16. The mathematics of algorithm analysis

Learning objectives:

- worst-case and average performance of an algorithm
- growth rate of a function
- asymptotics: O(), Ω(), .ΘΘ()
- asymptotic behavior of sums
- solution techniques for recurrence relations
- asymptotic performance of divide-and-conquer algorithms
- average number of inversions and average distance in a permutation
- trees and their properties

Growth rates and orders of magnitude

To understand a specific algorithm, it is useful to ask and answer the following questions, usually in this order: What is the problem to be solved? What is the main idea on which this algorithm is based? Why is it correct? How efficient is it?

The variety of problems is vast, and so is the variety of "main ideas" that lead one to design an algorithm and establish its correctness. True, there are general algorithmic principles or schemas which are problem-independent, but these rarely suffice: Interesting algorithms typically exploit specific features of a problem, so there is no unified approach to understanding the logic of algorithms. Remarkably, there *is* a unified approach to the efficiency analysis of algorithms, where efficiency is measured by a program's time and storage requirements. This is remarkable because there is great variety in (1) sets of input data and (2) environments (computers, operating systems, programming languages, coding techniques), and these differences have a great influence on the run time and storage consumed by a program. These two types of differences are overcome as follows.

Different sets of input data: worst-case and average performance

The most important characteristic of a set of data is its size, measured in terms of any unit convenient to the problem at hand. This is typically the number of primitive objects in the data, such as bits, bytes, integers, or any monotonic function thereof, such as the magnitude of an integer. *Examples:* For sorting, the number n of elements is natural; for square matrices, the number n of rows and columns is convenient; it is a monotonic function (square root) of the actual size n^2 of the data. An algorithm may well behave very differently on different data sets of equal size n—among all possible configurations of given size n some will be favorable, others less so. Both the *worst-case* data set of size n and the *average* over all data sets of size n provide well-defined and important measures of efficiency. *Example:* When sorting data sets about whose order nothing is known, average performance is well characterized by averaging run time over all n! permutations of the n elements.

Different environments: focus on growth rate and ignore constants

The work performed by an algorithm is expressed as a function of the problem size, typically measured by size n of the input data. By focusing on the growth rate of this function but ignoring specific constants, we succeed in losing a lot of detail information that changes wildly from one computing environment to another, while retaining some essential information that is remarkably invariant when moving a computation from a micro- to a supercomputer, from machine language to Pascal, from amateur to professional programmer. The definition of general measures for the complexity of problems and for the efficiency of algorithms is a major achievement of computer science. It is based on the notions of *asymptotic time and space complexity*. Asymptotics renounces exact measurement but states how the work grows as the problem size increases. This information often suffices to distinguish efficient algorithms from inefficient ones. The asymptotic behavior of an algorithm is described by the O(), Ω(), Θ(), and o() notations. To determine the amount of work to be performed by an algorithm we count operations that take constant time (independently of n) and data objects that require constant storage space. The time required by an addition, comparison, or exchange of two numbers is typically independent of how many numbers we are processing; so is the storage requirement for a number.

Assume that the time required by four algorithms A_1, A_2, A_3, and A_4 is $\log_2 n$, n, $n \cdot \log_2 n$, and n^2, respectively. The following table shows that for sizes of data sets that frequently occur in practice, from $n \approx 10^3$ to 10^6, the difference in growth rate translates into large numerical differences:

n	$A_1 = \log_2 n$	$A_2 = n$	$A_3 = n \cdot \log_2 n$	$A_4 = n^2$
$2^5 = 32$	5	$2^5 = 3^2$	$5 \cdot 2^5 = 160$	$2^{10} \approx 10^3$
$2^{10} = 1024$	10	$2^{10} \approx 10^3$	$10 \cdot 2^{10} \approx 10^4$	$2^{20} \approx 10^6$
$2^{20} \approx 10_6$	20	$2^{20} \approx 1^{06}$	$20 \cdot 2^{20} \approx 2 \cdot 10^7$	$2^{40} \approx 10^{12}$

For a specific algorithm these functions are to be multiplied by a constant factor proportional to the time it takes to execute the body of the innermost loop. When comparing different algorithms that solve the same problem, it may well happen that one innermost loop is 10 times faster or slower than another. It is rare that this difference approaches a factor of 100. Thus for $n \approx 1000$ an algorithm with time complexity $\Theta(n \cdot \log n)$ will almost always be much more efficient than an algorithm with time complexity $\Theta(n^2)$. For small n, say $n = 32$, an algorithm of time complexity $\Theta(n^2)$ may be more efficient than one of complexity $\Theta(n \cdot \log n)$ (e.g. if its constant is 10 times smaller).

When we wish to predict exactly how many seconds and bytes a program needs, asymptotic analysis is still useful but is only a small part of the work. We now have to go back over the formulas and keep track of all the constant factors discarded in cavalier fashion by the O() notation. We have to assign numbers to the time consumed by scores of primitive O(1) operations. It may be sufficient to estimate the time consuming primitives, such as floating-point operations; or it may be necessary to include those that are hidden by a high-level programming language and answer questions such as: How long does an array access a[i, j] take? A procedure call? Incrementing the index i in a loop "for i := 0 to n"?

Asymptotics

Asymptotics is a technique used to estimate and compare the growth behavior of functions. Consider the function

$$f(x) = \frac{1+x^2}{x}.$$

f(x) is said to behave like x for $x \to \infty$ and like 1/x for $x \to 0$. The motivation for such a statement is that both x and 1/x are intuitively simpler, more easily understood functions than f(x). A complicated function is unlike any simpler one across its entire domain, but it usually behaves like a simpler one as x approaches some particular value. Thus all asymptotic statements include the qualifier $x \to x_0$. For the purpose of algorithm analysis we are interested in the behavior of functions for large values of their argument, and all our definitions below assume $x \to \infty$.

The asymptotic behavior of functions is described by the O(), Ω(), Θ (), and o() notations, as in $f(x) \in O(g(x))$. Each of these notations assigns to a given function g the *set of all functions* that are related to g in a well-defined way. Intuitively, O(), Ω(), Θ(), and o() are used to compare the growth of functions, as ≤, ≥, =, and < are used to compare numbers. O(g) is the set of all functions that are ≤ g in a precise technical sense that corresponds to the intuitive notion "grows no faster than g". The definition involves some technicalities signaled by the preamble $\exists c > 0, \exists \Rightarrow \exists x_0 \in X, \forall x \geq x_0$. It says that we ignore constant factors and initial behavior and are interested only in a function's behavior from some point on. N_0 is the set of nonnegative integers, R_0 the set of nonnegative reals. In the following definitions X stands for either N_0 or R_0. Let $g: X \to X$.

Definition of O(), "big oh":

$O(g) := \{f: X \to X \mid \exists\, c > 0, \exists\, x_0 \in X, \forall\, x \geq x_0 : f(x) \leq c \cdot g(x)\}$

We say that $f \in O(g)$, or that f grows at most as fast as g(x) for $x \to \infty$.

Definition of Ω(), "omega":

$\Omega(g) := \{f: X \to X \mid \exists\, c > 0\ \ x_0 \in X, \forall\, x \geq x_0: f(x) \geq c \cdot g(x)\}$.

We say that $f \in O(g)$, or that f grows at least as fast as g(x) for $x \to \infty$.

Definition of Θ(), "theta":

$\Theta(g) := O(g) \cap \Omega(g)$.

We say that $f \in \Theta(g)$, or that f has the same growth rate as g(x) for $x \to \infty$.

Definition of o(), "small oh":

$$o(g) := \{f: X \to X \mid \lim_{x \to \infty} \frac{f(x)}{g(x)} = 0\}.$$

We say that $f \in o(g)$, or that f grows slower than g(x) for $x \to \infty$.

Notation: Most of the literature uses = in place of our ∈, such as in $x = O(x^2)$. If you do so, just remember that this = has none of the standard properties of an equality relation—it is neither commutative nor transitive. Thus $O(x^2) = x$ is not used, and from $x = O(x^2)$ and $x^2 = O(x^2)$ it does not follow that $x = x^2$. The key to avoiding confusion is the insight that O() is not a function but a set of functions.

Summation formulas

$\log_2$ denotes the logarithm to the base 2, ln the natural logarithm to the base e.

$$\sum_{i=1}^{n} i = \frac{n\cdot(n+1)}{2} = \frac{n^2+n}{2}$$

$$\sum_{i=1}^{n} i^2 = \frac{n\cdot(n+1)\cdot(2n+1)}{6} = \frac{n^3}{3}+\frac{n^2}{2}+\frac{n}{6}$$

$$\sum_{i=1}^{n} i^3 = \frac{n^2\cdot(n+1)^2}{4} = \frac{n^4}{4}+\frac{n^3}{2}+\frac{n^2}{4}$$

The asymptotic behavior of a sum can be derived by comparing the sum to an integral that can be evaluated in closed form. Let f(x) be a monotonically increasing, integrable function. Then

$$\int_a^b f(x)\,dx$$

is bounded below and above by sums (Exhibit 16.1):

$$\sum_{i=1}^{n} f(x_{i-1})\cdot(x_i - x_{i-1}) \le \int_{x_0}^{x_n} f(x)\,dx \le \sum_{i=1}^{n} f(x_i)\cdot(x_i - x_{i-1}).$$

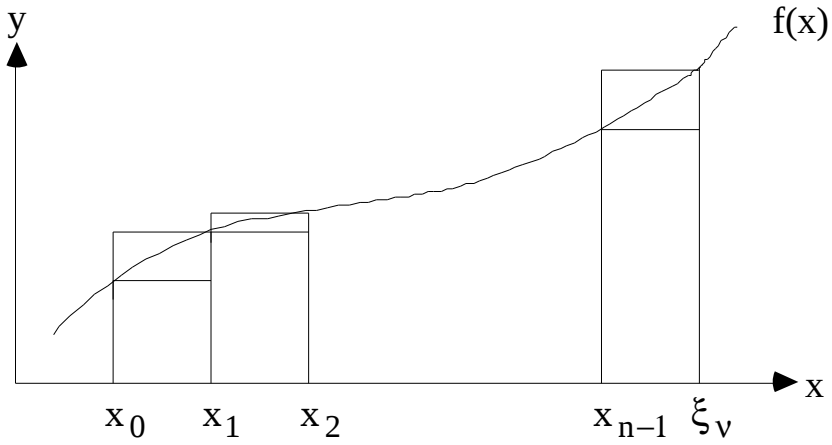

Exhibit 16.1: Bounding a definite integral by lower and upper sums.

Letting $x_i = i + 1$, this inequality becomes

$$\sum_{i=1}^{n} f(i) \le \int_{1}^{n+1} f(x)\,dx \le \sum_{i=1}^{n} f(i+1),$$

so

$$\int_{1}^{n+1} f(x)\,dx - f(n+1) + f(1) \le \sum_{i=1}^{n} f(i) \le \int_{1}^{n+1} f(x)\,dx. \qquad (*)$$

Example

By substituting

$$f(x) = x^k \quad \text{and} \quad \int x^k \, dx = \frac{x^{k+1}}{k+1}$$

with k > 0 in (∗) we obtain

$$\frac{(n+1)^{k+1}}{k+1} - \frac{1}{k+1} - (n+1)^k + 1 \;\le\; \sum_{i=1}^{n} i^k \;\le\; \frac{(n+1)^{k+1}}{k+1} - \frac{1}{k+1},$$

and therefore

$$\forall\, k > 0: \quad \sum_{i=1}^{n} i^k \;=\; \frac{n^{k+1}}{k+1} + g(n) \quad \text{with} \quad g(n) \in O(n^k).$$

Example

By substituting

$$f(x) = \ln x \quad \text{and} \quad \int \ln x \, dx = x \cdot \ln x - x$$

in (∗)we obtain

$$(n+1)\cdot\ln(n+1) - n - \ln(n+1) \le \sum_{i=1}^{n} \ln i \le (n+1)\,\text{cdot}\ln(n+1) - n,$$

and therefore

$$\sum_{i=1}^{n} \log_2 i = (n+1)\cdot\log_2(n+1) - \frac{n}{\ln 2} + g(n) \text{ with } g(n) \in O(\log n)$$

Example

By substituting

$$f(x) = \ln x \quad \text{and} \quad \int \ln x \, dx = x \cdot \ln x - x$$

in (∗)we obtain

$$\sum_{i=1}^{n} i \cdot \log_2 i = \frac{(n+1)^2}{2} \cdot \log_2(n+1) - \frac{(n+1)^2}{4 \cdot \ln 2} + g(n)$$

with $g(n) \in O(n \cdot \log n)$.

Recurrence relations

A *homogeneous linear* recurrence relation with constant coefficients is of the form

$$x_n = a_1 \cdot x_{n-1} + a_2 \cdot x_{n-2} + \ldots + a_k \cdot x_{n-k}$$

where the coefficients a_i are independent of n and $x_1, x_2, \ldots, x_{n-1}$ are specified. There is a general technique for solving linear recurrence relations with constant coefficients - that is, for determining x_n as a function of n. We will demonstrate this technique for the Fibonacci sequence which is defined by the recurrence

$x_n = x_{n-1} + x_{n-2}$, $x_0 = 0$, $x_1 = 1$.

We seek a solution of the form

$x_n = c \cdot r^n$

with constants c and r to be determined. Substituting this into the Fibonacci recurrence relation yields

$c \cdot r^n = c \cdot r^{n-1} + c \cdot r^{n-2}$

or

$c \cdot r^{n-2} \cdot (r^2 - r - 1) = 0$.

This equation is satisfied if either $c = 0$ or $r = 0$ or $r^2 - r - 1 = 0$. We obtain the trivial solution $x_n = 0$ for all n if $c = 0$ or $r = 0$. More interestingly, $r^2 - r - 1 = 0$ for

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

The sum of two solutions of a homogeneous linear recurrence relation is obviously also a solution, and it can be shown that any linear combination of solutions is again a solution. Therefore, the most general solution of the Fibonacci recurrence has the form

$$x_n = c_1 \cdot \left(\frac{1+\sqrt{5}}{2}\right)^n + c_2 \cdot \left(\frac{1-\sqrt{5}}{2}\right)^n$$

where c_1 and c_2 are determined as solutions of the linear equations derived from the initial conditions:

$$x_0 = 0 \quad \Rightarrow \quad c_1 + c_2 = 0$$

$$x_1 = 1 \quad \Rightarrow \quad c_1 \cdot \frac{1+\sqrt{5}}{2} + c_2 \cdot \frac{1-\sqrt{5}}{2} = 1$$

which yield

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

the complete solution for the Fibonacci recurrence relation is therefore

$$x_n = \frac{1}{\sqrt{5}} \cdot \left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}} \cdot \left(\frac{1-\sqrt{5}}{2}\right)^n.$$

Recurrence relations that are not linear with constant coefficients have no general solution techniques comparable to the one discussed above. General recurrence relations are solved (or their solutions are approximated or bounded) by trial-and-error techniques. If the trial and error is guided by some general technique, it will yield at least a good estimate of the asymptotic behavior of the solution of most recurrence relations.

Example

Consider the recurrence relation

$$x_n = \frac{2}{n} \sum_{i=0}^{n-1} x_i + a \cdot n + b \qquad (*)$$

with a > 0 and b > 0, which appears often in the average-case analysis of algorithms and data structures. When we know from the interpretation of this recurrence that its solution is monotonically nondecreasing, a systematic trial-and-error process leads to the asymptotic behavior of the solution. The simplest possible try is a constant, $x_n = c$. Substituting this into $(*)$ leads to

$$c \quad ?=? \quad 2 \cdot c + a \cdot n + b,$$

so $x_n = c$ is not a solution. Since the left-hand side x_n is smaller than an average of previous values on the right-hand side, the solution of this recurrence relation must grow faster than c. Next, we try a linear function $x_n = c \cdot n$:

$$\begin{aligned} c \cdot n \quad ?=? \quad & \frac{2}{n} \sum_{i=0}^{n-1} c \cdot i + a \cdot n + b \\ ?=? \quad & (c + a) \cdot n - c + b. \end{aligned}$$

At this stage of the analysis it suffices to focus on the leading terms of each side: $c \cdot n$ on the left and $(c + a) \cdot n$ on the right. The assumption $a > 0$ makes the right side larger than the left, and we conclude that a linear function also grows too slowly to be a solution of the recurrence relation. A new attempt with a function that grows yet faster, $x_n = c \cdot n^2$, leads to

$$\begin{aligned} c \cdot n^2 \quad ?=? \quad & \frac{2}{n} \sum_{i=0}^{n-1} c \cdot i^2 + a \cdot n + b \\ ?=? \quad & \frac{2}{3} \cdot c \cdot n^2 + (a - c) \cdot n + \frac{c}{3} + b. \end{aligned}$$

Comparing the leading terms on both sides, we find that the left side is now larger than the right, and conclude that a quadratic function grows too fast. Having bounded the growth rate of the solution from below and above, we try functions whose growth rate lies between that of a linear and a quadratic function, such as $x_n = c \cdot n^{1.5}$. A more sophisticated approach considers a family of functions of the form $x_n = c \cdot n^{1+\varepsilon}$ for any $\varepsilon > 0$: All of them grow too fast. This suggests $x_n = c \cdot n \cdot \log_2 n$, which gives

$$\begin{aligned} c \cdot n \cdot \log_2 n \quad ?=? \quad & \frac{2}{n} \sum_{i=0}^{n-1} c \cdot i \cdot \log_2 i + a \cdot n + b \\ ?=? \quad & \frac{2 \cdot c}{n} \left(\frac{n^2}{2} \cdot \log_2 n - \frac{n^2}{4 \cdot \ln 2} + g(n) \right) + a \cdot n + b \\ ?=? \quad & c \cdot n \cdot \log_2 n + \left(a - \frac{c}{2 \cdot \ln 2} \right) \cdot n + h(n) \end{aligned}$$

with $g(n) \in O(n \cdot \log n)$ and $h(n) \in O(\log n)$. To match the linear terms on each side, we must choose c such that

$$a - \frac{c}{2 \cdot \ln 2} = 0$$

or $c = a \cdot \ln 4 \approx 1.386 \cdot a$. Hence we now know that the solution to the recurrence relation $(*)$ has the form

$$x_n = (\ln 4) \cdot a \cdot n \cdot \log_2 n + g(n) \quad \text{with} \quad g(n) \in O(n).$$

Asymptotic performance of divide-and-conquer algorithms

We illustrate the power of the techniques developed in previous sections by analyzing the asymptotic performance not of a specific algorithm, but rather, of an entire class of divide-and-conquer algorithms. In "Divide and conquer recursion" we presented the following schema for divide-and-conquer algorithms that partition the set of data into two parts:

```
A(D):  if  simple(D) then return(A0(D))
                else 1. divide: partition D into D1 and D2;
                     2. conquer: R1 := A(D1);  R2 := A(D2);
                     3. combine: return(merge(R1, R2));
```

Assume further that the data set D can always be partitioned into two halves, D_1 and D_2, at every level of recursion. Two comments are appropriate:

1. For repeated halving to be possible it is not necessary that the size n of the data set D be a power of 2, $n = 2^k$. It is not important that D be partitioned into two exact halves—approximate halves will do. Imagine padding any data set D whose size is not a power of 2 with dummy elements, up to the next power of 2. Dummies can always be found that do not disturb the real computation: for example, by replicating elements or by appending sentinels. Padding is usually just a conceptual trick that may help in understanding the process, but need not necessarily generate any additional data.
2. Whether or not the divide step is guaranteed to partition D into two approximate halves, on the other hand, depends critically on the problem and on the data structures used. Example: Binary search in an ordered array partitions D into halves by probing the element at the midpoint; the same idea is impractical in a linked list because the midpoint is not directly accessible.

Under our assumption of halving, the time complexity T(n) of algorithm A applied to data D of size n satisfies the recurrence relation

$$T(n) = 2 \cdot T\left(\left\lceil \frac{n}{2} \right\rceil\right) + f(n)$$

where f(n) is the sum of the partitioning or splitting time and the "stitching time" required to merge two solutions of size n / 2 into a solution of size n. Repeated substitution yields

$$\begin{aligned} T(n) &= 4 \cdot T\left(\left\lceil \frac{n}{4} \right\rceil\right) + f(n) + 2 \cdot f\left(\left\lceil \frac{n}{2} \right\rceil\right) \\ &= 8 \cdot T\left(\left\lceil \frac{n}{8} \right\rceil\right) + f(n) + 2 \cdot f\left(\frac{n}{2}\right) + 4 \cdot f\left(\left\lceil \frac{n}{4} \right\rceil\right) \\ &\text{---} \\ &= n \cdot T(1) + \sum_{k=0}^{(\log_2 n)-1} 2^k \cdot f\left(\frac{n}{2^k}\right). \end{aligned}$$

The term $n \cdot T(1)$ expresses the fact that every data item gets looked at, the second sums up the splitting and stitching time. Three typical cases occur:

(a) Constant time splitting and merging f(n) = c yields

$T(n) = (T(1) + c) \cdot n.$

Example: Find the maximum of n numbers.

(b) Linear time splitting and merging $f(n) = a \cdot n + b$ yields

$T(n) = a \cdot n \cdot \log_2 n + (T(1) + b) \cdot n.$

Examples: Mergesort, quicksort.

(c) Expensive splitting and merging: $n \in o(f(n))$ yields

$T(n) = n \cdot T(1) + O(f(n) \cdot \log n)$

and therefore rarely leads to interesting algorithms.

Permutations

Inversions

Let $(a_k: 1 \le k \le n)$ be a permutation of the integers 1 .. n. A pair (a_i, a_j), $1 \le I < j \le n$, is called an *inversion* iff $a_i > a_j$. What is the average number of inversions in a permutation? Consider all permutations in pairs; that is, with any permutation A:

$a_1 = x_1; a_2 = x_2; \ldots ; a_n = x_n$

consider its inverse A', which contains the elements of A in inverse order:

$a_1 = x_n; a_2 = x_{n-1}; \ldots ; a_n = x_1.$

In one of these two permutations x_i and x_j are in the correct order, in the other, they form an inversion. Since there are $n \cdot (n - 1) / 2$ pairs of elements (x_i, x_j) with $1 \le i < j \le n$ there are, on average,

$$\mathbf{inv}_{\mathbf{average}} = \frac{1}{2} \cdot \mathbf{n} \cdot \frac{\mathbf{n-1}}{2} = \frac{\mathbf{n^2 - n}}{4}$$

inversions.

Average distance

Let $(a_k: 1 \le k \le n)$ be a permutation of the natural numbers from 1 to n. The distance of the element a_i from its correct position is $|a_i - i|$. The total distance of all elements from their correct positions is

$$\mathbf{TD}((\mathbf{a_k}: \mathbf{1 \le k \le n})) = \sum_{\mathbf{i=1}}^{\mathbf{n}} |\mathbf{a_i} - \mathbf{i}|.$$

Therefore, the average total distance (i.e. the average over all n! permutations) is

$$TD_{average} = \frac{1}{n!} \sum_{\substack{\text{all permutations} \\ (a_k : 1 \le k \le n)}} TD((a_k : 1 \le k \le n))$$

$$= \frac{1}{n!} \sum_{\substack{\text{all permutations} \\ (a_k : 1 \le k \le n)}} \sum_{i=1}^{n} |a_i - i|$$

$$= \frac{1}{n!} \sum_{i=1}^{n} \sum_{\substack{\text{all permutations} \\ (a_k : 1 \le k \le n)}} |a_i - i|.$$

Let $1 \le i \le n$ and $1 \le j \le n$. Consider all permutations for which a_i is equal to j. Since there are $(n - 1)!$ such permutations, we obtain

$$\sum_{\substack{\text{all permutations} \\ (a_k : 1 \le k \le n)}} |a_i - i| = (n-1)! \sum_{j=1}^{n} |j - i|.$$

Therefore,

$$TD_{average} = \frac{1}{n!} \sum_{i=1}^{n} \left((n-1)! \sum_{j=1}^{n} |j - i| \right)$$

$$= \frac{1}{n} \sum_{i=1}^{n} \left(\sum_{j=1}^{i-1} (i - j) + \sum_{j=i+1}^{n} (j - i) \right)$$

$$= \frac{n^2 - 1}{3}.$$

the average distance of an element a_i from its correct position is therefore

$$\frac{1}{n} \cdot \frac{n^2 - 1}{3} = \frac{n}{3} - \frac{1}{3 \cdot n}.$$

Trees

Trees are ubiquitous in discrete mathematics and computer science, and this section summarizes some of the basic concepts, terminology, and results. Although trees come in different versions, in the context of algorithms and data structures, "tree" almost always means an ordered rooted tree. An *ordered rooted tree* is either empty or it consists of a node, called a *root*, and a sequence of k ordered subtrees $T_1, T_2, \ldots, T_k$ (Exhibit 16.2). The nodes of an ordered tree that have only empty subtrees are called leaves or external nodes, the other nodes are called *internal node*s (Exhibit 16.3). The roots of the subtrees attached to a node are its *children*; and this node is their *parent*.

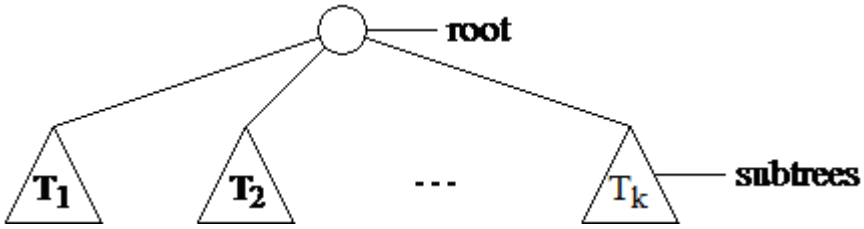

Exhibit 16.2: Recursive definition of a rooted, ordered tree.

The *level* of a node is defined recursively. The root of a tree is at level 0. The children of a node at level t are at level t + 1. The level of a node is the length of the path from the root of the tree to this node. The *height* of a tree is defined as the maximum level of all leaves. The *path length* of a tree is the sum of the levels of all its nodes (Exhibit 16.3).

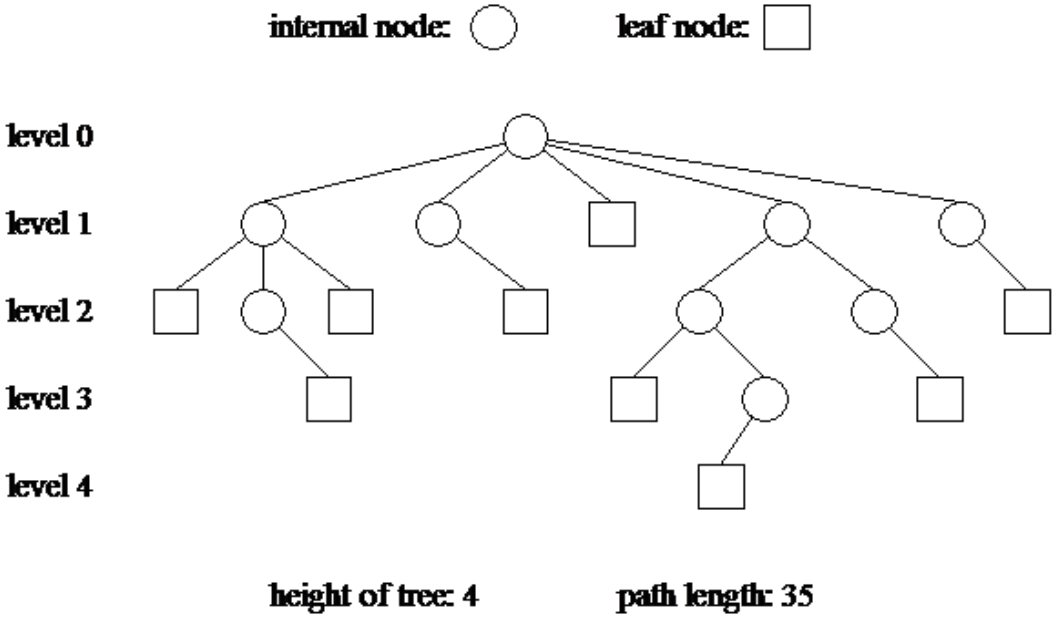

Exhibit 16.3: A tree of height = 4 and path length = 35.

A *binary tree* is an ordered tree whose nodes have at most two children. A 0-2 binary tree is a tree in which every node has zero or two children but not one. A 0-2 tree with n leaves has exactly n – 1 internal nodes. A binary tree of height h is called *complete* (completely balanced) if it has $2^{h+1} - 1$ nodes (Exhibit 16.4. A binary tree of height h is called *almost complete if* all its leaves are on levels h – 1 and h, and all leaves on level h are as far left as possible (Exhibit 16.4).

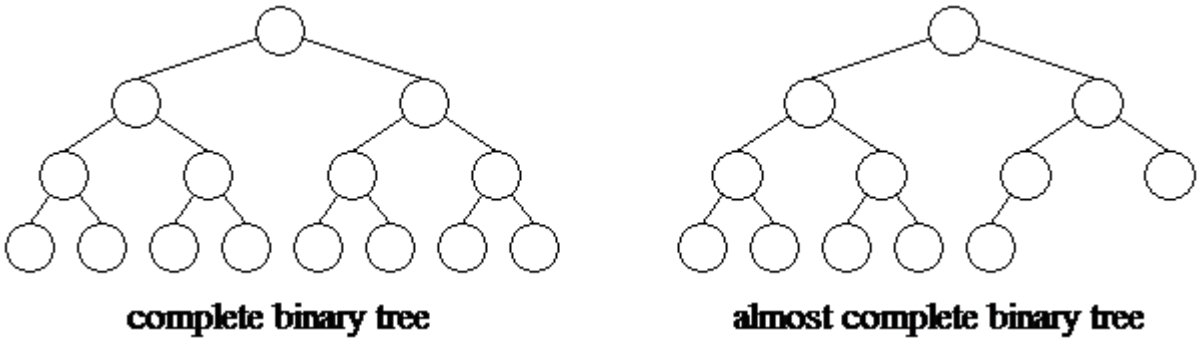

Exhibit 16.4: Examples of well-balanced binary trees.

Exercises

1. Suppose that we are comparing implementations of two algorithms on the same machine. For inputs of size n, the first algorithm runs in $9 \cdot n^2$ steps, while the second algorithm runs in $81 \cdot n \cdot \log_2 n$ steps. Assuming that the steps in both algorithms take the same time, for which values of n does the first algorithm beat the second algorithm?
2. What is the smallest value of n such that an algorithm whose running time is $256 \cdot n^2$ runs faster than an algorithm whose running time is 2^n on the same machine?
3. For each of the following functions $f_i(n)$, determine a function g(n) such that $f_i(n) \in \Theta(g(n))$. The function g(n) should be as simple as possible.

   $f_1(n) = 0.001 \cdot n^7 + n^2 + 2 \cdot n$

   $f_2(n) = n \cdot \log n + \log n + 1234 \cdot n$

   $f_3(n) = 5 \cdot n \cdot \log n + n^2 \cdot \log n + n^2$

   $f_4(n) = 5 \cdot n \cdot \log n + n^3 + n^2 \cdot \log n$
4. Prove formally that $1024 \cdot n^2 + 5 \cdot n \in \Theta(n^2)$.
5. Give an asymptotically tight bound for the following summation:

$$\sum_{i=1}^{n} i^k \cdot \log_2 i \quad \text{for } k > 0.$$

6. Find the most general solutions to the following recurrence relations.

$$\textbf{(a)} \quad x_n = x_{n-1} - \frac{1}{4} \cdot x_{n-2} \qquad \textbf{(b)} \quad x_n = x_{n-1} - \frac{1}{4} \cdot x_{n-2} + 2^{-n}$$

7. Solve the recurrence $T(\sqrt{n}) = 2 \cdot T() + \log_2 n$. *Hint:* Make a change of variables $m = \log_2 n$.
8. Compute the number of inversions and the total distance for the permutation (3 1 2 4).