but $cn + \Theta(n) > cn$.

The recursion-tree method for solving recurrences

- Use to generate a guess for the solution of a recurrence. Then verify by substitution method.
- Let us see how a recursion tree can provide a good guess for an upper-bound solution to the recurrence $T(n) = 3T(n/4) + \Theta(n^2)$. The following figure shows a recurrence tree for $T(n) = 3T(n/4) + cn^2$, where the constant $c > 0$ is the upper-bound constant in the $\Theta(n^2)$ term.



The recursion-tree method for solving recurrences



The recursion-tree method for solving recurrences

- For simplicity, assume that n is a power of 4 and the base case is $T(n) = \Theta(1)$.The subproblem size for a node at depth i is $n/4^i$. As we descend the tree from the root, the subproblem size hits $n = 1$ when $n/4^i = 1$ or, equivalently, when $i = \log_4 n$. Thus, the tree has internal nodes at depths 0, 1, 2, …, $\log_4 n - 1$ and leaves at depth $\log_4 n$.
- Each level has three times as many nodes as the level above, and so the number of nodes at depth i is 3^i. Because subproblem sizes reduce by a factor of 4 for each level further from the root, each internal node at depth $i = 0, 1, 2, \ldots, \log_4 n - 1$, has a cost of $c(n/4^i)^2$. Multiplying, we see that the total cost over all nodes at a given depth i is $3^i c(n/4^i)^2 = (3/16)^i cn^2$.
- The bottom level, at depth $\log_4 n$, contains $3^{\log_4 n} = n^{\log_4 3}$ leaves (using equation (3.21)). Each leaf contributes $\Theta(1)$, leading to a total cost of $\Theta(n^{\log_4 3})$.

The recursion-tree method for solving recurrences

- Now we add up the cost over all levels to determine the cost for the entire tree:

$$\begin{aligned} T(n) &= cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \cdots + \left(\frac{3}{16}\right)^{\log_4 n - 1} cn^2 + \Theta(n^{\log_4 3}) \\ &= \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\ &= \frac{(3/16)^{\log_4 n} - 1}{(3/16) - 1} cn^2 + \Theta(n^{\log_4 3}) \qquad \left(\text{by } \sum_{k=0}^{n} x^k = \frac{x^{n+1} - 1}{x - 1}\right) . \end{aligned}$$

- We can take advantage of small amounts of sloppiness and use an infinite decreasing geometric series as an upper bound ($\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$ when $|x| < 1$).

$$\begin{aligned} T(n) &= \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\ &< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\ &= \frac{1}{1 - (3/16)} cn^2 + \Theta(n^{\log_4 3}) \\ &= \frac{16}{13} cn^2 + \Theta(n^{\log_4 3}) \\ &= O(n^2) . \end{aligned}$$

The recursion-tree method for solving recurrences

- ***Idea***: Coefficients of cn^2 form a decreasing geometric series. Bound it by an infinite series and get a bound of 16/13 on the coefficients.
- Use substitution method to verify $O(n^2)$ upper bound. Show that $T(n) \le dn^2$ for constant $d > 0$:

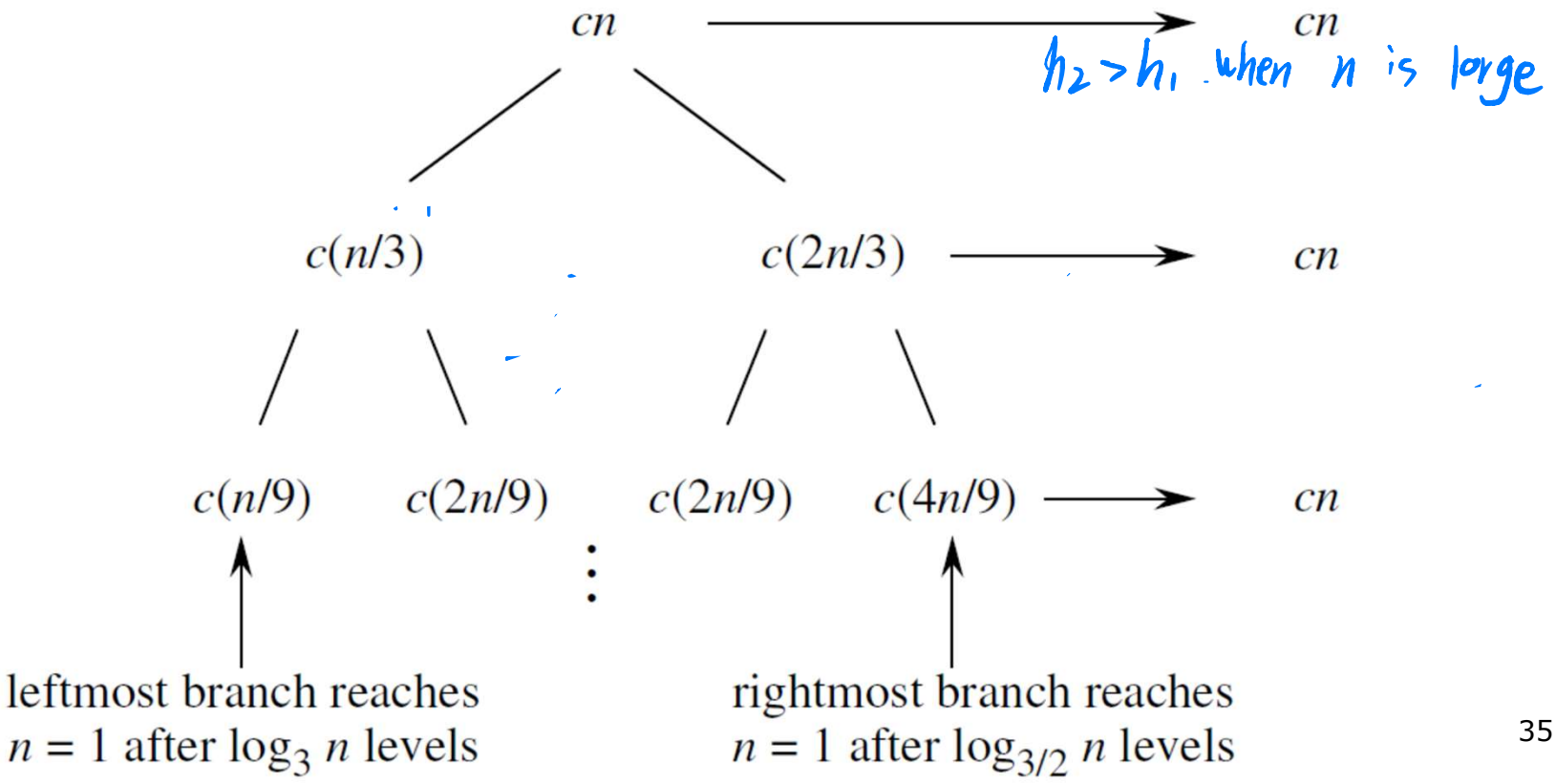
$$\begin{aligned} T(n) &\le 3T(n/4) + cn^2 \\ &\le 3d(n/4)^2 + cn^2 \\ &= \frac{3}{16} dn^2 + cn^2 \\ &\le dn^2 , \end{aligned}$$

by choosing $d \ge (16/13)c$. That gives an upper bound of $O(n^2)$.

- The lower bound of $\Omega(n^2)$ is obvious because the recurrence contains a $\Theta(n^2)$ term. Hence, $T(n) = \Theta(n^2)$.

The recursion-tree method for solving recurrences

- Let's find an asymptotic upper bound for another, more irregular recurrence $T(n) = T(n/3) + T(2n/3) + \Theta(n)$. The following figure shows a simpler recurrence tree for $T(n) = T(n/3) + T(2n/3) + cn$, where the constant $c > 0$ is the upper-bound constant in the $\Theta(n)$ term.



The recursion-tree method for solving recurrences

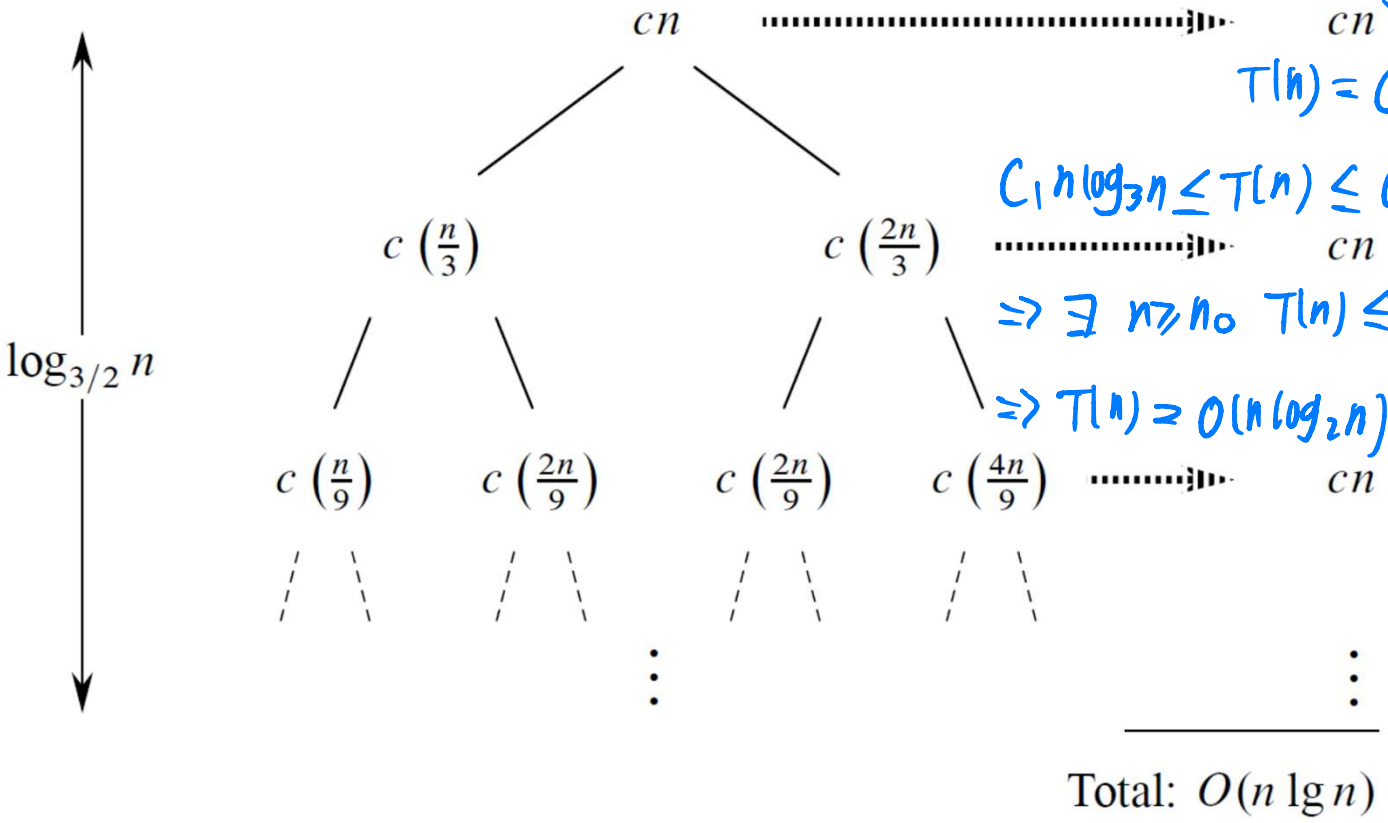
- The longest simple path from the root to a leaf is $n \to (2/3)n \to (2/3)^2 n \to \cdots \to 1$. Since $(2/3)^k n = 1$ when $k = \log_{3/2} n$, the height of the tree is $\log_{3/2} n$. Intuitively, we expect the solution to the recurrence to be at most the number of levels times the cost of each level, or $O(cn \log_{3/2} n) = O(n \lg n)$. The following figure shows only the top levels of the recursion tree.

need to find that $T(n) = \Omega(n \log_3 n)$ and $T(n) = O(n \log_{\frac{3}{2}} n)$

$C_1 n \log_3 n \le T(n) \le C_2 n \log_{\frac{3}{2}} n$

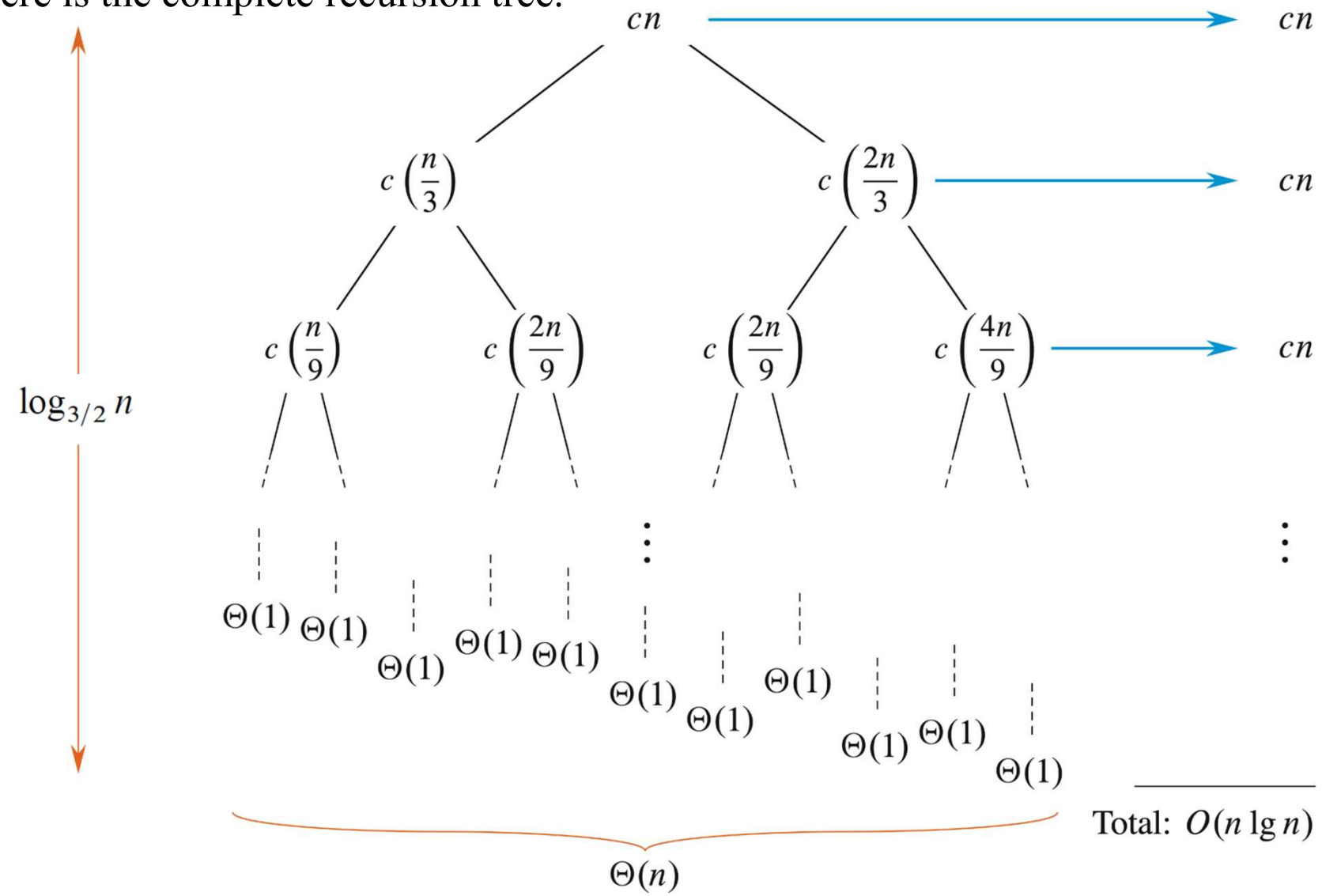
$\Rightarrow \exists\, n \ge n_0 \;\; T(n) \le C_3 n \log_2 n$

$\Rightarrow T(n) = O(n \log_2 n)$



The recursion-tree method for solving recurrences

- Here is the complete recursion tree.



The recursion-tree method for solving recurrences

- There are $\log_3 n$ full levels (going down the left side), and after $\log_{3/2} n$ levels, the problem size is down to 1 (going down the right side).
- Each level contributes $\leq cn$.
- Lower bound guess: $\geq dn \log_3 n = \Omega(n \lg n)$ for some positive constant d.
- Upper bound guess: $\leq dn \log_{3/2} n = O(n \lg n)$ for some positive constant d.
- Then *prove* by substitution.

1. ***Upper bound***: *Guess*: $T(n) \leq dn \lg n$.

Substitution:

$$
\begin{aligned}
T(n) &\leq T(n/3) + T(2n/3) + cn \\
&\leq d(n/3)\lg(n/3) + d(2n/3)\lg(2n/3) + cn \\
&= (d(n/3)\lg n - d(n/3)\lg 3) \\
&\qquad + (d(2n/3)\lg n - d(2n/3)\lg(3/2)) + cn \\
&= dn\lg n - d((n/3)\lg 3 + (2n/3)\lg(3/2)) + cn \\
&= dn\lg n - d((n/3)\lg 3 + (2n/3)\lg 3 - (2n/3)\lg 2) + cn \\
&= dn\lg n - dn(\lg 3 - 2/3) + cn \\
&\leq dn\lg n \qquad \text{if } -dn(\lg 3 - 2/3) + cn \leq 0 , \\
&\qquad\qquad\qquad\qquad\qquad\qquad d \geq \frac{c}{\lg 3 - 2/3} .
\end{aligned}
$$

Therefore, $T(n) = O(n \lg n)$.

The recursion-tree method for solving recurrences

2. ***Lower bound***:

Guess: $T(n) \geq d\,n \lg n$.

Substitution: Same as for the upper bound, but replacing $\leq$ by $\geq$. End up needing

$$0 < d \leq \frac{c}{\lg 3 - 2/3}\ .$$

Therefore, $T(n) = \Omega(n \lg n)$.

Since $T(n) = O(n \lg n)$ and $T(n) = \Omega(n \lg n)$, conclude that $T(n) = \Theta(n \lg n)$.

Master method

- Used for many divide-and-conquer ***master recurrences*** of the form

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

where $a \geq 1$, $b > 1$, and $f(n)$ is an asymptotically nonnegative function defined over all sufficiently large positive numbers.

- Master recurrences describe recursive algorithms that divide a problem of size n into a subproblems, each of size n/b. Each recursive subproblem takes time $T(n/b)$ (unless it is a base case). Call $f(n)$ the ***driving function***.

- In reality, subproblem sizes are integers, so that the real recurrence is more like

$$T(n) = a'T(\lfloor n/b \rfloor) + a''T(\lceil n/b \rceil) + f(n) ,$$

where $a', a'' \geq 0$ and $a' + a'' = a$. Ignoring floors and ceilings does not change the asymptotic solution to the recurrence.

Master method

- Based on the ***master theorem*** (*Theorem 4.1*):

Let a, b, $n_0 > 0$ be constants, $f(n)$ be a driving function defined and nonnegative on all sufficiently large reals. Define recurrence $T(n)$ on $n \in \mathbb{N}$ by

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

and where $aT(n/b)$ actually means

$a'T(\lfloor n/b \rfloor) + a''T(\lceil n/b \rceil)$ for some constants a', $a'' \geq 0$ satisfying $a = a' + a''$.

Master method

Then you can solve the recurrence by comparing $n^{\log_b a}$ vs. $f(n)$:

Case 1: $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$.
($f(n)$ is polynomially smaller than $n^{\log_b a}$.)
Solution: $T(n) = \Theta(n^{\log_b a})$.
(Intuitively: cost is dominated by leaves.)

Case 2: $f(n) = \Theta(n^{\log_b a} \lg^k n)$, where $k \geq 0$ is a constant.
($f(n)$ is within a polylog factor of $n^{\log_b a}$, but not smaller.)
Solution: $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.
(Intuitively: cost is $n^{\log_b a} \lg^k n$ at each level, and there are $\Theta(\lg n)$ levels.)
Simple case: $k = 0 \Rightarrow f(n) = \Theta(n^{\log_b a}) \Rightarrow T(n) = \Theta(n^{\log_b a} \lg n)$.

Case 3: $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$ and $f(n)$ satisfies the regularity condition $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n.
($f(n)$ is polynomially greater than $n^{\log_b a}$.)
Solution: $T(n) = \Theta(f(n))$.
(Intuitively: cost is dominated by root.)

Using the master method

To use the master method, you determine which case (if any) of the master theorem applies and write down the answer.

As a first example, consider the recurrence $T(n) = 9T(n/3) + n$. For this recurrence, we have $a = 9$ and $b = 3$, which implies that $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$. Since $f(n) = n = O(n^{2-\epsilon})$ for any constant $\epsilon \leq 1$, we can apply case 1 of the master theorem to conclude that the solution is $T(n) = \Theta(n^2)$.

Now consider the recurrence $T(n) = T(2n/3) + 1$, which has $a = 1$ and $b = 3/2$, which means that the watershed function is $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$. Case 2 applies since $f(n) = 1 = \Theta(n^{\log_b a} \lg^0 n) = \Theta(1)$. The solution to the recurrence is $T(n) = \Theta(\lg n)$.

Using the master method

For the recurrence $T(n) = 3T(n/4) + n \lg n$, we have $a = 3$ and $b = 4$, which means that $n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$. Since $f(n) = n \lg n = \Omega(n^{\log_4 3 + \epsilon})$, where ϵ can be as large as approximately 0.2, case 3 applies as long as the regularity condition holds for $f(n)$. It does, because for sufficiently large n, we have that $af(n/b) = 3(n/4)\lg(n/4) \leq (3/4)n \lg n = cf(n)$ for $c = 3/4$. By case 3, the solution to the recurrence is $T(n) = \Theta(n \lg n)$.

Next, let's look at the recurrence $T(n) = 2T(n/2) + n \lg n$, where we have $a = 2, b = 2$, and $n^{\log_b a} = n^{\log_2 2} = n$. Case 2 applies since $f(n) = n \lg n = \Theta(n^{\log_b a} \lg^1 n)$. We conclude that the solution is $T(n) = \Theta(n \lg^2 n)$.

Using the master method

Recurrence (2.3), $T(n) = 2T(n/2) + \Theta(n)$, on page 41, characterizes the running time of merge sort. Since $a = 2$ and $b = 2$, the watershed function is $n^{\log_b a} = n^{\log_2 2} = n$. Case 2 applies because $f(n) = \Theta(n)$, and the solution is $T(n) = \Theta(n \lg n)$.

Recurrence (4.9), $T(n) = 8T(n/2) + \Theta(1)$, on page 84, describes the running time of the simple recursive algorithm for matrix multiplication. We have $a = 8$ and $b = 2$, which means that the watershed function is $n^{\log_b a} = n^{\log_2 8} = n^3$. Since n^3 is polynomially larger than the driving function $f(n) = \Theta(1)$—indeed, we have $f(n) = O(n^{3-\epsilon})$ for any positive $\epsilon < 3$—case 1 applies. We conclude that $T(n) = \Theta(n^3)$.

Finally, recurrence (4.10), $T(n) = 7T(n/2) + \Theta(n^2)$, on page 87, arose from the analysis of Strassen's algorithm for matrix multiplication. For this recurrence, we have $a = 7$ and $b = 2$, and the watershed function is $n^{\log_b a} = n^{\lg 7}$. Observing that $\lg 7 = 2.807355\ldots$, we can let $\epsilon = 0.8$ and bound the driving function $f(n) = \Theta(n^2) = O(n^{\lg 7 - \epsilon})$. Case 1 applies with solution $T(n) = \Theta(n^{\lg 7})$.

Proof of the master theorem

Lemma 4.2

Let $a \geq 1$ and $b > 1$ be constants, and let $f(n)$ be a nonnegative function defined on exact powers of b. Define $T(n)$ on exact powers of b by the recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 , \\ aT(n/b) + f(n) & \text{if } n = b^i , \end{cases}$$

where i is a positive integer. Then

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

Proof of the master theorem

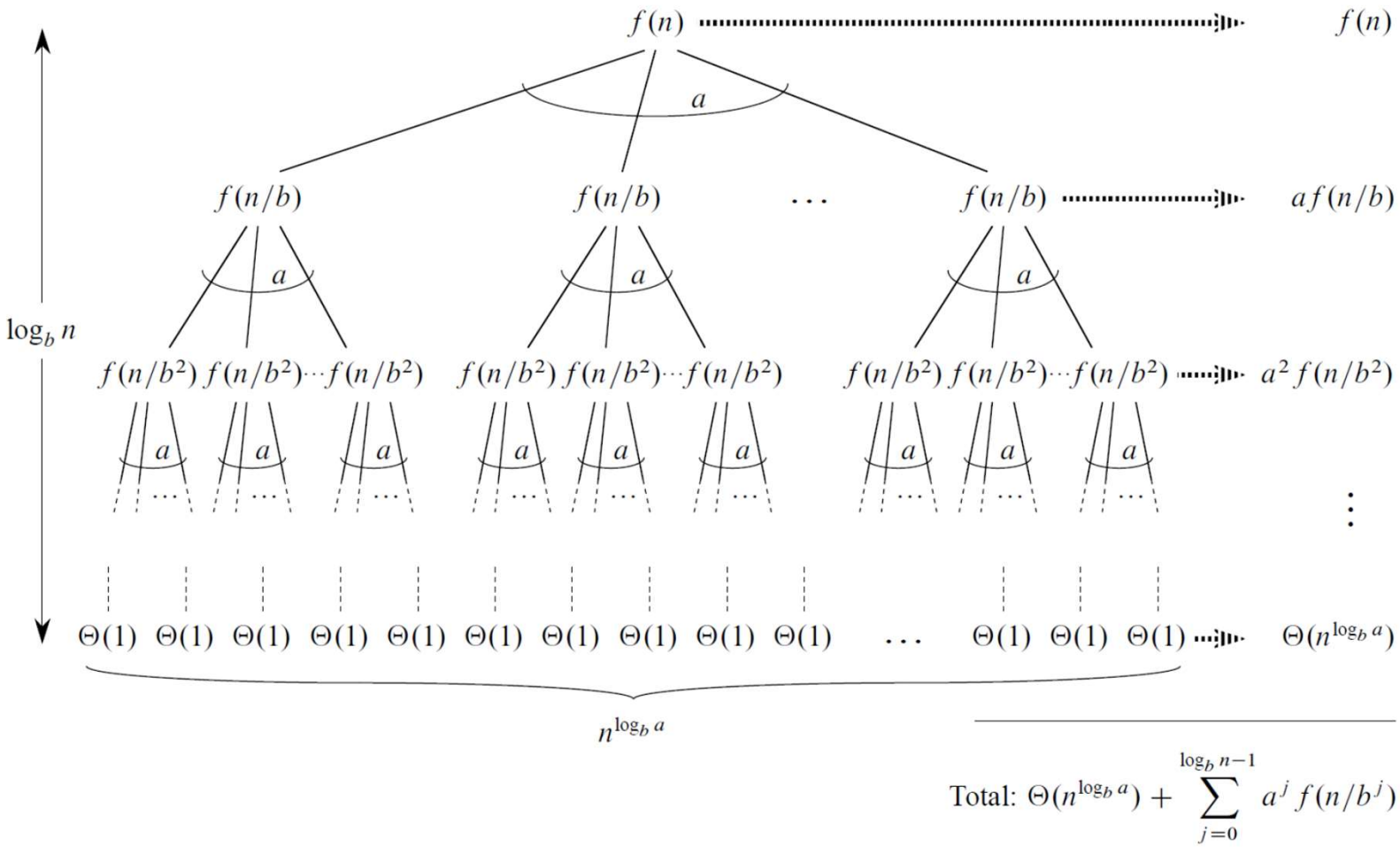


Figure 4.7 The recursion tree generated by $T(n) = aT(n/b) + f(n)$. The tree is a complete a-ary tree with $n^{\log_b a}$ leaves and height $\log_b n$. The cost of the nodes at each depth is shown at the right, and their sum is given in equation (4.21).

Summary

- Divide-and-conquer recurrences
- Strassen's algorithm
- Methods for solving recurrences
  - Substitution method
  - Recursion-tree method
  - Master method

Reading

- Sections 4.1 ~ 4.5
- Appendix D Matrices

Written exercise 4.1

- Use Strassen's algorithm to compute the matrix product

$$\begin{pmatrix} 1 & 3 \\ 7 & 5 \end{pmatrix} \begin{pmatrix} 6 & 8 \\ 4 & 2 \end{pmatrix} .$$

Show your work.

Written exercise 4.2

- Write pseudocode for Strassen's algorithm.

Written exercise 4.3

- Use the substitution method to show that each of the following recurrences defined on the reals has the asymptotic solution specified:

  a. $T(n) = T(n - 1) + n$ has solution $T(n) = O(n^2)$.

  b. $T(n) = T(n/2) + \Theta(1)$ has solution $T(n) = O(\lg n)$.

  c. $T(n) = 2T(n/3) + \Theta(n)$ has solution $T(n) = \Theta(n)$.

Written exercise 4.4

- For each of the following recurrences, sketch its recursion tree, and guess a good asymptotic upper bound on its solution. Then use the substitution method to verify:

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

  b. $T(n) = 3T(n - 1) + 1$.

Written exercise 4.5

- Use the master method to give tight asymptotic bounds for the following recurrences.

  a. $T(n) = 2T(n/4) + 1$.

  b. $T(n) = 2T(n/4) + \sqrt{n}$.

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