Chapter 4
Divide and Conquer

The slides for this course are based on the course textbook: Cormen, Leiserson, Rivest, and Stein, Introduction to Algorithms, 3rd edition, The MIT Press, McGraw-Hill, 2010.

Chapter 4 Topics

- Maximum subarray
- 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

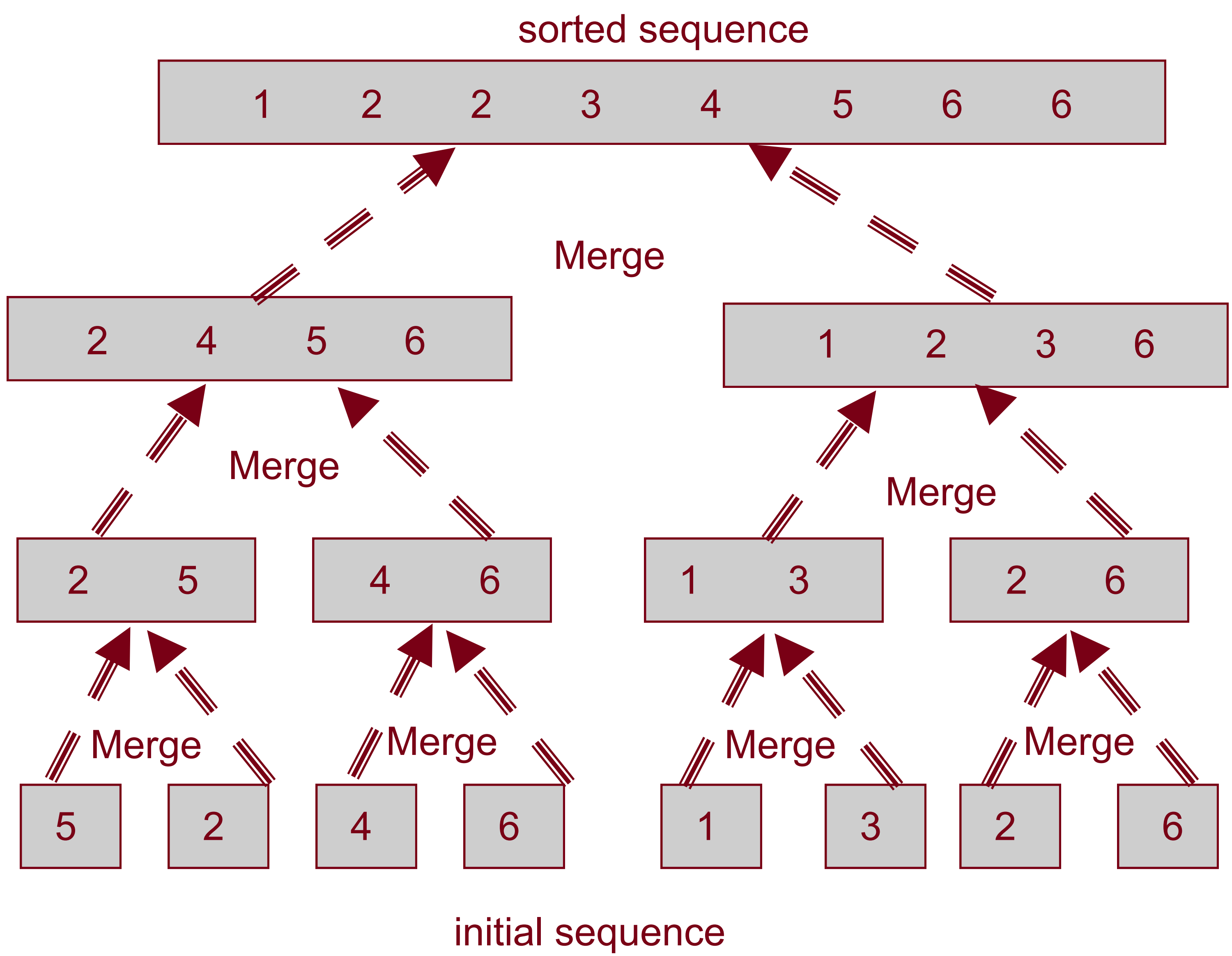
- A good divide and conquer algorithm generally implies an easy recursive version of the algorithm
- Three steps
  - Divide the problem into a number of subproblems
  - Conquer the subproblems by solving them recursively. When the subproblem size is small enough, just solve the subproblem.
  - Combine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

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

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:

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

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

- Linear search of a list

Recurrences for Algorithms, continued

- Binary search

“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

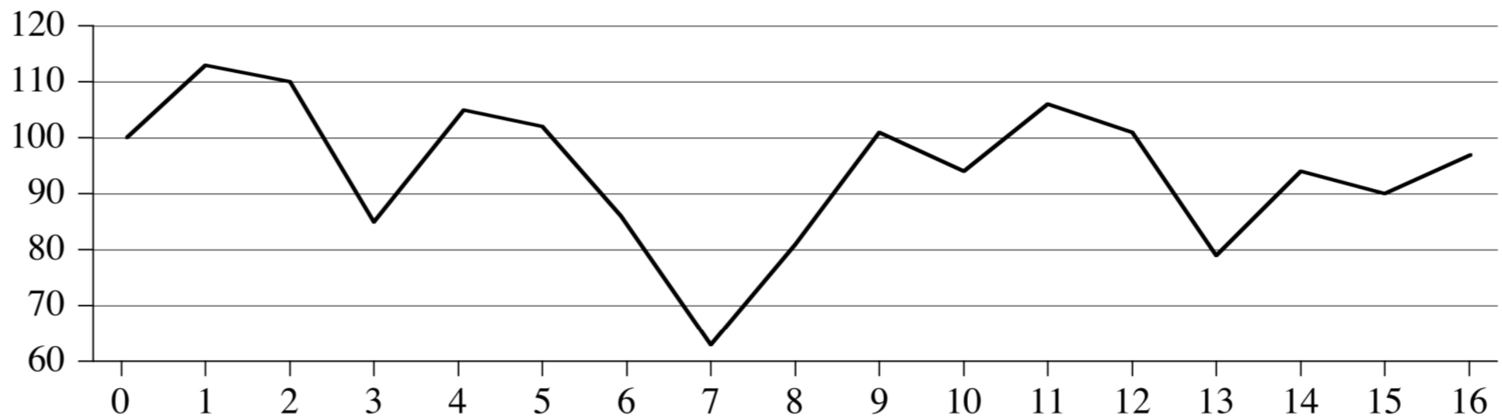
- The actual recurrence describing the worst-case running time for merge sort is:

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

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

Maximum-subarray Problem

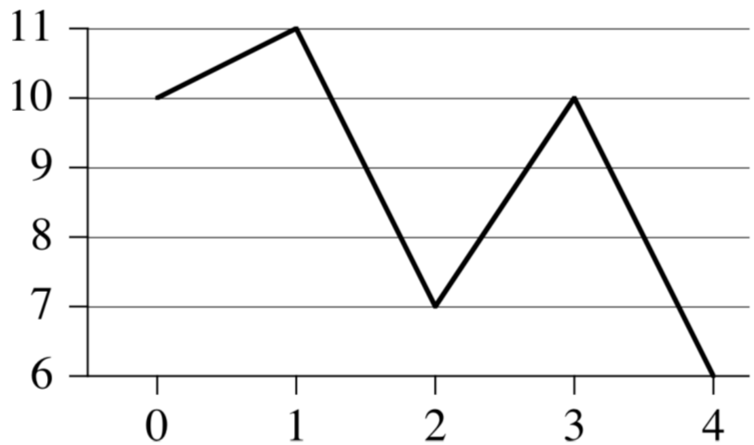
- Stock investment: Buy one unit of stock only one time and then sell it at a later date
- Goal: to maximize the profit



Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
Change		13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

One Potential Solution

- Find the highest price and search left to find the lowest prior price
- Find the lowest price and search right to find the highest later price
- Take the pair with the greater difference
- Do not work! See counterexample below.



Day	0	1	2	3	4
Price	10	11	7	10	6
Change		1	–4	3	–4

Maximum Subarray

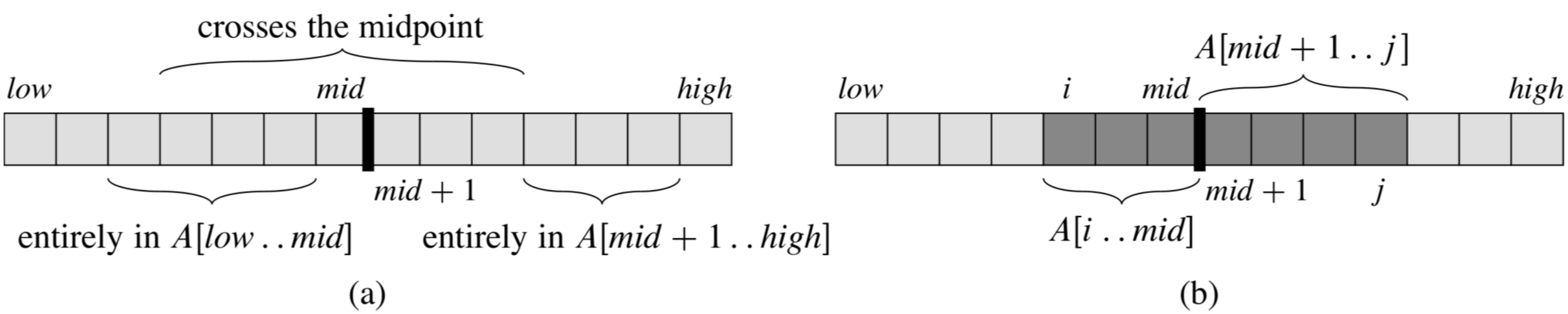
- Consider the daily change in price
- Maximum subarray problem: find the non-empty, contiguous subarray of A whose values have the largest sum.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	13	–3	–25	20	–3	–16	–23	18	20	–7	12	–5	–22	15	–4	7

maximum subarray

Divide and Conquer

- Suppose we want to find a maximum subarray of A[*low..high*]
- Divide and conquer will find the midpoint, say mid, of the subarray, and consider the subarrays A[*low..mid*] and A[*mid+1..high*]
- Any contiguous subarray A[*i..j*] must lie in one area out of three possibilities



Find Max Crossing Subarray

- First, it is easy to find a maximum subarray crossing the midpoint
- We just need to find maximum subarrays of the form A[$i..mid$] and A[$mid+1..j$] and combine them

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
  // Find a maximum subarray of the form A[i .. mid].
  left-sum = -∞
  sum = 0
  for i = mid downto low
      sum = sum + A[i]
      if sum > left-sum
          left-sum = sum
          max-left = i
  // Find a maximum subarray of the form A[mid + 1 .. j].
  right-sum = -∞
  sum = 0
  for j = mid + 1 to high
      sum = sum + A[j]
      if sum > right-sum
          right-sum = sum
          max-right = j
  // Return the indices and the sum of the two subarrays.
  return (max-left, max-right, left-sum + right-sum)
```

Find Maximum subarray

- We can then write a divide and conquer algorithm to solve the maximum subarray problem.
- Divide into three cases, and choose the best solution
  - Left subarray
  - Crossing subarray
  - Right subarray

```
FIND-MAXIMUM-SUBARRAY(A, low, high)
  if high == low
      return (low, high, A[low])          // base case: only one element
  else mid = ⌊(low + high)/2⌋
      (left-low, left-high, left-sum) =
          FIND-MAXIMUM-SUBARRAY(A, low, mid)
      (right-low, right-high, right-sum) =
          FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
      (cross-low, cross-high, cross-sum) =
          FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
      if left-sum ≥ right-sum and left-sum ≥ cross-sum
          return (left-low, left-high, left-sum)
      elseif right-sum ≥ left-sum and right-sum ≥ cross-sum
          return (right-low, right-high, right-sum)
      else return (cross-low, cross-high, cross-sum)
```

Analyzing the algorithm

- So the total running time is?

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + \underline{D(n)} + \underline{C(n)}$$

$$T(n) = 2T\left(\frac{n}{2}\right) + O(1) + n$$

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 $T(n) \ ? \ T(n-1)$
  - *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*.

    $T(n) \ ? \ T(n/2)$, if T holds for $n/2$ T holds for n

    - » Assume *T* holds for *n-1*
    - » Prove that *T* holds for *n* follows from this assumption

Example 1

- Given

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

- Prove $T(n) \in O(n^2)$ → tight upper bound
  - 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 of Example 1

$T(n) = an^2 + bn + c$ $\therefore$ $T(n) = \frac{1}{2}n^2 + \frac{n}{2}$

base: $T(1) = a \cdot 1^2 + b \cdot 1 + c = 1 \Rightarrow a + b + c = 1$ ①

induction step: suppose $T(n-1) = a(n-1)^2 + b(n-1) + c$

We need to show: $T(n) = T(n-1) + n$

$\Rightarrow an^2 + bn + c = a(n-1)^2 + b(n-1) + c + n$

$\Rightarrow an^2 + bn + c = an^2 - 2an + a + bn - b + c + n$

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

② $\cancel{b} = \cancel{b} + 1 - 2a \Rightarrow a = 1/2$ ③ $\cancel{c} = a - b + \cancel{c} \Rightarrow b = a = 1/2$

$\Rightarrow c = 0$

Example 2 – Establishing an Upper Bound

$$T(n) = \begin{cases} 1, & n = 1 \\ 4T(n/2) + n \end{cases}$$

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

Guess : $T(n) \in O(n^3)$ → loose upper bound

$\cancel{T(n) = an^3 + bn^2 + cn + d}$

$T(n) \leq C n^3$

base : $T(2) = 4T(1) + 2 \leq C \cdot 2^3 \Rightarrow C \geq \frac{6}{8}$

induction step : Assume $T(n/2) \leq C \cdot \left(\frac{n}{2}\right)^3$

We need to prove $T(n) \leq C \cdot n^3$

$T(n) = 4 \cdot T(n/2) + n \leq 4 \cdot C \cdot \left(\frac{n}{2}\right)^3 + n = \frac{C}{2} n^3 + n$

So $\frac{C}{2} n^3 + n \leq C n^3 \Rightarrow$

$\frac{C}{2} n^3 \geq n$

$C \geq \sqrt{2/n}$

≥ 2

Ex. 3 – Fallacious Argument

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

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

$T(n) \le cn^2$

base: skipped

Induction step: Assume $T\left(\frac{n}{2}\right) \le c\left(\frac{n}{2}\right)^2$

We need to prove $T(n) \le cn^2$

$T(n) = 4T(n/2) + n \le 4 \cdot c\left(\frac{n}{2}\right)^2 + n$

$= cn^2 + n$ failed!

Example 3 – Try again

Assume

$T(n) \le$ "desired term" – "positive term"

$T(n) \le c_1 n^2 - c_2 n \Rightarrow T(n) \in O(n^2)$

basecase: $n=2$

$T(n) = c_1 \cdot 2^2 - c_2 \cdot 2 \ge 1$

Inductive step: Assumption: $T(\frac{n}{2}) \le c_1 (\frac{n}{2})^2 - c_2 \frac{n}{2}$

We need to prove: $T(n) \le c_1 n^2 - c_2 n$ $\forall c_2 \ge 1$,

$T(n) = 4T(n/2) + n \le 4 \cdot c_1 (\frac{n}{2})^2 - 4 c_2 \frac{n}{2} + n$ $\quad T(n) \le c_1 n^2 - c_2 n$

$= c_1 n^2 - 2 c_2 n + n = c_1 n^2 - c_2 n - (c_2 - 1) n$

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 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.
- 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:

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

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

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

Substitute for n – 1 in the original relation:

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

We know that

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

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

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

Iterating the Recurrence

We see the following pattern:

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

$T(n) = (T(n – 2) + (n – 1)) + n$

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

. . .

$T(n) = T(n – (n – 2)) + 2 + 3 + \ldots + (n – 2) + (n – 1) + n$

$T(n) = T(n – (n – 1)) + 2 + 3 + \ldots + (n – 2) + (n – 1) + n$

$T(n) = T(n – (n - 0)) + 2 + 3 + \ldots + (n – 2) + (n – 1) + n$

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

$T(n) = T(0) + 1 + 2 + 3 + \ldots + (n – 2) + (n – 1) + n$

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

$T(n) = 0 + 1 + 2 + 3 + \ldots + (n – 2) + (n – 1) + n$ $= \frac{n(n+1)}{2}$

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

$T(n/2) = 2T(\frac{n/2}{2}) + c(\frac{n}{2})$

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

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

$$T(n) = 2\,\underline{T(n/2)} + cn$$

$$= 2\left(2\cdot T(n/2^2) + c\frac{n}{2}\right) + cn$$

$$= 2^2 T\left(\frac{n}{2^2}\right) + cn + cn$$

$$= 2^2\left(2\cdot T\left(\frac{n}{2^3}\right) + c\frac{n}{2^2}\right) + cn + cn$$

$$= 2^3 T\left(\frac{n}{2^3}\right) + cn + cn + cn$$

$$? = 2^{\log_2 n} T\left(\frac{n}{2^{\log_2 n}}\right) + \underbrace{cn + \cdots + cn}_{\log_2 n} \approx \log_2 n \cdot n + cn$$

$n \to n = 2^k$

$\frac{n}{2^k} = 1$

$k = \log_2 n$

$O(n \log n)$

Example 4

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

$= n(1 + 2 + 2^2 + \dots + 2^{\log_2 n}) + 4n^2$

$= n\,\dfrac{2^{\log_2 n + 1}}{2} + 4n^2 \leq O(n^2)$

$T(n/2) = \dfrac{n}{2} + 4T(n/2^2)$

$= n + 4\left(\dfrac{n}{2} + 4T\left(\dfrac{n}{2^2}\right)\right)$ $\qquad T\left(\dfrac{n}{2^2}\right) = \dfrac{n}{2^2} + 4T(n/2^3)$

$= n + 4 \cdot \dfrac{n}{2} + 4^2\, T\left(\dfrac{n}{2^2}\right)$ $\qquad 4^{\log_2 n} = 2^{2\log_2 n}$

$= n + 4 \cdot \dfrac{n}{2} + 4^2\left(\dfrac{n}{2^2} + 4T(n/2^3)\right)$ $\qquad = 2^{\log_2 n^2}$

$= n + 4 \cdot \dfrac{n}{2} + 4^2\,\dfrac{n}{2^2} + 4^3\,T\left(\dfrac{n}{2^3}\right)$ $\qquad = n^2$

$\dots$

$= n + 4\dfrac{n}{2} + 4^2\dfrac{n}{2^2} + 4^3 \cdot \dfrac{n}{2^3} + \dots$ $\qquad 1 = \dfrac{n}{2^k}$

$= \underbrace{n + 2n + 2^2 n + 2^3 n + \dots + n \cdot n}_{\log_2 n} + 4^{\log_2 n} \cdot \dfrac{n}{2^{\log_2 n}} + \cancel{4^{(\log_2 n + 1)}\, T(1)}$ $\qquad k = \log_2 n$

$+ 4n^2$

Example 5

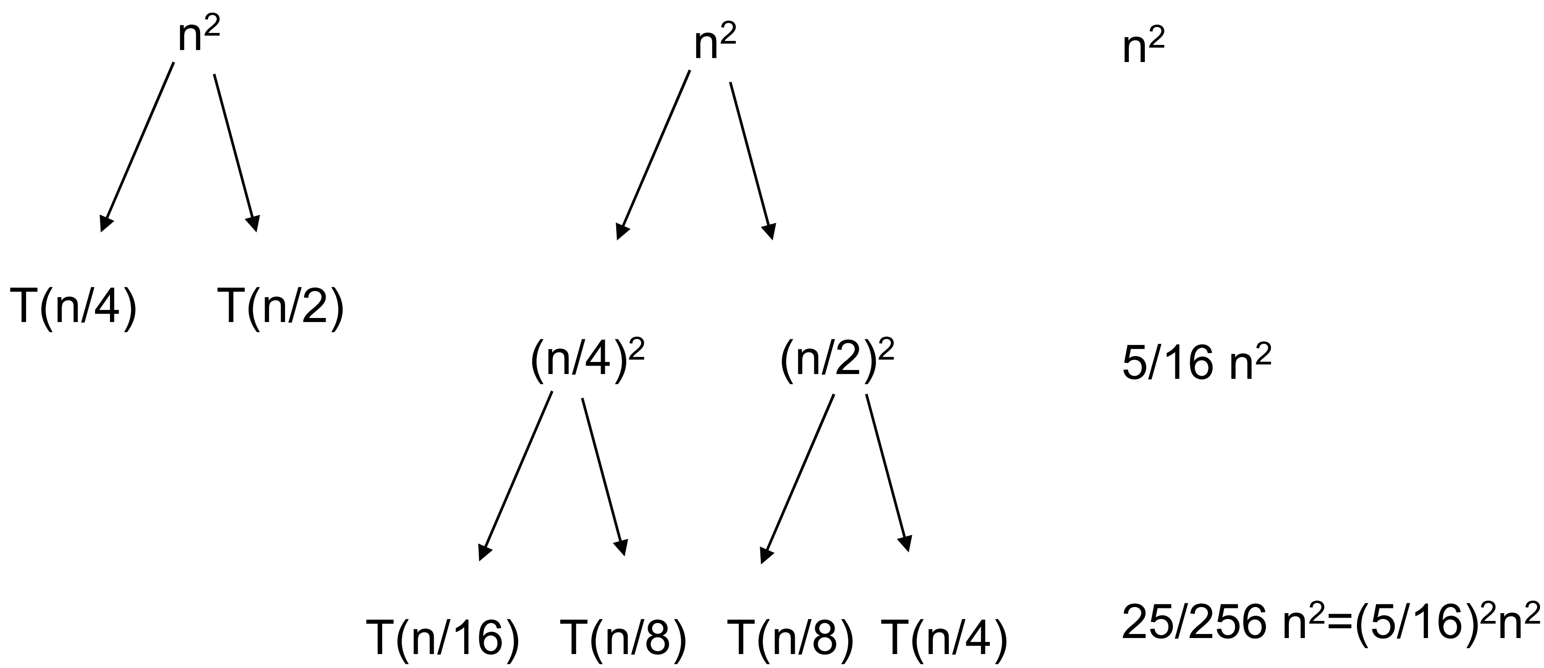
Recurrence: $T(n) = 4T(n/3) + n$

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

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 $\Theta(n^2)$

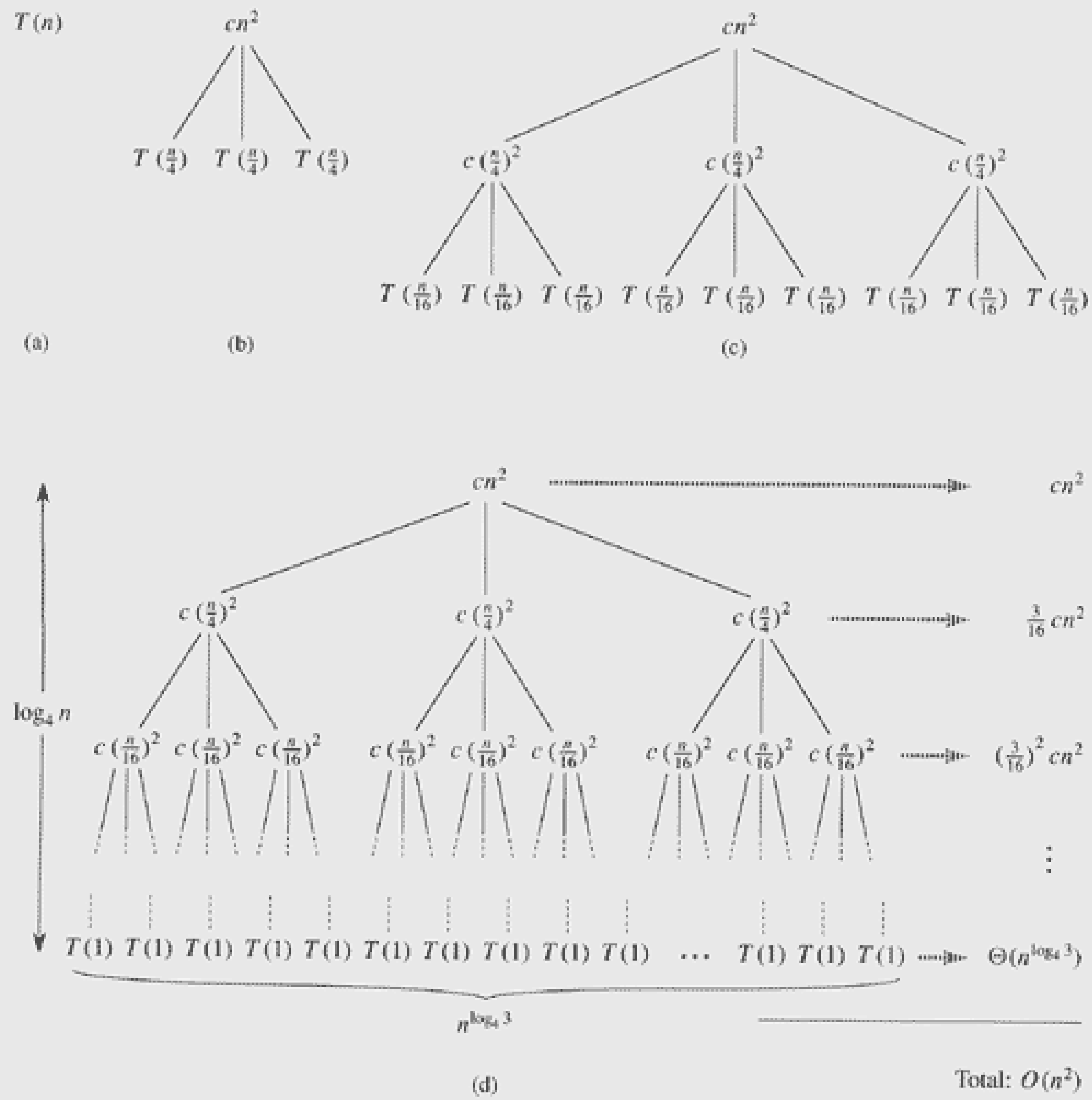


Figure 4.1 The construction of a recursion tree for the recurrence $T(n) = 3T(n/4) + cn^2$. Part **(a)** shows $T(n)$, which is progressively expanded in **(b)–(d)** to form the recursion tree. The fully expanded tree in part (d) has height $\log_4 n$ (it has $\log_4 n + 1$ levels).

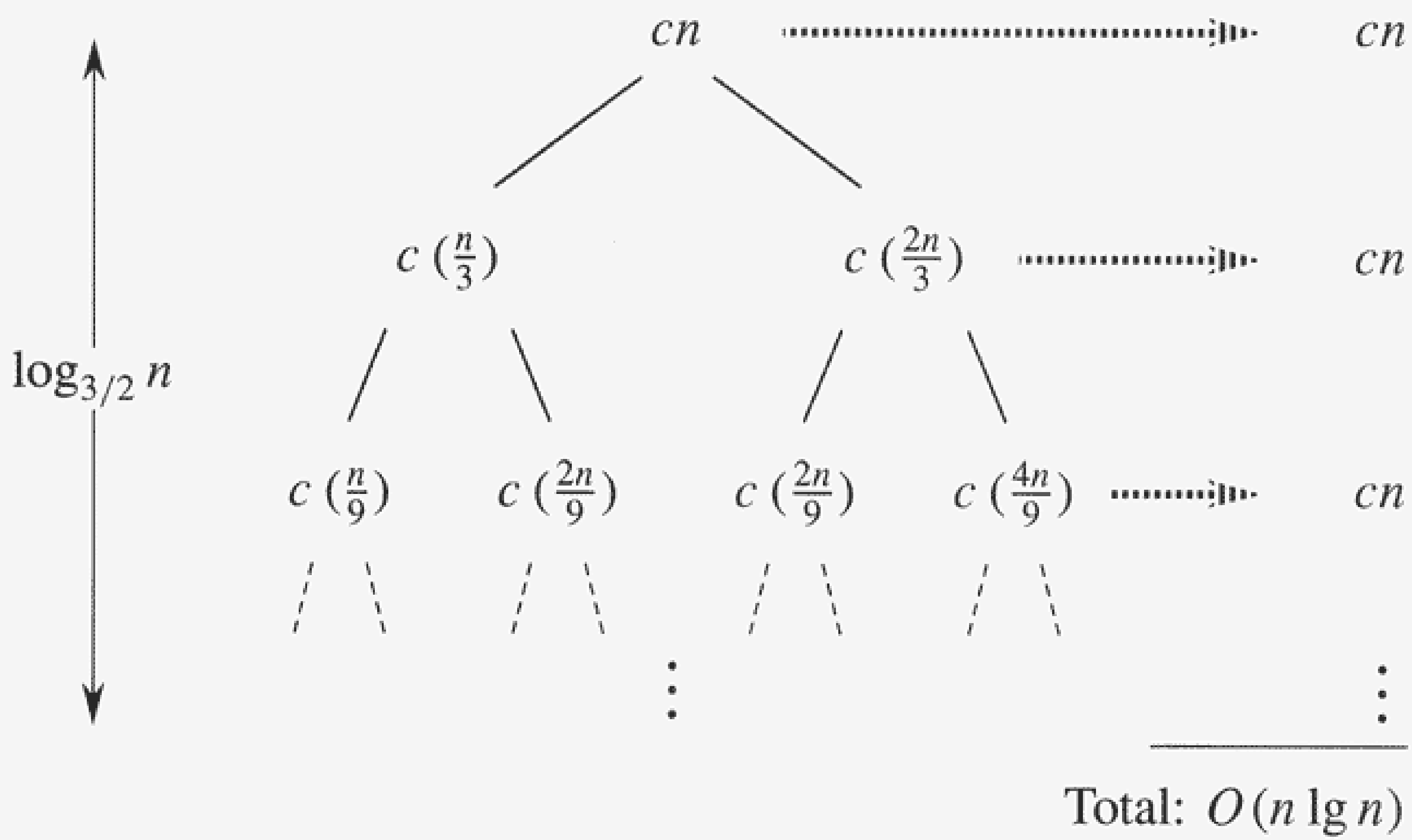


Figure 4.2 A recursion tree for the recurrence $T(n) = T(n/3) + T(2n/3) + cn$.

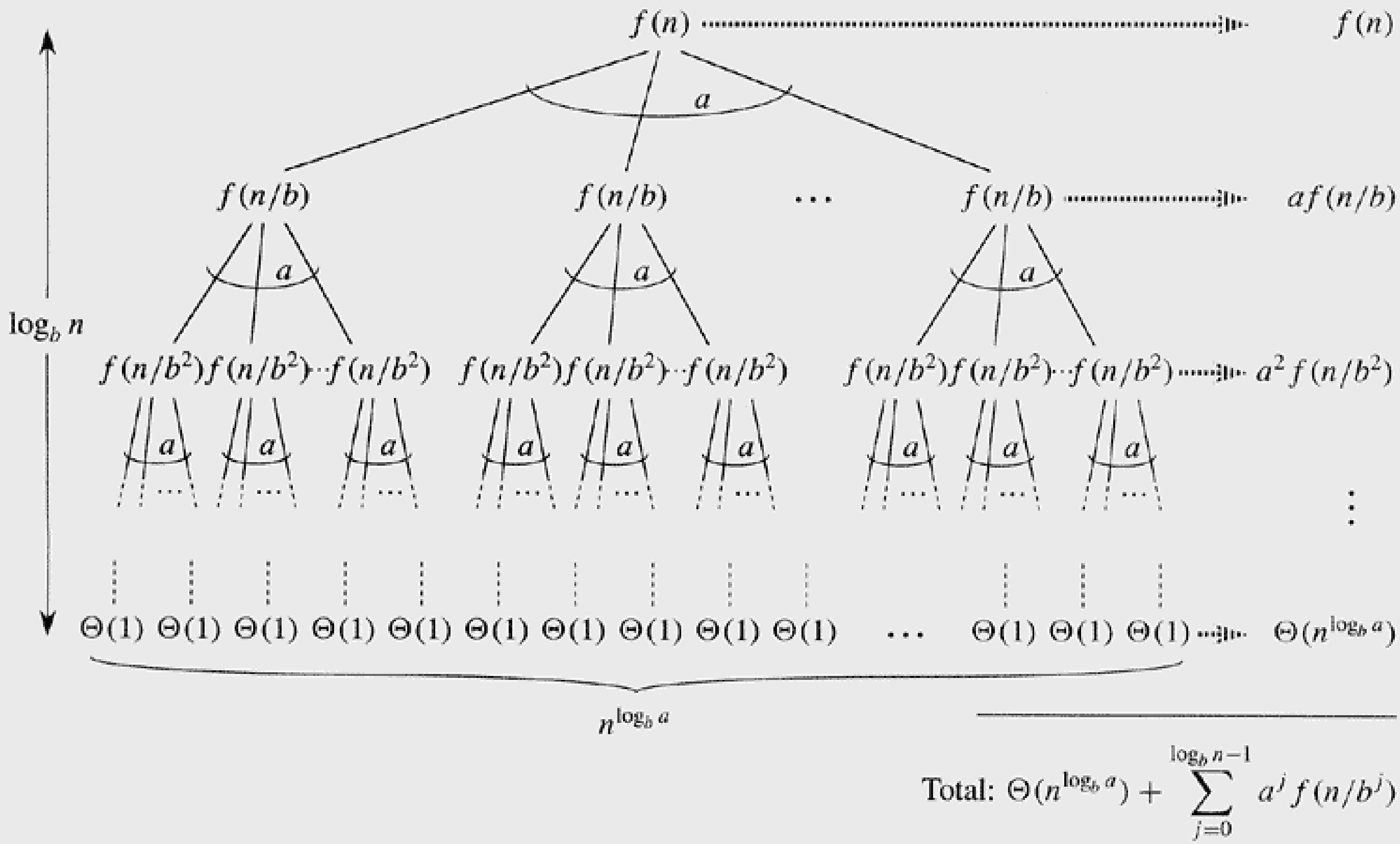


Figure 4.3 The recursion tree generated by $T(n) = aT(n/b) + f(n)$. The tree is a complete a-ary tree with $n^{\log_b a}$ leaves and height $\log_b n$. The cost of each level is shown at the right, and their sum is given in equation (4.6).

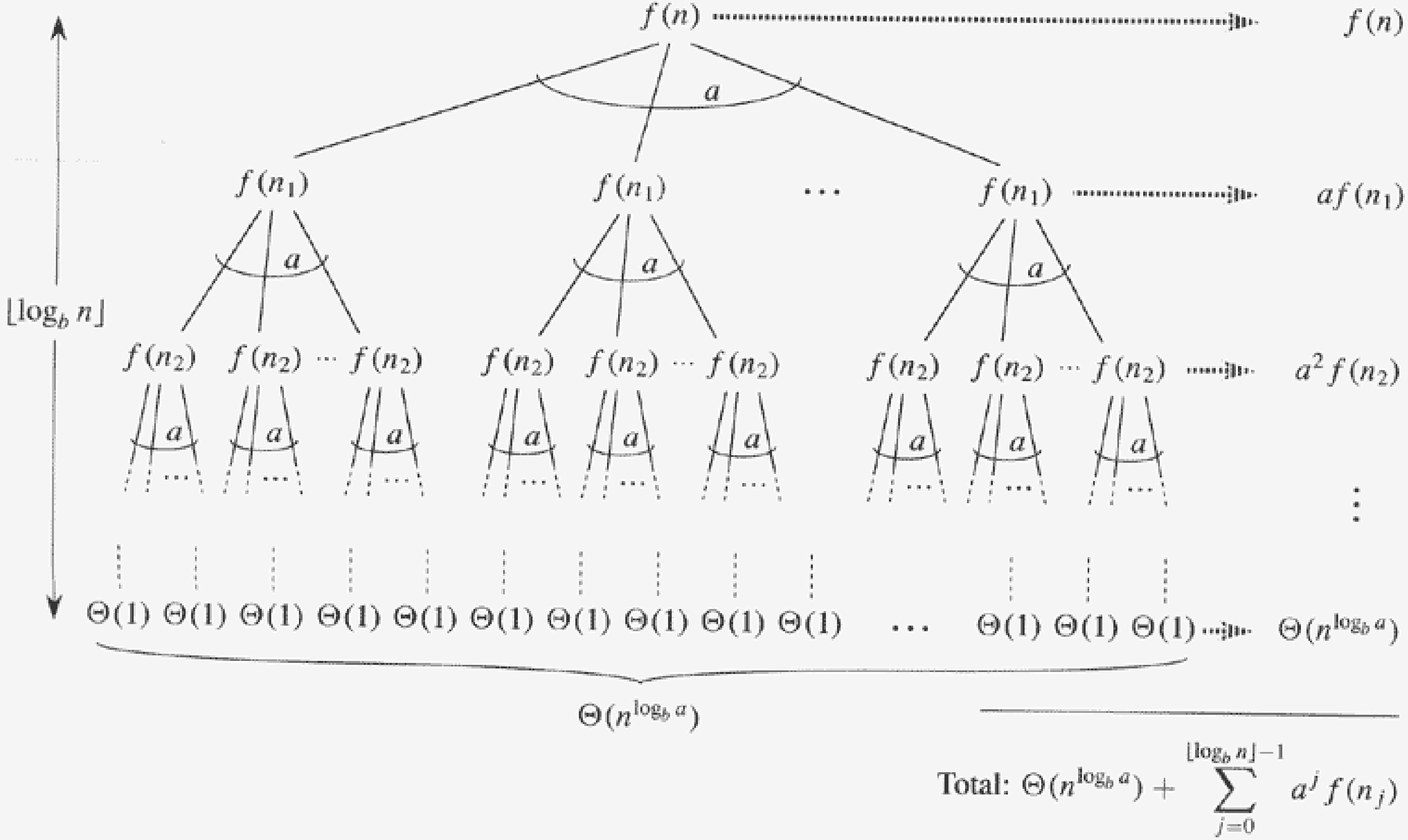


Figure 4.4 The recursion tree generated by $T(n) = aT(\lceil n/b \rceil) + f(n)$. The recursive argument n_j is given by equation (4.12).

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

n/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

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

$a = 2, b = 2$

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

- 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

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

$a = 2$, $b = 2$
$f(n) = \Theta(\text{n})$

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:

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

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 $af(n/b) \le cf(n)$ 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

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

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

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

$f(n) \overset{>}{\underset{<}{=}} n^{\log_b a}$

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

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

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

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

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

Here *f(n)* is polynomially larger 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

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

$a = 4$

$b = 2$

$f(n) = n$

$f(n) = n < n^{\log_2 4} = n^2$

$T(n) \in \Theta(n^2)$

Using the Master Method

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

$a = 64$

$b = 4$

$f(n) = n$ $<$ $n^{\log_b a} = n^3$

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

Using the Master Method

Using the master method, solve the recurrence

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

$a = 1$

$b = 3/2$

$f(n) = 1 =$

$n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$

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

Using the Master Method

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

Using the Master Method

Using the master method, solve the recurrence

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

$a = 1$

$b = 3$

$f(n) = n$ $>$ $n^{\log_b a} = 1$

$T(n) \in \Theta(n)$

Using the Master Method

② $T(n) = 4T(n/3) + n \lg n$ $\quad n^{\log_b a} = n^{\log_3 4}$

① $T(n) = 3T(n/4) + n \lg n$

$a = 3$

$b = 4$

$n^{\log_b a} = n^{\log_4 3}$

$f(n) = n \lg n$

$\lg n \; ? \; n^c$

$T(n) = n \lg n$

$n^1 \cdot n^{(\log_3 4 - 1)}$

c

$\cancel{n} \; \log_3 4$

$T(n) = n^{\log_3 4}$

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