Introduction to

3rd Edition

The Design and Analysis of Algorithms

2

Fundamentals of the Analysis of algorithm Efficiency

2.1 The Analysis Framework

There are two kinds of efficiency:

- *Time efficiency*, also called *time complexity*, indicates how fast an algorithm in question runs.

- *Space efficiency*, also called *space complexity*, refers to the amount of memory units required by the algorithm in addition to the space needed for its input and output.

Measuring an Input's Size

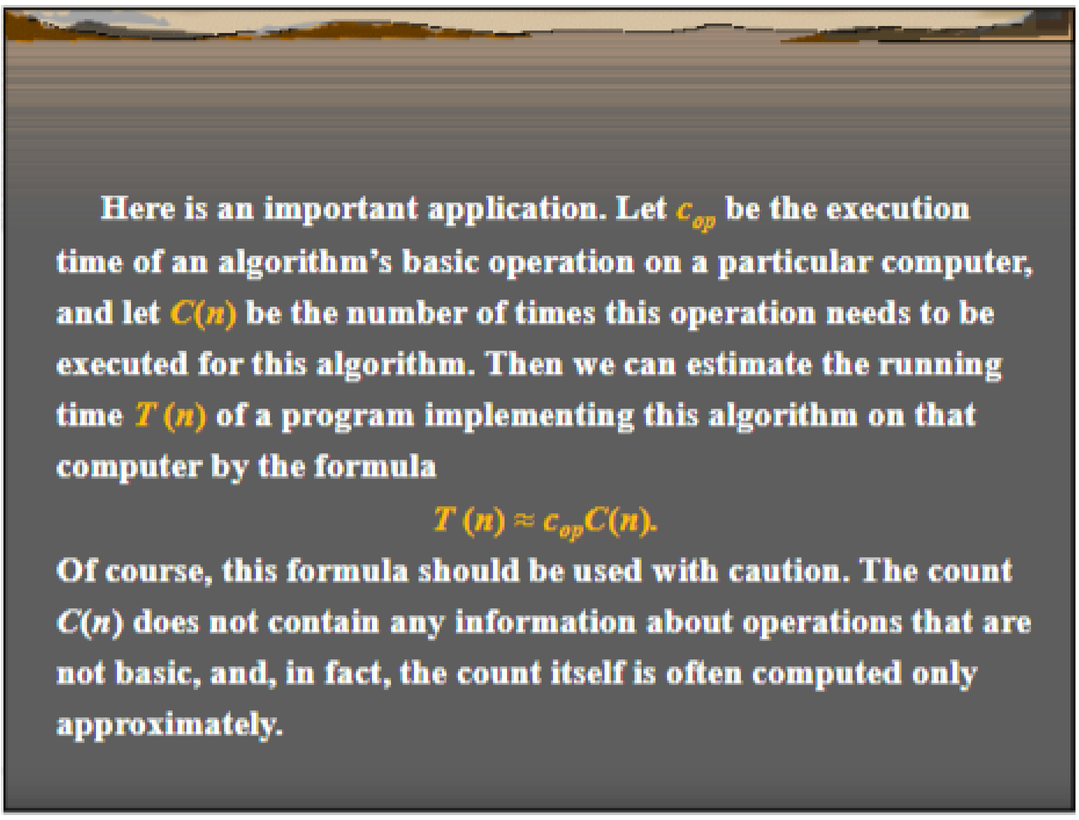
It is logical to investigate an algorithm's efficiency as a function of some parameter *n* indicating the algorithm' input size.

We should make a special note about measuring input size for algorithms solving problems such as checking primality of a positive integer *n*. Here, the input is just one number, and it is this number's magnitude that determines the input size. In such situations, it is preferable to measure size by the number *b* of bits in the *n*'s binary representation:

$$b = \lfloor \log_2 n \rfloor + 1. \qquad (2.1)$$

Units for Measuring Running Time

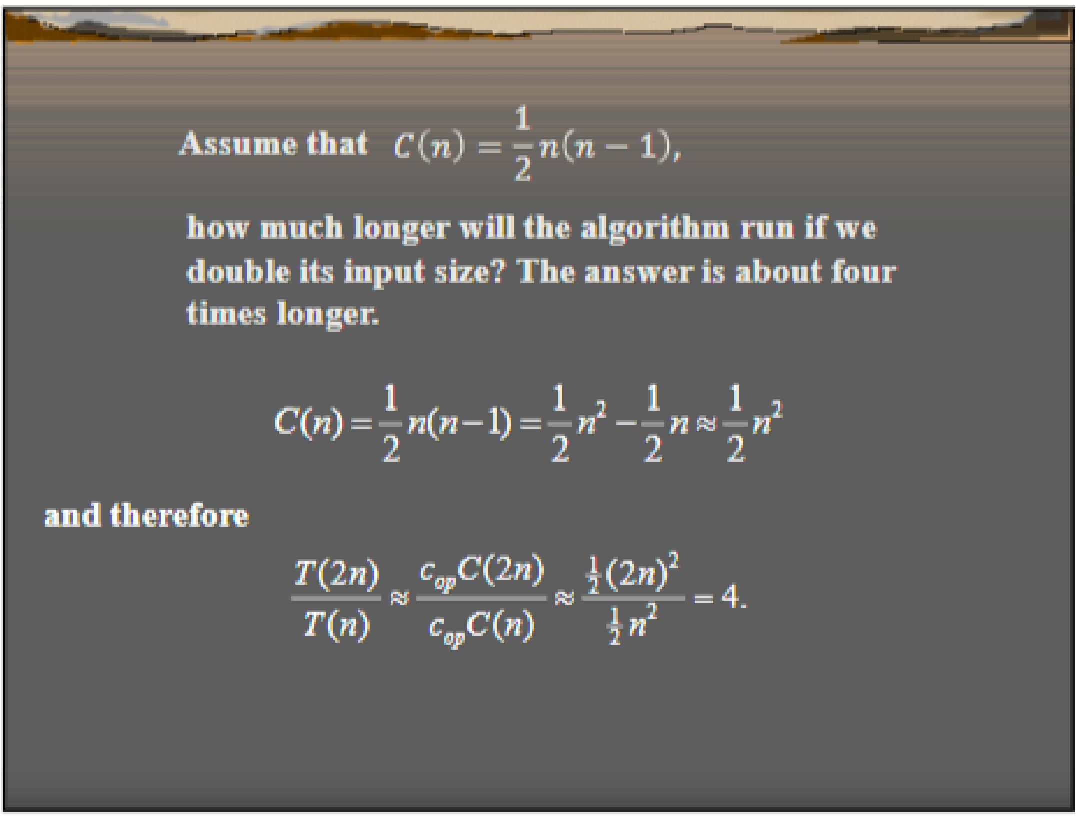
One possible approach is to count the number of times each of the algorithm's operations is executed. This approach is both excessively difficult and, as we shall see, usually unnecessary. The thing to do is to identify the most important operation of the algorithm, called the *basic operation* (基本運算), the operation contributing the most to the total running time (對總執行時間貢獻最大的運算), and compute the number of times the basic operation is executed.



Here is an important application. Let c_{op} be the execution time of an algorithm's basic operation on a particular computer, and let $C(n)$ be the number of times this operation needs to be executed for this algorithm. Then we can estimate the running time $T(n)$ of a program implementing this algorithm on that computer by the formula

$$T(n) \approx c_{op}C(n).$$

Of course, this formula should be used with caution. The count $C(n)$ does not contain any information about operations that are not basic, and, in fact, the count itself is often computed only approximately.



Assume that $C(n) = \frac{1}{2}n(n-1)$,

how much longer will the algorithm run if we double its input size? The answer is about four times longer.

$$C(n) = \frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \approx \frac{1}{2}n^2$$

and therefore

$$\frac{T(2n)}{T(n)} \approx \frac{c_{op}C(2n)}{c_{op}C(n)} \approx \frac{\frac{1}{2}(2n)^2}{\frac{1}{2}n^2} = 4.$$

Orders of Growth

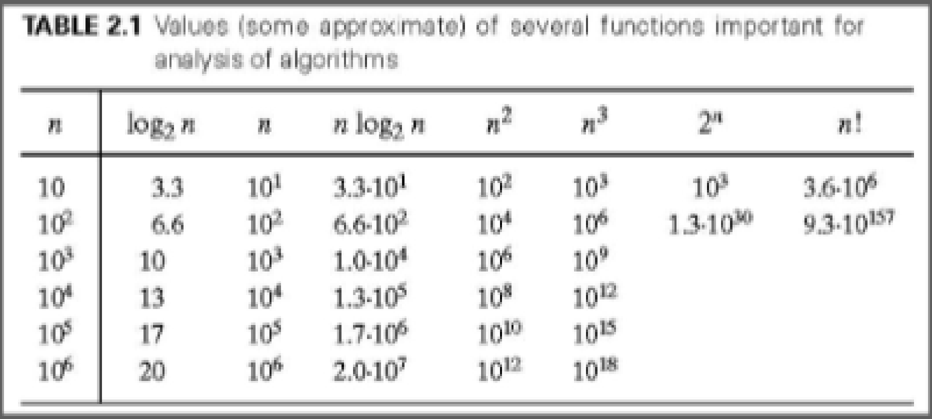
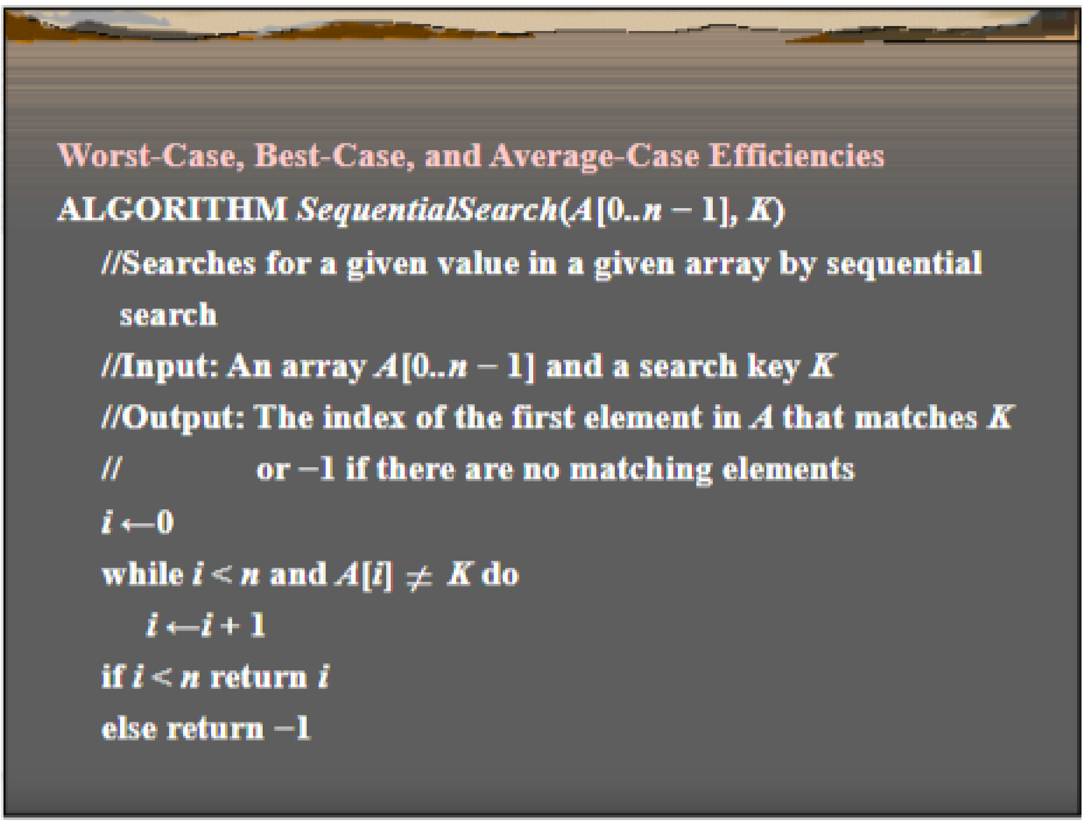


TABLE 2.1 Values (some approximate) of several functions important for analysis of algorithms

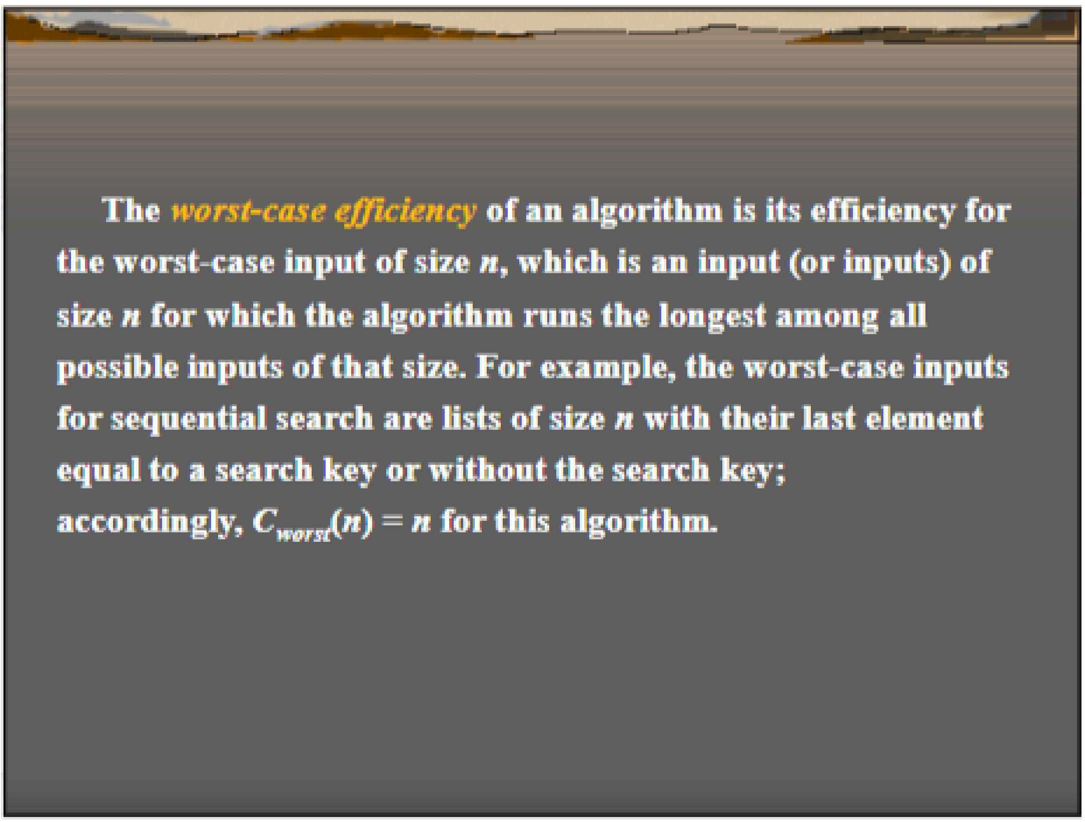
n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

- **Since $\log_a n = \log_a b \log_b n$, we can omit a logarithm's base and write simply log n in situations where we are interested just a function's order of growth.**

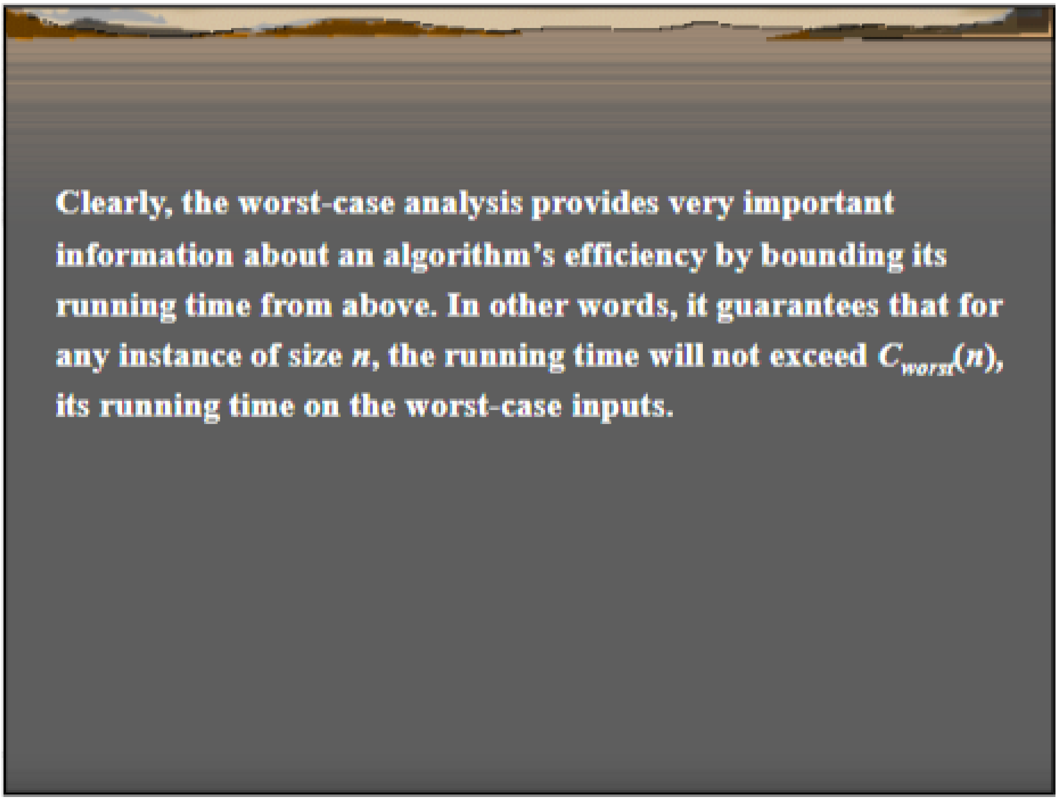


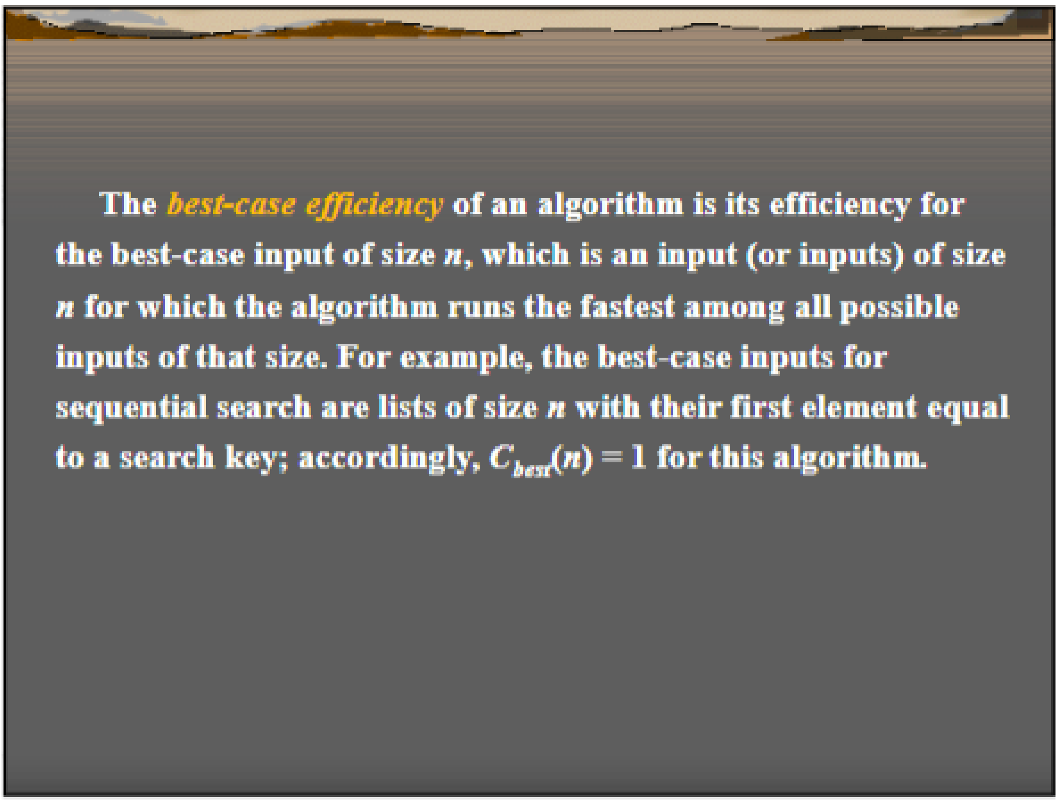
Worst-Case, Best-Case, and Average-Case Efficiencies

```
ALGORITHM SequentialSearch(A[0..n − 1], K)
    //Searches for a given value in a given array by sequential
      search
    //Input: An array A[0..n − 1] and a search key K
    //Output: The index of the first element in A that matches K
    //              or −1 if there are no matching elements
    i ← 0
    while i < n and A[i] ≠ K do
        i ← i + 1
    if i < n return i
    else return −1
```



The *worst-case efficiency* of an algorithm is its efficiency for the worst-case input of size n, which is an input (or inputs) of size n for which the algorithm runs the longest among all possible inputs of that size. For example, the worst-case inputs for sequential search are lists of size n with their last element equal to a search key or without the search key; accordingly, $C_{worst}(n) = n$ for this algorithm.


Clearly, the worst-case analysis provides very important information about an algorithm's efficiency by bounding its running time from above. In other words, it guarantees that for any instance of size n, the running time will not exceed $C_{worst}(n)$, its running time on the worst-case inputs.

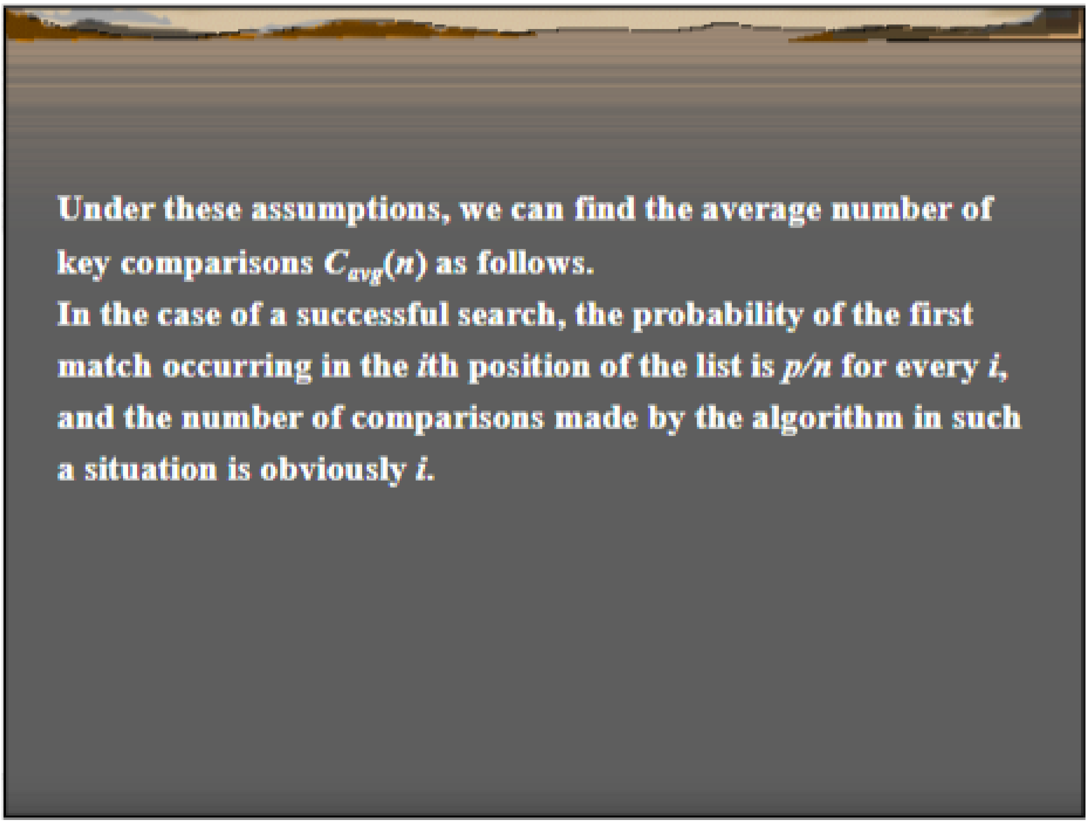

The *best-case efficiency* of an algorithm is its efficiency for the best-case input of size n, which is an input (or inputs) of size n for which the algorithm runs the fastest among all possible inputs of that size. For example, the best-case inputs for sequential search are lists of size n with their first element equal to a search key; accordingly, $C_{best}(n) = 1$ for this algorithm.

For example, the best-case inputs for sequential search are lists of size n with their first element equal to a search key; accordingly, $C_{best}(n) = 1$ for this algorithm.

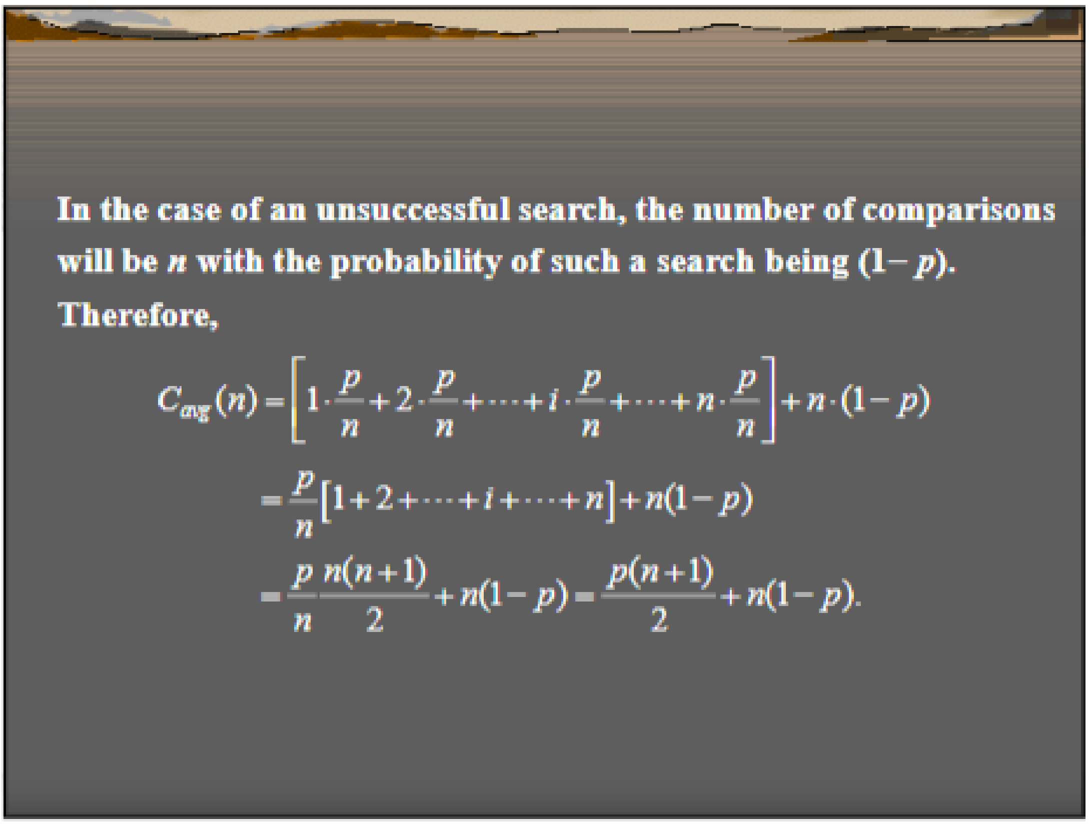
To analyze the algorithm's *average-case efficiency*, we must make some assumptions about possible inputs of size n. Let's consider again sequential search. The standard assumptions are that

(a) the probability of a successful search is equal to p ($0 \le p \le 1$);

(b) the probability of the first match occurring in the ith position of the list is the same for every i.



Under these assumptions, we can find the average number of key comparisons $C_{avg}(n)$ as follows.
In the case of a successful search, the probability of the first match occurring in the *i*th position of the list is *p/n* for every *i*, and the number of comparisons made by the algorithm in such a situation is obviously *i*.



In the case of an unsuccessful search, the number of comparisons will be *n* with the probability of such a search being $(1-p)$.
Therefore,

$$C_{avg}(n) = \left[1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + \cdots + i \cdot \frac{p}{n} + \cdots + n \cdot \frac{p}{n}\right] + n \cdot (1-p)$$

$$= \frac{p}{n}\left[1 + 2 + \cdots + i + \cdots + n\right] + n(1-p)$$

$$= \frac{p}{n}\frac{n(n+1)}{2} + n(1-p) = \frac{p(n+1)}{2} + n(1-p).$$

2.2 Asymptotic Notations and Basic Efficiency Classes

Informal Introduction

Informally, $O(g(n))$ is the set of all functions with a lower or same order of growth as $g(n)$ (to within a constant multiple, as n goes to infinity). Thus, to give a few examples, the following assertions are all true:

$$n \in O(n^2), \qquad 100n + 5 \in O(n^2), \qquad \frac{1}{2}n(n-1) \in O(n^2).$$

Indeed, the first two functions are linear and hence have a lower order of growth than $g(n) = n^2$, while the last one is quadratic and hence has the same order of growth as n^2. On the other hand,

$$n^3 \notin O(n^2), \qquad 0.00001n^3 \notin O(n^2), \qquad n^4 + n + 1 \notin O(n^2).$$

The second notation, $\Omega(g(n))$, stands for the set of all functions with a higher or same order of growth as $g(n)$ (to within a constant multiple, as n goes to infinity). For example,

$$n^3 \in \Omega(n^2), \qquad \frac{1}{2}n(n-1) \in \Omega(n^2), \qquad \text{but } 100n + 5 \notin \Omega(n^2).$$

O - notation

DEFINITION A function $t(n)$ is said to be in $O(g(n))$, denoted $t(n) \in O(g(n))$, if $t(n)$ is bounded above by some constant multiple of $g(n)$ for all large n, i.e., if there exist some positive constant c and some nonnegative integer n_0 such that

$$t(n) \le cg(n) \text{ for all } n \ge n_0.$$

The definition is illustrated in Figure 2.1 where, for the sake of visual clarity, n is extended to be a real number.

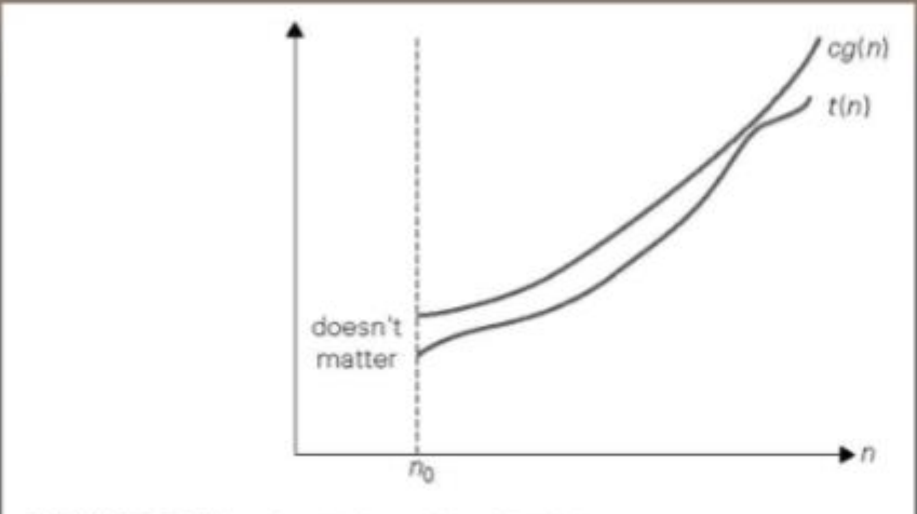


FIGURE 2.1 Big-oh notation: $t(n) \in O(g(n))$.

As an example, let us formally prove one of the assertions made in the introduction: $100n + 5 \in O(n^2)$. Indeed,

$$100n + 5 \le 100n + n \text{ (for all } n \ge 5) = 101n \le 101n^2.$$

Thus, as values of the constants c and n_0 required by the definition, we can take 101 and 5, respectively.

Note that the definition gives us a lot of freedom in choosing specific values for constants c and n_0. For example, we could also reason that

$$100n + 5 \le 100n + 5n \text{ (for all } n \ge 1) = 105n$$

to complete the proof with $c = 105$ and $n_0 = 1$.

Ω - notation

DEFINITION A function $t(n)$ is said to be in $\Omega(g(n))$, denoted $t(n) \in \Omega(g(n))$, if $t(n)$ is bounded below by some positive constant multiple of $g(n)$ for all large n, i.e., if there exist some positive constant c and some nonnegative integer n_0 such that

$$t(n) \geq cg(n) \text{ for all } n \geq n_0.$$

The definition is illustrated in Figure 2.2.

Here is an example of the formal proof that $n^3 \in \Omega(n^2)$:

$$n^3 \geq n^2 \text{ for all } n \geq 0,$$

i.e., we can select $c = 1$ and $n_0 = 0$.

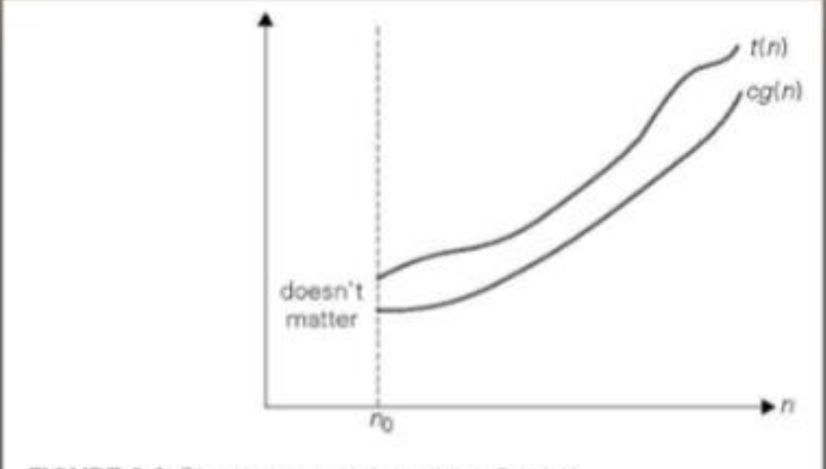


FIGURE 2.2 Big-omega notation: $t(n) \in \Omega(g(n))$.

Θ - notation

DEFINITION A function $t(n)$ is said to be in $\Theta(g(n))$, denoted $t(n) \in \Theta(g(n))$, if $t(n)$ is bounded both above and below by some positive constant multiples of $g(n)$ for all large n, i.e., if there exist some positive constants c_1 and c_2 and some nonnegative integer n_0 such that

$$c_2 g(n) \le t(n) \le c_1 g(n) \text{ for all } n \ge n_0.$$

The definition is illustrated in Figure 2.3.

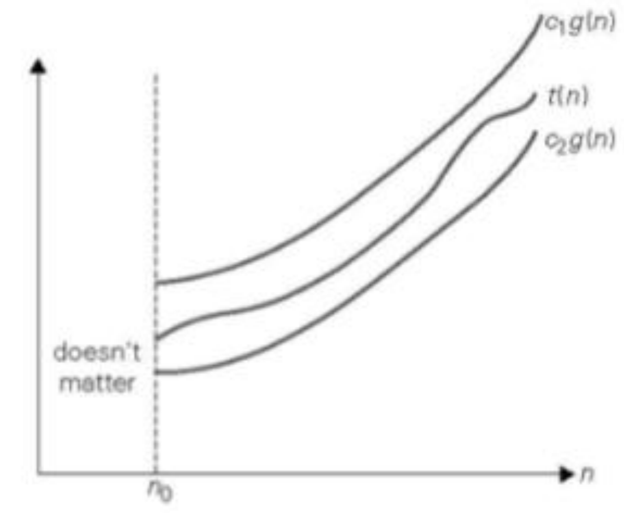


FIGURE 2.3 Big-theta notation: $t(n) \in \Theta(g(n))$.

For example, let us prove that $\frac{1}{2}n(n-1) \in \Theta(n^2)$. First, we prove the right inequality (the upper bound):

$$\frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \le \frac{1}{2}n^2 \quad \text{for all } n \ge 0.$$

Second, we prove the left inequality (the lower bound):

$$\frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \ge \frac{1}{2}n^2 - \frac{1}{2}n\frac{1}{2}n (\text{for all } n \ge 2) = \frac{1}{4}n^2.$$

Hence, we can select $c_2 = 1/4$, $c_1 = 1/2$, and $n_0 = 2$.

Useful Property Involving the Asymptotic Notations

THEOREM If $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$, then $t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$.

(The analogous assertions are true for the Ω and Θ notations as well.)

PROOF The proof extends to orders of growth the following simple fact about four arbitrary real numbers a_1, $b1$, a_2, b_2: if $a_1 \le b_1$ and $a_2 \le b_2$, then $a_1 + a_2 \le 2 \max\{b_1, b_2\}$.

Since $t_1(n) \in O(g_1(n))$, there exist some positive constant c_1 and some nonnegative integer n_1 such that

$$t_1(n) \le c_1 g_1(n) \quad \text{for all } n \ge n_1.$$

Similarly, since $t_2(n) \in O(g_2(n))$,

$$t_2(n) \le c_2 g_2(n) \quad \text{for all } n \ge n_2.$$

Let us denote $c_3 = \max\{c_1, c_2\}$ and consider $n \ge \max\{n_1, n_2\}$ so that we can use both inequalities. Adding them yields the following:

$$\begin{aligned} t_1(n) + t_2(n) &\le c_1 g_1(n) + c_2 g_2(n) \\ &\le c_3 g_1(n) + c_3 g_2(n) = c_3[g_1(n) + g_2(n)] \\ &\le c_3 2 \max\{g_1(n), g_2(n)\}. \end{aligned}$$

Hence, $t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$, with the constants c and n_0 required by the O definition being $2c_3 = 2\max\{c_1, c_2\}$ and $\max\{n_1, n_2\}$, respectively.

So what does this property imply for an algorithm that comprises two consecutively executed parts? It implies that the algorithm's overall efficiency is determined by the part with a higher order of growth, i.e., its least efficient part:

$$\left.\begin{array}{|c|}\hline t_1(n) \in O(g_1(n)) \\ \hline t_2(n) \in O(g_2(n)) \\ \hline \end{array}\right\} \quad t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\}).$$

Using Limits for Comparing Orders of Growth

Though the formal definitions of O, Ω , and Θ are indispensable for proving their abstract properties, they are rarely used for comparing the orders of growth of two specific functions. A much more convenient method for doing so is based on computing the limit of the ratio of two functions in question. Three principal cases may arise:

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \begin{cases} 0 & \text{implies that } t(n) \text{ has a smaller order of growth than } g(n), \\ c & \text{implies that } t(n) \text{ has the same order of growth as } g(n), \\ \infty & \text{implies that } t(n) \text{ has a larger order of growth than } g(n).^{3} \end{cases}$$

The limit-based approach is often more convenient than the one based on the definitions because it can take advantage of the powerful calculus techniques developed for computing limits, such as L'H ôpital's rule

$$\lim_{n\to\infty}\frac{t(n)}{g(n)}=\lim_{n\to\infty}\frac{t'(n)}{g'(n)}$$

and Stirling's formula

$$n!\approx\sqrt{2\pi n}\left(\frac{n}{e}\right)^n \text{ for large values of } n.$$

EXAMPLE 1 Compare the orders of growth of $\frac{1}{2}n(n-1)$ and n^2. (This is one of the examples we used at the beginning of this section to illustrate the definitions.)

$$\lim_{n\to\infty}\frac{\frac{1}{2}n(n-1)}{n^2}=\frac{1}{2}\lim_{n\to\infty}\frac{n^2-n}{n^2}=\frac{1}{2}\lim_{n\to\infty}(1-\frac{1}{n})=\frac{1}{2}.$$

Since the limit is equal to a positive constant, the functions have the same order of growth or, symbolically, $\frac{1}{2}n(n-1)\in\Theta(n^2)$.

EXAMPLE 2 **Compare the orders of growth of $\log_2 n$ and $\sqrt{n}$. (Unlike Example 1, the answer here is not immediately obvious.)**

$$\lim_{n\to\infty}\frac{\log_2 n}{\sqrt{n}} = \lim_{n\to\infty}\frac{(\log_2 n)'}{\left(\sqrt{n}\right)'} = \lim_{n\to\infty}\frac{(\log_2 e)\frac{1}{n}}{\frac{1}{2\sqrt{n}}} = 2\log_2 e \lim_{n\to\infty}\frac{1}{\sqrt{n}} = 0.$$

Since the limit is equal to zero, $\log_2 n$ has a smaller order of growth than $\sqrt{n}$. (Since $\lim_{n\to\infty}\frac{\log_2 n}{\sqrt{n}} = 0$, we can use the so-called *little-oh notation*: $\log_2 n \in o\left(\sqrt{n}\right)$. Unlike the big-Oh, the little-oh notation is rarely used in analysis of algorithms.)

Littel-*o*的定義如下：

我們說

$$f(n) \in o\big(g(n)\big)$$

如果對於所有正實數c, 存在正整數n_0,使得對於所有$n \ge n_0, 0 \le f(n) \le cg(n)$.

EXAMPLE 3 Compare the orders of growth of $n!$ and 2^n. (We discussed this informally in Section 2.1.) Taking advantage of Stirling's formula, we get

$$\lim_{n\to\infty}\frac{n!}{2^n}=\lim_{n\to\infty}\frac{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n}{2^n}=\lim_{n\to\infty}\sqrt{2\pi n}\,\frac{n^n}{2^n e^n}=\lim_{n\to\infty}\sqrt{2\pi n}\left(\frac{n}{2e}\right)^n=\infty.$$

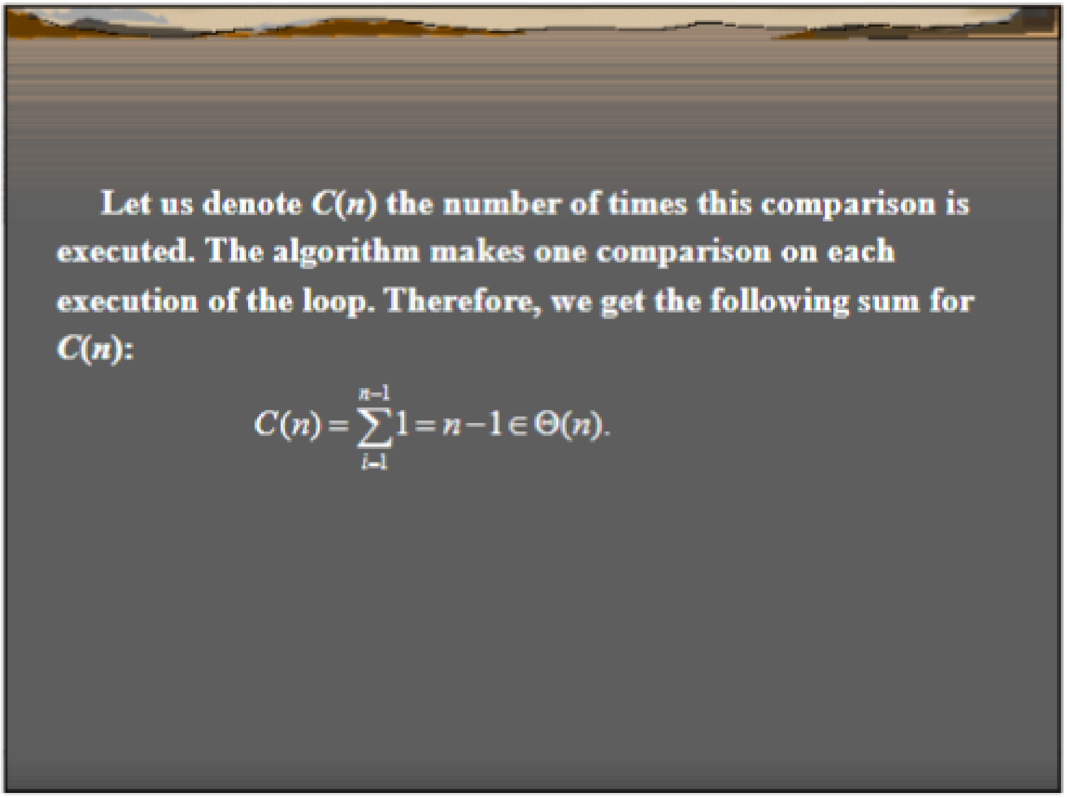
Thus, though 2^n grows very fast, $n!$ grows still faster. We can write symbolically that $n!\in\Omega(2^n)$; note, however, that while the big-Omega notation does not preclude the possibility that $n!$ and 2^n have the same order of growth, the limit computed here certainly does.

2.3 Mathematical Analysis of Nonrecursive Algorithms

EXAMPLE 1 Consider the problem of finding the value of the largest element in a list of n numbers. For simplicity, we assume that the list is implemented as an array. The following is pseudocode of a standard algorithm for solving the problem.

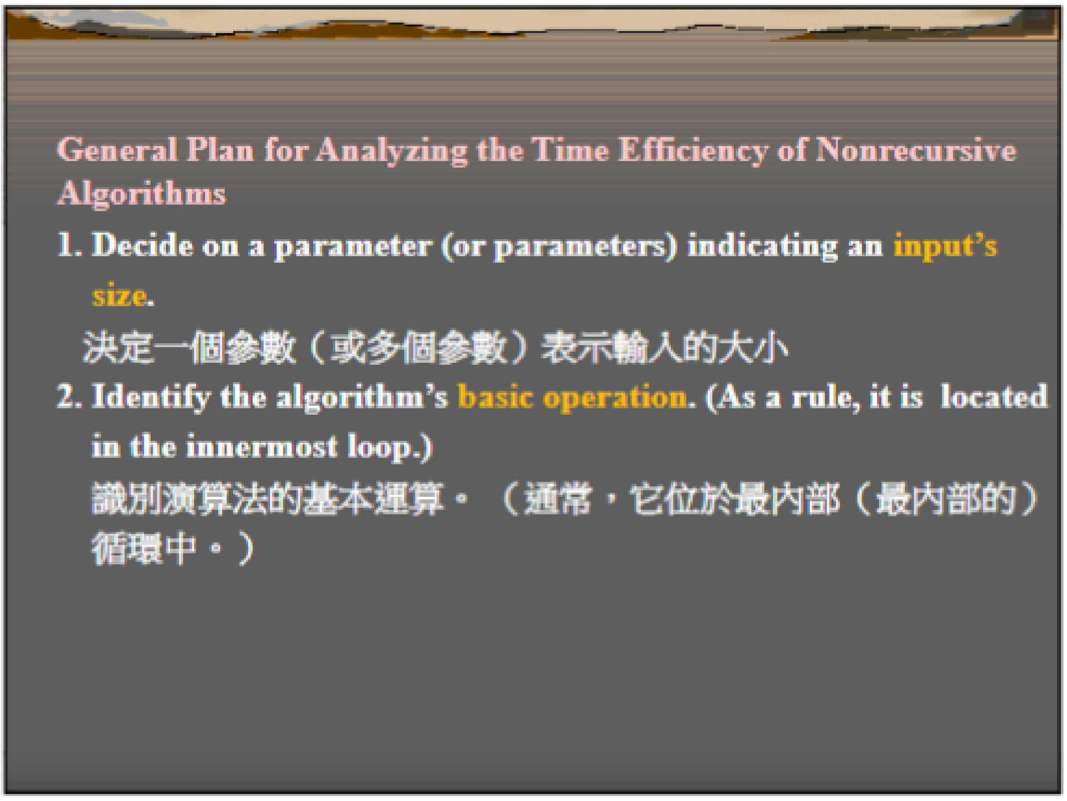
```
ALGORITHM MaxElement(A[0..n − 1])
    //Determines the value of the largest element in a given array
    //Input: An array A[0..n − 1] of real numbers
    //Output: The value of the largest element in A
    maxval ← A[0]
    for i ← 1 to n − 1 do
        if A[i] > maxval
            maxval ← A[i]
    return maxval
```

The obvious measure of an input's size here is the number of elements in the array, i.e., n. The operations that are going to be executed most often are in the algorithm's for loop. There are two operations in the loop's body: the comparison $A[i] > maxval$ and the assignment $maxval \leftarrow A[i]$. Which of these two operations should we consider basic? Since the comparison is executed on each repetition of the loop and the assignment is not, we should consider the comparison to be the algorithm's basic operation. Note that the number of comparisons will be the same for all arrays of size n; therefore, in terms of this metric, there is no need to distinguish among the worst, average, and best cases here.



Let us denote $C(n)$ the number of times this comparison is executed. The algorithm makes one comparison on each execution of the loop. Therefore, we get the following sum for $C(n)$:

$$C(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n).$$



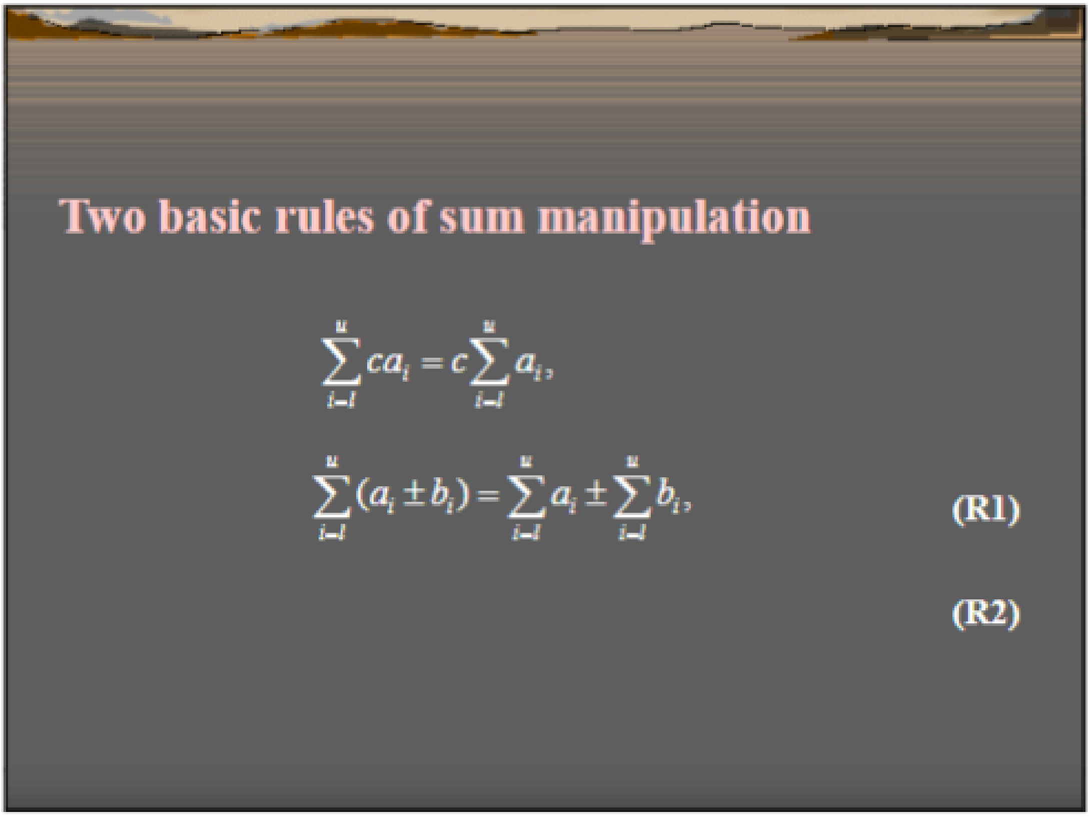
General Plan for Analyzing the Time Efficiency of Nonrecursive Algorithms

1. **Decide on a parameter (or parameters) indicating an input's size.**
   決定一個參數（或多個參數）表示輸入的大小
2. **Identify the algorithm's basic operation. (As a rule, it is located in the innermost loop.)**
   識別演算法的基本運算。（通常，它位於最內部（最內部的）循環中。）

3. Check whether the number of times the basic operation is executed depends only on the size of an input. If it also depends on some additional property, the worst-case, average-case, and, if necessary, best-case efficiencies have to be investigated separately.
檢查基本運算的執行次數是否僅取決於輸入的大小。如果它還取決於某些附加屬性，則必須分別研究最壞情況、平均情況以及必要時的最佳情況效率。

4. Set up a sum expressing the number of times the algorithm's basic operation is executed.
設置表示演算法基本運算執行次數的總和。

5. Using standard formulas and rules of sum manipulation, either find a closed-form formula for the count or, at the very least, establish its order of growth.
使用標準公式和求和運算規則，找到計數的封閉式公式，或者至少建立它的成長順序。



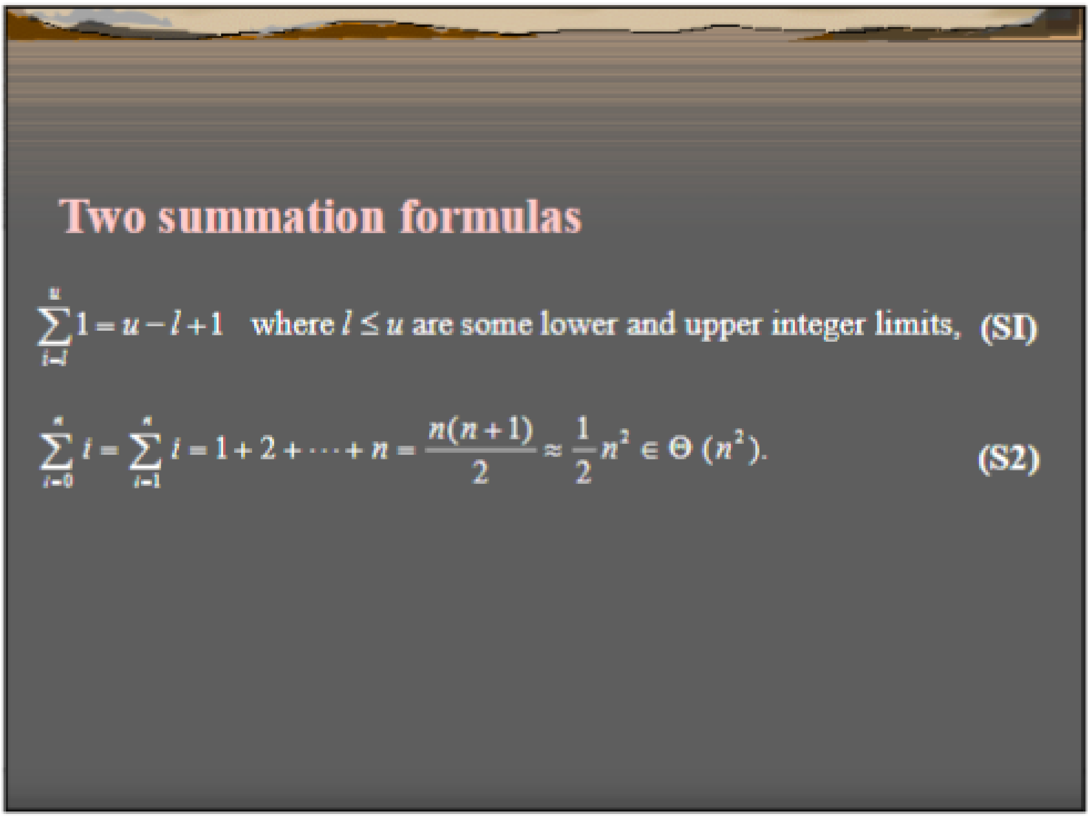
Two basic rules of sum manipulation

$$\sum_{i=l}^{u} ca_i = c\sum_{i=l}^{u} a_i,$$

$$\sum_{i=l}^{u} (a_i \pm b_i) = \sum_{i=l}^{u} a_i \pm \sum_{i=l}^{u} b_i,$$

(R1)

(R2)



Two summation formulas

$\sum_{i=l}^{u} 1 = u - l + 1$ where $l \le u$ are some lower and upper integer limits, (S1)

$\sum_{i=0}^{n} i = \sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2} \approx \frac{1}{2}n^2 \in \Theta(n^2).$ (S2)

EXAMPLE 2 Consider the *element uniqueness problem*: check whether all the elements in a given array of n elements are distinct. This problem can be solved by the following straightforward algorithm.

```
ALGORITHM    UniqueElements(A[0..n − 1])
    //Determines whether all the elements in a given array are
      distinct
    //Input: An array A[0..n − 1]
    //Output: Returns “true” if all the elements in A are distinct
    //             and “false” otherwise
    for i ← 0 to n − 2 do
        for j ← i + 1 to n − 1 do
            if A[i] = A[j] return false
    return true
```

The natural measure of the input's size here is again *n*, the number of elements in the array. Since the innermost loop contains a single operation (the comparison of two elements), we should consider it as the algorithm's basic operation.

Note that the number of element comparisons depends not only on *n* but also on whether there are equal elements in the array and, if there are, which array positions they occupy. We will limit our investigation to the worst case only.

By definition, the worst case input is an array for which the number of element comparisons $C_{worst}(n)$ is the largest among all arrays of size *n*.

An inspection of the innermost loop reveals that there are two kinds of worst-case inputs—inputs for which the algorithm does not exit the loop prematurely (過早地): arrays with no equal elements and arrays in which the last two elements are the only pair of equal elements.

For such inputs, one comparison is made for each repetition of the innermost loop, i.e., for each value of the loop variable j between its limits $i+1$ and $n-1$; this
is repeated for each value of the outer loop, i.e., for each value of the loop variable i between its limits 0 and $n-2$. Accordingly, we get

$$C_{worst}(n) = \sum_{i=0}^{n-2}\sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2}[(n-1)-(i+1)+1] = \sum_{i=0}^{n-2}(n-1-i)$$

$$= \sum_{i=0}^{n-2}(n-1) - \sum_{i=0}^{n-2} i = (n-1)(n-1) - \frac{(n-2)(n-1)}{2}$$

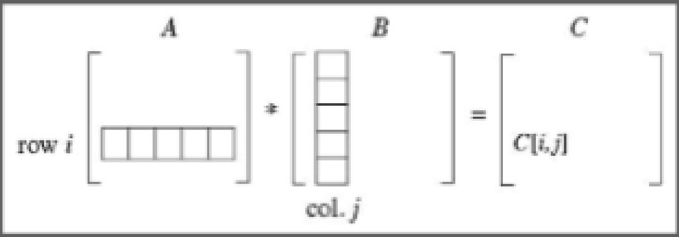
$$= \frac{(n-1)n}{2} \in \theta(n^2)$$

We also could have computed the sum $\sum_{i=0}^{n-2}(n-1-i)$ faster as follows:

$$\sum_{i=0}^{n-2}(n-1-i) = (n-1)+(n-2)+\cdots+1 = \frac{(n-1)n}{2},$$

where the last equality is obtained by applying summation formula (S2). Note that this result was perfectly predictable: in the worst case, the algorithm needs to compare all $n(n-1)/2$ distinct pairs of its n elements.

EXAMPLE 3 Given two $n \times n$ matrices A and B, find the time efficiency of the definition-based algorithm for computing their product $C = AB$. By definition, C is an $n \times n$ matrix whose elements are computed as the scalar (dot) products of the rows of matrix A and the columns of matrix B:



where $C[i, j] = A[i, 0]B[0, j] + \ldots + A[i, k]B[k, j] + \ldots + A[i, n-1]B[n-1, j]$ for every pair of indices $0 \le i, j \le n-1$.

```
ALGORITHM  Matrix Multiplication(A[0..n − 1, 0..n − 1], B[0..n − 1, 0..n − 1])
    //Multiplies two square matrices of order n by the definition-based algorithm
    //Input: Two n × n matrices A and B
    //Output: Matrix C = AB
    for i ←0 to n − 1 do
        for j ←0 to n − 1 do
            C[i, j ]←0.0
            for k←0 to n − 1 do
                C[i, j ]←C[i, j ]+ A[i, k] ∗ B[k, j]
    return C
```

We measure an input's size by matrix order *n*. There are two Arithmetical operations in the innermost loop here—multiplication and addition—that, in principle, can compete for designation as the algorithm's basic operation. Actually, we do not have to choose between them, because on each repetition of The innermost loop each of the two is executed exactly once. So by counting one we automatically count the other. Still, following a well-established tradition, we consider multiplication as the basic operation (see Section 2.1).

Let us set up a sum for the total number of multiplications $M(n)$ executed by the algorithm. (Since this count depends only on the size of the input matrices, we do not have to investigate the worst-case, average-case, and best-case efficiencies separately.)

Obviously, there is just one multiplication executed on each repetition of the algorithm's innermost loop, which is governed by the variable k ranging from the lower bound 0 to the upper bound $n - 1$. Therefore, the number of multiplications made for every pair of specific values of variables i and j is

$$\sum_{k=0}^{n-1} 1,$$

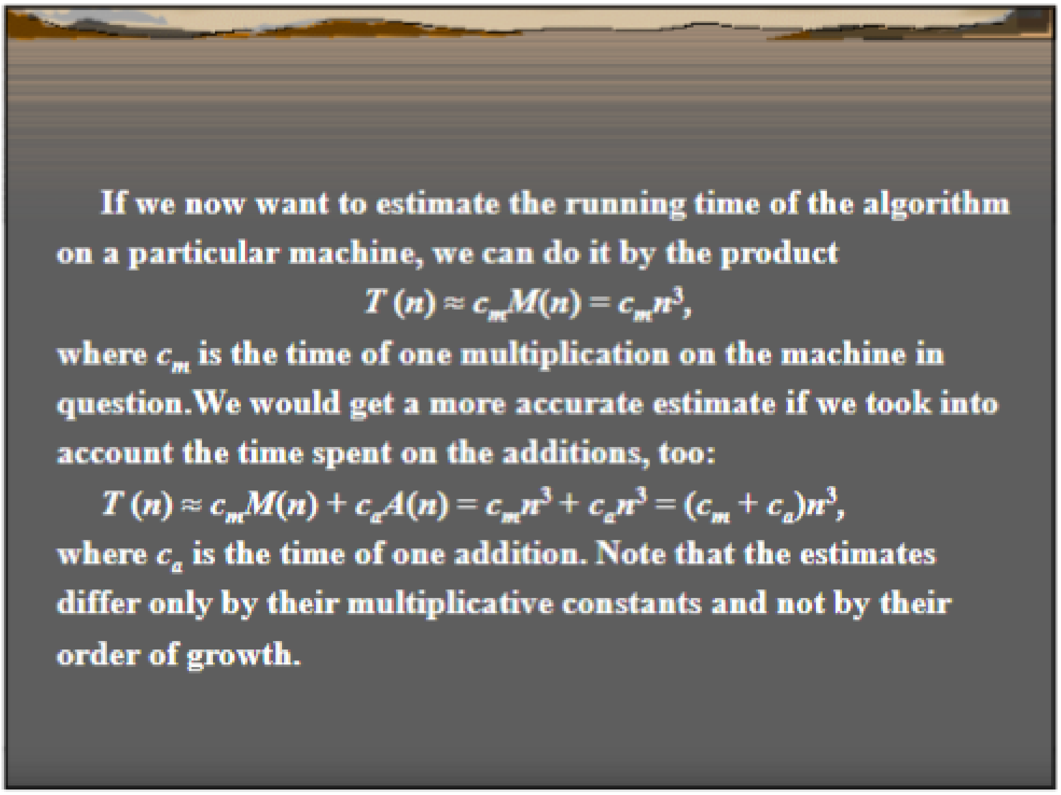
and the total number of multiplications $M(n)$ is expressed by the following triple sum:

$$M(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1.$$

Now, we can compute this sum by using formula (S1) and rule (R1) given above. Starting with the innermost sum $\sum_{k=0}^{n-1} 1$, which is equal to n (why?), we get

$$M(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1 = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} n = \sum_{i=0}^{n-1} n^2 = n^3.$$

This example is simple enough so that we could get this result without all the summation machinations. How? The algorithm computes n^2 elements of the product matrix. Each of the product's elements is computed as the scalar (dot) product of an n-element row of the first matrix and an n-element column of the second matrix, which takes n multiplications. So the total number of multiplications is $n \cdot n^2 = n^3$. (It is this kind of reasoning that we expected you to employ when answering this question in Problem 2 of Exercises 2.1.)



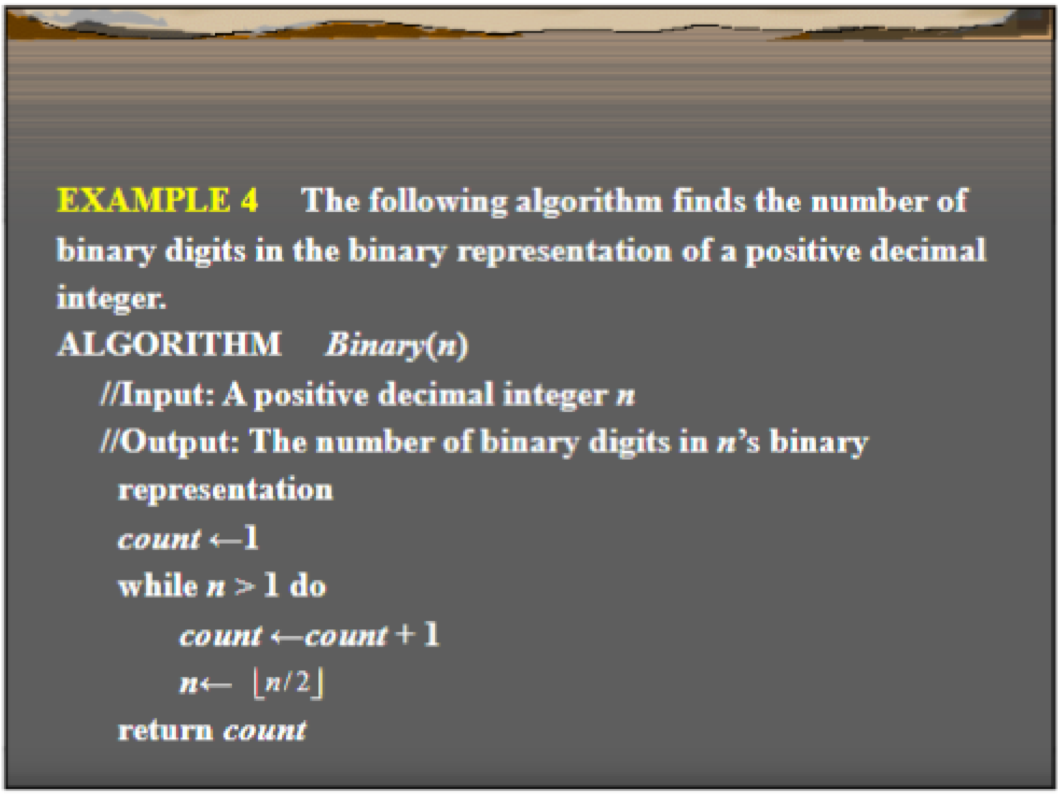
If we now want to estimate the running time of the algorithm on a particular machine, we can do it by the product

$$T(n) \approx c_m M(n) = c_m n^3,$$

where c_m is the time of one multiplication on the machine in question. We would get a more accurate estimate if we took into account the time spent on the additions, too:

$$T(n) \approx c_m M(n) + c_a A(n) = c_m n^3 + c_a n^3 = (c_m + c_a)n^3,$$

where c_a is the time of one addition. Note that the estimates differ only by their multiplicative constants and not by their order of growth.



EXAMPLE 4 The following algorithm finds the number of binary digits in the binary representation of a positive decimal integer.

```
ALGORITHM    Binary(n)
    //Input: A positive decimal integer n
    //Output: The number of binary digits in n's binary
      representation
      count ← 1
      while n > 1 do
          count ← count + 1
          n ← ⌊n/2⌋
      return count
```

First, notice that the most frequently executed operation here is not inside the while loop but rather the comparison $n > 1$ that determines whether the loop's body will be executed. Since the number of times the comparison will be executed is larger than the number of repetitions of the loop's body by exactly 1, the choice is not that important.

A more significant feature of this example is the fact that the loop variable takes on only a few values between its lower and upper limits; therefore, we have to use an alternative way of computing the number of times the loop is executed. Since the value of n is about halved on each repetition of the loop, the answer should be about $\log_2 n$. The exact formula for the number of times the comparison $n>1$ will be executed is actually $\lfloor \log_2 n \rfloor + 1$ —the number of bits in the binary representation of n according to formula (2.1).

We could also get this answer by applying the analysis technique based on recurrence relations; we discuss this technique in the next section because it is more pertinent to the analysis of recursive algorithms.

2.4 Mathematical Analysis of Recursive Algorithms

EXAMPLE 1 Compute the factorial function $F(n) = n!$ for an arbitrary nonnegative integer n. Since

$$n! = 1 \cdot \ldots \cdot (n-1) \cdot n = (n-1)! \cdot n \quad \text{for } n \geq 1$$

and $0! = 1$ by definition, we can compute $F(n) = F(n-1) \cdot n$ with the following recursive algorithm.

```
ALGORITHM    F(n)
    //Computes n! recursively
    //Input: A nonnegative integer n
    //Output: The value of n!
    if $n = 0$ return 1
    else return $F(n - 1) * n$
```

For simplicity, we consider n itself as an indicator of this algorithm's input size (rather than the number of bits in its binary expansion). The basic operation of the algorithm is multiplication,[5] whose number of executions we denote $M(n)$. Since the function $F(n)$ is computed according to the formula

$$F(n) = F(n-1) \cdot n \quad \text{for } n > 0,$$

the number of multiplications $M(n)$ needed to compute it must satisfy the equality

$$M(n) = \underbrace{M(n-1)}_{\substack{\text{to compute}\\ F(n-1)}} + \underbrace{1}_{\substack{\text{to multiply}\\ F(n-1) \text{ by } n}} \quad \text{for } n > 0.$$

Indeed, $M(n-1)$ multiplications are spent to compute $F(n-1)$, and one more multiplication is needed to multiply the result by n.

The last equation defines the sequence $M(n)$ that we need to find. This equation defines $M(n)$ not explicitly, i.e., as a function of n, but implicitly as a function of its value at another point, namely $n - 1$. Such equations are called *recurrence relations* or, for brevity, *recurrences*. Recurrence relations play an important role not only in analysis of algorithms but also in some areas of applied mathematics. They are usually studied in detail in courses on discrete mathematics or discrete structures; a very brief tutorial on them is provided in Appendix B. Our goal now is to solve the recurrence relation $M(n) = M(n-1) + 1$, i.e., to find an explicit formula for $M(n)$ in terms of n only.

Note, however, that there is not one but infinitely many sequences that satisfy this recurrence. (Can you give examples of, say, two of them?) To determine a solution uniquely, we need an *initial condition* that tells us the value with which the sequence starts. We can obtain this value by inspecting the condition that makes the algorithm stop its recursive calls:

if $n = 0$ return 1.

This tells us two things. First, since the calls stop when $n = 0$, the smallest value of n for which this algorithm is executed and hence $M(n)$ defined is 0. Second, by inspecting the pseudocode's exiting line, we can see that when $n = 0$, the algorithm performs no multiplications. Therefore, the initial condition we are after is

Thus, we succeeded in setting up the recurrence relation and initial condition for the algorithm's number of multiplications $M(n)$:

$$M(n) = M(n-1) + 1 \quad \text{for } n > 0, \qquad (2.2)$$
$$M(0) = 0.$$

Before we embark on a discussion of how to solve this recurrence, let us pause to reiterate an important point. We are dealing here with two recursively defined functions. The first is the factorial function $F(n)$ itself; it is defined by the recurrence

$$F(n) = F(n-1) \cdot n \quad \text{for every } n > 0,$$
$$F(0) = 1.$$

The second is the number of multiplications $M(n)$ needed to compute $F(n)$ by the recursive algorithm whose pseudocode was given at the beginning of the section. As we just showed, $M(n)$ is defined by recurrence (2.2). And it is recurrence (2.2) that we need to solve now.

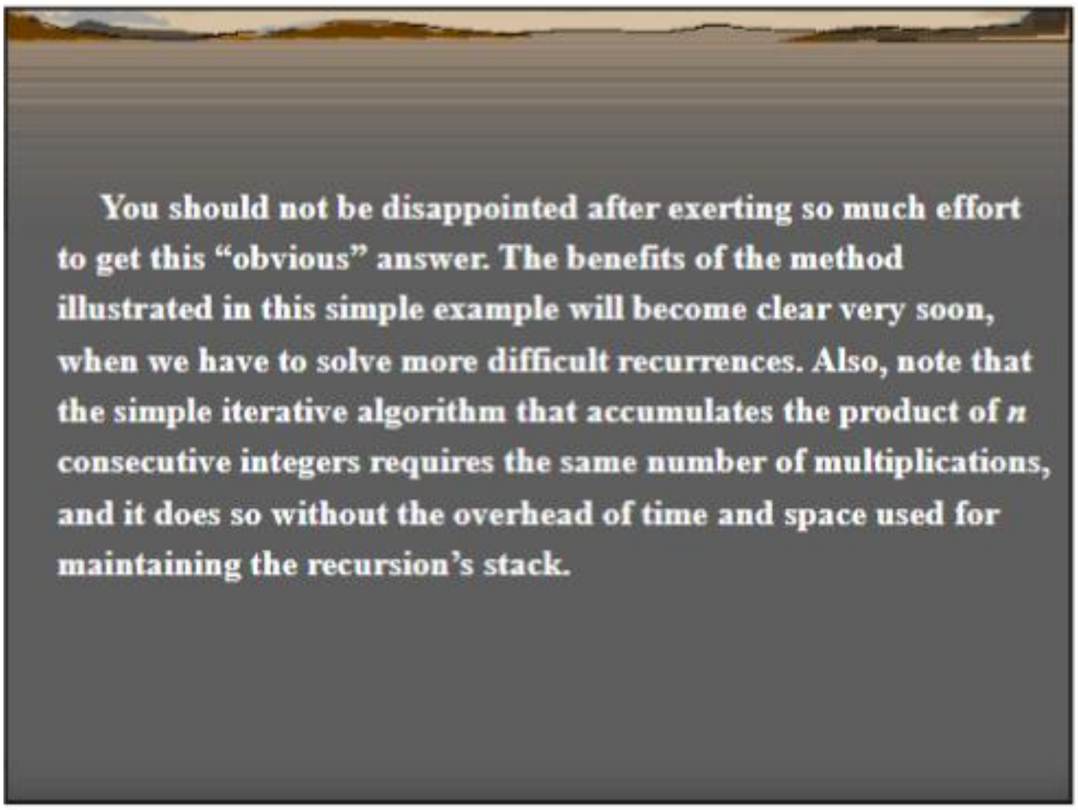
Though it is not difficult to "guess" the solution here (what sequence starts with 0 when $n = 0$ and increases by 1 on each step?), it will be more useful to arrive at it in a systematic fashion. From the several techniques available for solving recurrence relations, we use what can be called the *method of backward substitutions*. The method's idea (and the reason for the name) is immediately clear from the way it applies to solving our particular recurrence:

$$\begin{aligned} M(n) &= M(n-1)+1 && \text{substitute } M(n-1) = M(n-2)+1 \\ &= [M(n-2)+1]+1 = M(n-2)+2 && \text{substitute } M(n-2) = M(n-3)+1 \\ &= [M(n-3)+1]+2 = M(n-3)+3 \end{aligned}$$

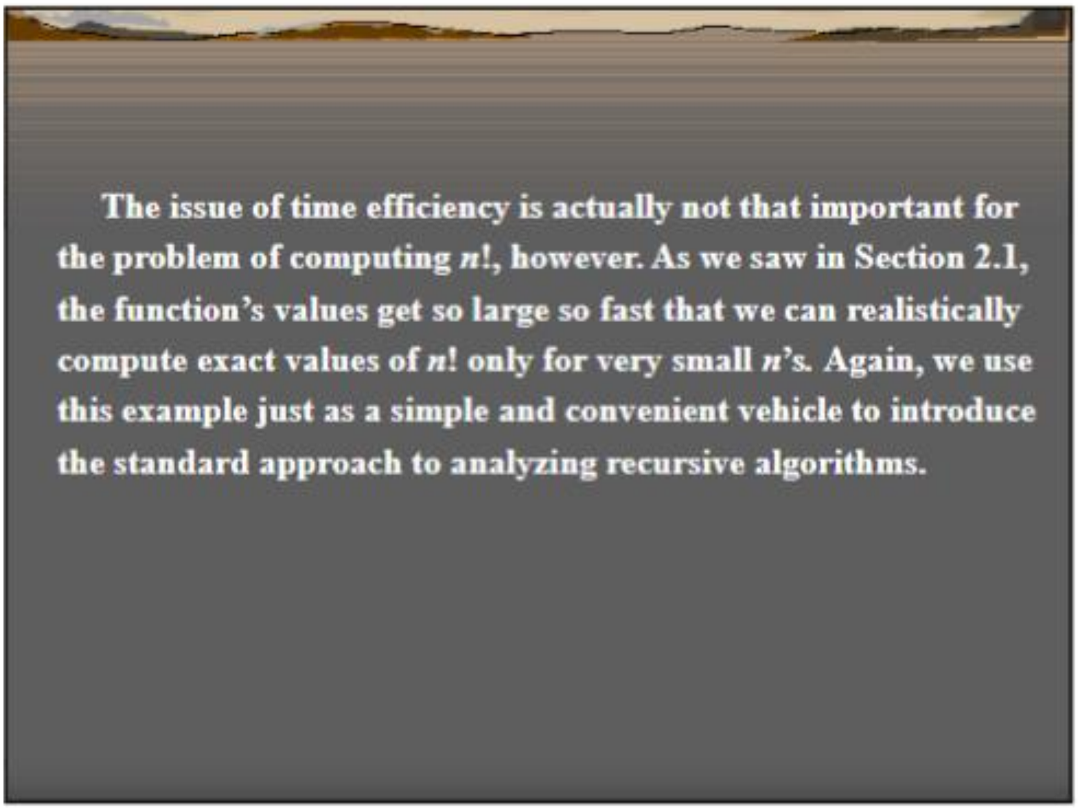
After inspecting the first three lines, we see an emerging pattern, which makes it possible to predict not only the next line (what would it be?) but also a general formula for the pattern: $M(n) = M(n - i) + i$. Strictly speaking, the correctness of this formula should be proved by mathematical induction, but it is easier to get to the solution as follows and then verify its correctness.

What remains to be done is to take advantage of the initial condition given. Since it is specified for $n = 0$, we have to substitute $i = n$ in the pattern's formula to get the ultimate result of our backward substitutions:

$$M(n) = M(n-1) + 1 = \cdots = M(n-i) + i = \cdots = M(n-n) + n = n.$$



You should not be disappointed after exerting so much effort to get this "obvious" answer. The benefits of the method illustrated in this simple example will become clear very soon, when we have to solve more difficult recurrences. Also, note that the simple iterative algorithm that accumulates the product of *n* consecutive integers requires the same number of multiplications, and it does so without the overhead of time and space used for maintaining the recursion's stack.



The issue of time efficiency is actually not that important for the problem of computing *n*!, however. As we saw in Section 2.1, the function's values get so large so fast that we can realistically compute exact values of *n*! only for very small *n*'s. Again, we use this example just as a simple and convenient vehicle to introduce the standard approach to analyzing recursive algorithms.

General Plan for Analyzing the Time Efficiency of Recursive Algorithms

1. Decide on a parameter (or parameters) indicating an input's size.
2. Identify the algorithm's basic operation.
3. Check whether the number of times the basic operation is executed can vary on different inputs of the same size; if it can, the worst-case, average-case, and best-case efficiencies must be investigated separately.

4. Set up a recurrence relation, with an appropriate initial condition, for the number of times the basic operation is executed.

5. Solve the recurrence or, at least, ascertain the order of growth of its solution.

EXAMPLE 2 As our next example, we consider another educational workhorse of recursive algorithms: the *Tower of Hanoi* puzzle. In this puzzle, we (or mythical monks, if you do not like to move disks) have n disks of different sizes that can slide onto any of three pegs. Initially, all the disks are on the first peg in order of size, the largest on the bottom and the smallest on top. The goal is to move all the disks to the third peg, using the second one as an auxiliary, if necessary. We can move only one disk at a time, and it is forbidden to place a larger disk on top of a smaller one.

The problem has an elegant recursive solution, which is illustrated in Figure 2.4. To move $n>1$ disks from peg 1 to peg 3 (with peg 2 as auxiliary), we first move recursively $n-1$ disks from peg 1 to peg 2 (with peg 3 as auxiliary), then move the largest disk directly from peg 1 to peg 3, and, finally, move recursively $n-1$ disks from peg 2 to peg 3 (using peg 1 as auxiliary). Of course, if $n = 1$, we simply move the single disk directly from the source peg to the destination peg.

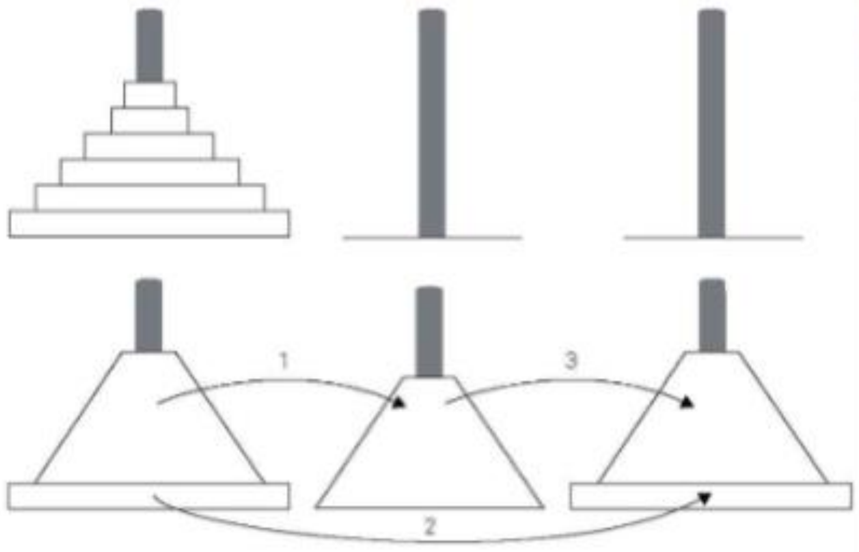


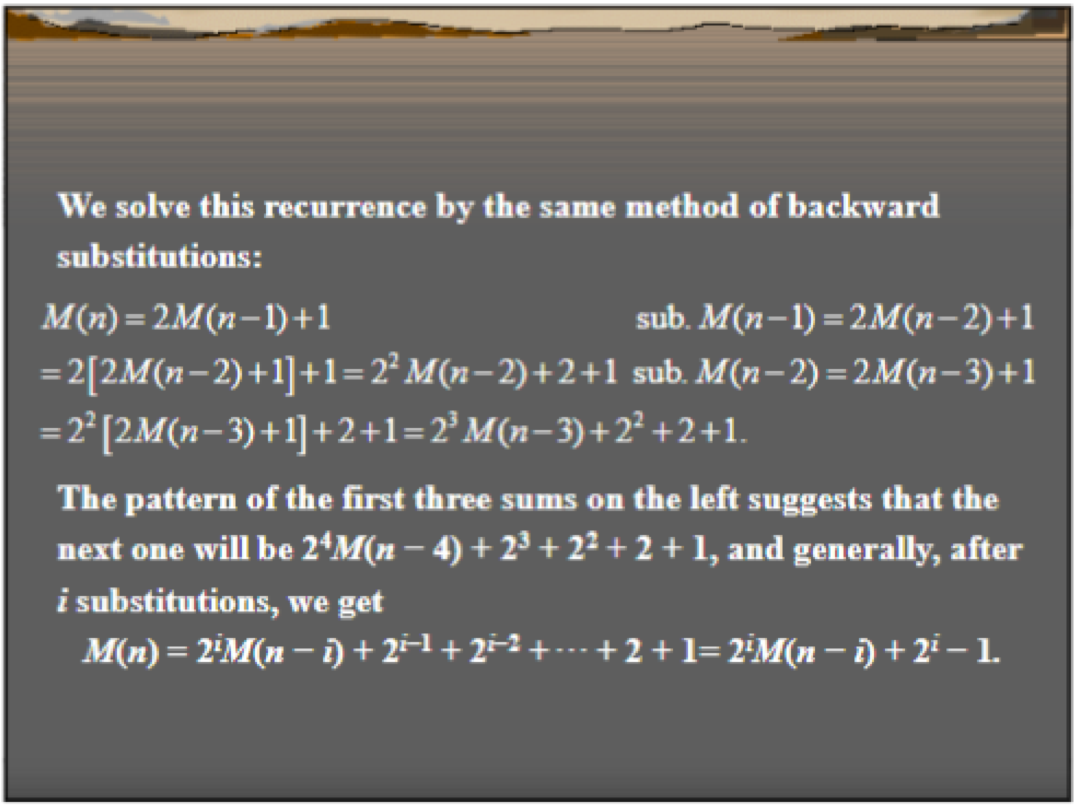
FIGURE 2.4 Recursive solution to the Tower of Hanoi puzzle.

Let us apply the general plan outlined above to the Tower of Hanoi problem. The number of disks *n* is the obvious choice for the input's size indicator, and so is moving one disk as the algorithm's basic operation. Clearly, the number of moves *M*(*n*) depends on *n* only, and we get the following recurrence equation for it:

$$M(n) = M(n-1) + 1 + M(n-1) \quad \text{for } n > 1.$$

With the obvious initial condition *M*(1) = 1, we have the following recurrence relation for the number of moves *M(n)*:

$$M(n) = 2M(n-1) + 1 \quad \text{for } n > 1, \qquad (2.3)$$
$$M(1) = 1.$$



We solve this recurrence by the same method of backward substitutions:

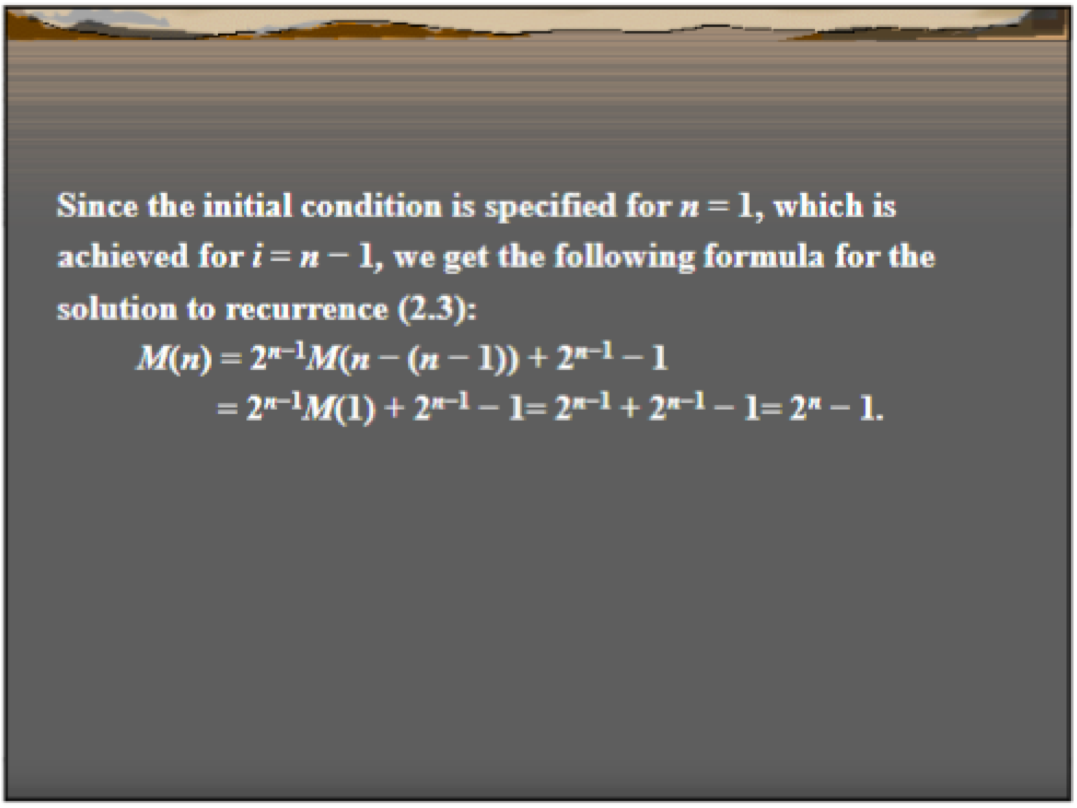
$M(n) = 2M(n-1)+1$ sub. $M(n-1) = 2M(n-2)+1$

$= 2[2M(n-2)+1]+1 = 2^2 M(n-2)+2+1$ sub. $M(n-2) = 2M(n-3)+1$

$= 2^2[2M(n-3)+1]+2+1 = 2^3 M(n-3)+2^2+2+1.$

The pattern of the first three sums on the left suggests that the next one will be $2^4M(n-4) + 2^3 + 2^2 + 2 + 1$, and generally, after i substitutions, we get

$$M(n) = 2^iM(n-i) + 2^{i-1} + 2^{i-2} + \cdots + 2 + 1 = 2^iM(n-i) + 2^i - 1.$$



Since the initial condition is specified for $n = 1$, which is achieved for $i = n - 1$, we get the following formula for the solution to recurrence (2.3):

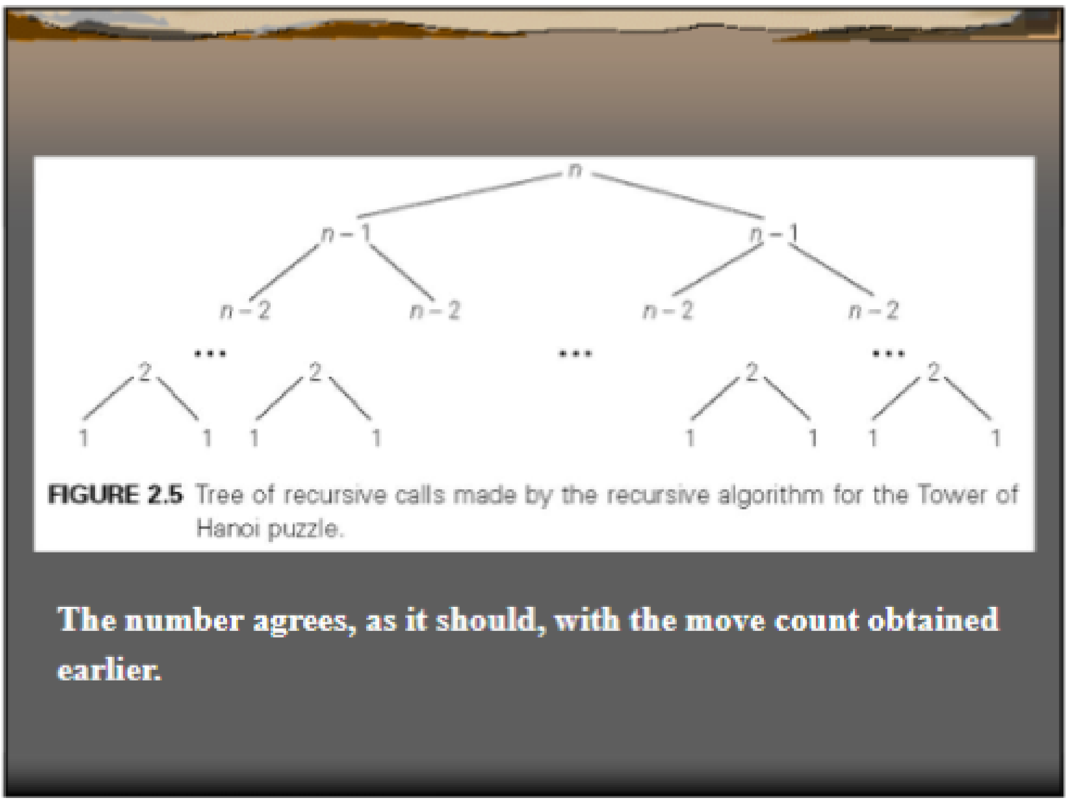
$$M(n) = 2^{n-1}M(n-(n-1)) + 2^{n-1} - 1$$
$$= 2^{n-1}M(1) + 2^{n-1} - 1 = 2^{n-1} + 2^{n-1} - 1 = 2^n - 1.$$

Thus, we have an exponential algorithm, which will run for an unimaginably long time even for moderate values of *n* (see Problem 5 in this section's exercises). This is not due to the fact that this particular algorithm is poor; in fact, it is not difficult to prove that this is the most efficient algorithm possible for this problem. It is the problem's intrinsic difficulty that makes it so computationally hard. Still, this example makes an important general point:

One should be careful with recursive algorithms because their succinctness may mask their inefficiency.

When a recursive algorithm makes more than a single call to itself, it can be useful for analysis purposes to construct a tree of its recursive calls. In this tree, nodes correspond to recursive calls, and we can label them with the value of the parameter (or, more generally, parameters) of the calls. For the Tower of Hanoi example, the tree is given in Figure 2.5. By counting the number of nodes in the tree, we can get the total number of calls made by the Tower of Hanoi algorithm:

$$C(n) = \sum_{l=0}^{n-1} 2^l \text{(where } l \text{ is the level in the tree in Figure 2.5)} = 2^n - 1.$$



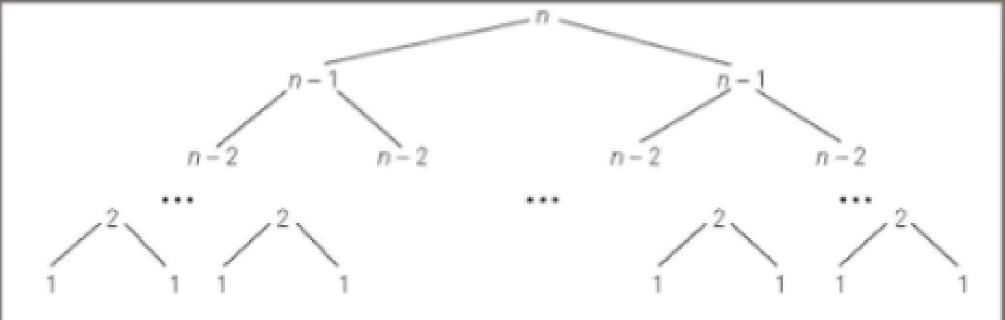
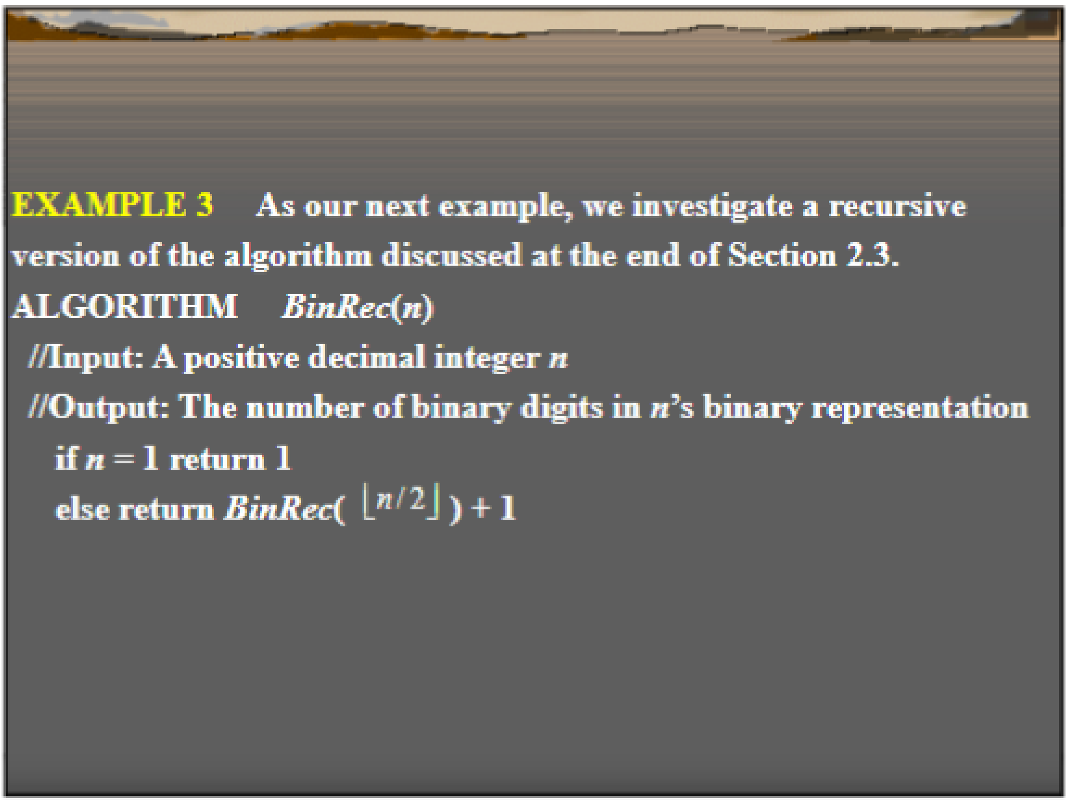


FIGURE 2.5 Tree of recursive calls made by the recursive algorithm for the Tower of Hanoi puzzle.

The number agrees, as it should, with the move count obtained earlier.



EXAMPLE 3 **As our next example, we investigate a recursive version of the algorithm discussed at the end of Section 2.3.**

```
ALGORITHM    BinRec(n)
  //Input: A positive decimal integer n
  //Output: The number of binary digits in n's binary representation
    if n = 1 return 1
    else return BinRec( ⌊n/2⌋ ) + 1
```

Let us set up a recurrence and an initial condition for the number of additions $A(n)$ made by the algorithm. The number of additions made in computing *BinRec*($\lfloor n/2 \rfloor$) is $A(\lfloor n/2 \rfloor)$, plus one more addition is made by the algorithm to increase the returned value by 1. This leads to the recurrence

$$A(n) = A(\lfloor n/2 \rfloor) + 1 \quad \text{for } n > 1. \qquad (2.4)$$

Since the recursive calls end when n is equal to 1 and there are no additions made then, the initial condition is

$$A(1) = 0.$$

The presence of $\lfloor n/2 \rfloor$ in the function's argument makes the method of backward substitutions stumble on values of n that are not powers of 2. Therefore, the standard approach to solving such a recurrence is to solve it only for $n = 2^k$ and then take advantage of the theorem called the *smoothness rule* (see Appendix B), which claims that under very broad assumptions the order of growth observed for $n = 2^k$ gives a correct answer about the order of growth for all values of n. (Alternatively, after getting a solution for powers of 2, we can sometimes fine-tune This solution to get a formula valid for an arbitrary n.) So let us apply this recipe to our recurrence, which for $n = 2^k$ takes the form

$$A(2^k) = A(2^{k-1}) + 1 \quad \text{for } k > 0,$$
$$A(2^0) = 0.$$

Now backward substitutions encounter no problems:

$$\begin{aligned}
A(2^k) &= A(2^{k-1}) + 1 && \text{substitute } A(2^{k-1}) = A(2^{k-2}) + 1 \\
&= [A(2^{k-2}) + 1] + 1 = A(2^{k-1}) + 2 && \text{substitute } A(2^{k-2}) = A(2^{k-3}) + 1 \\
&= [A(2^{k-3}) + 1] + 2 = A(2^{k-3}) + 3 && \cdots \\
&\cdots \\
&= A(2^{k-i}) + i \\
&\cdots \\
&= A(2^{k-k}) + k.
\end{aligned}$$

Thus, we end up with

$$A(2^k) = A(1) + k = k,$$

or, after returning to the original variable $n = 2^k$ and hence $k = \log_2 n$,

$$A(n) = \log_2 n \in \Theta(\log n).$$

In fact, one can prove (Problem 7 in this section's exercises) that the exact solution for an arbitrary value of n is given by just a slightly more refined formula $A(n) = \lfloor \log_2 n \rfloor$