Analysis of Algorithms

- Why to analyze algorithms?

Analysis of Algorithms

- Why to analyze algorithms?
  - When you have **several different algorithms to solve the same problem**, you have to decide which one is the **best suited** for your application.

Analysis of Algorithms

- Why to analyze algorithms?
  - When you have **several different algorithms to solve the same problem**, you have to decide which one is the **best suited** for your application.

  - Only after, you have determined the **efficiency of various algorithms**, you will be able to make a well-informed decision.

Analysis of Algorithms

- Why to analyze algorithms?
  - When you have **several different algorithms to solve the same problem**, you have to decide which one is the **best suited** for your application.
  - Only after, you have determined the **efficiency of various algorithms**, you will be able to make a well-informed decision.
  - But there is **no magic formula** for analyzing the efficiency of algorithms.

Analysis of Algorithms

- Why to analyze algorithms?
  - When you have **several different algorithms to solve the same problem**, you have to decide which one is the **best suited** for your application.
  - Only after, you have determined the **efficiency of various algorithms**, you will be able to make a well-informed decision.
  - But there is **no magic formula** for analyzing the efficiency of algorithms.
  - It is largely **a matter of judgment, intuition and experience.**

Analysis of Algorithms

- Why to analyze algorithms?
  - When you have **several different algorithms to solve the same problem**, you have to decide which one is the **best suited** for your application.
  - Only after, you have determined the **efficiency of various algorithms**, you will be able to make a well-informed decision.
  - But there is **no magic formula** for analyzing the efficiency of algorithms.
  - It is largely **a matter of judgment, intuition and experience.**
  - Nevertheless, there are some **basic techniques** that are often useful, **such as knowing how to deal with control structures and recurrence equations.**

Analysis of Algorithms

- Analyzing Sequencing
    - Sequencing

Analysis of Algorithms

- Analyzing Sequencing
  - Sequencing
    - Let P1 and P2 be two fragments of an algorithm. They may be single instructions or complicated sub algorithms.

Analysis of Algorithms

- Analyzing Sequencing
  - Sequencing
    - Let P1 and P2 be two fragments of an algorithm. They may be single instructions or complicated sub algorithms.
    - Let t1 and t2 be the time taken by P1 and P2, respectively. These times may depend on various parameters, such as the instance size.

Analysis of Algorithms

- Analyzing Sequencing
  - Sequencing
    - Let P1 and P2 be two fragments of an algorithm. They may be single instructions or complicated sub algorithms.
    - Let t1 and t2 be the time taken by P1 and P2, respectively. These times may depend on various parameters, such as the instance size.
    - The Sequencing Rule says that the time required to compute "P1;P2", that is first P1 and then P2, is simply t1 + t2.

Analysis of Algorithms

- Analyzing Sequencing
  - Sequencing
    - Let P1 and P2 be two fragments of an algorithm. They may be single instructions or complicated sub algorithms.
    - Let t1 and t2 be the time taken by P1 and P2, respectively. These times may depend on various parameters, such as the instance size.
    - The Sequencing Rule says that the time required to compute "P1;P2", that is first P1 and then P2, is simply t1 + t2.
  - Can we always consider P1 and P2 independent?

Analysis of Algorithms

- Analyzing Sequencing
  - Sequencing
    - Let P1 and P2 be two fragments of an algorithm. They may be single instructions or complicated sub algorithms.

    - Let t1 and t2 be the time taken by P1 and P2, respectively. These times may depend on various parameters, such as the instance size.

    - The Sequencing Rule says that the time required to compute "P1;P2", that is first P1 and then P2, is simply t1 + t2.

  - Can we always consider P1 and P2 independent?
    - It could happen that one of the parameters that control t2 depends on the result of the computation performed by P1. Thus, the analysis of "P1;P2" cannot always be performed by considering P1 and P2 independently.

Analysis of Algorithms

- Time complexity of a function (or set of statements) is considered as O(1) if it doesn't contain loop, recursion and call to any other non-constant time function.

- For example swap() function has O(1) time complexity.

Analysis of Algorithms

- A loop or recursion that runs a constant number of times is also considered as O(1).

- For example the following loop is O(1).

```
// Here c is a constant
  for (int i = 1; i <= c; i++) {
      // some O(1) expressions
  }
```

Analysis of Algorithms

- Time Complexity of a loop is considered as O(n) if the loop variables is incremented / decremented by a constant amount.

- For example following functions have O(n) time complexity.

```
 // Here c is a positive integer constant
  for (int i = 0; i < n; i += c) {
       // some O(1) expressions
  }

  for (int i = n; i > 0; i -= c) {
       // some O(1) expressions
  }
```

Analysis of Algorithms

- Time complexity of nested loops is equal to the number of times the innermost statement is executed.
- For example, the following sample loops have $O(n^2)$ time complexity
-
```
for (int i = 0; i <n; i += c) {
        for (int j = 0; j <n; j += c) {
              // some O(1) expressions
        }
    }
```

Analysis of Algorithms

- Analyzing Control Structure
  - Recursive Calls
    - As an example, consider the following recursive algorithm for Fibonacci sequence,

```
function Fibrec(n)
    if $n < 2$ then return $n$
    else return Fibrec($n - 1$)+Fibrec($n - 2$)
```

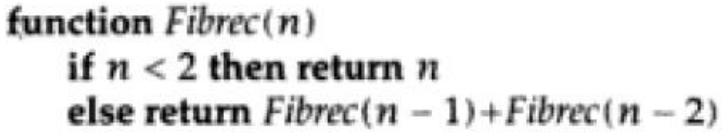
Analysis of Algorithms

- Analyzing Control Structure
  - Recursive Calls
    - Let T(n) be the time taken by a call on Fibrec(n).

```
function Fibrec(n)
    if n < 2 then return n
    else return Fibrec(n − 1)+Fibrec(n − 2)
```

Analysis of Algorithms

- Analyzing Control Structure
  - Recursive Calls



```
function Fibrec(n)
    if n < 2 then return n
    else return Fibrec(n - 1)+Fibrec(n - 2)
```

    - Let T(n) be the time taken by a call on Fibrec(n).
    - If n < 2, the algorithm simply returns n, which takes some constant time a. Otherwise, most of the work is spent in the two recursive calls, which take time T(n - 1) and T(n – 2), respectively.

Analysis of Algorithms

- Analyzing Control Structure
  - Recursive Calls

```
function Fibrec(n)
    if n < 2 then return n
    else return Fibrec(n - 1)+Fibrec(n - 2)
```

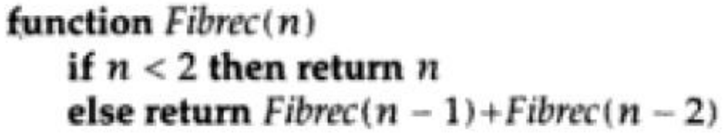
- Let T(n) be the time taken by a call on Fibrec(n).

- If n < 2, the algorithm simply returns n, which takes some constant time a. Otherwise, most of the work is spent in the two recursive calls, which take time T(n - 1) and T(n – 2), respectively.

- Moreover, one addition involving f_{n-1} and f_{n-2} (which are the values returned by the recursive calls) must be performed, in addition to the control of the recursion and the test "if n < 2".

Analysis of Algorithms

- Analyzing Control Structure
  - Recursive Calls



```
function Fibrec(n)
    if n < 2 then return n
    else return Fibrec(n - 1)+Fibrec(n - 2)
```

    - Let T(n) be the time taken by a call on Fibrec(n).

    - If n < 2, the algorithm simply returns n, which takes some constant time a. Otherwise, most of the work is spent in the two recursive calls, which take time T(n - 1) and T(n – 2), respectively.

    - Moreover, one addition involving f_{n-1} and f_{n-2} (which are the values returned by the recursive calls) must be performed, in addition to the control of the recursion and the test “if n < 2”.

    - Let h(n) stand for the work involved in this addition and control, that is time required by a call Fibrec(n) ignoring the time spent inside the two recursive calls.

Analysis of Algorithms

- Analyzing Control Structure
  - Recursive Calls

```
function Fibrec(n)
    if $n < 2$ then return $n$
    else return $Fibrec(n-1) + Fibrec(n-2)$
```

- Let T(n) be the time taken by a call on Fibrec(n).

- If n < 2, the algorithm simply returns n, which takes some constant time a. Otherwise, most of the work is spent in the two recursive calls, which take time T(n - 1) and T(n – 2), respectively.

- Moreover, one addition involving f_{n-1} and f_{n-2} (which are the values returned by the recursive calls) must be performed, in addition to the control of the recursion and the test "if n < 2".

- Let h(n) stand for the work involved in this addition and control, that is time required by a call Fibrec(n) ignoring the time spent inside the two recursive calls.

- By definition of T(n) and h(n), we obtain the following recurrence.

$$T(n) = \begin{cases} a & \text{if } n = 0 \text{ or } n = 1 \\ T(n-1) + T(n-2) + h(n) & \text{otherwise} \end{cases}$$

Analysis of Algorithms

- What is a recurrence or a recurrence equation?
  - A recurrence or recurrence equation is an equation or inequality that describes a function in terms of its value on smaller inputs.

  OR

  - A recurrence or recurrence equation is an equation or inequality that describes running time of a function in terms of running time of a function on smaller inputs .

Analysis of Algorithms

- Solving Recurrences
- Homogeneous Linear Recurrence with Constant Coefficients

$$a_0 t_n + a_1 t_{n-1} + \cdots + a_k t_{n-k} = 0$$

- *linear* because it does not contain terms of the form $t_{n-i}t_{n-j}$, t_{n-i}^2, and so on;
- *homogeneous* because the linear combination of the t_{n-i} is equal to zero; and
- *with constant coefficients* because the a_i are constants.

This equation of degree k in x is called the *characteristic equation* of the recurrence

$$p(x) = a_0 x^k + a_1 x^{k-1} + \cdots + a_k$$

- k roots

$$t_n = \sum_{i=1}^{k} c_i r_i^n \qquad\qquad t_n = \sum_{i=1}^{\ell} \sum_{j=0}^{m_i-1} c_{ij} n^j r_i^n$$

satisfies the recurrence for any choice of constants $c_1, c_2, \ldots, c_k$.

Analysis of Algorithms

- Solving Recurrences
- Homogeneous Recurrences

$$a_0 t_n + a_1 t_{n-1} + \cdots + a_k t_{n-k} = 0$$

$$p(x) = a_0 x^k + a_1 x^{k-1} + \cdots + a_k$$

Example 4.7.1. (Fibonacci) Consider the recurrence

$$f_n = \begin{cases} n & \text{if } n = 0 \text{ or } n = 1 \\ f_{n-1} + f_{n-2} & \text{otherwise} \end{cases}$$

First we rewrite this recurrence to fit the mould of Equation 4.7.

$$f_n - f_{n-1} - f_{n-2} = 0$$

The characteristic polynomial is

$$x^2 - x - 1$$

whose roots are

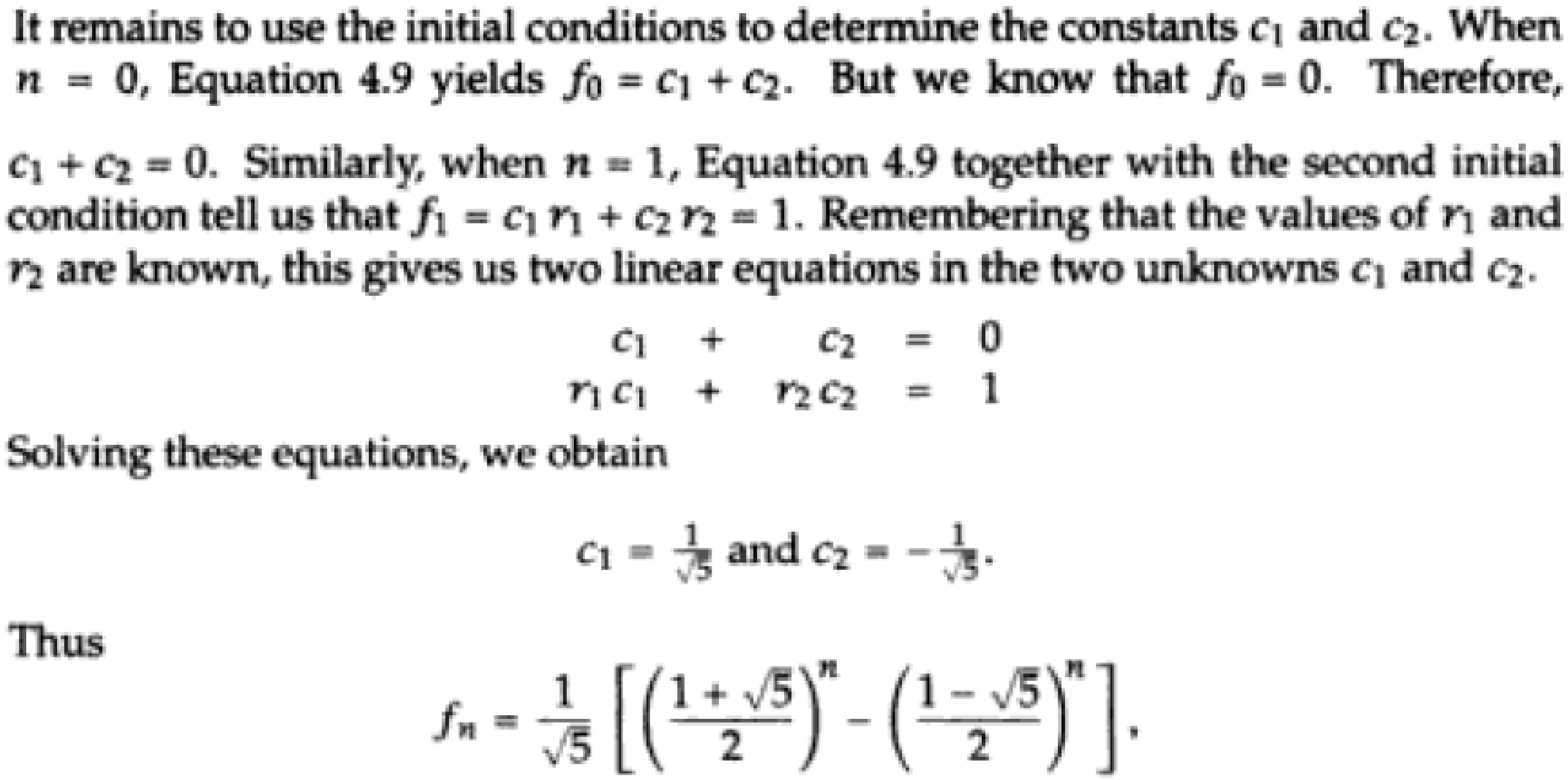
$$r_1 = \frac{1+\sqrt{5}}{2} \text{ and } r_2 = \frac{1-\sqrt{5}}{2}.$$

The general solution is therefore of the form

$$f_n = c_1 r_1^n + c_2 r_2^n. \qquad (4.9)$$

Analysis of Algorithms

- Solving Recurrences
- Homogeneous Recurrences



It remains to use the initial conditions to determine the constants c_1 and c_2. When $n = 0$, Equation 4.9 yields $f_0 = c_1 + c_2$. But we know that $f_0 = 0$. Therefore, $c_1 + c_2 = 0$. Similarly, when $n = 1$, Equation 4.9 together with the second initial condition tell us that $f_1 = c_1 r_1 + c_2 r_2 = 1$. Remembering that the values of r_1 and r_2 are known, this gives us two linear equations in the two unknowns c_1 and c_2.

$$\begin{array}{rcrcl} c_1 & + & c_2 & = & 0 \\ r_1 c_1 & + & r_2 c_2 & = & 1 \end{array}$$

Solving these equations, we obtain

$$c_1 = \tfrac{1}{\sqrt{5}} \text{ and } c_2 = -\tfrac{1}{\sqrt{5}}.$$

Thus

$$f_n = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right].$$

Analysis of Algorithms

- Solving Recurrences
- Homogeneous Recurrences

$$a_0 t_n + a_1 t_{n-1} + \cdots + a_k t_{n-k} = 0$$

$$p(x) = a_0 x^k + a_1 x^{k-1} + \cdots + a_k$$

Example 4.7.2. Consider the recurrence

$$t_n = \begin{cases} 0 & \text{if } n = 0 \\ 5 & \text{if } n = 1 \\ 3t_{n-1} + 4t_{n-2} & \text{otherwise} \end{cases}$$

First we rewrite the recurrence.

$$t_n - 3t_{n-1} - 4t_{n-2} = 0$$

The characteristic polynomial is

$$x^2 - 3x - 4 = (x+1)(x-4)$$

whose roots are $r_1 = -1$ and $r_2 = 4$. The general solution is therefore of the form

$$t_n = c_1(-1)^n + c_2 4^n.$$

The initial conditions give

$$\begin{array}{rcrcrl} c_1 & + & c_2 & = & 0 & \quad n = 0 \\ -c_1 & + & 4c_2 & = & 5 & \quad n = 1 \end{array}$$

Analysis of Algorithms

- Solving Recurrences
- Homogeneous Recurrences

Solving these equations, we obtain $c_1 = -1$ and $c_2 = 1$. Therefore

$$t_n = 4^n - (-1)^n.$$

Analysis of Algorithms

- Solving Recurrences
- Homogeneous Recurrences

$$a_0 t_n + a_1 t_{n-1} + \cdots + a_k t_{n-k} = 0$$

$$p(x) = a_0 x^k + a_1 x^{k-1} + \cdots + a_k$$

Example 4.7.3. Consider the recurrence

$$t_n = \begin{cases} n & \text{if } n = 0, 1 \text{ or } 2 \\ 5t_{n-1} - 8t_{n-2} + 4t_{n-3} & \text{otherwise} \end{cases}$$

First we rewrite the recurrence.

$$t_n - 5t_{n-1} + 8t_{n-2} - 4t_{n-3} = 0$$

The characteristic polynomial is

$$x^3 - 5x^2 + 8x - 4 = (x-1)(x-2)^2.$$

The roots are therefore $r_1 = 1$ of multiplicity $m_1 = 1$ and $r_2 = 2$ of multiplicity $m_2 = 2$, and the general solution is

$$t_n = c_1 1^n + c_2 2^n + c_3 n 2^n.$$

The initial conditions give

Analysis of Algorithms

- Solving Recurrences
- Homogeneous Recurrences

$$\begin{array}{ccccccccc} c_1 & + & c_2 & & & = & 0 & & n=0 \\ c_1 & + & 2c_2 & + & 2c_3 & = & 1 & & n=1 \\ c_1 & + & 4c_2 & + & 8c_3 & = & 2 & & n=2 \end{array}$$

Solving these equations, we obtain $c_1 = -2$, $c_2 = 2$ and $c_3 = -\frac{1}{2}$. Therefore

$$t_n = 2^{n+1} - n2^{n-1} - 2.$$

Analysis of Algorithms

- Solving Recurrences
- Inhomogeneous Recurrences

$$a_0 t_n + a_1 t_{n-1} + \cdots + a_k t_{n-k} = b^n p(n)$$

- b is a constant; and
- $p(n)$ is a polynomial in n of degree d.

  - Corresponding characteristic polynomial is

$$(a_0 x^k + a_1 x^{k-1} + \cdots + a_k)(x-b)^{d+1}$$

  - More general Form:

$$a_0 t_n + a_1 t_{n-1} + \cdots + a_k t_{n-k} = b_1^n p_1(n) + b_2^n p_2(n) + \cdots$$

  - Corresponding characteristic polynomial is

$$(a_0 x^k + a_1 x^{k-1} + \cdots + a_k)(x-b_1)^{d_1+1}(x-b_2)^{d_2+1}\cdots,$$

Analysis of Algorithms

- Solving Recurrences
- Inhomogeneous Recurrences

$$a_0 t_n + a_1 t_{n-1} + \cdots + a_k t_{n-k} = b^n p(n)$$

$$(a_0 x^k + a_1 x^{k-1} + \cdots + a_k)(x-b)^{d+1}$$

Example 4.7.4. Consider the recurrence

$$t_n - 2t_{n-1} = 3^n. \tag{4.11}$$

In this case, $b = 3$ and $p(n) = 1$, a polynomial of degree 0. A little cunning allows us to reduce this example to the homogeneous case with which we are now familiar. To see this, first multiply the recurrence by 3, obtaining

$$3t_n - 6t_{n-1} = 3^{n+1}.$$

Now replace n by $n-1$ in this recurrence to get

$$3t_{n-1} - 6t_{n-2} = 3^n. \tag{4.12}$$

Finally, subtract Equation 4.12 from 4.11 to obtain

$$t_n - 5t_{n-1} + 6t_{n-2} = 0, \tag{4.13}$$

which can be solved by the method of Section 4.7.2. The characteristic polynomial is

$$x^2 - 5x + 6 = (x-2)(x-3)$$

and therefore all solutions are of the form

$$t_n = c_1 2^n + c_2 3^n. \tag{4.14}$$

Analysis of Algorithms

- Solving Recurrences
- Inhomogeneous Recurrences

Equations 4.11and 4.13 are *not* equivalent: Equation 4.13 can be solved given arbitrary values for t_0 and t_1 (the initial conditions), whereas our original Equation 4.11 implies that $t_1 = 2t_0 + 3$.

$$\begin{array}{rclr} c_1 + c_2 & = & t_0 & n = 0 \\ 2c_1 + 3c_2 & = & 2t_0 + 3 & n = 1 \end{array} \qquad (4.15)$$

Solving these, we obtain $c_1 = t_0 - 3$ and $c_2 = 3$. Therefore, the general solution is

$$t_n = (t_0 - 3)\,2^n + 3^{n+1}$$

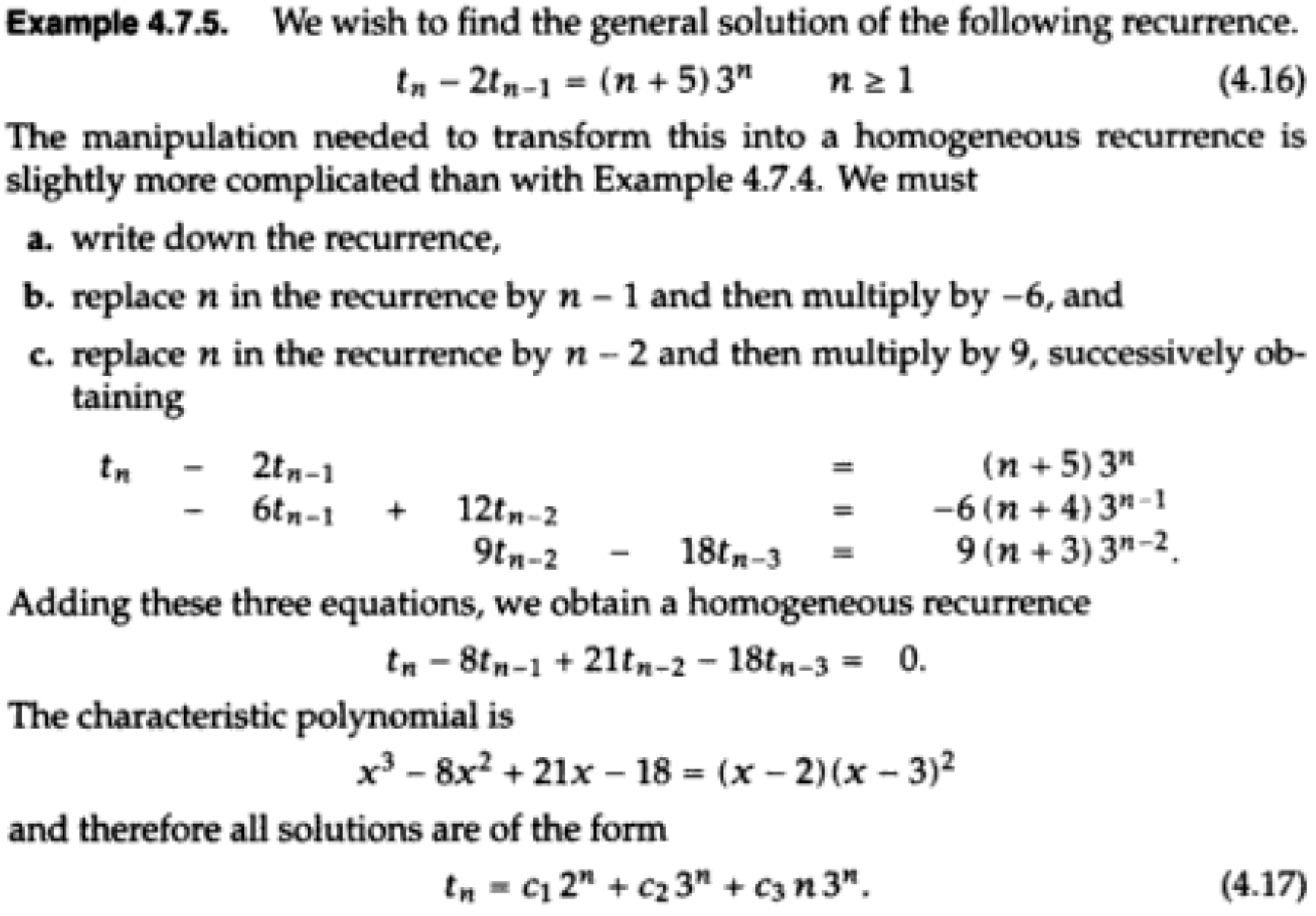
and thus $t_n \in \Theta(3^n)$ regardless of the initial condition.

Analysis of Algorithms

- Solving Recurrences
- Inhomogeneous Recurrences

$$a_0 t_n + a_1 t_{n-1} + \cdots + a_k t_{n-k} = b^n p(n)$$

$$(a_0 x^k + a_1 x^{k-1} + \cdots + a_k)(x-b)^{d+1}$$



Example 4.7.5. We wish to find the general solution of the following recurrence.

$$t_n - 2t_{n-1} = (n+5)3^n \qquad n \ge 1 \tag{4.16}$$

The manipulation needed to transform this into a homogeneous recurrence is slightly more complicated than with Example 4.7.4. We must

a. write down the recurrence,

b. replace n in the recurrence by $n-1$ and then multiply by -6, and

c. replace n in the recurrence by $n-2$ and then multiply by 9, successively obtaining

$$\begin{array}{cccccccc} t_n & - & 2t_{n-1} & & & & = & (n+5)3^n \\ & - & 6t_{n-1} & + & 12t_{n-2} & & = & -6(n+4)3^{n-1} \\ & & & & 9t_{n-2} & - \; 18t_{n-3} & = & 9(n+3)3^{n-2}. \end{array}$$

Adding these three equations, we obtain a homogeneous recurrence

$$t_n - 8t_{n-1} + 21t_{n-2} - 18t_{n-3} = 0.$$

The characteristic polynomial is

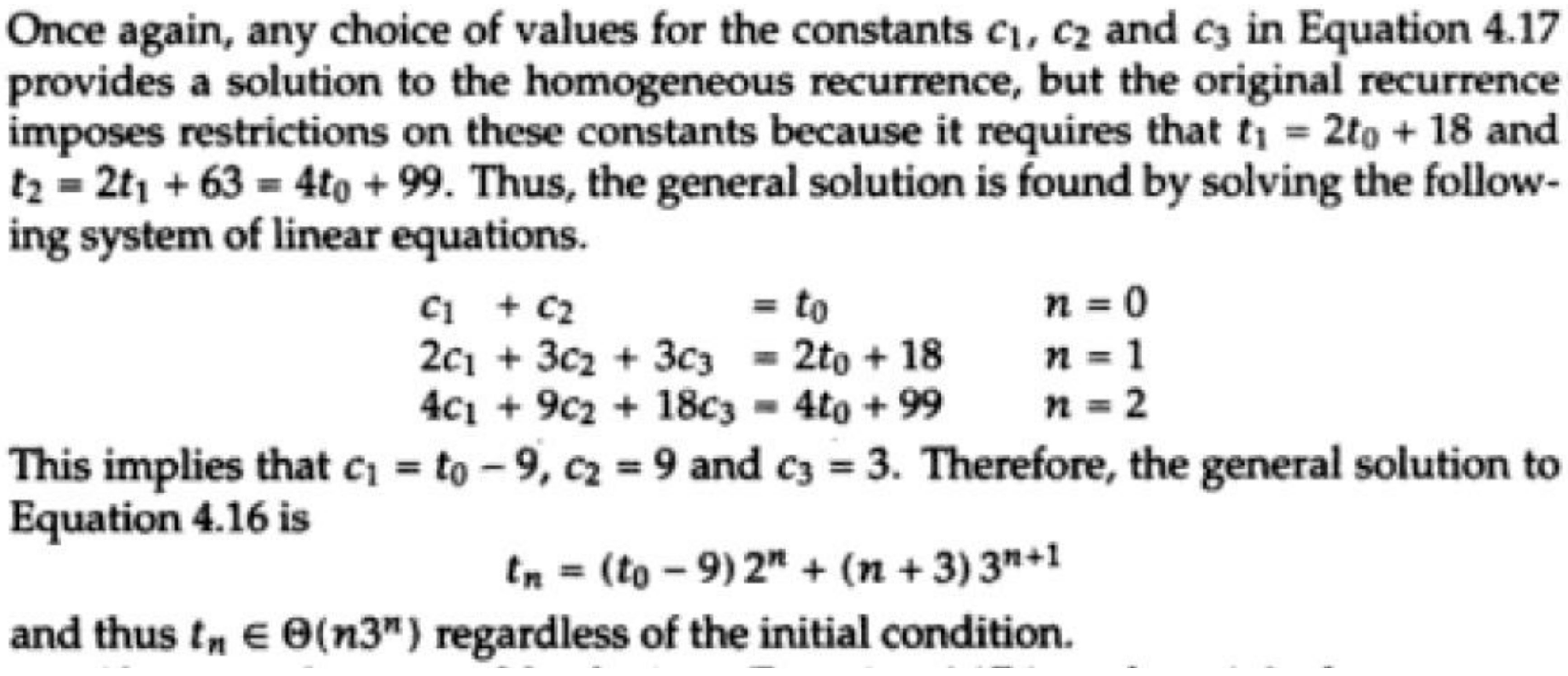
$$x^3 - 8x^2 + 21x - 18 = (x-2)(x-3)^2$$

and therefore all solutions are of the form

$$t_n = c_1 2^n + c_2 3^n + c_3 n 3^n. \tag{4.17}$$

Analysis of Algorithms

- Solving Recurrences
- Inhomogeneous Recurrences



Once again, any choice of values for the constants c_1, c_2 and c_3 in Equation 4.17 provides a solution to the homogeneous recurrence, but the original recurrence imposes restrictions on these constants because it requires that $t_1 = 2t_0 + 18$ and $t_2 = 2t_1 + 63 = 4t_0 + 99$. Thus, the general solution is found by solving the following system of linear equations.

$$\begin{array}{llll} c_1 + c_2 & = t_0 & & n = 0 \\ 2c_1 + 3c_2 + 3c_3 & = 2t_0 + 18 & & n = 1 \\ 4c_1 + 9c_2 + 18c_3 & = 4t_0 + 99 & & n = 2 \end{array}$$

This implies that $c_1 = t_0 - 9$, $c_2 = 9$ and $c_3 = 3$. Therefore, the general solution to Equation 4.16 is

$$t_n = (t_0 - 9)\,2^n + (n + 3)\,3^{n+1}$$

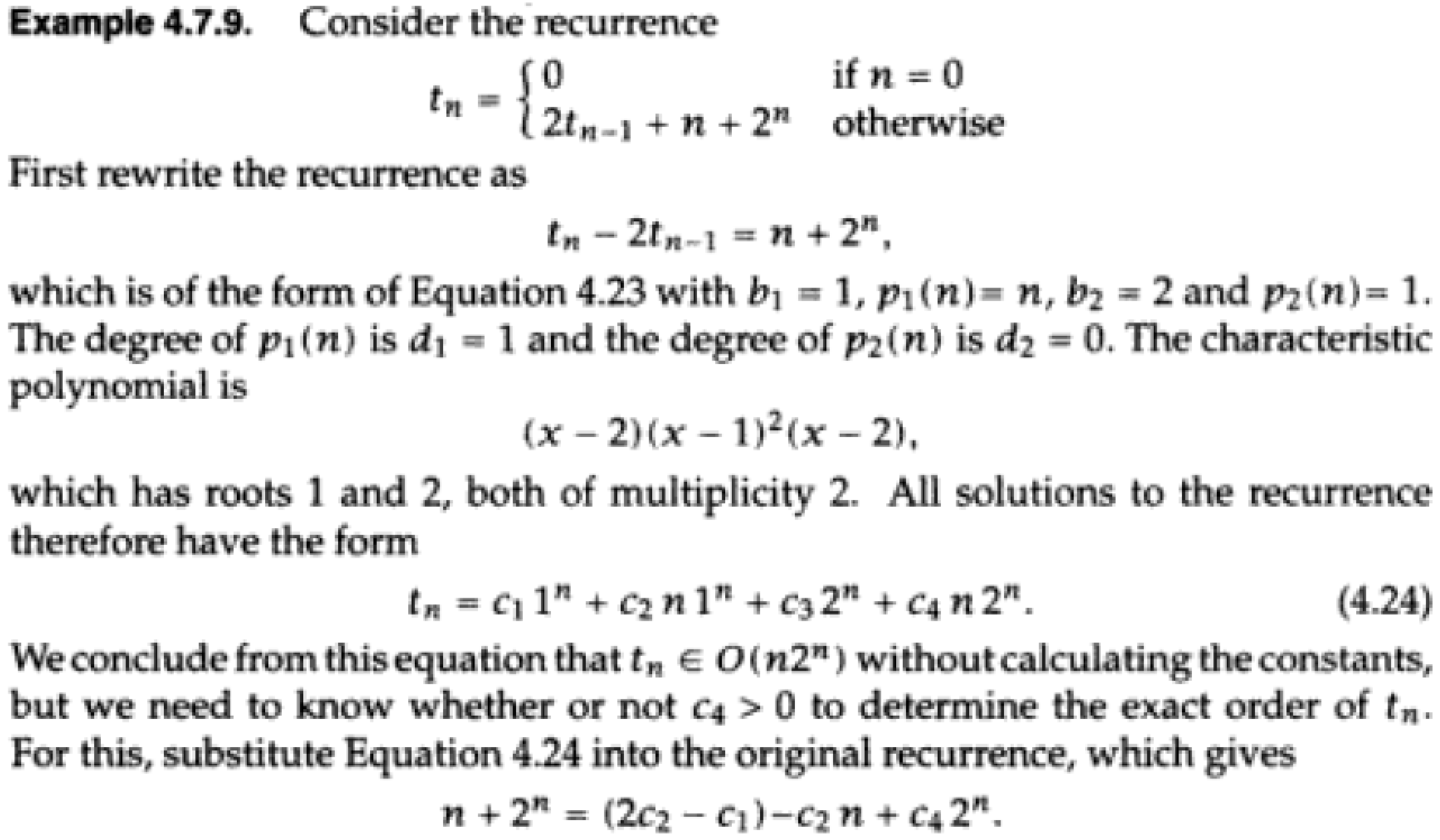
and thus $t_n \in \Theta(n3^n)$ regardless of the initial condition.

Analysis of Algorithms

- Solving Recurrences
- Inhomogeneous Recurrences

$$a_0 t_n + a_1 t_{n-1} + \cdots + a_k t_{n-k} = b^n p(n)$$

$$(a_0 x^k + a_1 x^{k-1} + \cdots + a_k)(x - b)^{d+1}$$



Example 4.7.9. Consider the recurrence

$$t_n = \begin{cases} 0 & \text{if } n = 0 \\ 2t_{n-1} + n + 2^n & \text{otherwise} \end{cases}$$

First rewrite the recurrence as

$$t_n - 2t_{n-1} = n + 2^n,$$

which is of the form of Equation 4.23 with $b_1 = 1$, $p_1(n) = n$, $b_2 = 2$ and $p_2(n) = 1$. The degree of $p_1(n)$ is $d_1 = 1$ and the degree of $p_2(n)$ is $d_2 = 0$. The characteristic polynomial is

$$(x-2)(x-1)^2(x-2),$$

which has roots 1 and 2, both of multiplicity 2. All solutions to the recurrence therefore have the form

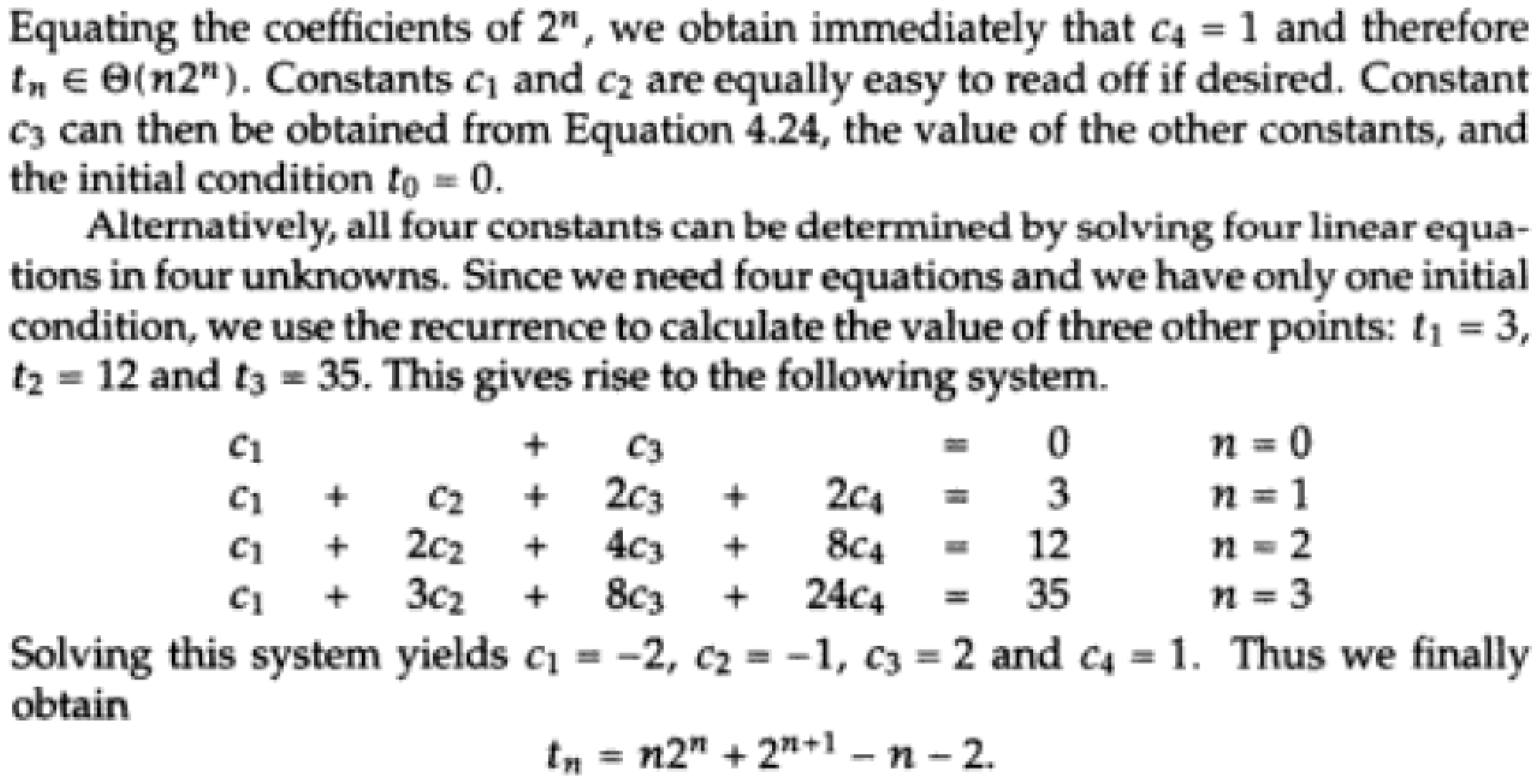
$$t_n = c_1 1^n + c_2 n 1^n + c_3 2^n + c_4 n 2^n. \quad (4.24)$$

We conclude from this equation that $t_n \in O(n2^n)$ without calculating the constants, but we need to know whether or not $c_4 > 0$ to determine the exact order of t_n. For this, substitute Equation 4.24 into the original recurrence, which gives

$$n + 2^n = (2c_2 - c_1) - c_2 n + c_4 2^n.$$

Analysis of Algorithms

- Solving Recurrences
- Inhomogeneous Recurrences



Equating the coefficients of 2^n, we obtain immediately that $c_4 = 1$ and therefore $t_n \in \Theta(n2^n)$. Constants c_1 and c_2 are equally easy to read off if desired. Constant c_3 can then be obtained from Equation 4.24, the value of the other constants, and the initial condition $t_0 = 0$.

Alternatively, all four constants can be determined by solving four linear equations in four unknowns. Since we need four equations and we have only one initial condition, we use the recurrence to calculate the value of three other points: $t_1 = 3$, $t_2 = 12$ and $t_3 = 35$. This gives rise to the following system.

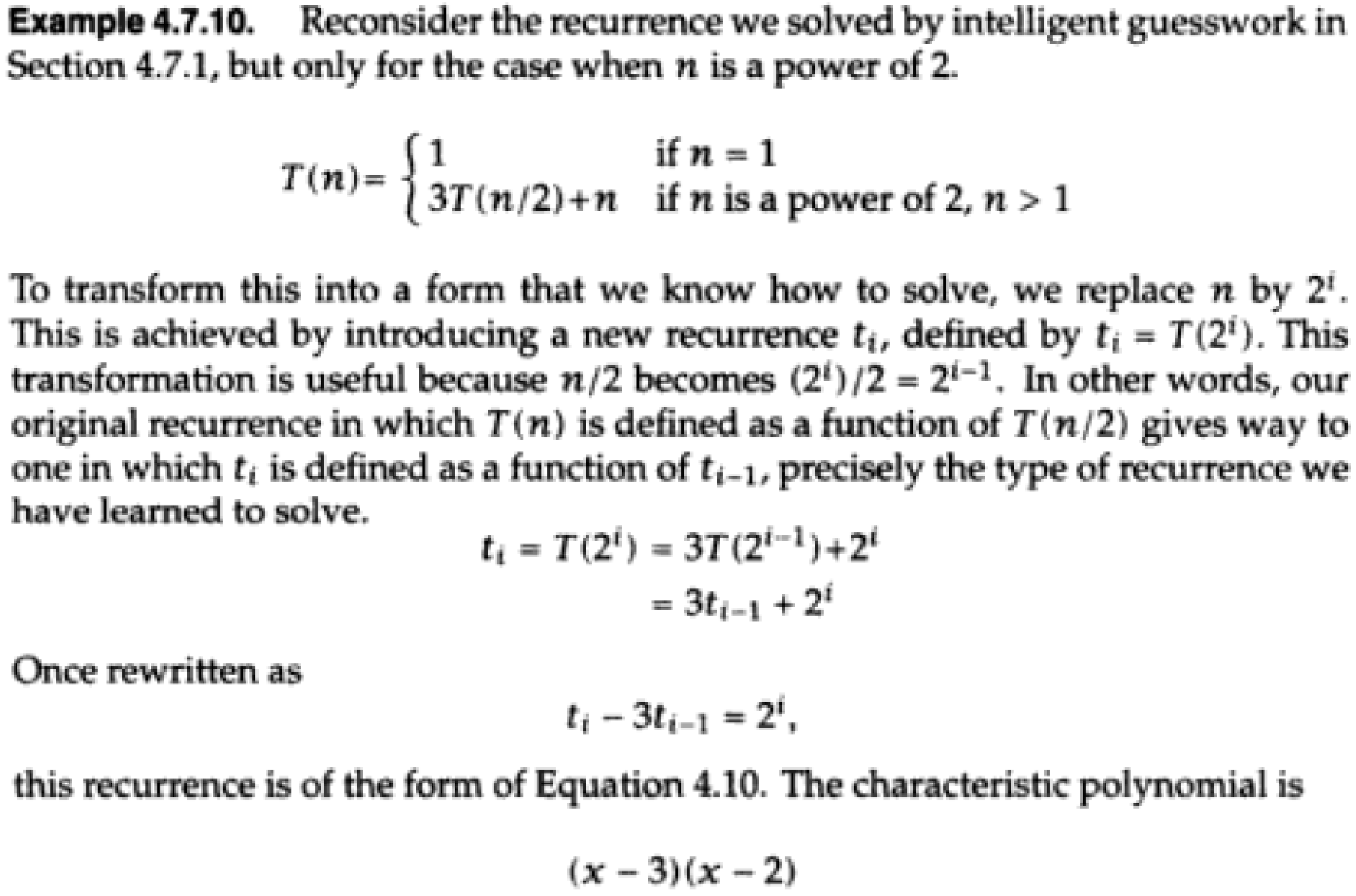
$$\begin{array}{ccccccccccc}
c_1 & & & + & c_3 & & & = & 0 & & n=0 \\
c_1 & + & c_2 & + & 2c_3 & + & 2c_4 & = & 3 & & n=1 \\
c_1 & + & 2c_2 & + & 4c_3 & + & 8c_4 & = & 12 & & n=2 \\
c_1 & + & 3c_2 & + & 8c_3 & + & 24c_4 & = & 35 & & n=3
\end{array}$$

Solving this system yields $c_1 = -2$, $c_2 = -1$, $c_3 = 2$ and $c_4 = 1$. Thus we finally obtain

$$t_n = n2^n + 2^{n+1} - n - 2.$$

Analysis of Algorithms

- Solving Recurrences
- Change of Variable



Example 4.7.10. Reconsider the recurrence we solved by intelligent guesswork in Section 4.7.1, but only for the case when n is a power of 2.

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 3T(n/2) + n & \text{if } n \text{ is a power of 2, } n > 1 \end{cases}$$

To transform this into a form that we know how to solve, we replace n by 2^i. This is achieved by introducing a new recurrence t_i, defined by $t_i = T(2^i)$. This transformation is useful because $n/2$ becomes $(2^i)/2 = 2^{i-1}$. In other words, our original recurrence in which $T(n)$ is defined as a function of $T(n/2)$ gives way to one in which t_i is defined as a function of t_{i-1}, precisely the type of recurrence we have learned to solve.

$$\begin{aligned} t_i = T(2^i) &= 3T(2^{i-1}) + 2^i \\ &= 3t_{i-1} + 2^i \end{aligned}$$

Once rewritten as

$$t_i - 3t_{i-1} = 2^i,$$

this recurrence is of the form of Equation 4.10. The characteristic polynomial is

$$(x-3)(x-2)$$

Analysis of Algorithms

- Solving Recurrences
- Change of Variable

and hence all solutions for t_i are of the form

$$t_i = c_1 3^i + c_2 2^i.$$

We use the fact that $T(2^i) = t_i$ and thus $T(n) = t_{\lg n}$ when $n = 2^i$ to obtain

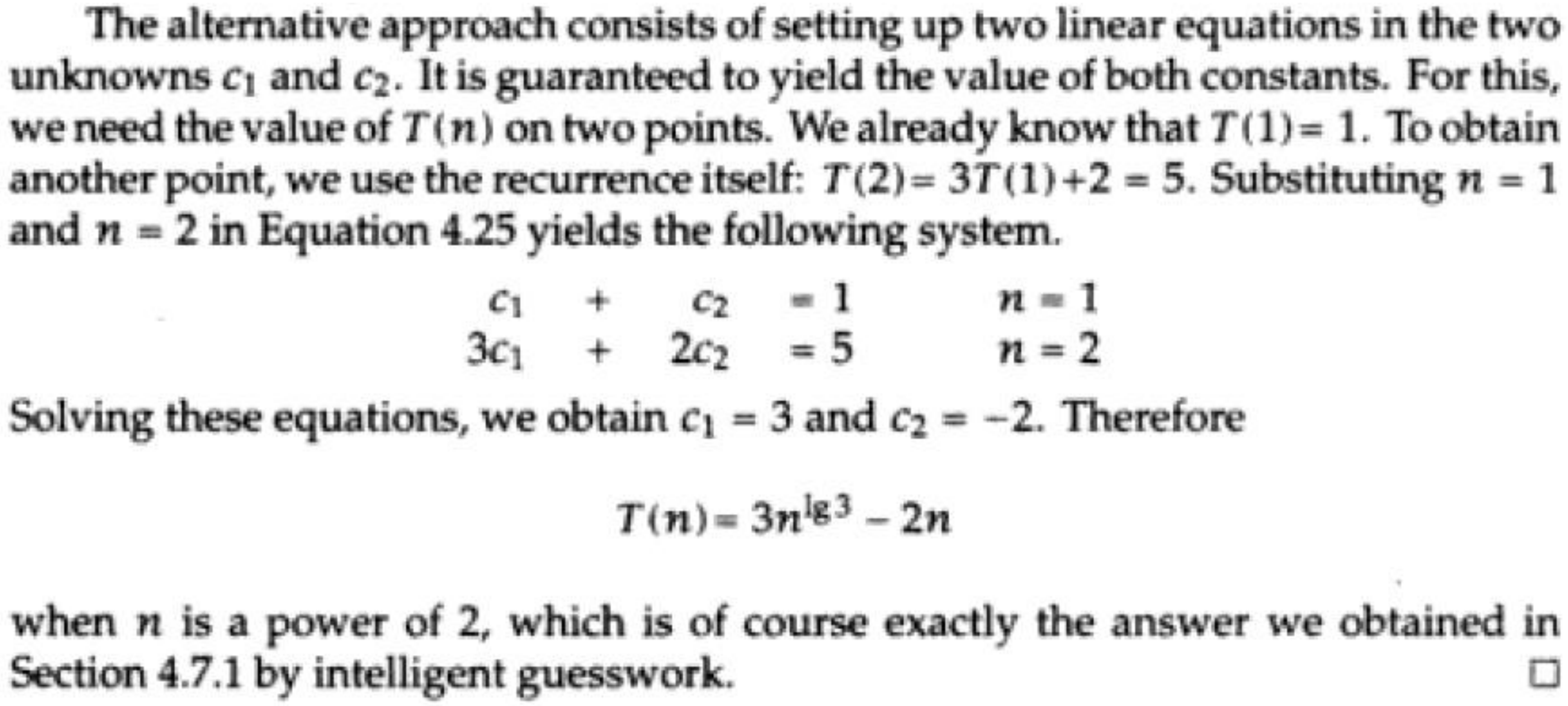
$$\begin{aligned} T(n) &= c_1 3^{\lg n} + c_2 2^{\lg n} \\ &= c_1 n^{\lg 3} + c_2 n \end{aligned} \qquad (4.25)$$

when n is a power of 2, which is sufficient to conclude that

$$T(n) \in O(n^{\lg 3} \mid n \text{ is a power of } 2).$$

Analysis of Algorithms

- Solving Recurrences
- Change of Variable



The alternative approach consists of setting up two linear equations in the two unknowns c_1 and c_2. It is guaranteed to yield the value of both constants. For this, we need the value of $T(n)$ on two points. We already know that $T(1) = 1$. To obtain another point, we use the recurrence itself: $T(2) = 3T(1)+2 = 5$. Substituting $n = 1$ and $n = 2$ in Equation 4.25 yields the following system.

$$\begin{array}{lclcll} c_1 & + & c_2 & = 1 & \qquad & n = 1 \\ 3c_1 & + & 2c_2 & = 5 & \qquad & n = 2 \end{array}$$

Solving these equations, we obtain $c_1 = 3$ and $c_2 = -2$. Therefore

$$T(n) = 3n^{\lg 3} - 2n$$

when n is a power of 2, which is of course exactly the answer we obtained in Section 4.7.1 by intelligent guesswork. □

Analysis of Algorithms

- Solving Recurrences
- Change of Variable

Example 4.7.11. Consider the recurrence

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

when n is a power of 2, $n \geq 2$. We proceed as in the previous example.

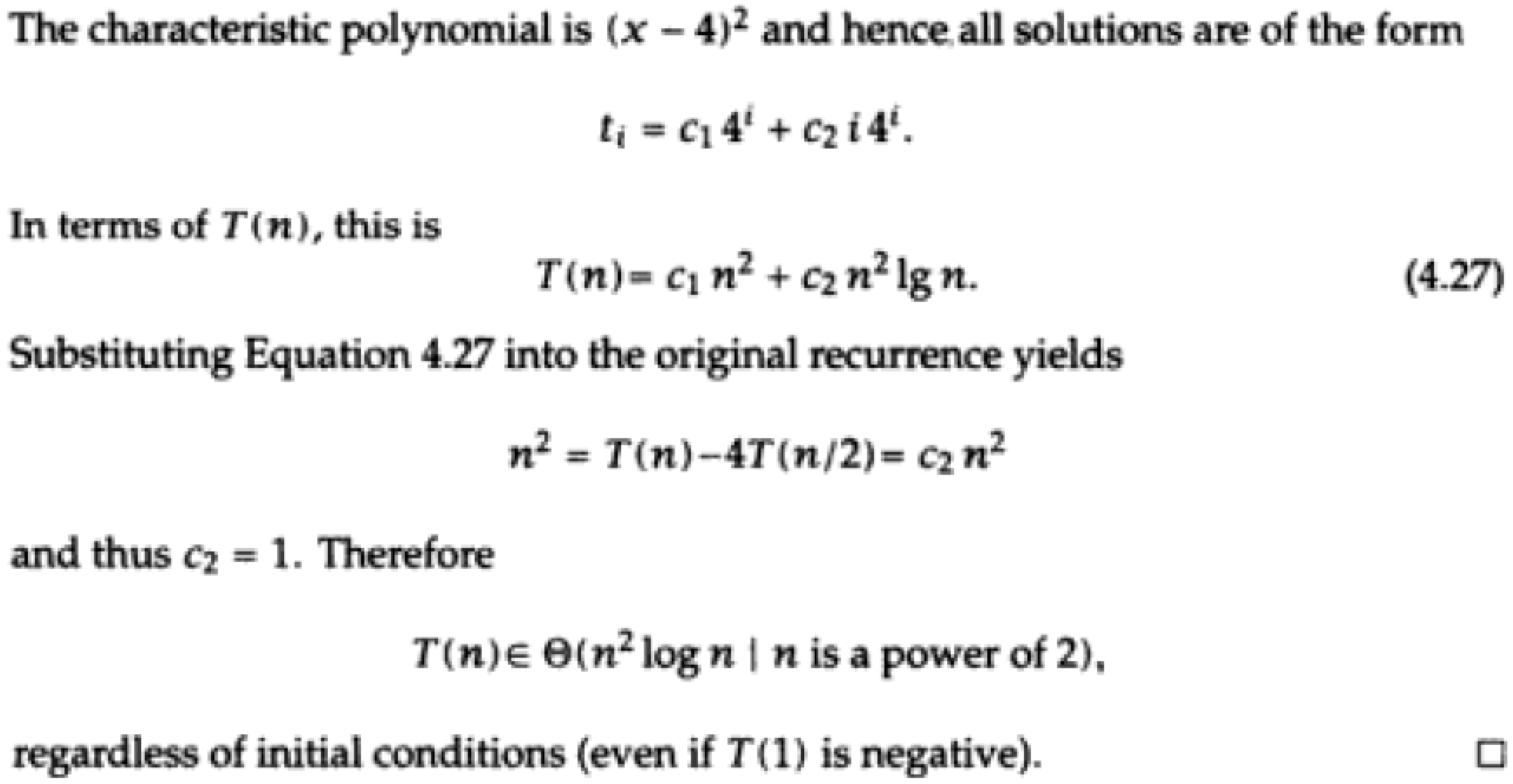
$$\begin{aligned} t_i = T(2^i) &= 4T(2^{i-1}) + (2^i)^2 \\ &= 4t_{i-1} + 4^i \end{aligned}$$

We rewrite this in the form of Equation 4.10.

$$t_i - 4t_{i-1} = 4^i$$

Analysis of Algorithms

- Solving Recurrences
- Change of Variable



The characteristic polynomial is $(x-4)^2$ and hence all solutions are of the form

$$t_i = c_1 4^i + c_2 i 4^i.$$

In terms of $T(n)$, this is

$$T(n) = c_1 n^2 + c_2 n^2 \lg n. \tag{4.27}$$

Substituting Equation 4.27 into the original recurrence yields

$$n^2 = T(n) - 4T(n/2) = c_2 n^2$$

and thus $c_2 = 1$. Therefore

$$T(n) \in \Theta(n^2 \log n \mid n \text{ is a power of } 2),$$

regardless of initial conditions (even if $T(1)$ is negative). □

Analysis of Algorithms

- Solving Recurrences
- Change of Variable

Example 4.7.12. Consider the recurrence

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

when n is a power of 2, $n \geq 2$. As before, we obtain

$$\begin{aligned} t_i = T(2^i) &= 2T(2^{i-1}) + i\,2^i \\ &= 2t_{i-1} + i\,2^i \end{aligned}$$

We rewrite this in the form of Equation 4.10.

$$t_i - 2t_{i-1} = i\,2^i$$

The characteristic polynomial is $(x-2)(x-2)^2 = (x-2)^3$ and hence all solutions are of the form

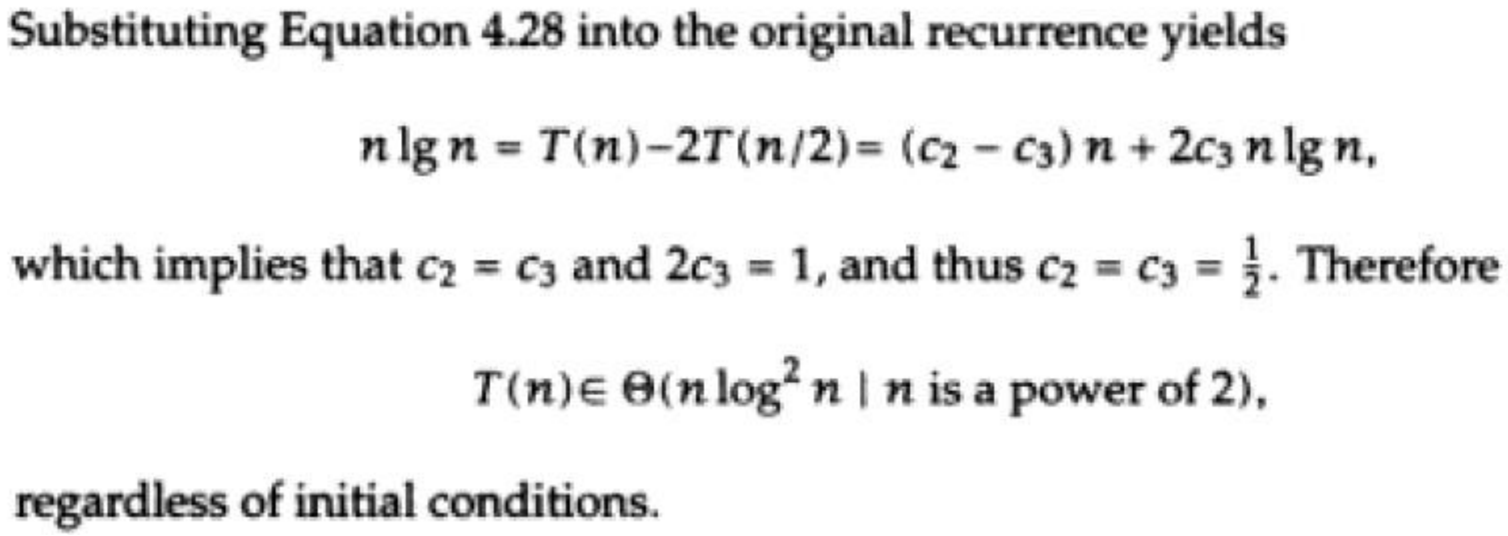
$$t_i = c_1 2^i + c_2\, i\, 2^i + c_3\, i^2 2^i.$$

In terms of $T(n)$, this is

$$T(n) = c_1 n + c_2\, n \lg n + c_3\, n \lg^2 n. \qquad (4.28)$$

Analysis of Algorithms

- Solving Recurrences
- Change of Variable



Substituting Equation 4.28 into the original recurrence yields

$$n \lg n = T(n) - 2T(n/2) = (c_2 - c_3)\, n + 2c_3\, n \lg n,$$

which implies that $c_2 = c_3$ and $2c_3 = 1$, and thus $c_2 = c_3 = \frac{1}{2}$. Therefore

$$T(n) \in \Theta(n \log^2 n \mid n \text{ is a power of } 2),$$

regardless of initial conditions.

Analysis of Algorithms

- Solving Recurrences
- Master Method

The master method provides a "cookbook" method for solving recurrences of the form

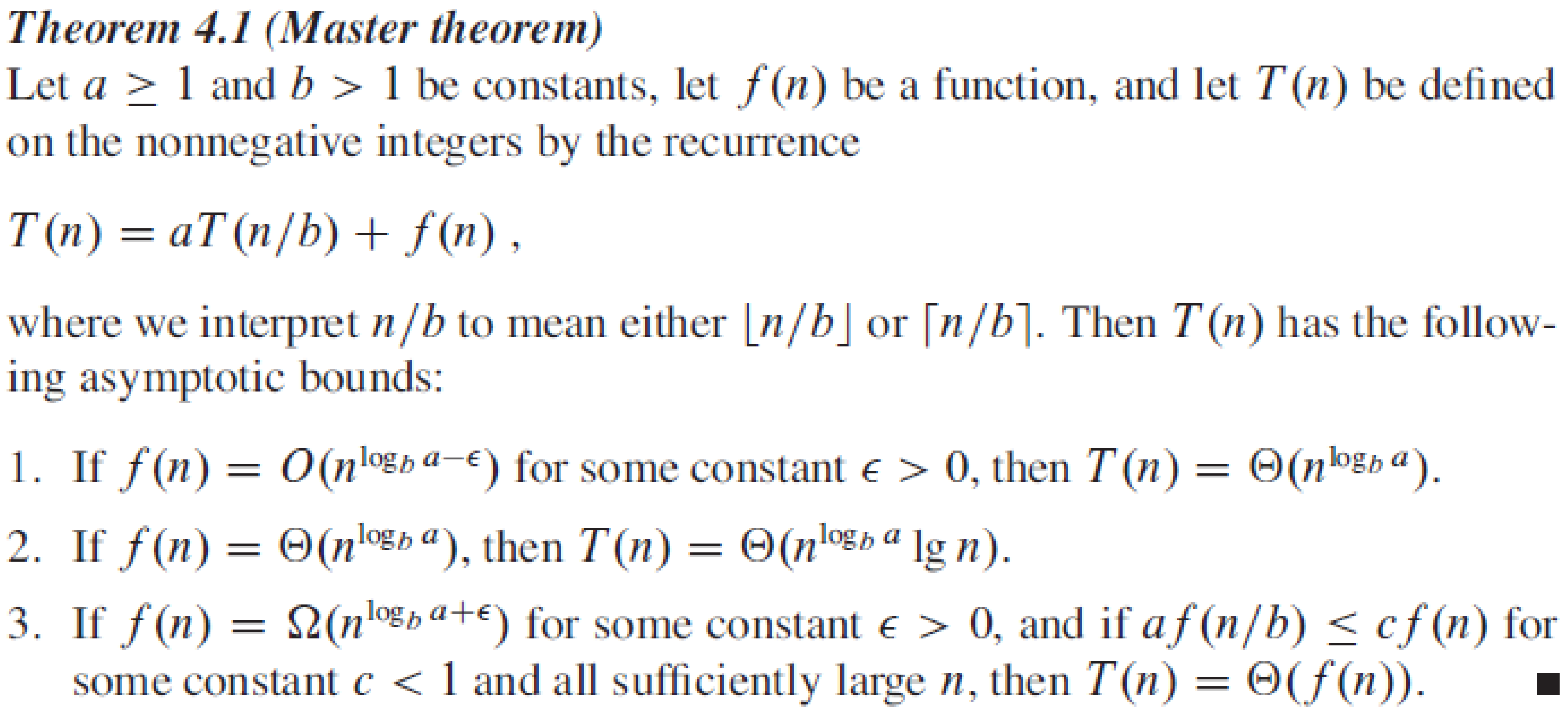
$$T(n) = aT(n/b) + f(n) , \tag{4.20}$$

where $a \geq 1$ and $b > 1$ are constants and $f(n)$ is an asymptotically positive

The recurrence (4.20) describes the running time of an algorithm that divides a problem of size n into a subproblems, each of size n/b, where a and b are positive constants. The a subproblems are solved recursively, each in time $T(n/b)$. The function $f(n)$ encompasses the cost of dividing the problem and combining the results of the subproblems.

Analysis of Algorithms

- Solving Recurrences
- Master Method



Theorem 4.1 (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

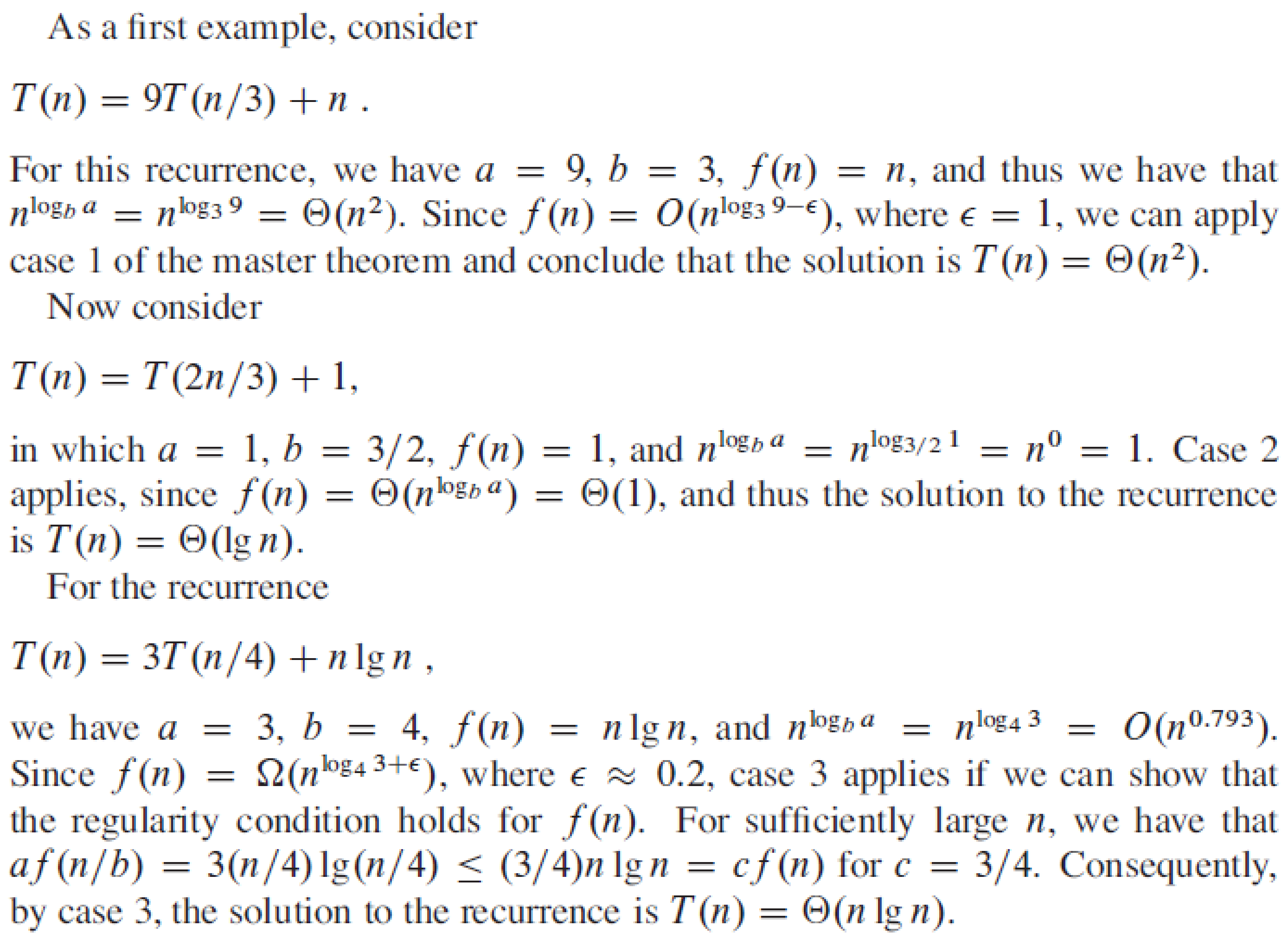
$$T(n) = aT(n/b) + f(n) ,$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, 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 $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

Analysis of Algorithms

- Solving Recurrences
- Using the Master Method



As a first example, consider

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

For this recurrence, we have $a = 9$, $b = 3$, $f(n) = n$, and thus we have that $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$. Since $f(n) = O(n^{\log_3 9 - \epsilon})$, where $\epsilon = 1$, we can apply case 1 of the master theorem and conclude that the solution is $T(n) = \Theta(n^2)$.

Now consider

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

in which $a = 1$, $b = 3/2$, $f(n) = 1$, and $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$. Case 2 applies, since $f(n) = \Theta(n^{\log_b a}) = \Theta(1)$, and thus the solution to the recurrence is $T(n) = \Theta(\lg n)$.

For the recurrence

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

we have $a = 3$, $b = 4$, $f(n) = n \lg n$, and $n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$. Since $f(n) = \Omega(n^{\log_4 3 + \epsilon})$, where $\epsilon \approx 0.2$, case 3 applies if we can show that the regularity condition holds for $f(n)$. 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 $c = 3/4$. Consequently, by case 3, the solution to the recurrence is $T(n) = \Theta(n \lg n)$.

Analysis of Algorithms

- Solving Recurrences
- Using the Master Method

The master method does not apply to the recurrence

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

even though it appears to have the proper form: $a = 2$, $b = 2$, $f(n) = n \lg n$, and $n^{\log_b a} = n$. You might mistakenly think that case 3 should apply, since $f(n) = n \lg n$ is asymptotically larger than $n^{\log_b a} = n$. The problem is that it is not *polynomially* larger. The ratio $f(n)/n^{\log_b a} = (n \lg n)/n = \lg n$ is asymptotically less than n^{ϵ} for any positive constant ϵ. Consequently, the recurrence falls into the gap between case 2 and case 3. (See Exercise 4.6-2 for a solution.)

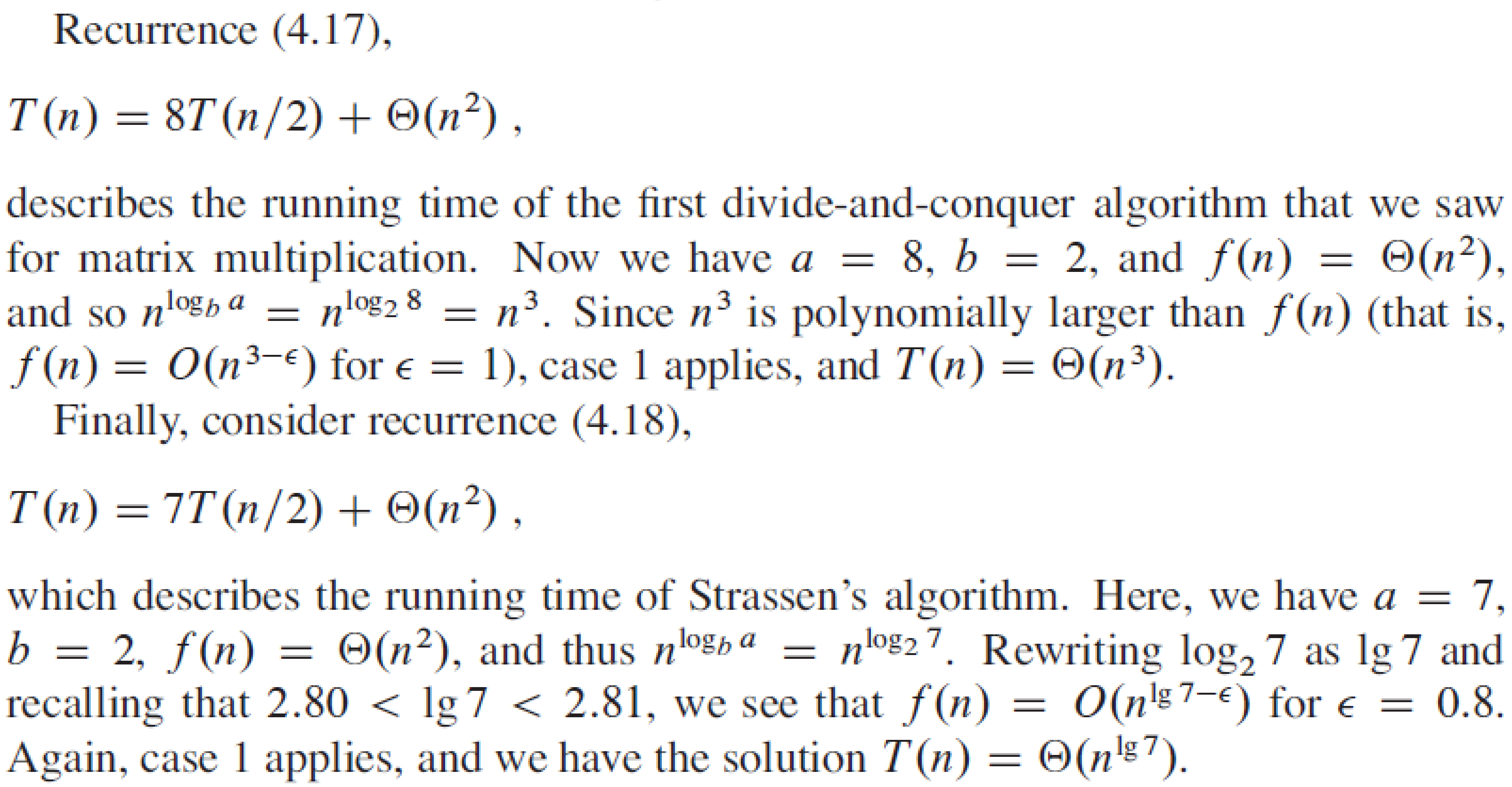
Let's use the master method to solve the recurrences we saw in Sections 4.1 and 4.2. Recurrence (4.7),

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

characterizes the running times of the divide-and-conquer algorithm for both the maximum-subarray problem and merge sort. (As is our practice, we omit stating the base case in the recurrence.) Here, we have $a = 2$, $b = 2$, $f(n) = \Theta(n)$, and thus we have that $n^{\log_b a} = n^{\log_2 2} = n$. Case 2 applies, since $f(n) = \Theta(n)$, and so we have the solution $T(n) = \Theta(n \lg n)$.

Analysis of Algorithms

- Solving Recurrences
- Using the Master Method



Recurrence (4.17),

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

describes the running time of the first divide-and-conquer algorithm that we saw for matrix multiplication. Now we have $a = 8$, $b = 2$, and $f(n) = \Theta(n^2)$, and so $n^{\log_b a} = n^{\log_2 8} = n^3$. Since n^3 is polynomially larger than $f(n)$ (that is, $f(n) = O(n^{3-\epsilon})$ for $\epsilon = 1$), case 1 applies, and $T(n) = \Theta(n^3)$.

Finally, consider recurrence (4.18),

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

which describes the running time of Strassen's algorithm. Here, we have $a = 7$, $b = 2$, $f(n) = \Theta(n^2)$, and thus $n^{\log_b a} = n^{\log_2 7}$. Rewriting $\log_2 7$ as $\lg 7$ and recalling that $2.80 < \lg 7 < 2.81$, we see that $f(n) = O(n^{\lg 7-\epsilon})$ for $\epsilon = 0.8$. Again, case 1 applies, and we have the solution $T(n) = \Theta(n^{\lg 7})$.

Analysis of Algorithms

- Solving Recurrences
- Range Transformations

Example 4.7.15. Consider the following recurrence, which defines $T(n)$ when n is a power of 2.

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

The first step is a change of variable: let t_i denote $T(2^i)$.

$$\begin{aligned} t_i = T(2^i) &= 2^i\,T^2(2^{i-1}) \\ &= 2^i\,t_{i-1}^2 \end{aligned}$$

At first glance, none of the techniques we have seen applies to this recurrence since it is not linear; furthermore the coefficient 2^i is not a constant. To transform the range, we create yet another recurrence by using u_i to denote $\lg t_i$.

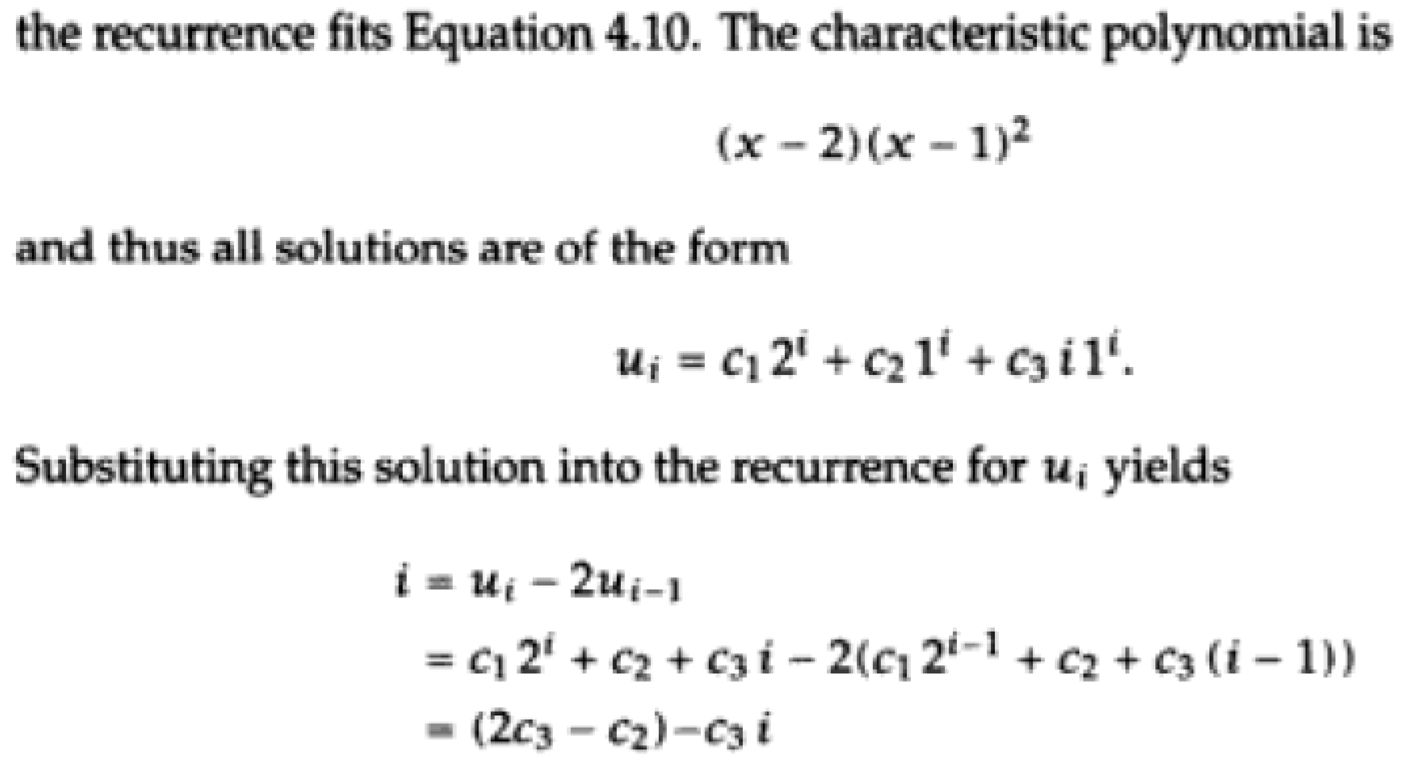
$$\begin{aligned} u_i = \lg t_i &= i + 2\lg t_{i-1} \\ &= i + 2u_{i-1} \end{aligned}$$

This time, once rewritten as

$$u_i - 2u_{i-1} = i,$$

Analysis of Algorithms

- Solving Recurrences
- Range Transformations



the recurrence fits Equation 4.10. The characteristic polynomial is

$$(x-2)(x-1)^2$$

and thus all solutions are of the form

$$u_i = c_1 2^i + c_2 1^i + c_3 i 1^i.$$

Substituting this solution into the recurrence for u_i yields

$$\begin{aligned} i &= u_i - 2u_{i-1} \\ &= c_1 2^i + c_2 + c_3 i - 2(c_1 2^{i-1} + c_2 + c_3 (i-1)) \\ &= (2c_3 - c_2) - c_3 i \end{aligned}$$

Analysis of Algorithms

- Solving Recurrences
- Range Transformations

and thus $c_3 = -1$ and $c_2 = 2c_3 = -2$. Therefore, the general solution for u_i, if the initial condition is not taken into account, is $u_i = c_1 2^i - i - 2$. This gives us the general solution for t_i and $T(n)$.

$$t_i = 2^{u_i} = 2^{c_1 2^i - i - 2}$$

$$T(n) = t_{\lg n} = 2^{c_1 n - \lg n - 2} = \frac{2^{c_1 n}}{4n}$$

We use the initial condition $T(1) = 1/3$ to determine c_1: $T(1) = 2^{c_1}/4 = 1/3$ implies that $c_1 = \lg(4/3) = 2 - \lg 3$. The final solution is therefore

$$T(n) = \frac{2^{2n}}{4n\, 3^n}.$$

Analysis of Algorithms

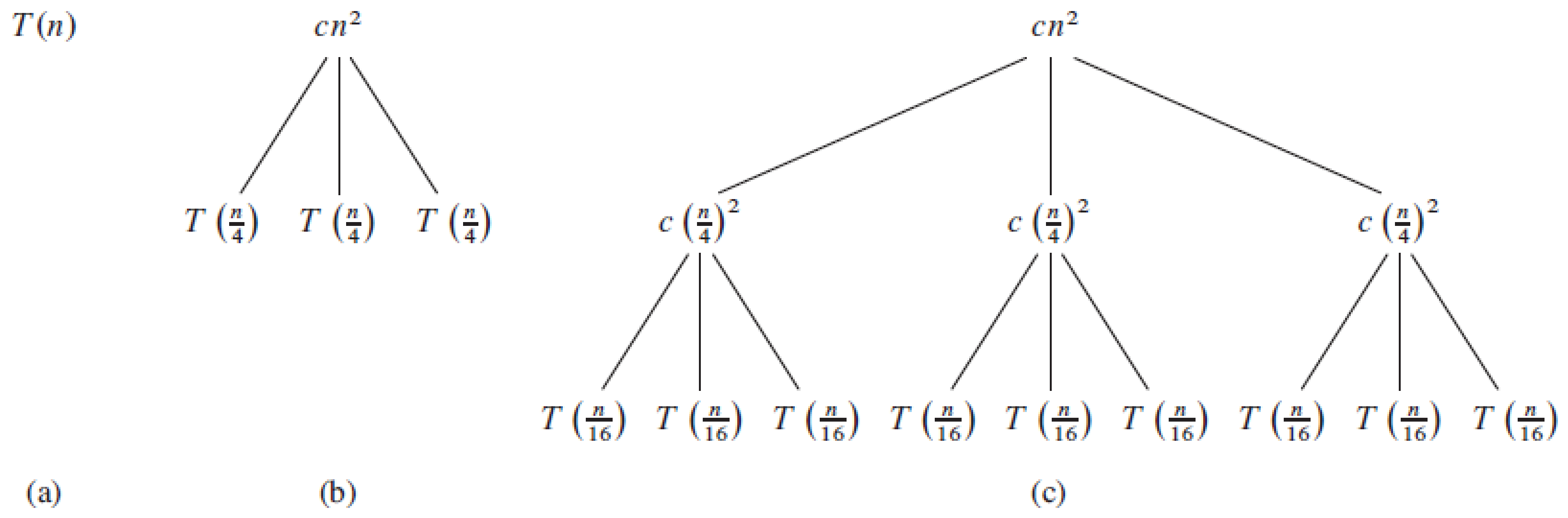
- Solving Recurrences
  - Recursion Tree Method
    - This method is used to generate a **good guess** and therefore a small amount of **sloppiness** can be tolerated.
    - In a recurrence tree, **each node** represents the **cost of a single sub-problem.**
    - Costs within each level of the tree are summed to obtain a set of **per-level costs**, and then all the per level costs are summed to determine the **total cost of all levels** of the recursion.

Analysis of Algorithms

- Solving Recurrences
  - Recursion Tree Method

$$T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2).$$

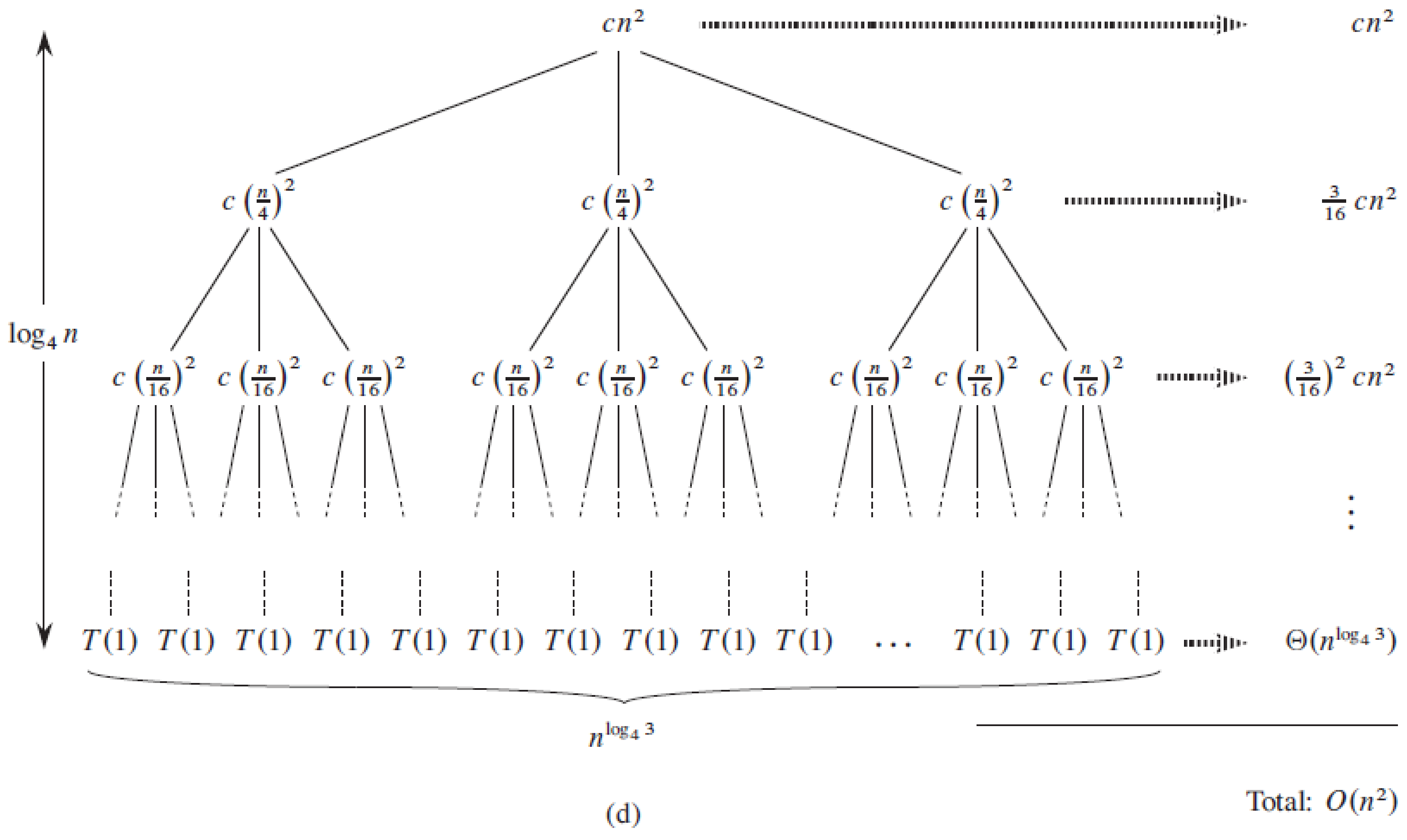
$$T(n) = 3T(n/4) + cn^2$$



$T(n)$
cn^2
$T\left(\frac{n}{4}\right)$ $T\left(\frac{n}{4}\right)$ $T\left(\frac{n}{4}\right)$
cn^2
$c\left(\frac{n}{4}\right)^2$ $c\left(\frac{n}{4}\right)^2$ $c\left(\frac{n}{4}\right)^2$
$T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$
(a)
(b)
(c)

Analysis of Algorithms

- Solving Recurrences
  - Recursion Tree Method



Analysis of Algorithms

- Solving Recurrences
  - Recursion Tree Method
    - Notice that sub-problem sizes decrease by a factor of 4 each time we go down one level.
    - Sub-problem size for a node at depth "i" is $n/4^i$
    - So, $(n/4^i) = 1$ means that sub-problem size has reached 1, i.e. when $i = \log_4 n$, sub-problem size has reached 1, i.e. $\log_4 n$ is the depth of the tree.
    - Each level has three times more nodes than the level above, and so the number of nodes at depth "i" is 3^i. Each node at depth "i", for $i = 0, 1, 2, \ldots, \log_4 n - 1$ has a cost of $c \times (n/4^i)^2$. So, total cost over all nodes at depth "i" is $3^i \times c \times (n/4^i)^2 = c \times n^2 \times (3/16)^i$.
    - The bottom level (at $i = \log_4 n$) has $3^i = 3^{\log_4 n} = n^{\log_4 3}$ nodes, each contributing cost T(1), for a total cost of $n^{\log_4 3} \times T(1)$, which is $\theta\left(n^{\log_4 3}\right)$, since we assume that T(1) is a constant.

Analysis of Algorithms

- Solving Recurrences
  - Recursion Tree Method

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

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

Analysis of Algorithms

- Solving Recurrences

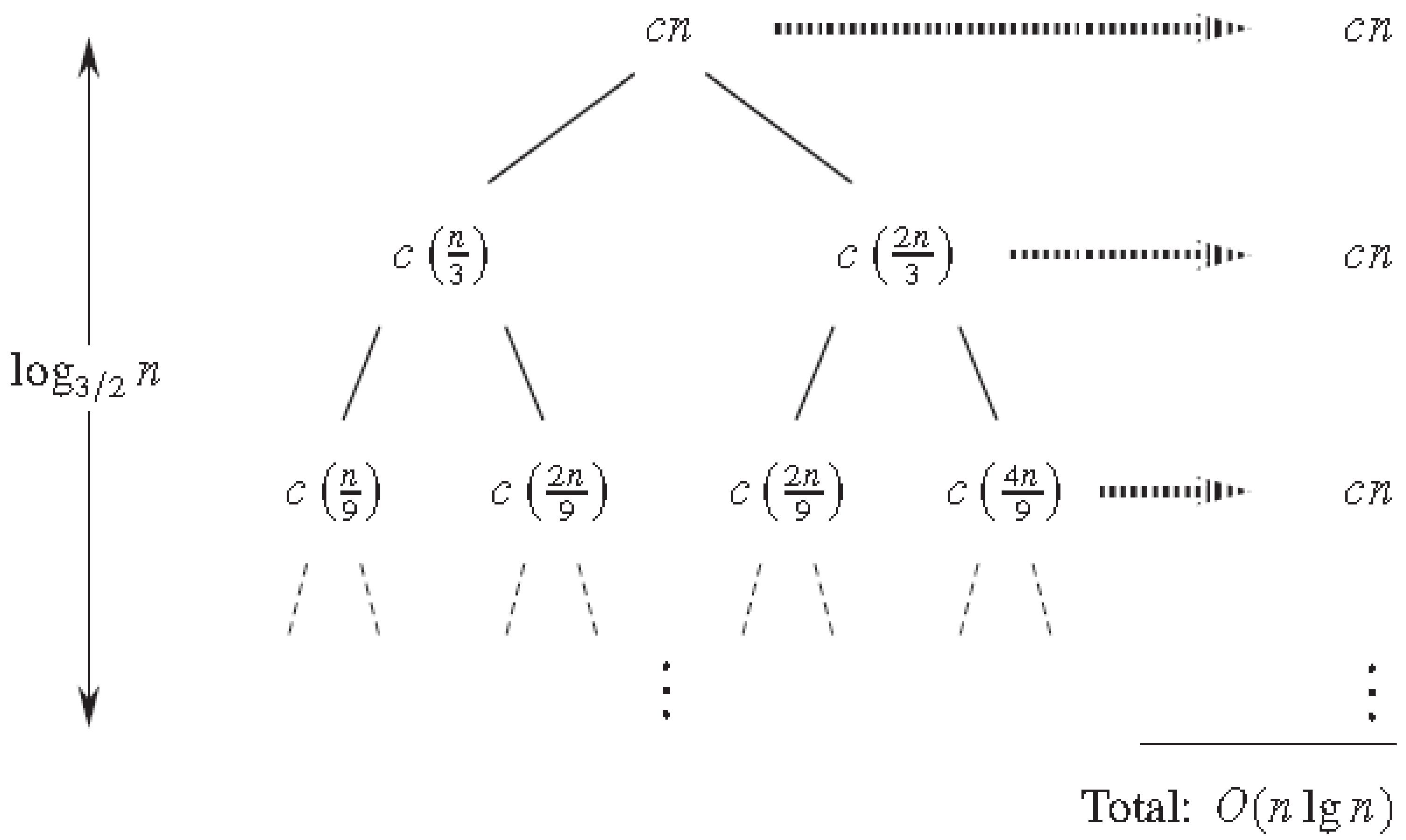
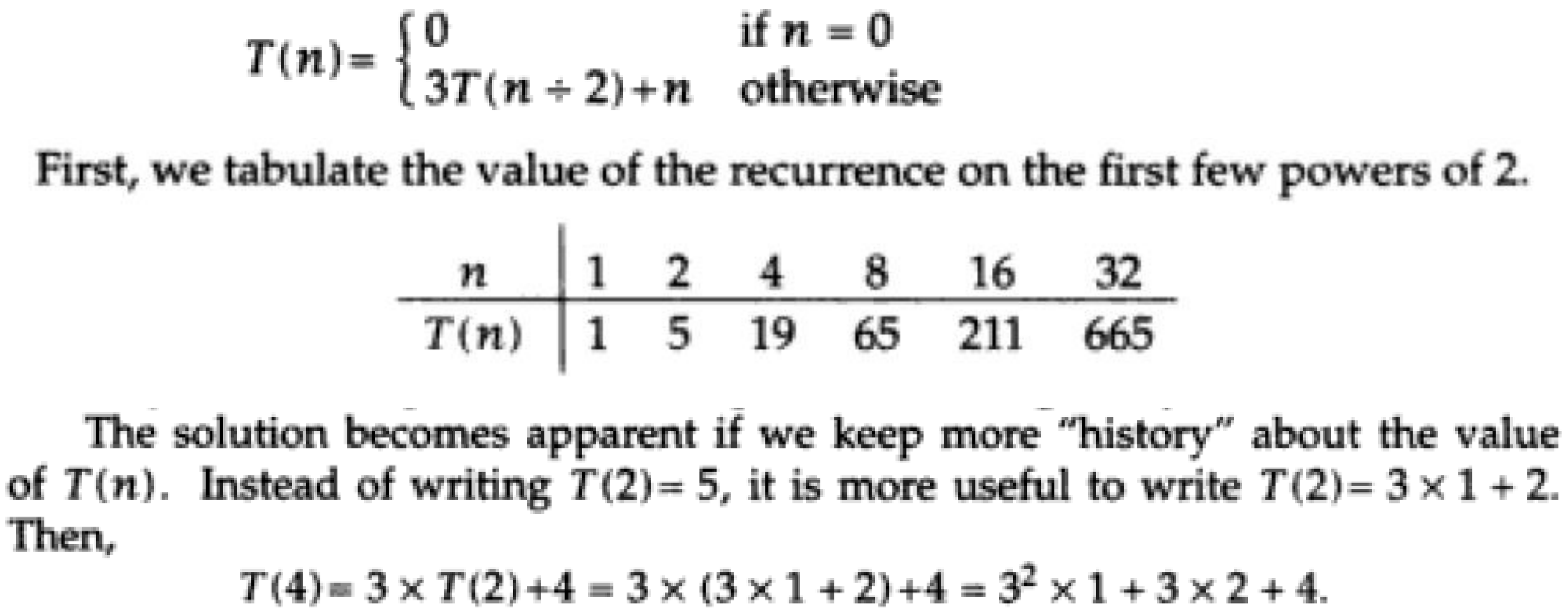

cn
cn
$c\left(\frac{n}{3}\right)$
$c\left(\frac{2n}{3}\right)$
cn
$c\left(\frac{n}{9}\right)$
$c\left(\frac{2n}{9}\right)$
$c\left(\frac{2n}{9}\right)$
$c\left(\frac{4n}{9}\right)$
cn
$\log_{3/2} n$
Total: $O(n \lg n)$

Figure 4.6 A recursion tree for the recurrence $T(n) = T(n/3) + T(2n/3) + cn$.

Analysis of Algorithms

- Solving Recurrences
  - Intelligent Guess Work



$$T(n) = \begin{cases} 0 & \text{if } n = 0 \\ 3T(n \div 2) + n & \text{otherwise} \end{cases}$$

First, we tabulate the value of the recurrence on the first few powers of 2.

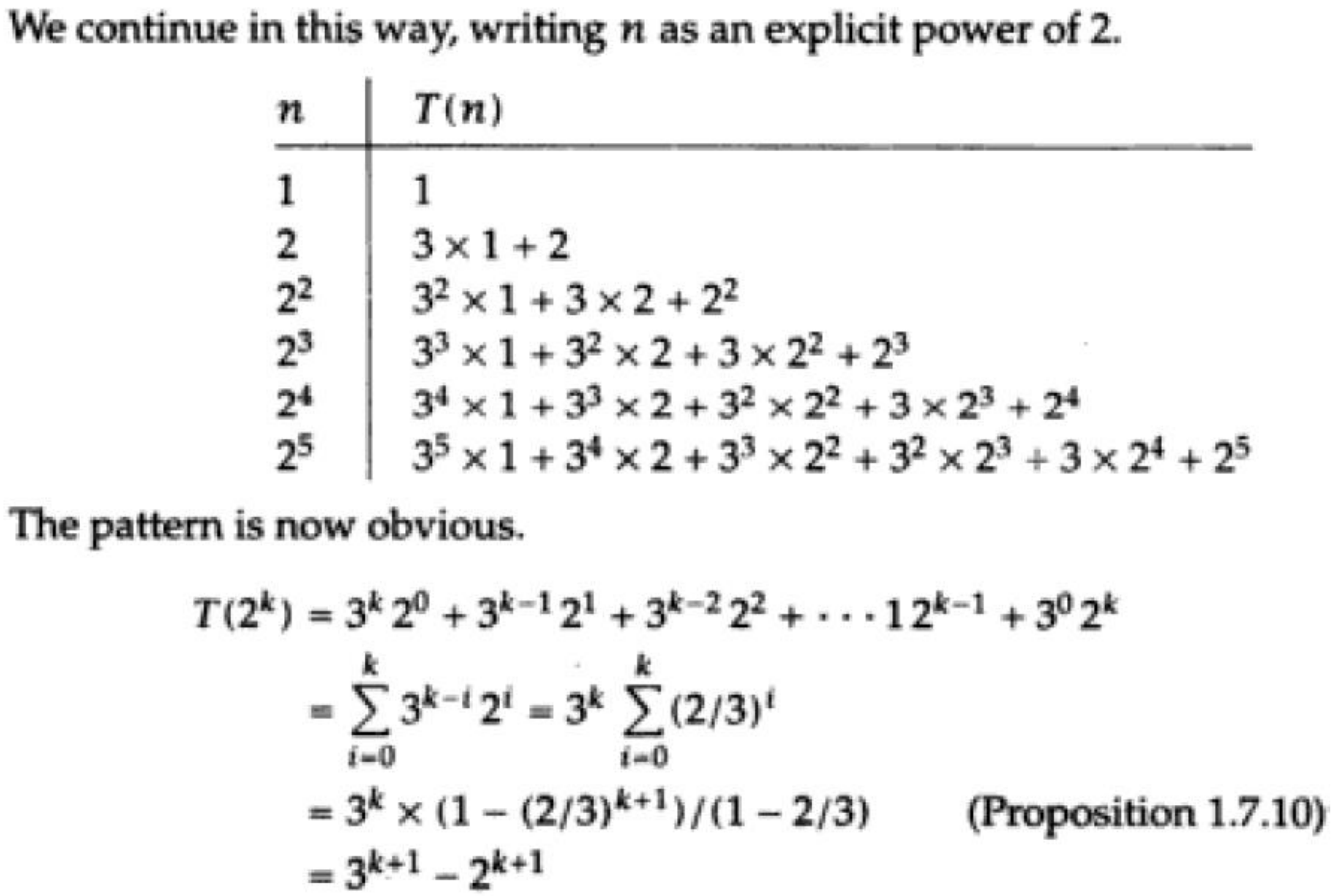
n	1	2	4	8	16	32
$T(n)$	1	5	19	65	211	665

The solution becomes apparent if we keep more "history" about the value of $T(n)$. Instead of writing $T(2) = 5$, it is more useful to write $T(2) = 3 \times 1 + 2$. Then,

$$T(4) = 3 \times T(2) + 4 = 3 \times (3 \times 1 + 2) + 4 = 3^2 \times 1 + 3 \times 2 + 4.$$

Analysis of Algorithms

- Solving Recurrences
- Intelligent Guess Work



We continue in this way, writing n as an explicit power of 2.

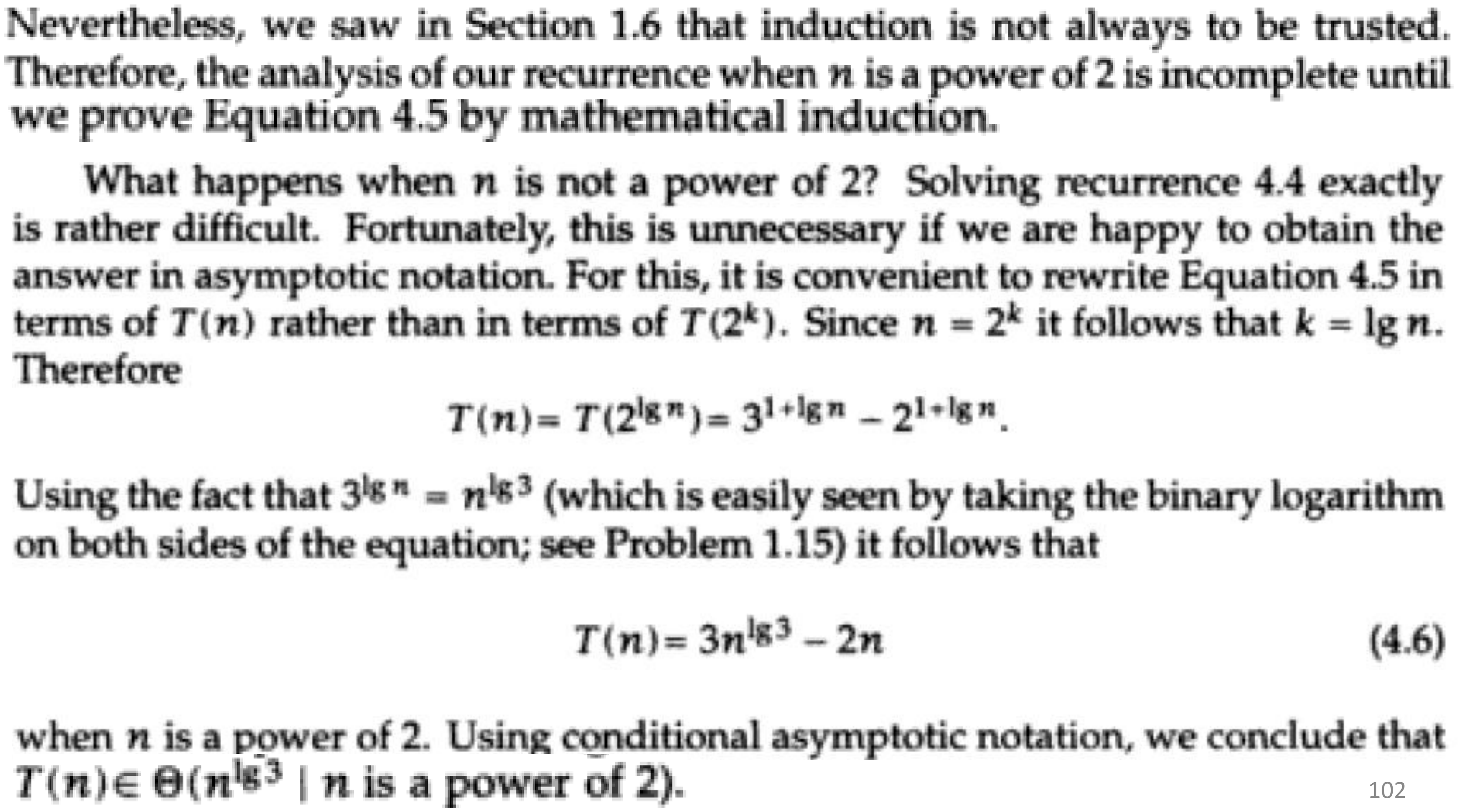
n	$T(n)$
1	1
2	$3 \times 1 + 2$
2^2	$3^2 \times 1 + 3 \times 2 + 2^2$
2^3	$3^3 \times 1 + 3^2 \times 2 + 3 \times 2^2 + 2^3$
2^4	$3^4 \times 1 + 3^3 \times 2 + 3^2 \times 2^2 + 3 \times 2^3 + 2^4$
2^5	$3^5 \times 1 + 3^4 \times 2 + 3^3 \times 2^2 + 3^2 \times 2^3 + 3 \times 2^4 + 2^5$

The pattern is now obvious.

$$\begin{aligned} T(2^k) &= 3^k 2^0 + 3^{k-1} 2^1 + 3^{k-2} 2^2 + \cdots 1 2^{k-1} + 3^0 2^k \\ &= \sum_{i=0}^{k} 3^{k-i} 2^i = 3^k \sum_{i=0}^{k} (2/3)^i \\ &= 3^k \times (1 - (2/3)^{k+1})/(1 - 2/3) \qquad \text{(Proposition 1.7.10)} \\ &= 3^{k+1} - 2^{k+1} \end{aligned}$$

Analysis of Algorithms

- Solving Recurrences
- Intelligent Guess Work



Nevertheless, we saw in Section 1.6 that induction is not always to be trusted. Therefore, the analysis of our recurrence when n is a power of 2 is incomplete until we prove Equation 4.5 by mathematical induction.

What happens when n is not a power of 2? Solving recurrence 4.4 exactly is rather difficult. Fortunately, this is unnecessary if we are happy to obtain the answer in asymptotic notation. For this, it is convenient to rewrite Equation 4.5 in terms of $T(n)$ rather than in terms of $T(2^k)$. Since $n = 2^k$ it follows that $k = \lg n$. Therefore

$$T(n) = T(2^{\lg n}) = 3^{1+\lg n} - 2^{1+\lg n}.$$

Using the fact that $3^{\lg n} = n^{\lg 3}$ (which is easily seen by taking the binary logarithm on both sides of the equation; see Problem 1.15) it follows that

$$T(n) = 3n^{\lg 3} - 2n \qquad (4.6)$$

when n is a power of 2. Using conditional asymptotic notation, we conclude that $T(n) \in \Theta(n^{\lg 3} \mid n \text{ is a power of } 2)$.