Chapter 4
Divide and Conquer

The slides for this course are based on the course textbook: Cormen, Leiserson, Rivest, and Stein, *Introduction to Algorithms*, 2nd edition, The MIT Press, McGraw-Hill, 2001.

- Many of the slides were provided by the publisher for use with the textbook. They are copyrighted, 2001.
- These slides are for classroom use only, and may be used only by students in this specific course this semester. They are NOT a substitute for reading the textbook!

Chapter 4 Topics

- The substitution method
- The recursion-tree method
- The master method

Designing Algorithms

- There are a number of design paradigms for algorithms that have proven useful for many types of problems
- Insertion sort – incremental approach
- Other examples of design approaches
  - divide and conquer
  - greedy algorithms
  - dynamic programming

Divide and Conquer

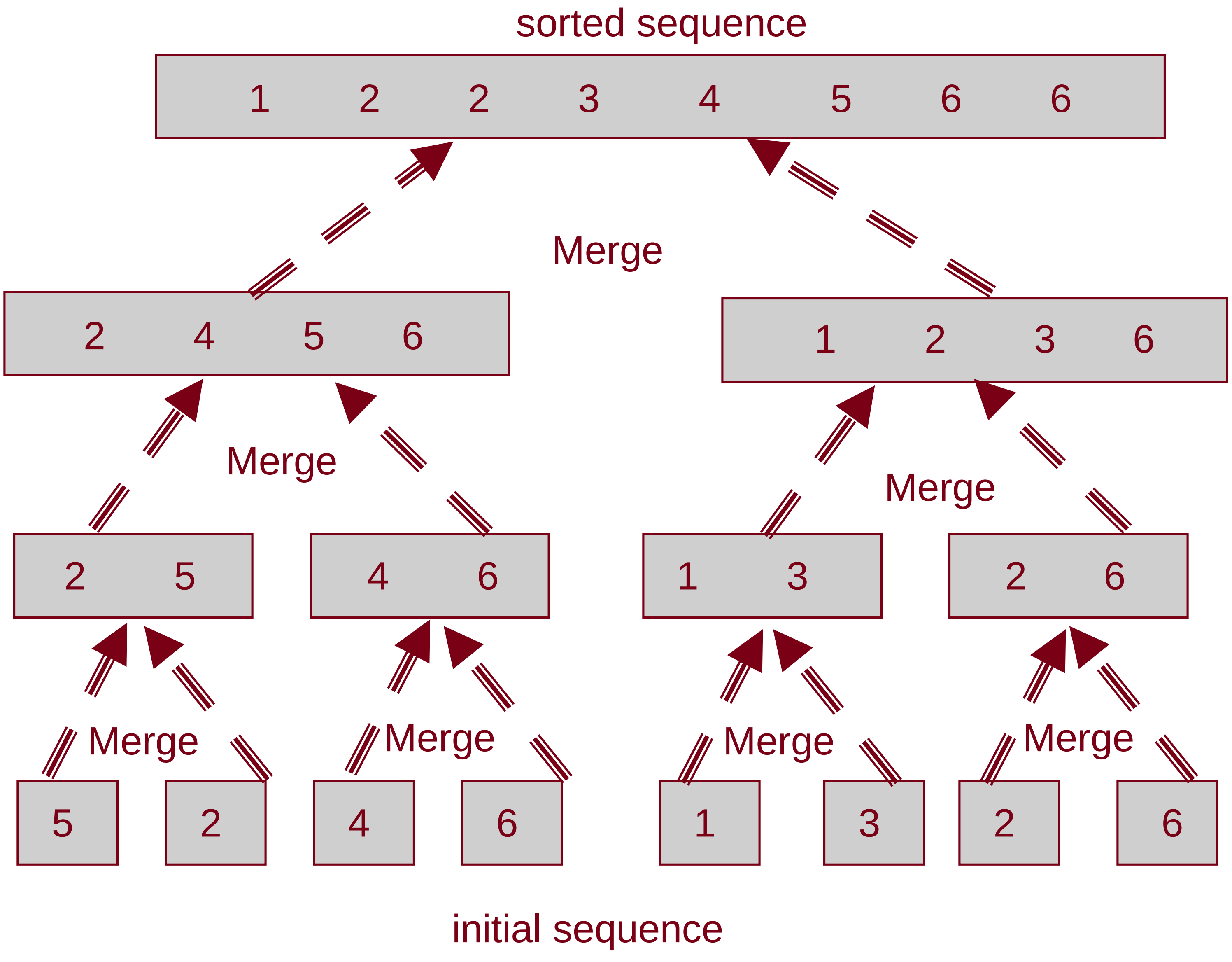
- A good divide and conquer algorithm generally implies an easy recursive version of the algorithm
- Three steps
  - Divide the problem into a number of subproblems
  - Conquer the subproblems by solving them recursively. When the subproblem size is small enough, just solve the subproblem.
  - Combine - the solutions of subproblems to form the solution of the original problem

Merge Sort

- Divide
  - divide an n-element sequence into two *n/2* element sequences
- Conquer
  - if the resulting list is of length 1 it is sorted
  - else call the merge sort recursively
- Combine
  - merge the two sorted sequences

```
MERGE-SORT (A,p,r)
    if p < r
        then q ← ⌊(p+r)/2⌋
            MERGE-SORT(A,p,q)
            MERGE-SORT(A,q+1,r)
            MERGE(A,p,q,r)
```

To sort A[1..n], invoke MERGE-SORT with MERGE-SORT(A,1,length(A))


sorted sequence
1 2 2 3 4 5 6 6
Merge
2 4 5 6
1 2 3 6
Merge
Merge
2 5
4 6
1 3
2 6
Merge
Merge
Merge
Merge
5
2
4
6
1
3
2
6
initial sequence

Recurrences

Definition –

a recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs

Recurrence for Divide and Conquer Algorithms

$$T(n) = \begin{cases} \Theta(1) & \leftarrow \textit{Base case} \\ \underbrace{aT(n/b)}_{\textit{Conquer cost}} + \underbrace{D(n)}_{\textit{Divide cost}} + \underbrace{C(n)}_{\textit{Combine cost}} \end{cases}$$

Analysis of Merge-Sort

Here is what we got as the running time:

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

We can ignore the $\Theta(1)$ factor, as it is irrelevant compared to $\Theta(n)$, and we can rewrite this recurrence as:

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

Recurrence for Merge Sort

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

- $\Theta(1)$ represents the running time of the base case.
- The “divide” phase really only involves resetting the lower and upper bounds of the current subarray, which has almost no cost associated with it.
- $T(n/2)$ is the cost of each of the “conquer” parts of the algorithm, and we have two parts, for a cost of $2T(n/2)$.
- $\Theta(n)$ is the cost of the “combine” part, the merge function.

Why Recurrences?

- The complexity of many interesting algorithms is easily expressed as a recurrence – especially divide and conquer algorithms
- The complexity of recursive algorithms is readily expressed as a recurrence.

Why solve recurrences?

- To make it easier to compare the complexity of two algorithms
- To make it easier to compare the complexity of the algorithm to standard reference functions.

Example Recurrences for Algorithms

- Insertion sort

$$T(n) = \begin{cases} 1 & \text{for } n \leq 1 \\ T(n-1) + n & \text{otherwise} \end{cases}$$

- Linear search of a list

$$T(n) = \begin{cases} 1 & \text{for } n \leq 1 \\ T(n-1) + 1 & \text{otherwise} \end{cases}$$

Recurrences for Algorithms, continued

- Binary search

$$T(n) = \begin{cases} 1 & \text{for n} \leq 1 \\ \mathrm{T(n/2)} + 1 & \text{otherwise} \end{cases}$$

“Casual” About Some Details

- Boundary conditions
  - These are usually constant for small n
- Floors and ceilings
  - Usually makes no difference in solution
  - Usually assume n is an “appropriate” integer (i.e., a power of 2) and assume that the function behaves the same way if floors and ceilings were taken into consideration

Merge Sort Assumptions

- The actual recurrence describing the worst-case running time for merge sort is:

$$T(n) = \begin{cases} \Theta(1) & \text{for n} \le 1 \\ \mathrm{T}(\lfloor n/2 \rfloor) + \mathrm{T}(\lceil n/2 \rceil) + \Theta(n) & \text{otherwise} \end{cases}$$

- But we typically assume that $n = 2^k$ where k is an integer and use the simpler recurrence.

Methods for Solving Recurrences

- Constructive induction
- Iterative substitution
  - Recurrence trees
- Master Theorem

Constructive Induction

- Use mathematical induction to derive an answer
- Steps
  1. Guess the form of the solution
  2. Use mathematical induction to find constants or show that they can be found and to prove that the answer is correct

Constructive induction

- Goal
  - Derive a function of *n* (or other variables used to express the size of the problem) that is not a recurrence so we can establish an upper and/or lower bound on the recurrence
  - We may get an exact solution or we may just get upper or lower bounds on the solution

Constructive Induction

- Suppose T includes a parameter n and n is a natural number (positive integer)
- Instead of proving directly that T holds for all values of n, prove
  - T holds for a base case b (often $n = 1$)
  - For every $n > b$, if T holds for $n-1$, then T holds for n.
    - » Assume T holds for $n-1$
    - » Prove that T holds for n follows from this assumption

Example 1

- Given

$$T(n) = \begin{cases} 1 & \text{for n} \leq 1 \\ \text{T(n - 1)} + \text{n} & \text{otherwise} \end{cases}$$

- Prove $T(n) \in O(n^2)$
  - Note that this is the recurrence for insertion sort and we have already shown that this is $O(n^2)$ using other methods

$$T(n) = \sum_{i=1}^{n} i = \frac{n(n+1)}{2} \in O(n^2)$$

Proof for Example 1

- Guess that the solution for T(n) is a quadratic equation

$$T(n) = an^2 + bn + c$$

- Assume this solution holds for *n-1*

$$T(n-1) = a(n-1)^2 + b(n-1) + c$$

- Now consider the case for *n*. Begin with the recurrence for *T(n)*

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

Proof for Ex. 1, continued

$T(n) = T(n – 1) + n$

We assumed that

$$T(n – 1) = a(n - 1)^2 + b(n – 1) + c$$

so we substitute this in the above equation:

$$T(n) = a(n - 1)^2 + b(n – 1) + c + n$$

Now let's multiply this out:

$$(n – 1)^2 = n^2 – 2n + 1, \text{ so}$$

$T(n) = an^2 – 2an + a + bn – b + c + n$, and

$T(n) = an^2 – 2an + bn + n + a – b + c$, and

$T(n) = an^2 + (-2a + b + 1)n + (a – b + c)$

Proof for Ex. 1, continued

We now can see that

$T(n) = an^2 + (-2a + b + 1)n + a - b + c$.

We know that a, b, and c are just names for arbitrary constants, so set $a = a$, $b = (-2a + b + 1)$, and $c = (a - b + c)$.

Now we can calculate a:

$$b = (-2a + b + 1)$$

$$0 = -2a + 1 = 1 - 2a$$

$$2a = 1$$

$$a = 1/2$$

Proof for Ex. 1, continued

And now we can calculate b:

$$c = (a - b + c)$$

$$0 = a - b$$

$$0 = \frac{1}{2} - b$$

$$b = 1/2$$

Proof for Ex. 1 continued

The values for a and b are now constrained, but the value for c is not. However, we now have a more complete hypothesis, and we can use this new hypothesis and the definition of the recurrence to get a value for c.

We know that:

$$T(n) = \tfrac{1}{2} n^2 + \tfrac{1}{2} n + c$$

and substituting 0 for n we get

$$T(0) = \tfrac{1}{2} 0^2 + \tfrac{1}{2} 0 + c = c$$

but

$$T(0) = 0 \quad \text{(the case when } n = 0\text{)}$$

so

$$T(0) = c = 0$$

Proof for Ex. 1 continued

We know that:

$$T(n) = ½\, n^2 \ + \ ½\, n \ + \ c$$

Substituting 0 for c we get

$$T(n) = ½\, n^2 \ + \ ½\, n \ \text{ for } n \geq 0$$

which, in Big-O notation is: $O(n^2)$

Compare this to what we determined to be the running time of Insertion Sort by a direct analysis of the algorithm:

$$T(n) = \sum_{i=1}^{n} i = \frac{n(n+1)}{2} \in O(n^2)$$

Example 2 – Establishing an Upper Bound

Recurrence : $T(n) = 4T(n/2) + n$

Guess : $T(n) \in O(n^3)$

Assumption : $n = 2^k$ where k is an integer

In this case we want to prove that $T(n) \le cn^3 \quad \forall n \ge n_0$

Assume $T(n/2) \le c(n/2)^3 \quad \forall n \ge n_0$

Starting with the recurrence for $T(n)$

$$
\begin{aligned}
T(n) &= 4T(n/2) + n \\
&\le 4c(n/2)^3 + n \\
&\le 1/2\,cn^3 + n
\end{aligned}
$$

This is not quite what we need : $T(n) \le c(n)^3$

Ex. 2 – Establishing an Upper Bound

We want to prove that $T(n) \leq cn^3 \quad \forall n \geq n_0$

$T(n) \leq 1/2cn^3 + n$

Trick

$T(n) \leq 1/2cn^3 + n$

$\leq cn^3 - (\tfrac{1}{2}cn^3 - n)$

$\leq cn^3 \quad \forall c > 2 \quad \text{and } n > 1$

The "trick" is recognizing that if x ≤ y - z then it must be true that x ≤ y (provided that z is positive).

General heuristic – try to write the expression in the form

< answer you want > - < something greater than 0 >

Ex. 2 – Establishing an Upper Bound

We still need a boundary condition specified. We have shown that $T(n) \leq cn^3$ for all $c > 2$ and $n \geq 1$.

Now select a c value that is large enough to satisfy a boundary condition. In this case we can select $c = 3$ for a boundary condition of $n = 1$.

Note that we have established an upper bound, but it is not a tight upper bound. See the next example.

Ex. 3 – Fallacious Argument

Recurrence: $T(n) = 4T(n/2) + n$

Guess: $T(n) \in O(n^2)$

Assumption: $n = 2^k$ where k is an integer

In this case we want to prove that $T(n) \leq cn^2 \quad \forall n \geq n_0$

Assume $T(n/2) \leq c(n/2)^2 \quad \forall n \geq n_0$

Starting with the recurrence for $T(n)$

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

$\therefore T(n) \in O(n^2)$

But this is incorrect, because $cn^2 + n \leq cn^2$ only holds for $n \leq 0$ and it must hold for all n greater than the base

Example 3 – Try again

When you get to this point

$T(n) \le cn^2 + n$

Revise the inductive hypothesis

Heuristic:

When you find yourself in the situation

$$T(n) \le < \text{term you want} > + < \text{something} + >$$

start over with a new inductive hypothesis in which you substract a lower order term.

Guess $T(n) \le c_1 n^2 - c_2 n$

Assume $T(n/2) \le c_1 (n/2)^2 - c_2 (n/2)$

Starting with recurrence

$T(n) = 4T(n/2) + n$

Ex. 3–Try again, continued

Starting with the recurrence

$$
\begin{aligned}
T(n) &= 4T(n/2) + n \\
&\le 4(c_1(n/2)^2 - c_2(n/2)) + n \\
&\le c_1 n^2 - 2c_2 n + n \\
&\le c_1 n^2 - c_2 n - (c_2 n - n)
\end{aligned}
$$

Now the first two terms are in the correct form and the last term is positive for all values of $c_2 \ge 1$ so

$T(n) \le c_1 n^2 - c_2 n$ for all $c_2 \ge 1$

Select c_1 to be large enough to handle the intial conditions.

Boundary Conditions

- Boundary conditions are not usually important because we don't need an actual *c* value (if polynomially bounded)
- But sometimes it makes a big difference
  - Exponential solutions
  - Suppose we are searching for a solution to:

    $T(n) = T(n/2)^2$
  - and we find the partial solution:

    $T(n) = c^n$

Boundary Conditions, cont.

If the boundary condition is

$$T(n) = 2$$

this implies that $T(n) \in \Theta(2^n)$.

But if the boundary condition is

$$T(n) = 3$$

this implies that $T(n) \in \Theta(3^n)$,

and $\Theta(3^n) \neq \Theta(2^n)$.

The results are even more dramatic if $\mathrm{T}(1) = 1$

$$T(1) = 1 \Rightarrow T(n) = \Theta(1^n) = \Theta(1)$$

Boundary Conditions

The solutions to the recurrences below have very different upper bounds:

$$T(n) = \begin{cases} 1 & \text{for } n = 1 \\ T(n/2)^2 & \text{otherwise} \end{cases}$$

$$T(n) = \begin{cases} 2 & \text{for } n = 1 \\ T(n/2)^2 & \text{otherwise} \end{cases}$$

$$T(n) = \begin{cases} 3 & \text{for } n = 1 \\ T(n/2)^2 & \text{otherwise} \end{cases}$$

Iterating the Recurrence

- Called *iterative substitution*
- Sometimes referred to as *plug and chug*.
- In iterative substitution we substitute the original form of the recurrence everywhere T occurs on the right side of the recurrence equation.
- Repeat as needed until a pattern appears.
- The math can be messy with this method.
- Sometimes we can use this method to get an estimate that we can use for the substitution method.

Iterating the Recurrence

Look at the recurrence relation:

$$T(n) = \begin{cases} 0 & \text{if } n = 0 \\ T(n - 1) + n & \text{if } n > 0 \end{cases}$$

Substituting n – 1 for n in the relation above we get:

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

Substitute for n – 1 in the original relation:

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

We know that

$$T(n - 2) = T(n - 3) + (n - 2)$$

So substitute this for T(n – 2) above:

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

Iterating the Recurrence

We see the following pattern:

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

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

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

. . .

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

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

$$\mathbf{T(n) = T(n - (n - 0)) + 2 + 3 + \ldots + (n - 2) + (n - 1) + n}$$

We can rewrite $(n - (n - 0))$ as $(n - n)$ or as (0), thus:

$$\mathbf{T(n) = T(0) + 1 + 2 + 3 + \ldots + (n - 2) + (n - 1) + n}$$

But we know that $T(0) = 0$ is the base case, so:

$$\mathbf{T(n) = 0 + 1 + 2 + 3 + \ldots + (n - 2) + (n - 1) + n}$$

Iterating the Recurrence

The summation of

$$\mathbf{T(n) = 0 + 1 + 2 + 3 + \ldots + (n - 2) + (n - 1) + n}$$

is

$$\mathbf{T(n) = (n\ (n + 1)\ /2) = \tfrac{1}{2}\ n^2 + \tfrac{1}{2}\ n}$$

which we recognize as $O(n^2)$.

Iterating the Recurrence

Let's look at the recurrence equation for Merge Sort again:

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

Let's substitute 2T(n/2) + cn for T(n/2) in the expression above:

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

Let's substitute 2T(n/2) + cn again:

$$\begin{aligned} &= 2^2(2T((n/2^2)/2 + c\,((n/2)/2) + 2cn \\ &= 2^3T(n/2^3) + 3cn \end{aligned}$$

What pattern emerges?

Iterating the Recurrence

$2^1T(n/2^1) + 1cn$

$2^2T(n/2^2) + 2cn$

$2^3T(n/2^3) + 3cn$

$\downarrow$

$2^iT(n/2^i) + icn$

Assume that $n = 2^i$ (n is an integer power of 2); then $i = \log_2 n$.

Substituting $\log_2 n$ for i gives:

$$2^{\log_2 n} \cdot T(n/n) + \log_2 n \cdot c \cdot n$$

Remember that $a^{\log_b n} = n^{\log_b a}$, so we have

$$n^{\log_2 2} \cdot T(n/n) + \log_2 n \cdot c \cdot n$$

Iterating the Recurrence

$n^{\log_2 2}$ is n^1 or simply n, so we have:

$$n \cdot T(n/n) + \log_2 n \cdot c \cdot n$$

We know that n/n = 1, so we have:

$$n \cdot T(1) + \log_2 n \cdot c \cdot n$$

We know that T(1) is the base case for this recurrence, and T(n) = c if n = 1, so we have:

$$n \cdot c + \log_2 n \cdot c \cdot n$$

Rearranging the right and left sides of the summation gives:

$$c \cdot n \cdot \log_2 n + c \cdot n$$

Which is $O(n \log_2 n)$

Example 4

$T(n) = n + 4T(n/2)$

Start iterating the recurrence

$T(n) = n + 4(n/2 + 4T(n/4))$

$\quad = n + 2n + 16T(n/4)$

Iterate the recurrence again

$T(n) = n + 2n + 16(n/4 + 4T(n/8))$

$\quad = n + 2n + 4n + 64T(n/8)$

We observe that the *ith* term in the series is $2^i n$

How far do we iterate before we reach a boundary condition?

If we use 1 as the boundary condition, it will be when we reach

$n/2^i = 1$

Example 4, continued

When

$n/2^i = 1$ then $i = \lg n$

Now, since we know that the *ith* term is $2^{\mathrm{i}} n$

we can rewrite the series as

$$T(n) = n + 2n + 4n + \ldots + 2^{\lg n} nT(1)$$

Remember that $a^{\log_b n} = n^{\log_b a}$

$$T(n) = n + 2n + 4n + \ldots + n^{\lg 2} n$$

$$= n + 2n + 4n + \ldots + n^2$$

$$= n + 2n + 4n + \ldots + 2^{\lg n - 1} n + n^2$$

$$T(n) == n + 2n + 4n + \ldots + 2^{\lg n - 1} n + n^2 T(1)$$

Factor out a geometric progression

$$\sum_{i=0}^{n} x^k = \frac{x^{n+1} - 1}{x - 1} \quad \text{for } x \neq 1$$

$$T(n) = n(2^0 + 2^1 + 2^2 \ldots + 2^{\lg n - 1}) + n^2\, T(1)$$

$$= n\left(\frac{2^{\lg n} - 1}{2 - 1}\right) + \Theta(n^2)$$

$$= n(n-1) + \Theta(n^2)$$

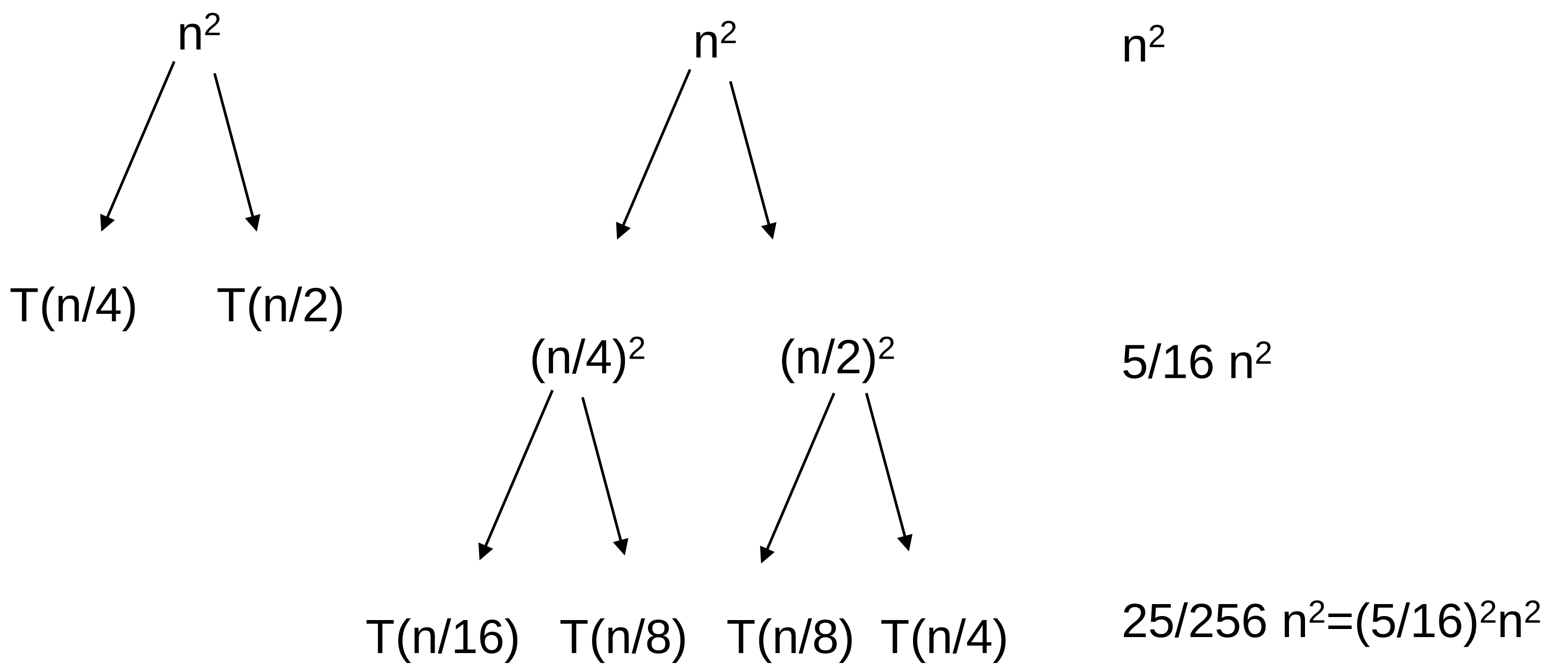
$$= \Theta(n^2) + \Theta(n^2)$$

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

Recurrence Trees

- Allow you to visualize the process of iterating the recurrence
- Allows you make a good guess for the substitution method
- Or to organize the bookkeeping for iterating the recurrence
- Example

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



Since the values decrease geometrically, the total is at most a constant factor more than the largest term and hence the solution is $\Theta(n^2)$

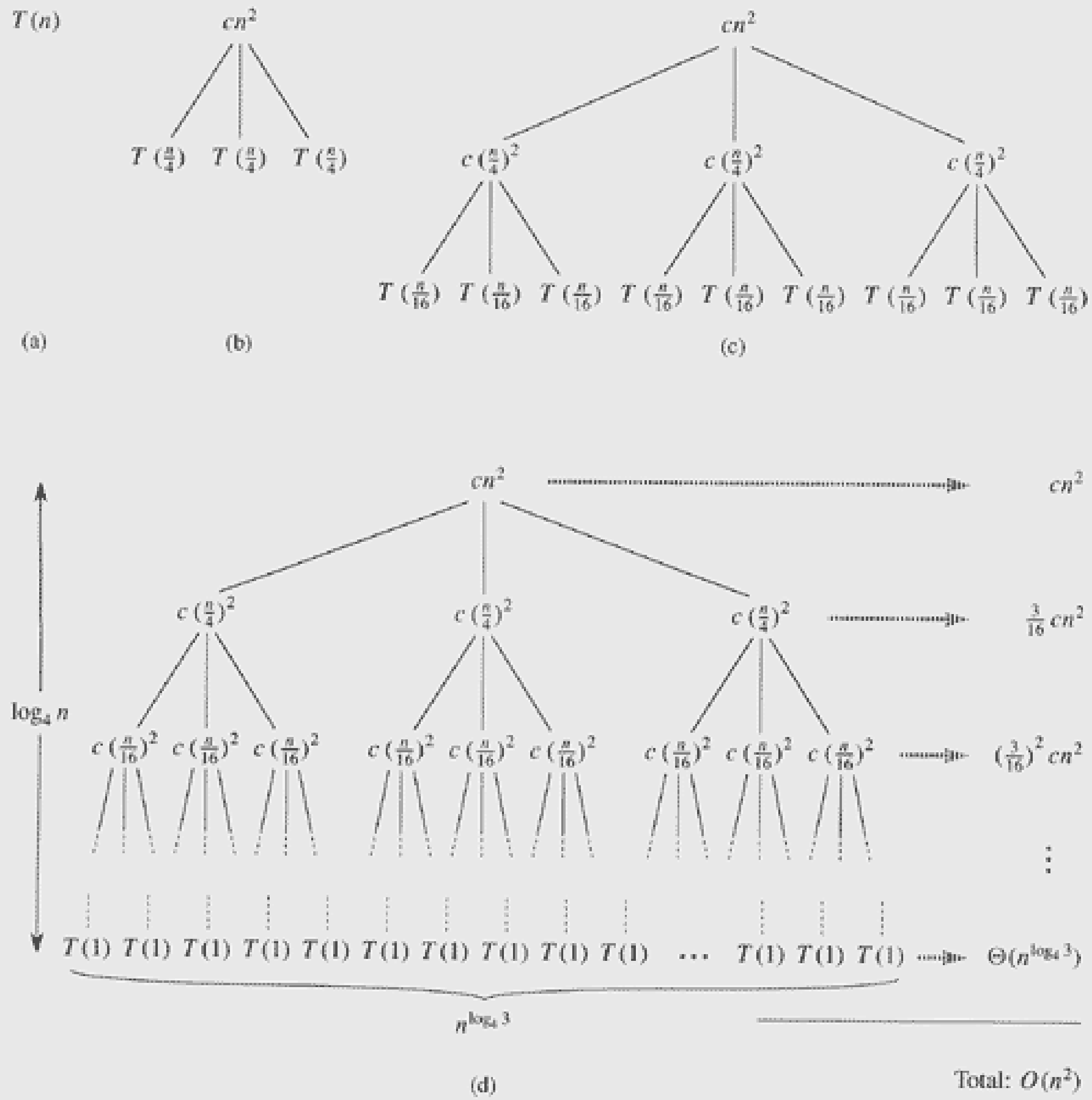


Figure 4.1 The construction of a recursion tree for the recurrence $T(n) = 3T(n/4) + cn^2$. Part **(a)** shows $T(n)$, which is progressively expanded in **(b)–(d)** to form the recursion tree. The fully expanded tree in part (d) has height $\log_4 n$ (it has $\log_4 n + 1$ levels).

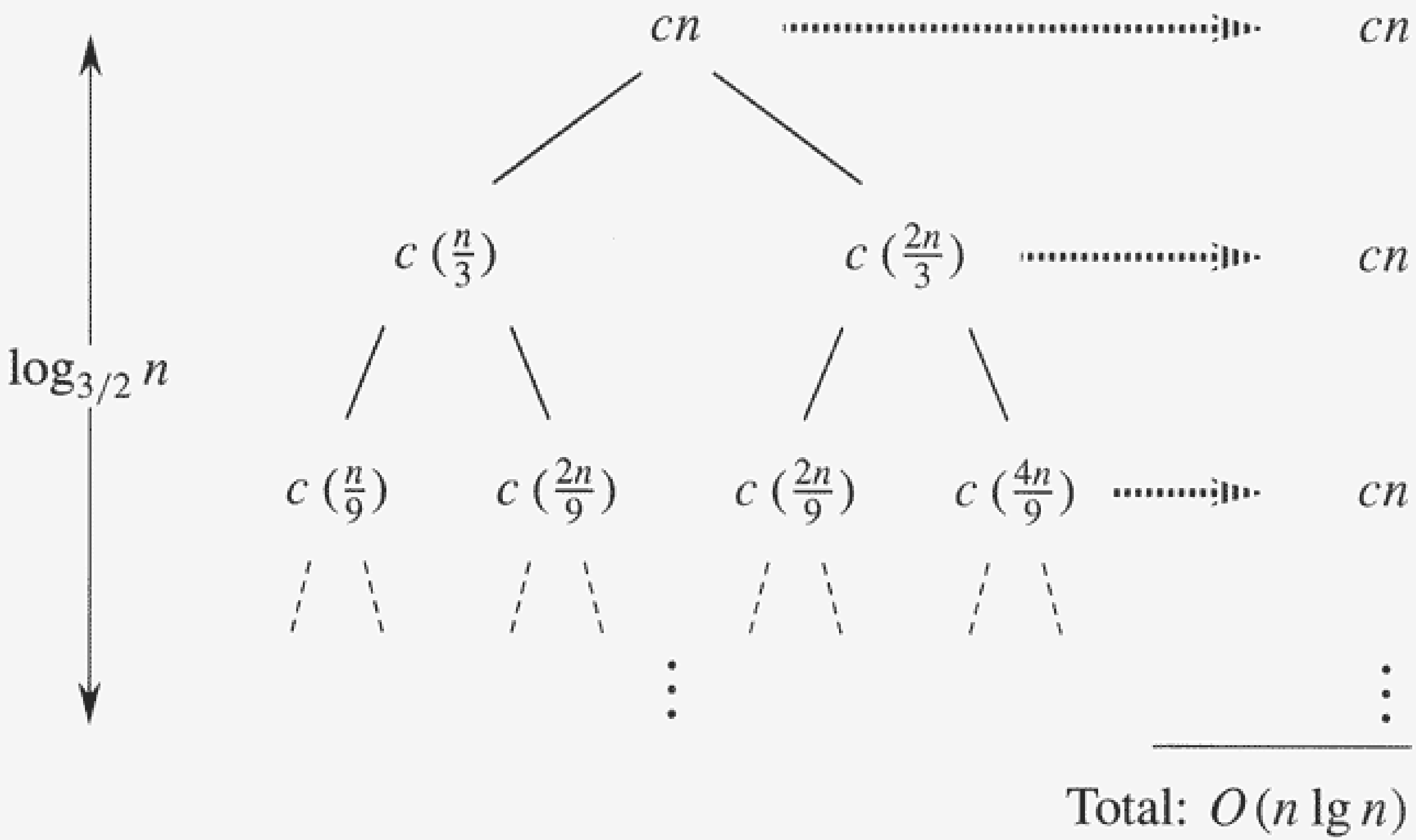


Figure 4.2 A recursion tree for the recurrence $T(n) = T(n/3) + T(2n/3) + cn$.

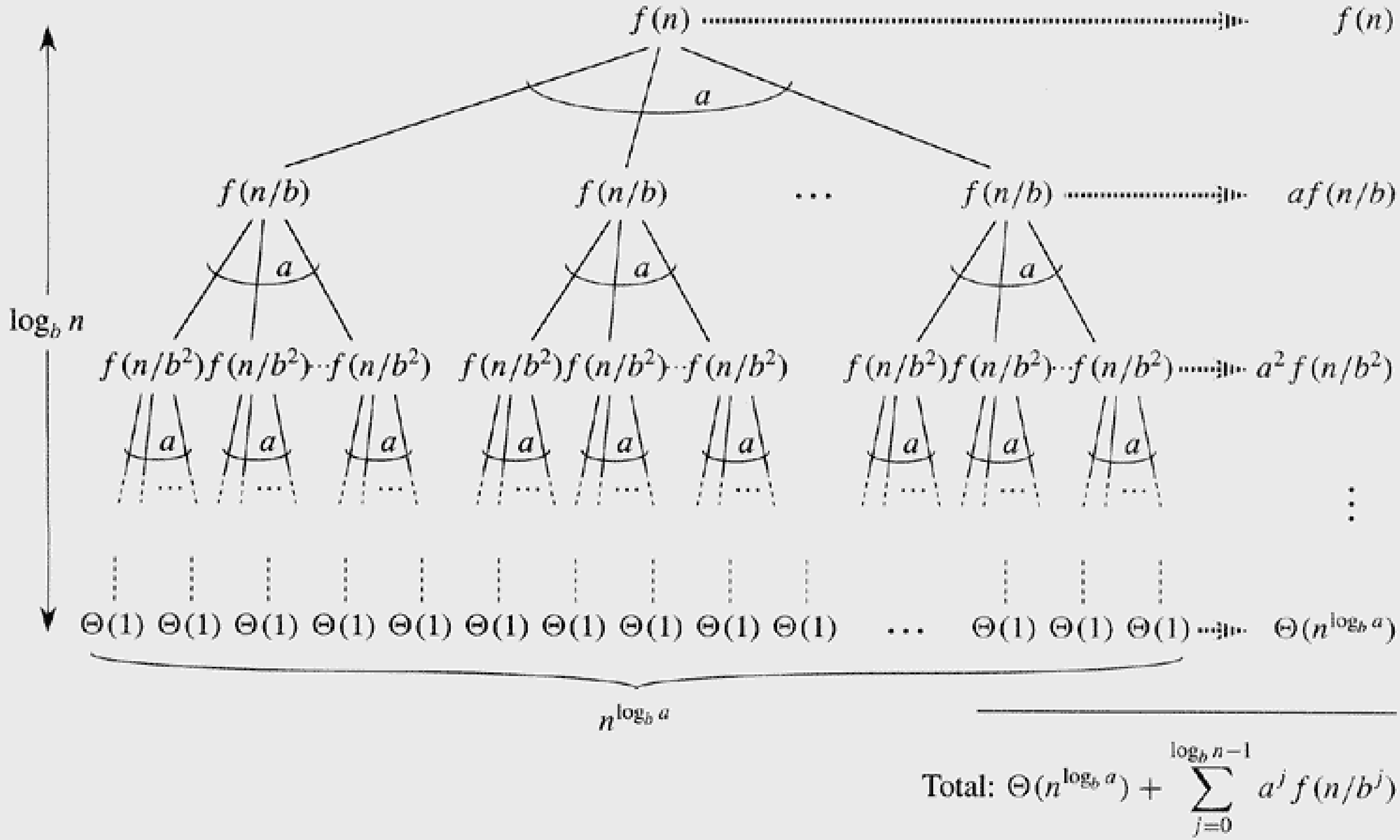


Figure 4.3 The recursion tree generated by $T(n) = aT(n/b) + f(n)$. The tree is a complete a-ary tree with $n^{\log_b a}$ leaves and height $\log_b n$. The cost of each level is shown at the right, and their sum is given in equation (4.6).

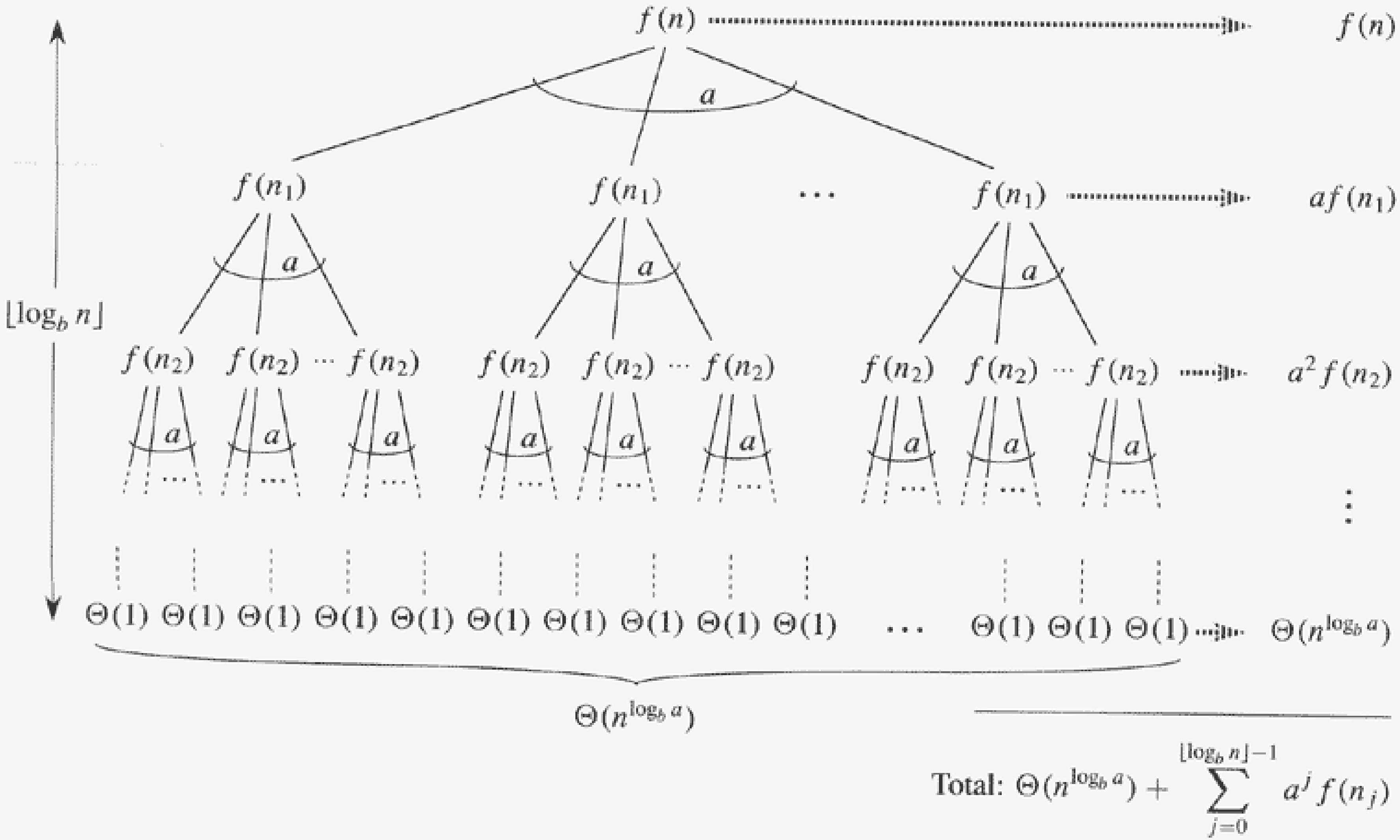


Figure 4.4 The recursion tree generated by $T(n) = aT(\lceil n/b \rceil) + f(n)$. The recursive argument n_j is given by equation (4.12).

The master method

Provides a cookbook method for solving recurrences of the form

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

where a $\geq$ 1 and b > 1 and *f(n)* is an asymptotically positive function.

Divide and Conquer Algorithms

- The form of the master theorem is very convenient because divide and conquer algorithms have recurrences of the form

$$T(n) = aT(n/b) + D(n) + C(n)$$

where

a is the number of subproblems at each step

$1/b$ is the size of each subproblem

$D(n)$ is the cost of dividing into subproblems

$C(n)$ is the cost of combining the solutions to subproblems

Form of the Master Theorem

- Combines *D(n)* and *C(n)* into *f(n)*
- For example, in Merge-Sort

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

$a = 2$, $b = 2$

$f(n) = \Theta(n)$

- We will ignore floors and ceilings. The proof of the Master Theorem includes a proof that this is ok.

Form of the Master Theorem

- Combines *D(n)* and *C(n)* into *f(n)*
- For example, in Merge-Sort

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

$a = 2$, $b = 2$

$f(n) = \Theta(\text{n})$

We will ignore floors and ceilings. The proof of the Master Theorem includes a proof that this is ok.

Form of the Master Theorem

- The Master Method is used for recurrence equations of the form:

$$T(n) = \begin{cases} c & \text{for n} < \text{d} \\ \text{aT(n/b)} + f(\text{n}) & \text{for n} \geq 1 \end{cases}$$

Master theorem

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the non-negative integers by the recurrence

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

where we interpret n/b to mean either the floor or ceiling of n/b. Then $T(n)$ can be bounded asymptotically as follows:

Master theorem

Case 1: if $f(n) = O\left(n^{\log_b a - \varepsilon}\right)$ for some constant $\varepsilon > 0$, then

$$T(n) = \Theta\left(n^{\log_b a}\right)$$

Case 2: if $f(n) = \Theta\left(n^{\log_b a}\right)$, then

$$T(n) = \Theta\left(n^{\log_b a} \lg n\right)$$

Case 3: if $f(n) = \Omega\left(n^{\log_b a + \varepsilon}\right)$ for some constant $\varepsilon > 0$, and

if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all

sufficiently large n, then

$$T(n) = \Theta\left(f(n)\right)$$

3 cases

1. If there is a small constant ε > 0, such that

$$f(n) = O\left(n^{\log_b a - \varepsilon}\right)$$

then T(n) is

$$\Theta\left(n^{\log_b a}\right)$$

Here *f(n)* is polynomially <u>smaller</u> than the special function $n^{\log_b a}$

3 cases

2. If

$$f(n) = \Theta\left(n^{\log_b a}\right)$$

then T(n) is

$$\Theta\left(n^{\log_b a} \lg n\right)$$

Here *f(n)* is asymptotically <u>equal to</u> the special function

$$n^{\log_b a}$$

3 cases

3. If there are small constants ε > 0 and c < 1, such that $af(n/b) \le cf(n)$

$$f(n) = \Omega\left(n^{\log_b a + \varepsilon}\right)$$

for all sufficiently large n, then T(n) is

$$\Theta(f(n))$$

Here *f(n)* is polynomially <u>larger</u> than the special function $n^{\log_b a}$

What does the master theorem say?

Compare two functions :

$$f(n) \qquad \text{and} \qquad n^{\log_b a}$$

When $f(n)$ grows asymptotically slower (Case 1)

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

When the growth rates are the same (Case 2)

$$T(n) = \Theta(f(n)\lg n) = \Theta(n^{\log_b a} \lg n)$$

When $f(n)$ grows asymptotically faster (Case 3)

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

Using the Master Method

Using the master method, solve the recurrence

$T(n) = 4T(n/2) + n$

Remember the form the recurrence must have:

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

Here $a = 4$, $b = 2$, and $f(n) = \mathrm{n}$

Plug these values into our special function $n^{\log_b a}$

and we get $n^{\log_2 4}$ or $= n^2$. Does $f(n) = O(n^{2-\varepsilon})$?

Yes, if $\varepsilon = 1$. So this is Case 1, and

$$T(n) = \Theta\left(n^{\log_2 4}\right) = \Theta\left(n^2\right)$$

Using the Master Method

How do we know that this is Case 1, and not Case 2 or Case 3? Look at *f(n)*. Does:

$$f(n) = O\left(n^{\log_b a - \varepsilon}\right)$$ yes

$$f(n) = \Theta\left(n^{\log_b a}\right)$$ no

$$f(n) = \Omega\left(n^{\log_b a + \varepsilon}\right)$$ no

Using the Master Method

$$T(n) = 64T(n/4) + n$$

$$a = 64 \qquad b = 4 \qquad f(n) = n$$

$$n^{\log_b a} = n^{\log_4 64} = n^3 = \Theta(n^3)$$

Since $f(n) = O(n^3)$ where $\varepsilon = 2$,

case 1 applies and

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

Using the Master Method

Using the master method, solve the recurrence

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

Remember the form the recurrence must have:

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

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

Plug these values into our special function

and we get $n^{\log_{3/2} 1}$ or = $n^0 = 1$. Does $f(n) = \Theta(1)$?

Yes. So this is Case 2, and

$$T(n) = \Theta(1 \bullet \lg n) = \Theta(\lg n)$$

Using the Master Method

$$T(n) = T(3n/4) + 1$$

$$a = 1 \qquad b = 4/3 \qquad f(n) = 1$$

$$n^{\log_b a} = n^{\log_{4/3} 1} = n^0 = 1$$

Case 2 applies and

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

Using the Master Method

Using the master method, solve the recurrence

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

Remember the form the recurrence must have:

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

Here $a = 1$, $b = 3$, and $f(n) = \mathrm{n}$

Plug these values into our special function and we get $n^{\log_3 1}$ or $= \mathrm{n}^0 = 1$. Does $f(n) = \Omega(\mathrm{n}^{0+\varepsilon})$? Yes; $\varepsilon = 1$, and $af(n/b) = \mathrm{n}/3 = (1/3)\mathrm{f}(\mathrm{n})$, giving $\mathrm{c} = 1/3$. So this is Case 3, and

$$\mathrm{T(n)} = \Theta(\mathrm{f(n)}) = \Theta(\mathrm{n})$$

Using the Master Method

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

$$a = 3 \qquad b = 4 \qquad f(n) = n\lg n$$

$$n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$$

Since $f(n) = \Omega(n^{\log_4 3+\varepsilon})$,

case 3 applies and

$$T(n) = \Theta(\mathrm{n}\lg n)$$

Conclusion

- We talked about:
  - ✓The substitution method (2 types)
  - ✓The recursion-tree method
  - ✓The master method
- Be able to solve recurrences using all three of these methods.

The Master Theorem

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let T(n) be defined on the nonegative integers by the recurrence $T(n) = aT(n/b) + f(n)$

where n/b can be either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$

Then T(n) can be bounded asymptotically as follows:

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

   then $T(n) = \Theta(n^{\log_b a})$
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$
3. If $f(n) = \Omega(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$, and

   if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all

   sufficiently large n, then $T(n) = \Theta(\mathrm{f(n)})$