UNIT I INTRODUCTION

9

Notion of an Algorithm – Fundamentals of Algorithmic Problem Solving – Important Problem Types – Fundamentals of the Analysis of Algorithmic Efficiency –Analysis Framework – Asymptotic Notations and Basic Efficiency Classes-Empirical analysis - Mathematical analysis for Recursive and Non-recursive algorithms - Visualization

Need for studying algorithms:

- The study of algorithms is the cornerstone of computer science.
- It can be recognized as the core of computer science.
- Computer programs would not exist without algorithms.
- With computers becoming an essential part of our professional & personal life's, studying algorithms becomes a necessity, more so for computer science engineers.
- Another reason for studying algorithms is that if we know a standard set of important algorithms ,
- They further our analytical skills & help us in developing new algorithms for required applications

ALGORITHM

An algorithm is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.

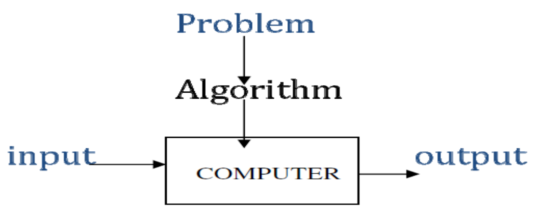
(OR)

An algorithm is finite set of instructions that is followed, accomplishes a particular task.

In addition, all algorithms must satisfy the following criteria:

1. Input. Zero or more quantities are externally supplied.
2. Output. At least one quantity is produced.
3. Definiteness. Each instruction is clear and produced.
4. Finiteness. If we trace out the instruction of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.
5. Effectiveness. Every instruction must be very basic

NOTION OF ALGORITHM



- Algorithms for the same problem can be based on very different ideas and can solve the problem with dramatically different speeds.
- Euclid of Alexandria (third century B.C.) outlined an algorithm for solving this problem in one of the volumes of his *Elements* most famous for its systematic exposition of geometry.
- In modern terms, ***Euclid's algorithm*** is based on applying repeatedly the equality gcd(m, n) = gcd$(n, m \bmod n)$,
- where m mod n is the remainder of the division of m by n, until m mod n is equal to 0. Since gcd$(m, 0) = m$ (why?), the last value of m is also the greatest common divisor of the initial m and n.
- For example, gcd$(60, 24)$ can be computed as follows: gcd$(60, 24)$ = gcd$(24, 12)$ = gcd$(12, 0)$ = 12.

Euclid's algorithm for computing gcd(m, n)

Step 1 If $n = 0$, return the value of m as the answer and stop; otherwise, proceed to Step 2.

Step 2 Divide m by n and assign the value of the remainder to r.

Step 3 Assign the value of n to m and the value of r to n. Go to Step 1.

Alternatively, we can express the same algorithm in pseudocode:

```
ALGORITHM    Euclid(m, n)
//Computes gcd(m, n) by Euclid's algorithm
//Input: Two nonnegative, not-both-zero integers m and n //Output: Greatest common divisor of m and n
while n  = 0 do
r ← m mod n m ← n
n ← r return m
```

Consecutive integer checking algorithm for computing gcd(m, n)

Step 1 Assign the value of min$\{m, n\}$ to t.

Step 2 Divide m by t. If the remainder of this division is 0, go to Step 3; otherwise,

go to Step 4.

Step 3 Divide n by t. If the remainder of this division is 0, return the value of t as the answer and stop; otherwise, proceed to Step 4.

Step 4 Decrease the value of t by 1. Go to Step 2

The third procedure for finding the greatest common divisor should be familiar to you from middle school.

Middle-school procedure for computing gcd(m, n)

Step 1 Find the prime factors of m. **Step 2** Find the prime factors of n.

Step 3 Identify all the common factors in the two prime expansions found in Step 1 and Step 2. (If p is a common factor occurring p_m and p_n times in m and n, respectively, it should be repeated min$\{p_m, p_n\}$ times.)

Step 4 Compute the product of all the common factors and return it as the greatest common divisor of the numbers given.

Thus, for the numbers 60 and 24, we get

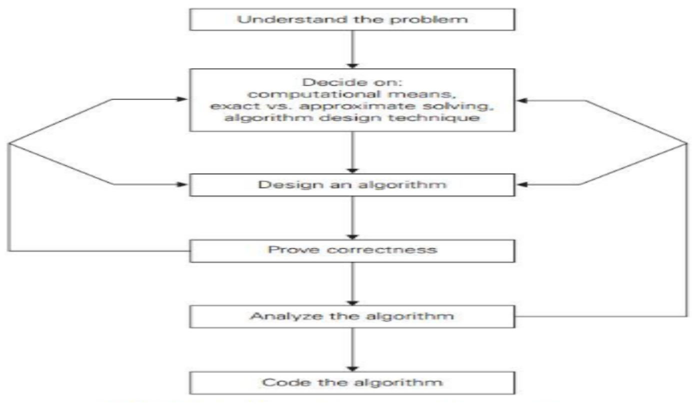
$60 = 2 \cdot 2 \cdot 3 \cdot 5$

$24 = 2 \cdot 2 \cdot 2 \cdot 3$ gcd$(60, 24) = 2 \cdot 2 \cdot 3 = 12$.

FUNDAMENTALS OF ALGORITHMIC PROBLEM SOLVING

Understanding the Problem

An input to an algorithm specifies an ***instance*** of the problem the algorithm solves. It is very important to specify exactly the set of instances the algorithm needs to handle.



Ascertaining the Capabilities of the Computational Device

- Once you completely understand a problem, you need to ascertain the capabilities of the computational device the algorithm is intended for.
- The vast majority of algorithms in use today are still destined to be programmed for a computer closely resembling the von Neumann machine—a computer architecture outlined by the prominent Hungarian-American mathematician John von Neumann (1903– 1957)
- The essence of this architecture is captured by the so-called ***random-access machine*** (***RAM***). Its central assumption is that instructions are executed one after another, one operation at a time.
- Accordingly, algorithms designed to be executed on such machines are called ***sequential algorithms***.
- The central assumption of the RAM model does not hold for some newer computers that can execute operations concurrently, i.e., in parallel. Algorithms that take advantage of this capability are called ***parallel algorithms***.

Choosing between Exact and Approximate Problem Solving

- The next principal decision is to choose between solving the problem exactly or solving it approximately.
- In the former case, an algorithm is called an ***exact algorithm***; in the latter case, an algorithm is called an ***approximation algorithm***.

Algorithm Design Techniques

- Now, with all the components of the algorithmic problem solving in place, how do you design an algorithm to solve a given problem?

What is an algorithm design technique?

- An ***algorithm design technique*** (or "strategy" or "paradigm") is a general approach to solving problems algorithmically that is applicable to a variety of problems from different areas of computing.

Designing an Algorithm and Data Structures

- While the algorithm design techniques do provide a powerful set of general approaches to algorithmic problem solving, designing an algorithm for a particular problem may still be a challenging task.

Algorithms + Data Structures = Programs

Methods of Specifying an Algorithm

- Once you have designed an algorithm, you need to specify it in some fashion.
- Using a natural language has an obvious appeal; however, the inherent ambiguity of any natural language makes a succinct and clear description of algorithms surprisingly difficult.
- ***Pseudocode*** is a mixture of a natural language and programming language-like constructs.
- Pseudocode is usually more precise than natural language, and its usage often yields more succinct algorithm descriptions.
- In the earlier days of computing, the dominant vehicle for specifying algorithms was a ***flowchart***, a method of expressing an algorithm by a collection of connected geometric shapes containing descriptions of the algorithm's steps.

Proving an Algorithm's Correctness

- Once an algorithm has been specified, you have to prove its ***correctness***.
- A common technique for proving correctness is to use mathematical induction because an algorithm's iterations provide a natural sequence of steps needed for such proofs.

Analyzing an Algorithm

- After correctness, by far the most important is ***efficiency***.
- In fact, there are two kinds of algorithm efficiency: ***time efficiency***, indicating how fast the algorithm runs, and ***space efficiency***, indicating how much extra memory it uses.
- Another desirable characteristic of an algorithm is ***simplicity***.
- Yet another desirable characteristic of an algorithm is ***generality***. There are, in fact, two issues here: generality of the problem the algorithm solves and the set of inputs it accepts.

Coding an Algorithm

- Most algorithms are destined to be ultimately implemented as computer pro-grams. Programming an algorithm presents both a peril and an opportunity.
- As a practical matter, the validity of programs is still established by testing.
- Testing of computer programs is an art rather than a science, but that does not mean that there is nothing in it to learn.

IMPORTANT PROBLEM TYPES

- Sorting
- Searching
- String processing
- Graph problems
- Combinatorial problems
- Geometric problems
- Numerical problems

Sorting

- The ***sorting problem*** is to rearrange the items of a given list in non-decreasing order.
- Of course, for this problem to be meaningful, the nature of the list items must allow such an ordering.
- For example, we can choose to sort student records in alphabetical order of names or by student number or by student grade-point average. Such a specially chosen piece of information is called a ***key***.
- Two properties of sorting algorithms deserve special mention.

- A sorting algorithm is called ***stable*** if it preserves the relative order of any two equal elements in its input. In other words, if an input list contains two equal elements in positions ***i*** and ***j*** where ***i < j,*** then in the sorted list they have to be in positions ***i*** and ***j*** ,respectively, such that ***i < j .***
- The second notable feature of a sorting algorithm is the amount of extra memory the algorithm requires.
- An algorithm is said to be ***in-place*** if it does not require extra memory, except, possibly, for a few memory units.

Searching

- The ***searching problem*** deals with finding a given value, called a ***search key***, in a given set (or a multiset, which permits several elements to have the same value).

String Processing

- A ***string*** is a sequence of characters from an alphabet.
- Strings of particular interest are text strings, which comprise letters, numbers, and special characters; bit strings, which comprise zeros and ones; and gene sequences, which can be modelled by strings of characters from the four-character alphabet {A, C, G, T}.
- One particular problem—that of searching for a given word in a text—has attracted special attention from researchers. They call it ***string matching***.

Graph Problems

- One of the oldest and most interesting areas in algorithmics is graph algorithms.
- Informally, a ***graph*** can be thought of as a collection of points called vertices, some of which are connected by line segments called edges.

Combinatorial Problems

- From a more abstract perspective, the traveling salesman problem and the graph-coloring problem are examples of ***combinatorial problems***.
- These are problems that ask, explicitly or implicitly, to find a combinatorial object—such as a permutation, a combination, or a subset—that satisfies certain constraints.

Geometric Problems

- ***Geometric algorithms*** deal with geometric objects such as points, lines, and polygons.
- Two classic problems of computational geometry: the closest-pair problem and the convex-hull problem.
- The ***closest-pair problem*** is self-explanatory: given ***n*** points in the plane, find the closest pair among them.
- The ***convex-hull problem*** asks to find the smallest convex polygon that would include all the points of a given set.

Numerical Problems

- ***Numerical problems***, another large special area of applications, are problems that involve mathematical objects of continuous nature: solving equations and systems of equations, computing definite integrals, evaluating functions, and so on.

FUNDAMENTALS OF THE ANALYSIS OF ALGORITHMIC EFFICIENCY

THE ANALYSIS FRAMEWORK

1. Measuring an Input's Size

2. Units for Measuring Running Time

3. Orders of Growth

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

- ***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

- Almost all algorithms run longer on larger inputs.
- For example, it takes longer to sort larger arrays, multiply larger matrices, and so on.
- Therefore, it is logical to investigate an algorithm's efficiency as a function of some parameter ***n*** indicating the algorithm's input size.
- For example, it will be the size of the list for problems of sorting, searching, finding the list's smallest element, and most other problems dealing with lists.

- For the problem of evaluating a polynomial $p(x) = a_n x^n + \cdots + a_0$ of degree n, it will be the polynomial's degree or the number of its coefficients, which is larger by 1 than its degree.
- For example, how should we measure an input's size for a spell-checking algorithm? If the algorithm examines individual characters of its input, we should measure the size by the number of characters; if it works by processing words, we should count their number in the input.
- In such situations, it is preferable to measure size by the number b of bits in the n's binary representation:

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

Units for Measuring Running Time

- The next issue concerns units for measuring an algorithm's running time.
- Of course, we can simply use some standard unit of time measurement—a second, or millisecond, and so on—to measure the running time of a program implementing the algorithm.

There are obvious drawbacks to such an approach, however:

1. dependence on the speed of a particular computer
2. dependence on the quality of a program implementing the algorithm and of the compiler used in generating the machine code, and
3. the difficulty of clocking the actual running time of the pro-gram.

- One possible approach is to count the number of times each of the algorithm's operations is executed.
- The thing to do is to identify the most important operation of the algorithm, called the ***basic operation***, the operation contributing the most to the total running time, and compute the number of times the basic operation is executed.
- 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

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

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

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}		

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

- Consider, as an example, sequential search.
- This is a straightforward algorithm that searches for a given item (some search key ***K***) in a list of ***n*** elements by checking successive elements of the list until either a match with the search key is found or the list is exhausted.
- Here is the algorithm's pseudocode, in which, for simplicity, a list is implemented as an array. It also assumes that the second condition ***X***[***i***] = ***K*** will not be checked if the first one, which checks that the array's index does not exceed its upper bound, fails.

ALGORITHM *SequentialSearch(X*[1…..n]*, K)*

//Searches for a given value in a given array by sequential search //Input: An array $X[1\ldots n]$ 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

```
for i← 1 to n do
if(X[i]==K)
return i
else
return −1
```

REFER NOTES FOR BEST CASE ,WORST CASE and AVERAGE CASE

$$\begin{aligned} C_{avg}(n) &= [1\cdot\frac{p}{n} + 2\cdot\frac{p}{n} + \cdots + i\cdot\frac{p}{n} + \cdots + n\cdot\frac{p}{n}] + n\cdot(1-p) \\ &= \frac{p}{n}[1 + 2 + \cdots + i + \cdots + n] + n(1-p) \\ &= \frac{p}{n}\frac{n(n+1)}{2} + n(1-p) = \frac{p(n+1)}{2} + n(1-p). \end{aligned}$$

- For example, if $p = 1$ (the search must be successful), the average number of key comparisons made by sequential search is $(n + 1)/2$; that is, the algorithm will inspect, on average, about half of the list's elements.
- If $p = 0$ (the search must be unsuccessful), the average number of key comparisons will be n because the algorithm will inspect all n elements on all such inputs.

ASYMPTOTIC NOTATIONS AND BASIC EFFICIENCY CLASSE

- ***Big O-notation***
- ***Big Omega -notation***
- ***Big 0-notation***
- ***Useful Property Involving the Asymptotic Notations***
- ***Using Limits for Comparing Orders of Growth***
- ***Basic Efficiency Classes***

***O*-notation**

DEFINITION A function ***t (n)*** is said to be in ***O(g(n))***, denoted ***t (n)*** ∈ ***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

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

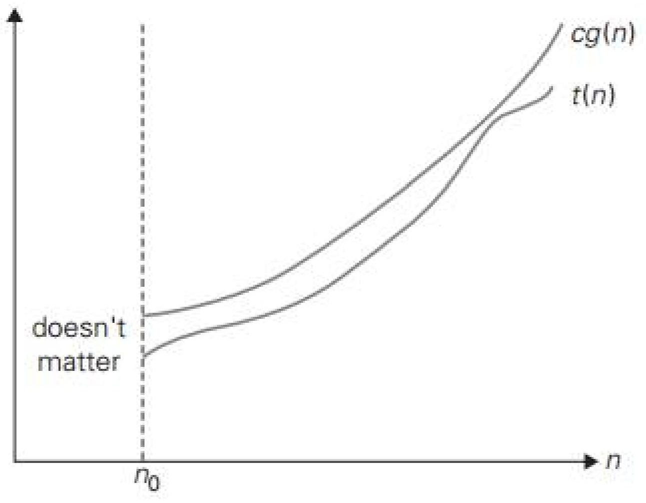


FIGURE 2.1 Big-oh notation: $t(n) \in O(g(n))$.

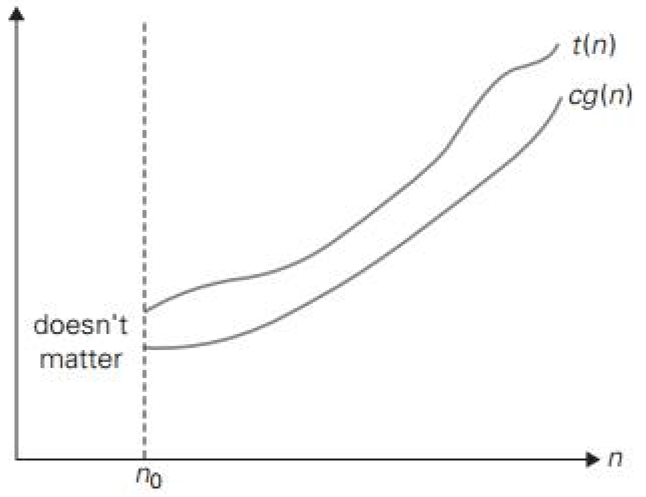


FIGURE 2.2 Big-omega notation: $t(n) \in \Omega(g(n))$.

Ω-notation

DEFINITION A function ***t (n)*** is said to be in ***(g(n))***, denoted ***t (n)*** ∈ ***(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

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

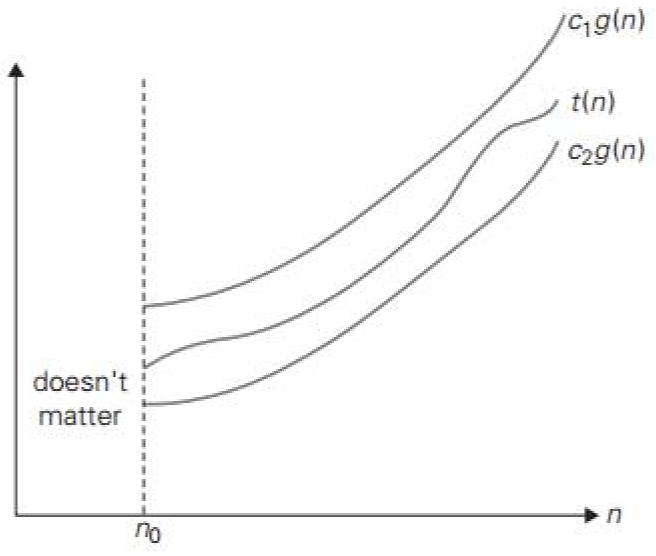


FIGURE 2.3 Big-theta notation: $t(n) \in \Theta(g(n))$.

Θ-notation

DEFINITION A function ***t (n)*** is said to be in ***(g(n))***, denoted ***t (n)*** ∈ ***(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

$c_2g(n) \leq t(n) \leq c_1g(n)$ for all $n \geq n_0$.

USEFUL PROPERTY INVOLVING THE ASYMPTOTIC NOTATIONS

The following property, in particular, is useful in analyzing algorithms that comprise two consecutively executed parts.

THEOREM

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

PROOF

The proof extends to orders of growth the following simple fact about four arbitrary real numbers a_1, b_1, 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 non-negative integer n_1 such that

$t_1(n) \le c_1 g_1(n)$ for all $n \ge n_1$.

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

$t_2(n) \le c_2 g_2(n)$ 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:

$t_1(n) + t_2(n) \le c_1 g_1(n) + c_2 g_2(n)$

$c_3 g_1(n) + c_3 g_2(n) = c_3[g_1(n) + g_2(n)]$

$c_3 2 \max\{g_1(n), g_2(n)\}$.

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.

It implies that the algorithm's overall efficiency is deter-mined by the part with a higher order of growth, i.e., its least efficient part:

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

If, for example, a sorting algorithm used in the first part makes no more than $\frac{1}{2} n(n-1)$ comparisons (and hence is in $O(n^2)$) while the second part makes no more than $n - 1$ comparisons (and hence is in $O(n)$), the efficiency of the entire algorithm will be in $O(\max\{n^2, n\}) = O(n^2)$.

USING LIMITS FOR COMPARING ORDERS OF GROWTH

- Though the formal definitions of O, Θ, and Ω 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:

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

Note that the first two cases mean that $t(n) \in O(g(n))$, the last two mean that $t(n) \in \Omega(g(n))$, and the second case means that $t(n) \in \Theta(g(n))$.

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'Hôpital's rule

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

and Stirling's formula

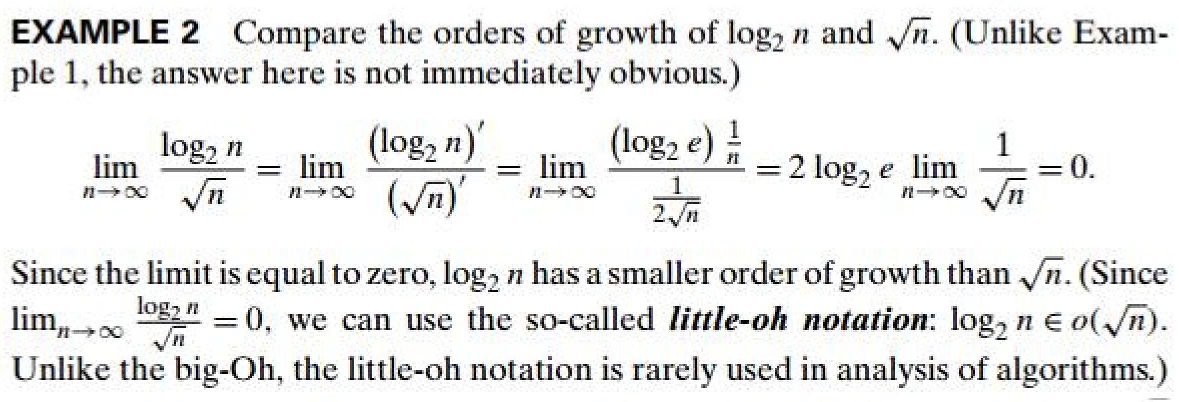
$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \quad \text{for large values of } n.$$

Here are three examples of using the limit-based approach to comparing orders of growth of two functions.

EXAMPLE 1 *Compare the orders of growth of* $\frac{1}{2}$***n(n*** *− 1)* ***and*** n^2.

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

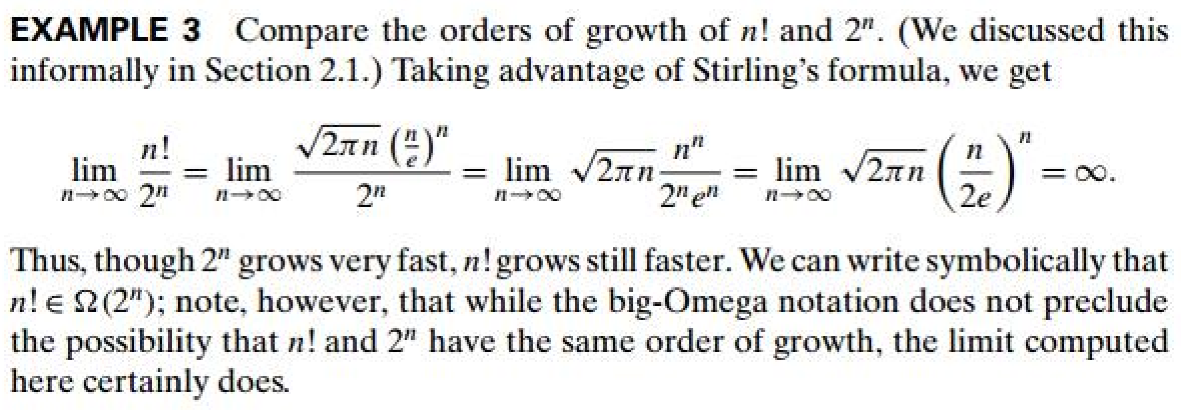
Since the limit is equal to a positive constant, the functions have the same order of growth or, symbolically, $\frac{1}{2}n(n-1) \in \Theta(n^2)$.



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

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

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***: $\log_2 n \in o(\sqrt{n})$. Unlike the big-Oh, the little-oh notation is rarely used in analysis of algorithms.)



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.

MATHEMATICAL ANALYSIS OF NON RECURSIVE ALGORITHMS

GENERAL PLAN FOR ANALYZING THE TIME EFFICIENCY OF NONRECURSIVE ALGORITHMS

- Decide on a parameter (or parameters) indicating an input's size.
- Identify the algorithm's basic operation. (As a rule, it is located in the inner-most loop.)
- 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.
- Set up a sum expressing the number of times the algorithm's basic operation is executed.
- 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.

$$\sum_{i=l}^{u} ca_i = c\sum_{i=l}^{u} a_i, \qquad \textbf{(R1)}$$

$$\sum_{i=l}^{u} (a_i \pm b_i) = \sum_{i=l}^{u} a_i \pm \sum_{i=l}^{u} b_i, \qquad \textbf{(R2)}$$

and two summation formulas

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

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

Note that the formula $\sum_{i=1}^{n-1} 1 = n - 1$, which we used in Example 1, is a special case of formula (S1) for $l = 1$ and $u = n - 1$.

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.

- There are two operations in the loop's body: the comparison A[i] > maxval and the assignment maxval ← 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.
- Let us denote C(n) the number of times this comparison is executed and try to find a formula expressing it as a function of size n .

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

This is an easy sum to compute because it is nothing other than 1 repeated n − 1 times. Thus,

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

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.

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

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

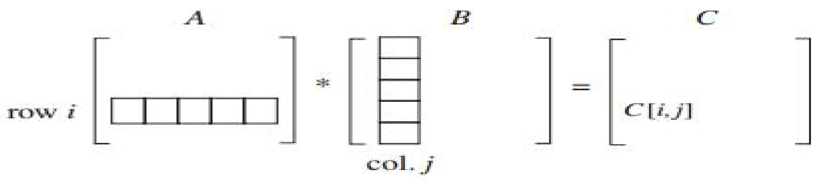
$$= (n-1)^2 - \frac{(n-2)(n-1)}{2} = \frac{(n-1)n}{2} \approx \frac{1}{2} n^2 \in \Theta(n^2).$$

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

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

EXAMP

LE 3 Given two n × 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 × n matrix whose elements are computed as the scalar (dot) products of the rows of matrix A and the columns of matrix B:



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

```
ALGORITHM  MatrixMultiplication(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
- Let us set up a sum for the total number of multiplications M(n) executed by the algorithm.

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

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

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

- 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 cm 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 ca is the time of one addition.

MATHEMATICAL ANALYSIS OF RECURSIVE ALGORITHMS

GENERAL PLAN FOR ANALYZING THE TIME EFFICIENCY OF RECURSIVE ALGORITHMS

- Decide on a parameter (or parameters) indicating an input’s size.
- Identify the algorithm’s basic operation.

- 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.
- Set up a recurrence relation, with an appropriate initial condition, for the number of times the basic operation is executed.
- Solve the recurrence or, at least, ascertain the order of growth of its solution.

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

$n! = 1 \cdot \ldots \cdot (n-1) \cdot n = (n-1)! \cdot n$ 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
```

- we consider n itself as an indicator of this algorithm's input size
- The basic operation of the algorithm is multiplication,whose number of executions we denote $M(n)$.
- Since the function $F(n)$ is computed according to the formula
  $F(n) = F(n-1) \cdot n$ for $n > 0$,
- The number of multiplications $M(n)$ needed to compute it must satisfy the equality

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

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

- 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.

- The initial condition is

$$\underset{\text{the calls stop when } n = 0 \longrightarrow \uparrow}{M(0)} = \underset{\uparrow \longleftarrow \text{ no multiplications when } n = 0}{0.}$$

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

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

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

- From the several techniques available for solving recurrence relations, we use what can be called the ***method of backward substitutions***.

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

- The general formula for the pattern: $\boldsymbol{M(n) = M(n - i) + i}$.
- Since it is specified for $\boldsymbol{n} = 0$, we have to substitute $\boldsymbol{i} = \boldsymbol{n}$ in the pattern's formula to get the ultimate result of our backward substitutions:

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

EXAMPLE 2 ***Tower of Hanoi*** puzzle.

- In this puzzle, we have $\boldsymbol{n}$ 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.
- To move $\boldsymbol{n} > 1$ disks from peg 1 to peg 3 (with peg 2 as auxiliary), we first move recursively $\boldsymbol{n} - 1$ 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 $\boldsymbol{n} - 1$ disks from peg 2 to peg 3 (using peg 1 as auxiliary).
- Of course, if $\boldsymbol{n} = 1$, we simply move the single disk directly from the source peg to the destination peg.

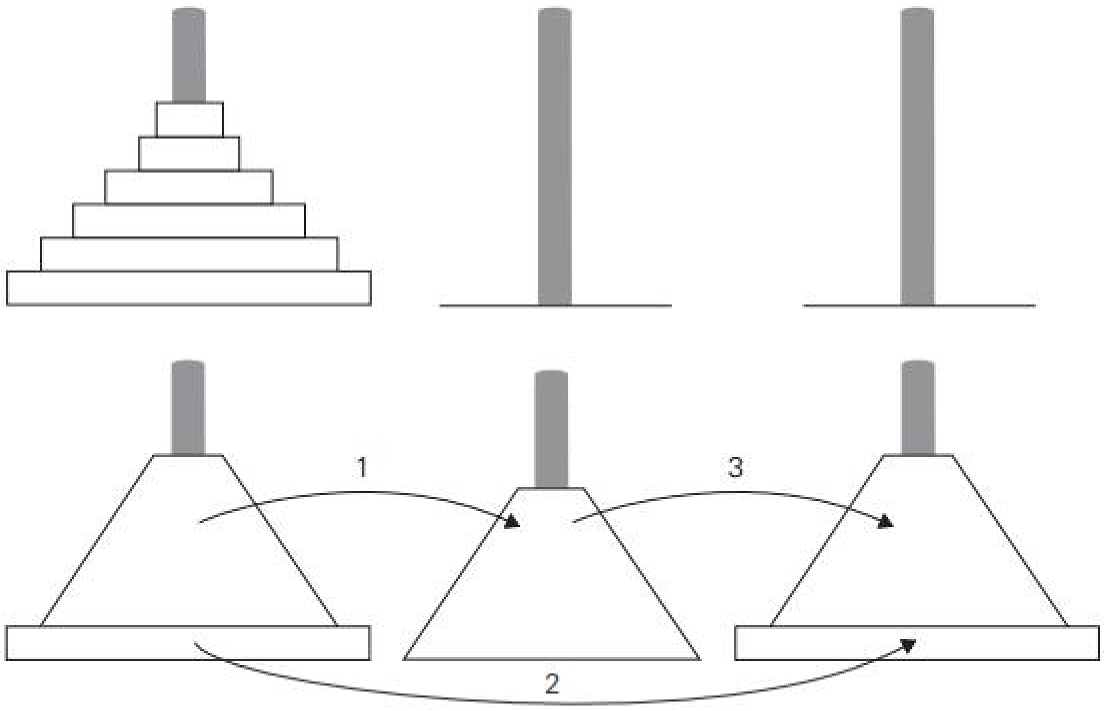


FIGURE 2.4 Recursive solution to the Tower of Hanoi puzzle.

- 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 $M(n)$:

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

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

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

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):

$$\begin{aligned} M(n) &= 2^{n-1}M(n-(n-1)) + 2^{n-1} - 1 \\ &= 2^{n-1}M(1) + 2^{n-1} - 1 = 2^{n-1} + 2^{n-1} - 1 = 2^n - 1. \end{aligned}$$

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

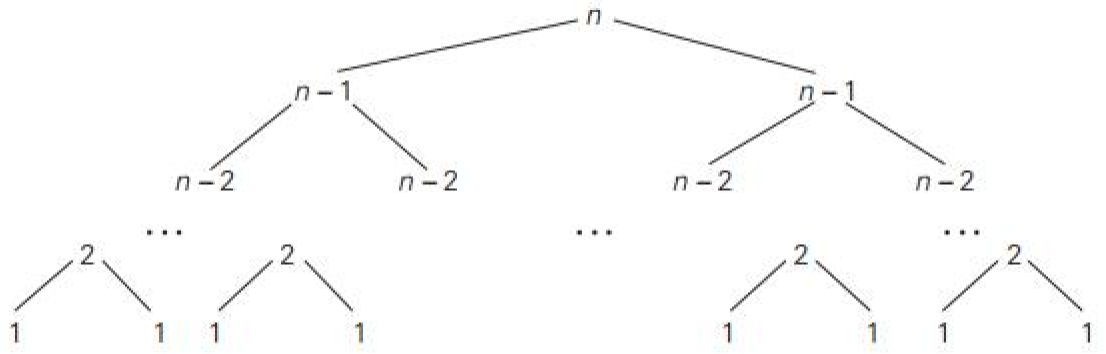


FIGURE 2.5 Tree of recursive calls made by the recursive algorithm for the Tower of Hanoi puzzle.

EXAMPLE 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

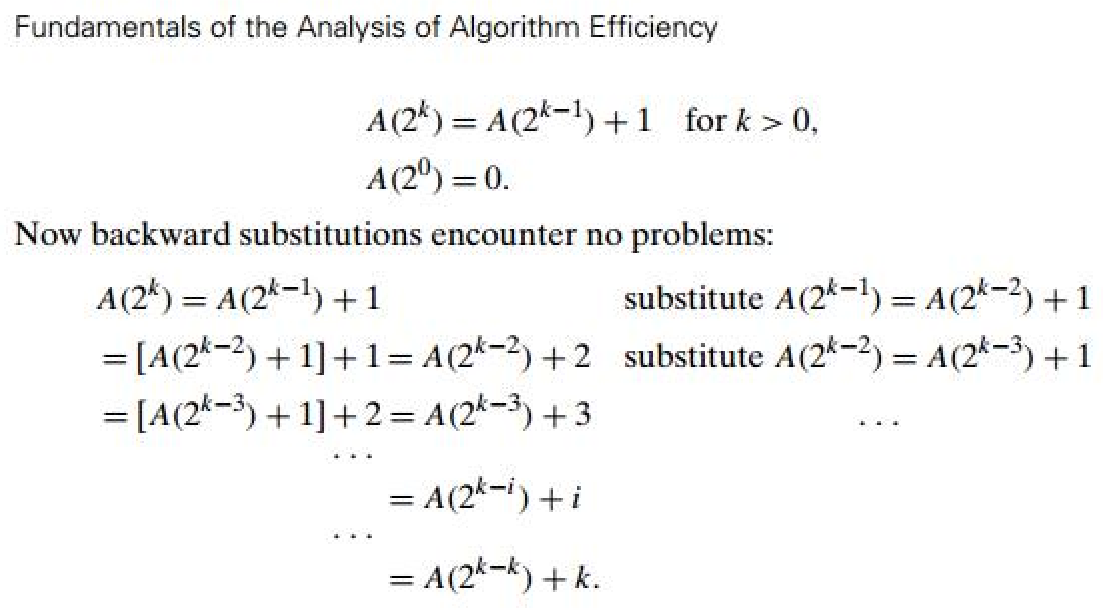
- Let us set up a recurrence and an initial condition for the number of additions ***A(n)*** made by the algorithm.
- The number of additions made in computing *BinRec**(n/*2 ***)*** is ***A(n/*2 *)***, plus one more addition is made by the algorithm to increase the returned value by 1.
- This leads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

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

- 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 ***n/*2** in the function's argument makes the method of back-ward 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



Fundamentals of the Analysis of Algorithm Efficiency

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

Now backward substitutions encounter no problems:

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

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

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

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

$A(n) = \log_2 n$

EMPIRICAL ANALYSIS OF ALGORITHMS

GENERAL PLAN FOR THE EMPIRICAL ANALYSIS OF ALGORITHM TIME EFFICIENCY

1. Understand the experiment's purpose
2. Decide on the efficiency metric M to be measured and the measurement unit (an operation count vs. a time unit).
3. Decide on characteristics of the input sample (its range, size, and so on).
4. Prepare a program implementing the algorithm (or algorithms) for the experimentation.
5. Generate a sample of inputs.
6. Run the algorithm (or algorithms) on the sample's inputs and record the data observed.
7. Analyze the data obtained.
    - The first alternative is to insert a counter (or counters) into a program implementing the algorithm to count the number of times the algorithm's basic operation is executed.
    - This is usually a straightforward operation;
    - The second alternative is to time the program implementing the algorithm in question.
    - The easiest way to do this is to use a system's command, such as the time command in UNIX.
    - Alternatively, one can measure the running time of a code fragment by asking for the system time right before the fragment's start (t_{start}) and just after its completion (t_{finish}), and then computing the difference between the two ($t_{finish} - t_{start}$).

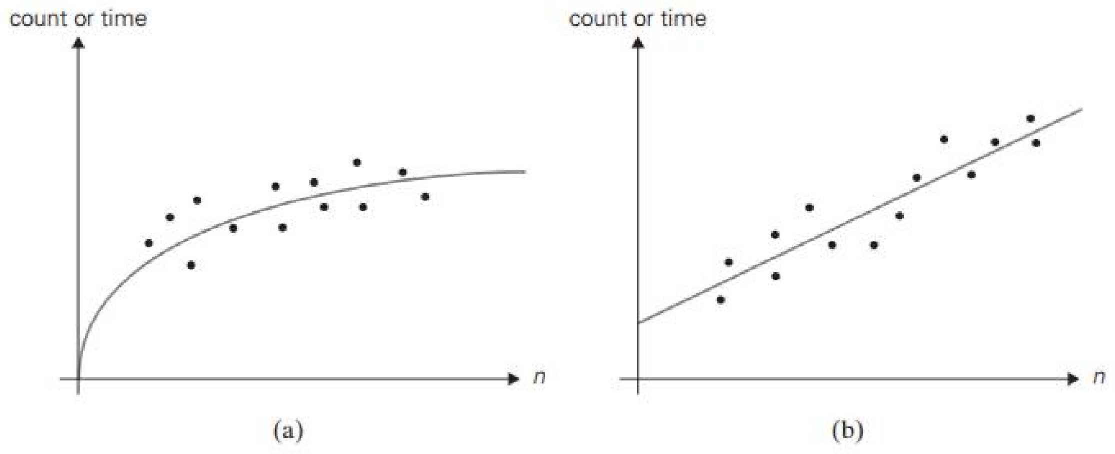
- First, a system's time is typically not very accurate, and you might get somewhat different results on repeated runs of the same program on the same inputs.
- Second, given the high speed of modern computers, the running time may fail to register at all and be reported as zero.
- The standard trick to overcome this obstacle is to run the program in an extra loop many times, measure the total running time, and then divide it by the number of the loop's repetitions.
- Third, on a computer running under a time-sharing system such as UNIX, the reported time may include the time spent by the CPU on other programs, which obviously defeats the purpose of the experiment.
- Thus, measuring the physical running time has several disadvantages, both principal and technical, not shared by counting the executions of a basic operation.
- On the other hand, the physical running time provides very specific information about an algorithm's performance in a particular computing environment, which can be of more importance to the experimenter than, say, the algorithm's asymptotic efficiency class.
- In addition, measuring time spent on different segments of a program can pinpoint a bottleneck in the program's performance that can be missed by an abstract deliberation about the algorithm's basic operation.
- Getting such data—called ***profiling***—is an important resource in the empirical analysis of an algorithm's running time;
- Whether you decide to measure the efficiency by basic operation counting or by time clocking, you will need to decide on a sample of inputs for the experiment.
- Often, the goal is to use a sample representing a "typical" input; so the challenge is to understand what a "typical" input is.
- The principal advantage of size changing according to a pattern is that its impact is easier to analyze
- For example, if a sample's sizes are generated by doubling, you can compute the ratios $M(2n)/M(n)$ of the observed metric M (the count or the time) to see whether the ratios exhibit a behavior typical of algorithms in one of the basic efficiency classes
- The major disadvantage of nonrandom sizes is the possibility that the algorithm under investigation exhibits atypical behavior on the sample chosen.
- For example, if all the sizes in a sample are even and your algorithm runs much more slowly on odd-size inputs, the empirical results will be quite misleading.
- Another important issue concerning sizes in an experiment's sample is whether several instances of the same size should be
- Alternatively, you can implement one of several known algorithms for generating (pseudo)random numbers.
- The most widely used and thoroughly studied of such algorithms is the ***linear congruential method***.

ALGORITHM *Random(n, m, seed, a, b)*

//Generates a sequence of ***n*** pseudorandom numbers according to the linear congruential method

```
//Input: A positive integer n and positive integer parameters m, seed, a, b
//Output: A sequence r_1, . . . , r_n of n pseudorandom integers uniformly distributed among integer values between 0 and m − 1 //Note: Pseudorandom numbers between 0 and 1 can be obtained by treating the integers generated as digits after the decimal point
r_0 ← seed
for i ← 1 to n do
    r_i ← (a * r_{i−1} + b) mod m
```

- The empirical data obtained as the result of an experiment need to be recorded and then presented for an analysis.
- Data can be presented numerically in a table or graphically in a ***scatterplot***, i.e., by points in a Cartesian coordinate system.
- It is a good idea to use both these options whenever it is feasible because both methods have their unique strengths and weaknesses.
- The principal advantage of tabulated data lies in the opportunity to manipulate it easily.
- On the other hand, the form of a scatterplot may also help in ascertaining the algorithm's probable efficiency class.
- One of the possible applications of the empirical analysis is to predict the algorithm's performance on an instance not included in the experiment sample.
- The principal strength of the mathematical analysis is its independence of specific inputs; its principal weakness is its limited applicability, especially for investigating the average-case efficiency.
- The principal strength of the empirical analysis lies in its applicability to any algorithm, but its results can depend on the particular sample of instances and the computer used in



the experiment.

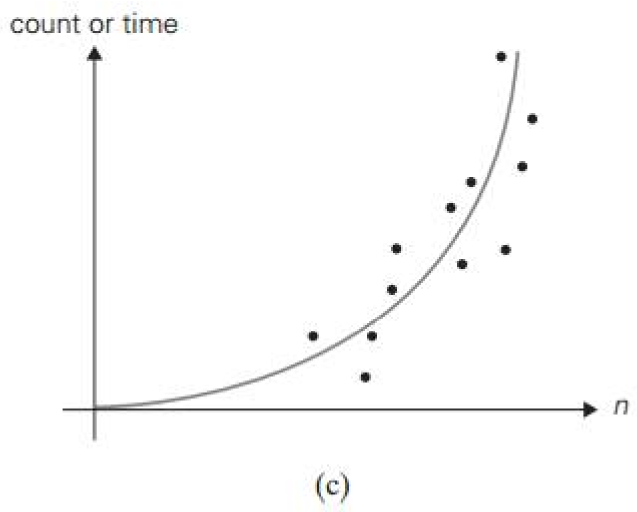
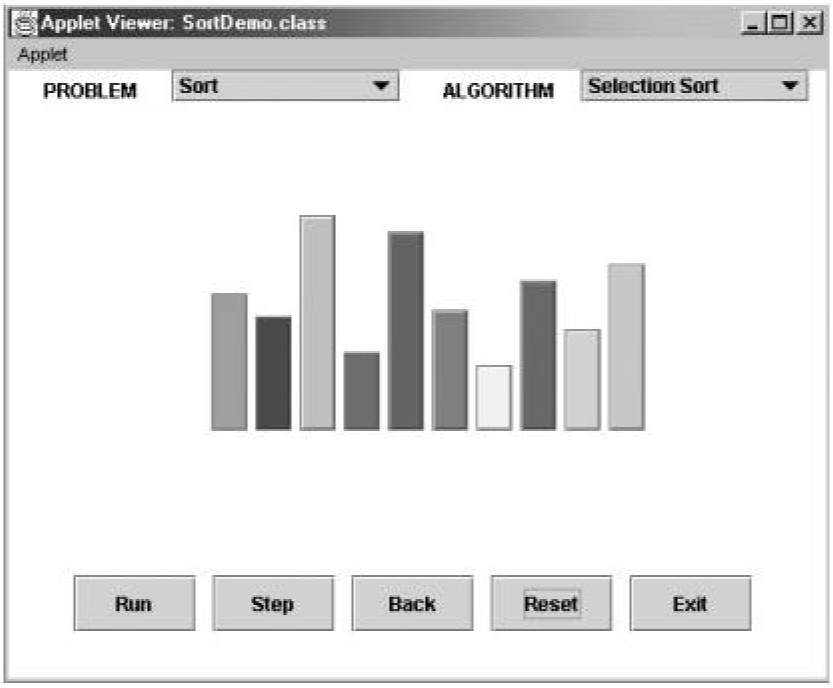


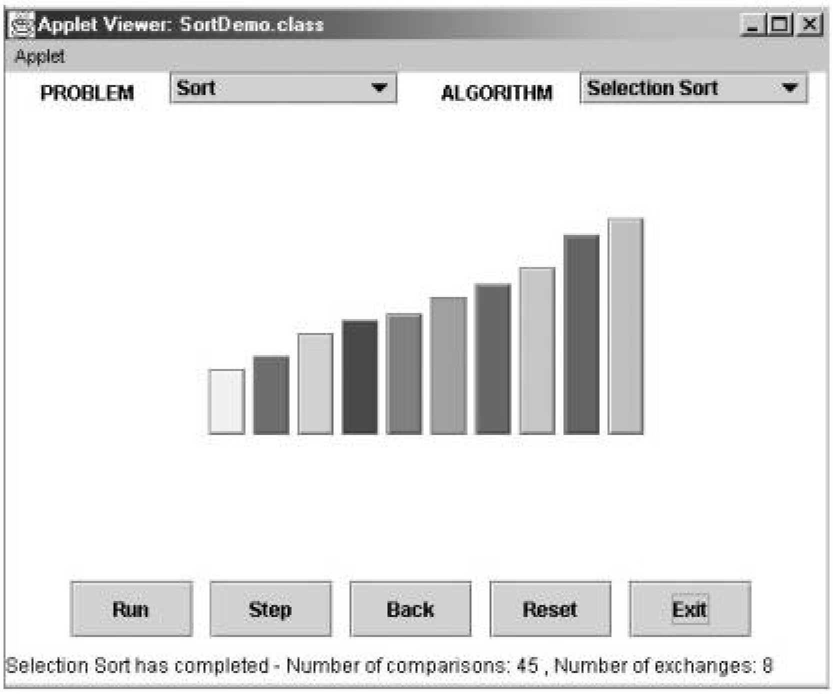
FIGURE 2.7 Typical scatter plots. (a) Logarithmic. (b) Linear. (c) One of the convex functions.

ALGORITHM VISUALIZATION

- In addition to the mathematical and empirical analyses of algorithms, there is yet a third way to study algorithms.
- It is called ***algorithm visualization*** and can be defined as the use of images to convey some useful information about algorithms.
- That information can be a visual illustration of an algorithm's operation, of its performance on different kinds of inputs, or of its execution speed versus that of other algorithms for the same problem.
- To accomplish this goal, an algorithm visualization uses graphic elements—points, line segments, two- or three-dimensional bars, and so on—to represent some "interesting events" in the algorithm's operation.
- There are two principal variations of algorithm visualization:
  Static algorithm visualization
  Dynamic algorithm visualization, also called algorithm animation
- Static algorithm visualization shows an algorithm's progress through a series of still images.
- Algorithm animation, on the other hand, shows a continuous, movie-like presentation of an algorithm's operations.
- Animation is an arguably more sophisticated option, which, of course, is much more difficult to implement.
- Early efforts in the area of algorithm visualization go back to the 1970s.
- The watershed event happened in 1981 with the appearance of a 30-minute color sound film titled *Sorting Out Sorting*.

- The success of *Sorting Out Sorting* made sorting algorithms a perennial favorite for algorithm animation.
- Indeed, the sorting problem lends itself quite naturally to visual presentation via vertical or horizontal bars or sticks of different heights or lengths, which need to be rearranged according to their sizes.

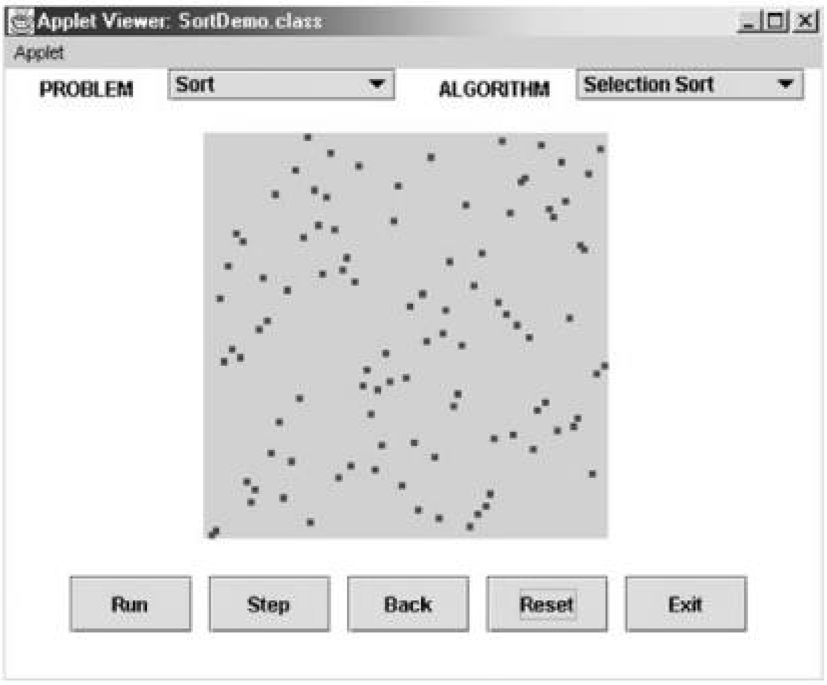




Selection Sort has completed - Number of comparisons: 45 , Number of exchanges: 8

FIGURE 2.8 Initial and final screens of a typical visualization of a sorting algorithm using the bar representation.

- Since the appearance of *Sorting Out Sorting*, a great number of algorithm animations have been created, especially after the appearance of Java and the World Wide Web in the 1990s.



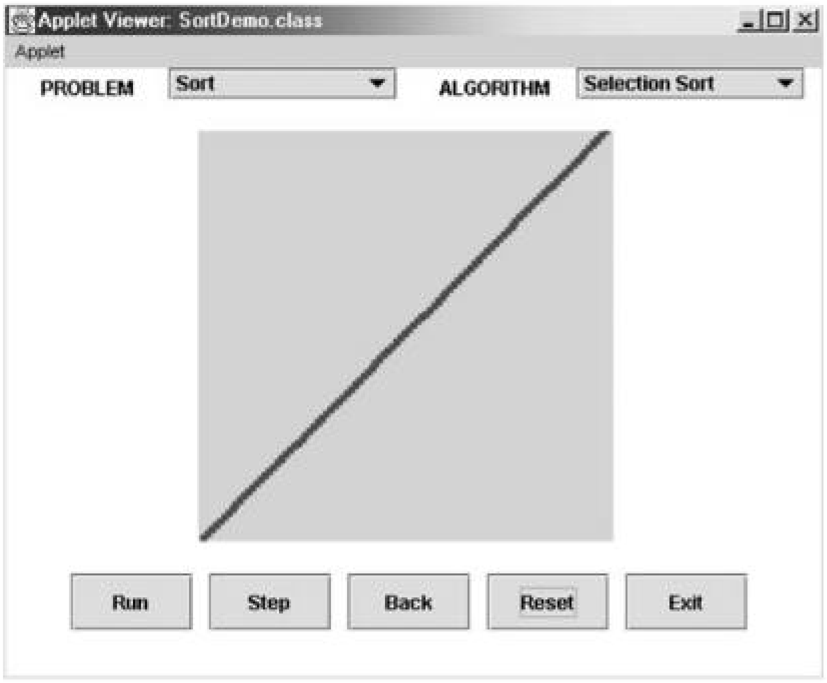


FIGURE 2.9 Initial and final screens of a typical visualization of a sorting algorithm using the scatterplot representation.

- There are two principal applications of algorithm visualization: research and education. Potential benefits for researchers are based on expectations that algorithm visualization may help uncover some unknown features of algorithms.
- The application of algorithm visualization to education seeks to help students learning algorithms.
- The available evidence of its effectiveness is decisively mixed.
- A deeper understanding of human perception of images will be required before the true potential of algorithm visualization is fulfilled.