2

Introduction

When you want to become a sculptor[1] *you have to learn some basic techniques: where to get the right stones, how to move them, how to handle the chisel, how to erect scaffolding, ... Knowing these techniques will not make you a famous artist, but even if you have a really exceptional talent, it will be very difficult to develop into a successful artist without knowing them. It is not necessary to master all of the basic techniques before sculpting the first piece. But you always have to be willing to go back to improve your basic techniques.*

This introductory chapter plays a similar role in this book. We introduce basic concepts that make it simpler to discuss and analyze algorithms in the subsequent chapters. There is no need for you to read this chapter from beginning to end before you proceed to later chapters. On first reading, we recommend that you should read carefully to the end of Sect. 2.3 and skim through the remaining sections. We begin in Sect. 2.1 by introducing some notation and terminology that allow us to argue about the complexity of algorithms in a concise way. We then introduce a simple machine model in Sect. 2.2 that allows us to abstract from the highly variable complications introduced by real hardware. The model is concrete enough to have predictive value and abstract enough to allow elegant arguments. Section 2.3 then introduces a high-level pseudocode notation for algorithms that is much more convenient for expressing algorithms than the machine code of our abstract machine. Pseudocode is also more convenient than actual programming languages, since we can use high-level concepts borrowed from mathematics without having to worry about exactly how they can be compiled to run on actual hardware. We frequently annotate programs to make algorithms more readable and easier to prove correct. This is the subject of Sect. 2.4. Section 2.5 gives the first comprehensive example: binary search in a sorted array. In Sect. 2.6, we introduce mathematical techniques for analyzing the complexity of programs, in particular, for analyzing nested loops and recursive pro-

[1] The above illustration of Stonehenge is from [156].

cedure calls. Additional analysis techniques are needed for average-case analysis; these are covered in Sect. 2.7. Randomized algorithms, discussed in Sect. 2.8, use coin tosses in their execution. Section 2.9 is devoted to graphs, a concept that will play an important role throughout the book. In Sect. 2.10, we discuss the question of when an algorithm should be called efficient, and introduce the complexity classes **P** and **NP**. Finally, as in every chapter of this book, there are sections containing implementation notes (Sect. 2.11) and historical notes and further findings (Sect. 2.12).

2.1 Asymptotic Notation

The main purpose of algorithm analysis is to give performance guarantees, for example bounds on running time, that are at the same time accurate, concise, general, and easy to understand. It is difficult to meet all these criteria simultaneously. For example, the most accurate way to characterize the running time T of an algorithm is to view T as a mapping from the set I of all inputs to the set of nonnegative numbers $\mathbb{R}_+$. For any problem instance i, $T(i)$ is the running time on i. This level of detail is so overwhelming that we could not possibly derive a theory about it. A useful theory needs a more global view of the performance of an algorithm.

We group the set of all inputs into classes of "similar" inputs and summarize the performance on all instances in the same class into a single number. The most useful grouping is by *size*. Usually, there is a natural way to assign a size to each problem instance. The size of an integer is the number of digits in its representation, and the size of a set is the number of elements in the set. The size of an instance is always a natural number. Sometimes we use more than one parameter to measure the size of an instance; for example, it is customary to measure the size of a graph by its number of nodes and its number of edges. We ignore this complication for now. We use $\mathrm{size}(i)$ to denote the size of instance i, and I_n to denote the instances of size n for $n \in \mathbb{N}$. For the inputs of size n, we are interested in the maximum, minimum, and average execution times:[2]

$$\begin{array}{ll} \textbf{worst case:} & T(n) = \max\{T(i) : i \in I_n\} \\ \textbf{best case:} & T(n) = \min\{T(i) : i \in I_n\} \\ \textbf{average case:} & T(n) = \dfrac{1}{|I_n|} \displaystyle\sum_{i \in I_n} T(i)\,. \end{array}$$

We are interested most in the worst-case execution time, since it gives us the strongest performance guarantee. A comparison of the best case and the worst case tells us how much the execution time varies for different inputs in the same class. If the discrepancy is big, the average case may give more insight into the true performance of the algorithm. Section 2.7 gives an example.

We shall perform one more step of data reduction: we shall concentrate on *growth rate* or *asymptotic analysis*. Functions $f(n)$ and $g(n)$ have the *same growth rate* if

[2] We shall make sure that $\{T(i) : i \in I_n\}$ always has a proper minimum and maximum, and that I_n is finite when we consider averages.

there are positive constants c and d such that $c \leq f(n)/g(n) \leq d$ for all sufficiently large n, and $f(n)$ *grows faster* than $g(n)$ if, for all positive constants c, we have $f(n) \geq c \cdot g(n)$ for all sufficiently large n. For example, the functions n^2, $n^2 + 7n$, $5n^2 - 7n$, and $n^2/10 + 10^6 n$ all have the same growth rate. Also, they grow faster than $n^{3/2}$, which in turn grows faster than $n \log n$. The growth rate talks about the behavior for large n. The word "asymptotic" in "asymptotic analysis" also stresses the fact that we are interested in the behavior for large n.

Why are we interested only in growth rates and the behavior for large n? We are interested in the behavior for large n because the whole purpose of designing efficient algorithms is to be able to solve large instances. For large n, an algorithm whose running time has a smaller growth rate than the running time of another algorithm will be superior. Also, our machine model is an abstraction of real machines and hence can predict actual running times only up to a constant factor, and this suggests that we should not distinguish between algorithms whose running times have the same growth rate. A pleasing side effect of concentrating on growth rate is that we can characterize the running times of algorithms by simple functions. However, in the sections on implementation, we shall frequently take a closer look and go beyond asymptotic analysis. Also, when using one of the algorithms described in this book, you should always ask yourself whether the asymptotic view is justified.

The following definitions allow us to argue precisely about *asymptotic behavior*. Let $f(n)$ and $g(n)$ denote functions that map nonnegative integers to nonnegative real numbers:

$$\begin{aligned}
\mathrm{O}(f(n)) &= \{g(n) : \exists c > 0 : \exists n_0 \in \mathbb{N}_+ : \forall n \geq n_0 : g(n) \leq c \cdot f(n)\}\,,\\
\Omega(f(n)) &= \{g(n) : \exists c > 0 : \exists n_0 \in \mathbb{N}_+ : \forall n \geq n_0 : g(n) \geq c \cdot f(n)\}\,,\\
\Theta(f(n)) &= \mathrm{O}(f(n)) \cap \Omega(f(n))\,,\\
\mathrm{o}(f(n)) &= \{g(n) : \forall c > 0 : \exists n_0 \in \mathbb{N}_+ : \forall n \geq n_0 : g(n) \leq c \cdot f(n)\}\,,\\
\omega(f(n)) &= \{g(n) : \forall c > 0 : \exists n_0 \in \mathbb{N}_+ : \forall n \geq n_0 : g(n) \geq c \cdot f(n)\}\,.
\end{aligned}$$

The left-hand sides should be read as "big O of f", "big omega of f", "theta of f", "little o of f", and "little omega of f", respectively.

Let us see some examples. $\mathrm{O}(n^2)$ is the set of all functions that grow at most quadratically, $\mathrm{o}(n^2)$ is the set of functions that grow less than quadratically, and $\mathrm{o}(1)$ is the set of functions that go to zero as n goes to infinity. Here "1" stands for the function $n \mapsto 1$, which is one everywhere, and hence $f \in \mathrm{o}(1)$ if $f(n) \leq c \cdot 1$ for any positive c and sufficiently large n, i.e., $f(n)$ goes to zero as n goes to infinity. Generally, $\mathrm{O}(f(n))$ is the set of all functions that "grow no faster than" $f(n)$. Similarly, $\Omega(f(n))$ is the set of all functions that "grow at least as fast as" $f(n)$. For example, the Karatsuba algorithm for integer multiplication has a worst-case running time in $\mathrm{O}(n^{1.58})$, whereas the school algorithm has a worst-case running time in $\Omega(n^2)$, so that we can say that the Karatsuba algorithm is asymptotically faster than the school algorithm. The "little o" notation $\mathrm{o}(f(n))$ denotes the set of all functions that "grow strictly more slowly than" $f(n)$. Its twin $\omega(f(n))$ is rarely used, and is only shown for completeness.

The growth rate of most algorithms discussed in this book is either a polynomial or a logarithmic function, or the product of a polynomial and a logarithmic function. We use polynomials to introduce our readers to some basic manipulations of asymptotic notation.

Lemma 2.1. *Let* $p(n) = \sum_{i=0}^{k} a_i n^i$ *denote any polynomial and assume* $a_k > 0$. *Then* $p(n) \in \Theta\left(n^k\right)$.

Proof. It suffices to show that $p(n) \in \mathrm{O}\left(n^k\right)$ and $p(n) \in \Omega\left(n^k\right)$. First observe that for $n > 0$,

$$p(n) \le \sum_{i=0}^{k} |a_i| n^i \le n^k \sum_{i=0}^{k} |a_i| ,$$

and hence $p(n) \le (\sum_{i=0}^{k} |a_i|) n^k$ for all positive n. Thus $p(n) \in \mathrm{O}\left(n^k\right)$.

Let $A = \sum_{i=0}^{k-1} |a_i|$. For positive n we have

$$p(n) \ge a_k n^k - A n^{k-1} = \frac{a_k}{2} n^k + n^{k-1} \left(\frac{a_k}{2} n - A \right)$$

and hence $p(n) \ge (a_k/2) n^k$ for $n > 2A/a_k$. We choose $c = a_k/2$ and $n_0 = 2A/a_k$ in the definition of $\Omega\left(n^k\right)$, and obtain $p(n) \in \Omega\left(n^k\right)$. $\square$

Exercise 2.1. Right or wrong? (a) $n^2 + 10^6 n \in \mathrm{O}\left(n^2\right)$, (b) $n \log n \in \mathrm{O}(n)$, (c) $n \log n \in \Omega(n)$, (d) $\log n \in \mathrm{o}(n)$.

Asymptotic notation is used a lot in algorithm analysis, and it is convenient to stretch mathematical notation a little in order to allow sets of functions (such as $\mathrm{O}\left(n^2\right)$) to be treated similarly to ordinary functions. In particular, we shall always write $h = \mathrm{O}(f)$ instead of $h \in \mathrm{O}(f)$, and $\mathrm{O}(h) = \mathrm{O}(f)$ instead of $\mathrm{O}(h) \subseteq \mathrm{O}(f)$. For example,

$$3n^2 + 7n = \mathrm{O}\left(n^2\right) = \mathrm{O}\left(n^3\right) .$$

Be warned that sequences of equalities involving O-notation should only be read from left to right.

If h is a function, F and G are sets of functions, and $\circ$ is an operator such as $+$, $\cdot$, or $/$, then $F \circ G$ is a shorthand for $\{f \circ g : f \in F, g \in G\}$, and $h \circ F$ stands for $\{h\} \circ F$. So $f(n) + \mathrm{o}(f(n))$ denotes the set of all functions $f(n) + g(n)$ where $g(n)$ grows strictly more slowly than $f(n)$, i.e., the ratio $(f(n) + g(n))/f(n)$ goes to one as n goes to infinity. Equivalently, we can write $(1 + \mathrm{o}(1)) f(n)$. We use this notation whenever we care about the constant in the leading term but want to ignore *lower-order terms*.

Lemma 2.2. *The following rules hold for* O*-notation:*

$$\begin{aligned} c f(n) &= \Theta(f(n)) \text{ for any positive constant,} \\ f(n) + g(n) &= \Omega(f(n)) , \\ f(n) + g(n) &= \mathrm{O}(f(n)) \text{ if } g(n) = \mathrm{O}(f(n)) , \\ \mathrm{O}(f(n)) \cdot \mathrm{O}(g(n)) &= \mathrm{O}(f(n) \cdot g(n)) . \end{aligned}$$

Exercise 2.2. Prove Lemma 2.2.

Exercise 2.3. Sharpen Lemma 2.1 and show that $p(n) = a_k n^k + \mathrm{o}(n^k)$.

Exercise 2.4. Prove that $n^k = \mathrm{o}(c^n)$ for any integer k and any $c > 1$. How does $n^{\log\log n}$ compare with n^k and c^n?

2.2 The Machine Model

In 1945, John von Neumann (Fig. 2.1) introduced a computer architecture [201] which was simple, yet powerful. The limited hardware technology of the time forced him to come up with an elegant design that concentrated on the essentials; otherwise, realization would have been impossible. Hardware technology has developed tremendously since 1945. However, the programming model resulting from von Neumann's design is so elegant and powerful that it is still the basis for most of modern programming. Usually, programs written with von Neumann's model in mind also work well on the vastly more complex hardware of today's machines.

Fig. 2.1. John von Neumann born Dec. 28, 1903 in Budapest, died Feb. 8, 1957 in Washington, DC

The variant of von Neumann's model used in algorithmic analysis is called the *RAM* (random access machine) model. It was introduced by Sheperdson and Sturgis [179]. It is a *sequential* machine with uniform memory, i.e., there is a single processing unit, and all memory accesses take the same amount of time. The memory or *store*, consists of infinitely many cells $S[0]$, $S[1]$, $S[2]$, $\ldots$; at any point in time, only a finite number of them will be in use.

The memory cells store "small" integers, also called *words*. In our discussion of integer arithmetic in Chap. 1, we assumed that "small" meant one-digit. It is more reasonable and convenient to assume that the interpretation of "small" depends on the size of the input. Our default assumption is that integers bounded by a polynomial in the size of the data being processed can be stored in a single cell. Such integers can be represented by a number of bits that is logarithmic in the size of the input. This assumption is reasonable because we could always spread out the contents of a single cell over logarithmically many cells with a logarithmic overhead in time and space and obtain constant-size cells. The assumption is convenient because we want to be able to store array indices in a single cell. The assumption is necessary because allowing cells to store arbitrary numbers would lead to absurdly overoptimistic algorithms. For example, by repeated squaring, we could generate a number with 2^n bits in n steps. Namely, if we start with the number $2 = 2^1$, squaring it once gives $4 = 2^2 = 2^{2^1}$, squaring it twice gives $16 = 2^4 = 2^{2^2}$, and squaring it n times gives 2^{2^n}.

Our model supports a limited form of parallelism. We can perform simple operations on a logarithmic number of bits in constant time.

In addition to the main memory, there are a small number of *registers* $R_1, \ldots, R_k$. Our RAM can execute the following *machine instructions*:

- $R_i := S[R_j]$ *loads* the contents of the memory cell indexed by the contents of R_j into register R_i.
- $S[R_j] := R_i$ *stores* register R_i into the memory cell indexed by the contents of R_j.
- $R_i := R_j \odot R_\ell$ is a binary register operation where "$\odot$" is a placeholder for a variety of operations. The *arithmetic* operations are the usual $+$, $-$, and $*$ but also the bitwise operations **|** (OR), & (AND), **>>** (shift right), **<<** (shift left), and $\oplus$ (exclusive OR, XOR). The operations **div** and **mod** stand for integer division and the remainder, respectively. The *comparison* operations $\leq$, $<$, $>$, and $\geq$ yield *true* (= 1) or *false* (= 0). The *logical* operations $\wedge$ and $\vee$ manipulate the *truth values* 0 and 1. We may also assume that there are operations which interpret the bits stored in a register as a floating-point number, i.e., a finite-precision approximation of a real number.
- $R_i := \odot R_j$ is a *unary* operation using the operators $-$, $\neg$ (logical NOT), or ~ (bitwise NOT).
- $R_i := C$ assigns a *constant* value to R_i.
- `JZ` j, R_i continues execution at memory address j if register R_i is zero.
- `J` j continues execution at memory address j.

Each instruction takes one time step to execute. The total execution time of a program is the number of instructions executed. A program is a list of instructions numbered starting at one. The addresses in jump-instructions refer to this numbering. The input for a computation is stored in memory cells $S[1]$ to $S[R_1]$.

It is important to remember that the RAM model is an abstraction. One should not confuse it with physically existing machines. In particular, real machines have a finite memory and a fixed number of bits per register (e.g., 32 or 64). In contrast, the word size and memory of a RAM scale with input size. This can be viewed as an abstraction of the historical development. Microprocessors have had words of 4, 8, 16, and 32 bits in succession, and now often have 64-bit words. Words of 64 bits can index a memory of size 2^{64}. Thus, at current prices, memory size is limited by cost and not by physical limitations. Observe that this statement was also true when 32-bit words were introduced.

Our complexity model is also a gross oversimplification: modern processors attempt to execute many instructions in parallel. How well they succeed depends on factors such as data dependencies between successive operations. As a consequence, an operation does not have a fixed cost. This effect is particularly pronounced for memory accesses. The worst-case time for a memory access to the main memory can be hundreds of times higher than the best-case time. The reason is that modern processors attempt to keep frequently used data in *caches* – small, fast memories close to the processors. How well caches work depends a lot on their architecture, the program, and the particular input.

We could attempt to introduce a very accurate cost model, but this would miss the point. We would end up with a complex model that would be difficult to handle. Even a successful complexity analysis would lead to a monstrous formula depending on many parameters that change with every new processor generation. Although such a formula would contain detailed information, the very complexity of the formula would make it useless. We therefore go to the other extreme and eliminate all model parameters by assuming that each instruction takes exactly one unit of time. The result is that constant factors in our model are quite meaningless – one more reason to stick to asymptotic analysis most of the time. We compensate for this drawback by providing implementation notes, in which we discuss implementation choices and trade-offs.

2.2.1 External Memory

The biggest difference between a RAM and a real machine is in the memory: a uniform memory in a RAM and a complex memory hierarchy in a real machine. In Sects. 5.7, 6.3, and 7.6, we shall discuss algorithms that have been specifically designed for huge data sets which have to be stored on slow memory, such as disks. We shall use the *external-memory model* to study these algorithms.

The external-memory model is like the RAM model except that the fast memory S is limited in size to M words. Additionally, there is an external memory with unlimited size. There are special *I/O operations*, which transfer B consecutive words between slow and fast memory. For example, the external memory could be a hard disk, M would then be the size of the main memory, and B would be a block size that is a good compromise between low latency and high bandwidth. With current technology, $M = 2$ Gbyte and $B = 2$ Mbyte are realistic values. One I/O step would then take around 10 ms which is $2 \cdot 10^7$ clock cycles of a 2 GHz machine. With another setting of the parameters M and B, we could model the smaller access time difference between a hardware cache and main memory.

2.2.2 Parallel Processing

On modern machines, we are confronted with many forms of parallel processing. Many processors have 128–512-bit-wide *SIMD* registers that allow the parallel execution of a **s**ingle **i**nstruction on **m**ultiple **d**ata objects. *Simultaneous multithreading* allows processors to better utilize their resources by running multiple threads of activity on a single processor core. Even mobile devices often have multiple processor cores that can independently execute programs, and most servers have several such *multicore* processors accessing the same *shared memory*. Coprocessors, in particular those used for graphics processing, have even more parallelism on a single chip. High-performance computers consist of multiple server-type systems interconnected by a fast, dedicated network. Finally, more loosely connected computers of all types interact through various kinds of network (the Internet, radio networks, . . .) in *distributed systems* that may consist of millions of nodes. As you can imagine, no single simple model can be used to describe parallel programs running on these many levels of parallelism. We shall therefore restrict ourselves to occasional informal arguments as to why a certain sequential algorithm may be more or less easy to adapt to parallel processing. For example, the algorithms for high-precision arithmetic in Chap. 1 could make use of SIMD instructions.

2.3 Pseudocode

Our RAM model is an abstraction and simplification of the machine programs executed on microprocessors. The purpose of the model is to provide a precise definition of running time. However, the model is much too low-level for formulating complex algorithms. Our programs would become too long and too hard to read. Instead, we formulate our algorithms in *pseudocode*, which is an abstraction and simplification of imperative programming languages such as C, C++, Java, C#, and Pascal, combined with liberal use of mathematical notation. We now describe the conventions used in this book, and derive a timing model for pseudocode programs. The timing model is quite simple: *basic pseudocode instructions take constant time, and procedure and function calls take constant time plus the time to execute their body*. We justify the timing model by outlining how pseudocode can be translated into equivalent RAM code. We do this only to the extent necessary to understand the timing model. There is no need to worry about compiler optimization techniques, since constant factors are outside our theory. The reader may decide to skip the paragraphs describing the translation and adopt the timing model as an axiom. The syntax of our pseudocode is akin to that of Pascal [99], because we find this notation typographically nicer for a book than the more widely known syntax of C and its descendants C++ and Java.

2.3.1 Variables and Elementary Data Types

A *variable declaration* "$v = x : T$" introduces a variable v of type T, and initializes it with the value x. For example, "$answer = 42 : \mathbb{N}$" introduces a variable *answer* assuming integer values and initializes it to the value 42. When the type of a variable is clear from the context, we shall sometimes omit it from the declaration. A type is either a basic type (e.g., integer, Boolean value, or pointer) or a composite type. We have predefined composite types such as arrays, and application-specific classes (see below). When the type of a variable is irrelevant to the discussion, we use the unspecified type *Element* as a placeholder for an arbitrary type. We take the liberty of extending numeric types by the values $-\infty$ and ∞ whenever this is convenient. Similarly, we sometimes extend types by an undefined value (denoted by the symbol $\perp$), which we assume to be distinguishable from any "proper" element of the type T. In particular, for pointer types it is useful to have an undefined value. The values of the pointer type "**Pointer to** T" are handles of objects of type T. In the RAM model, this is the index of the first cell in a region of storage holding an object of type T.

A declaration "$a : Array\,[i..j]$ **of** T" introduces an *array* a consisting of $j - i + 1$ *elements* of type T, stored in $a[i], a[i+1], \ldots, a[j]$. Arrays are implemented as contiguous pieces of memory. To find an element $a[k]$, it suffices to know the starting address of a and the size of an object of type T. For example, if register R_a stores the starting address of array $a[0..k]$ and the elements have unit size, the instruction sequence "$R_1 := R_a + 42; R_2 := S[R_1]$" loads $a[42]$ into register R_2. The size of an array is fixed at the time of declaration; such arrays are called *static*. In Sect. 3.2, we show how to implement *unbounded arrays* that can grow and shrink during execution.

A declaration "c : **Class** *age* : $\mathbb{N}$, *income* : $\mathbb{N}$ **end**" introduces a variable c whose values are pairs of integers. The components of c are denoted by $c.age$ and $c.income$. For a variable c, **addressof** c returns the address of c. We also say that it returns a handle to c. If p is an appropriate pointer type, $p :=$ **addressof** c stores a handle to c in p and $*p$ gives us back c. The fields of c can then also be accessed through $p \rightarrow age$ and $p \rightarrow income$. Alternatively, one may write (but nobody ever does) $(*p).age$ and $(*p).income$.

Arrays and objects referenced by pointers can be allocated and deallocated by the commands **allocate** and **dispose**. For example, $p :=$ **allocate** *Array* $[1..n]$ **of** T allocates an array of n objects of type T. That is, the statement allocates a contiguous chunk of memory of size n times the size of an object of type T, and assigns a handle of this chunk (= the starting address of the chunk) to p. The statement **dispose** p frees this memory and makes it available for reuse. With **allocate** and **dispose**, we can cut our memory array S into disjoint pieces that can be referred to separately. These functions can be implemented to run in constant time. The simplest implementation is as follows. We keep track of the used portion of S by storing the index of the first free cell of S in a special variable, say *free*. A call of **allocate** reserves a chunk of memory starting at *free* and increases *free* by the size of the allocated chunk. A call of **dispose** does nothing. This implementation is time-efficient, but not space-efficient. Any call of **allocate** or **dispose** takes constant time. However, the total space consumption is the total space that has ever been allocated and not the maximal space simultaneously used, i.e., allocated but not yet freed, at any one time. It is not known whether an arbitrary sequence of **allocate** and **dispose** operations can be realized space-efficiently and with constant time per operation. However, for all algorithms presented in this book, **allocate** and **dispose** can be realized in a time- and space-efficient way.

We borrow some composite data structures from mathematics. In particular, we use tuples, sequences, and sets. *Pairs*, *triples*, and other *tuples* are written in round brackets, for example $(3,1)$, $(3,1,4)$, and $(3,1,4,1,5)$. Since tuples only contain a constant number of elements, operations on them can be broken into operations on their constituents in an obvious way. *Sequences* store elements in a specified order; for example "$s = \langle 3,1,4,1 \rangle$: *Sequence* **of** $\mathbb{Z}$" declares a sequence s of integers and initializes it to contain the numbers 3, 1, 4, and 1 in that order. Sequences are a natural abstraction of many data structures, such as files, strings, lists, stacks, and queues. In Chap. 3, we shall study many ways to represent sequences. In later chapters, we shall make extensive use of sequences as a mathematical abstraction with little further reference to implementation details. The empty sequence is written as $\langle\rangle$.

Sets play an important role in mathematical arguments and we shall also use them in our pseudocode. In particular, you shall see declarations such as "$M = \{3,1,4\}$

: *Set* **of** $\mathbb{N}$" that are analogous to declarations of arrays or sequences. Sets are usually implemented as sequences.

2.3.2 Statements

The simplest statement is an assignment $x := E$, where x is a variable and E is an expression. An assignment is easily transformed into a constant number of RAM instructions. For example, the statement $a := a + bc$ is translated into "$R_1 := R_b * R_c$; $R_a := R_a + R_1$", where R_a, R_b, and R_c stand for the registers storing a, b, and c, respectively. From C, we borrow the shorthands `++` and `--` for incrementing and decrementing variables. We also use parallel assignment to several variables. For example, if a and b are variables of the same type, "$(a,b) := (b,a)$" swaps the contents of a and b.

The conditional statement "**if** C **then** I **else** J", where C is a Boolean expression and I and J are statements, translates into the instruction sequence

$$\mathit{eval}(C);\ \texttt{JZ}\ \mathit{sElse},\ R_c;\ \mathit{trans}(I);\ \texttt{J}\ \mathit{sEnd};\ \mathit{trans}(J)\ ,$$

where $\mathit{eval}(C)$ is a sequence of instructions that evaluate the expression C and leave its value in register R_c, $\mathit{trans}(I)$ is a sequence of instructions that implement statement I, $\mathit{trans}(J)$ implements J, *sElse* is the address of the first instruction in $\mathit{trans}(J)$, and *sEnd* is the address of the first instruction after $\mathit{trans}(J)$. The sequence above first evaluates C. If C evaluates to false (= 0), the program jumps to the first instruction of the translation of J. If C evaluates to true (= 1), the program continues with the translation of I and then jumps to the instruction after the translation of J. The statement "**if** C **then** I" is a shorthand for "**if** C **then** I **else** ;", i.e., an if–then–else with an empty "else" part.

Our written representation of programs is intended for humans and uses less strict syntax than do programming languages. In particular, we usually group statements by indentation and in this way avoid the proliferation of brackets observed in programming languages such as C that are designed as a compromise between readability for humans and for computers. We use brackets only if the program would be ambiguous otherwise. For the same reason, a line break can replace a semicolon for the purpose of separating statements.

The loop "**repeat** I **until** C" translates into $\mathit{trans}(I)$; $\mathit{eval}(C)$; `JZ` *sI*, R_c, where *sI* is the address of the first instruction in $\mathit{trans}(I)$. We shall also use many other types of loop that can be viewed as shorthands for repeat loops. In the following list, the shorthand on the left expands into the statements on the right:

while C **do** I	**if** C **then repeat** I **until** $\neg C$
for $i := a$ **to** b **do** I	$i := a$; **while** $i \le b$ **do** I; i++
for $i := a$ **to** ∞ **while** C **do** I	$i := a$; **while** C **do** I; i++
foreach $e \in s$ **do** I	**for** $i := 1$ **to** $\lvert s\rvert$ **do** $e := s[i]$; I

Many low-level optimizations are possible when loops are translated into RAM code. These optimizations are of no concern for us. For us, it is only important that the execution time of a loop can be bounded by summing the execution times of each of its iterations, including the time needed for evaluating conditions.

2.3.3 Procedures and Functions

A subroutine with the name *foo* is declared in the form "**Procedure** *foo*(D) I", where I is the body of the procedure and D is a sequence of variable declarations specifying the parameters of *foo*. A call of *foo* has the form *foo*(P), where P is a parameter list. The parameter list has the same length as the variable declaration list. Parameter passing is either "by value" or "by reference". Our default assumption is that basic objects such as integers and Booleans are passed by value and that complex objects such as arrays are passed by reference. These conventions are similar to the conventions used by C and guarantee that parameter passing takes constant time. The semantics of parameter passing is defined as follows. For a value parameter x of type T, the actual parameter must be an expression E of the same type. Parameter passing is equivalent to the declaration of a local variable x of type T initialized to E. For a reference parameter x of type T, the actual parameter must be a variable of the same type and the formal parameter is simply an alternative name for the actual parameter.

As with variable declarations, we sometimes omit type declarations for parameters if they are unimportant or clear from the context. Sometimes we also declare parameters implicitly using mathematical notation. For example, the declaration **Procedure** *bar*$(\langle a_1, \ldots, a_n \rangle)$ introduces a procedure whose argument is a sequence of n elements of unspecified type.

Most procedure calls can be compiled into machine code by simply substituting the procedure body for the procedure call and making provisions for parameter passing; this is called *inlining*. Value passing is implemented by making appropriate assignments to copy the parameter values into the local variables of the procedure. Reference passing to a formal parameter $x : T$ is implemented by changing the type of x to **Pointer to** T, replacing all occurrences of x in the body of the procedure by $(*x)$ and initializing x by the assignment $x :=$ **addressof** y, where y is the actual parameter. Inlining gives the compiler many opportunities for optimization, so that inlining is the most efficient approach for small procedures and for procedures that are called from only a single place.

Functions are similar to procedures, except that they allow the return statement to return a value. Figure 2.2 shows the declaration of a recursive function that returns $n!$ and its translation into RAM code. The substitution approach fails for *recursive* procedures and functions that directly or indirectly call themselves – substitution would never terminate. Realizing recursive procedures in RAM code requires the concept of a *recursion stack*. Explicit subroutine calls over a stack are also used for large procedures that are called multiple times where inlining would unduly increase the code size. The recursion stack is a reserved part of the memory; we use *RS* to denote it. *RS* contains a sequence of *activation records*, one for each active procedure call. A special register R_r always points to the first free entry in this stack. The activation record for a procedure with k parameters and ℓ local variables has size $1 + k + \ell$. The first location contains the return address, i.e., the address of the instruction where execution is to be continued after the call has terminated, the next k locations are reserved for the parameters, and the final ℓ locations are for the local variables. A procedure call is now implemented as follows. First, the calling procedure *caller*

```
**Function** $factorial(n) : \mathbb{Z}$
    **if** $n = 1$ **then return** $1$ **else return** $n \cdot factorial(n-1)$
```

```
factorial:                                  // the first instruction of factorial
$R_n := RS[R_r - 1]$                        // load $n$ into register $R_n$
JZ thenCase, $R_n$                          // jump to then case, if $n$ is zero
$RS[R_r] =$ aRecCall                        // else case; return address for recursive call
$RS[R_r + 1] := R_n - 1$                    // parameter is $n-1$
$R_r := R_r + 2$                            // increase stack pointer
J factorial                                 // start recursive call
aRecCall:                                   // return address for recursive call
$R_{result} := RS[R_r - 1] * R_{result}$    // store $n * factorial(n-1)$ in result register
J return                                    // goto return
thenCase:                                   // code for then case
$R_{result} := 1$                           // put 1 into result register
return:                                     // code for return
$R_r := R_r - 2$                            // free activation record
J $RS[R_r]$                                 // jump to return address
```

Fig. 2.2. A recursive function *factorial* and the corresponding RAM code. The RAM code returns the function value in register R_{result}.

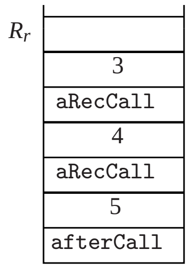

Fig. 2.3. The recursion stack of a call *factorial*(5) when the recursion has reached *factorial*(3)

pushes the return address and the actual parameters onto the stack, increases R_r accordingly, and jumps to the first instruction of the called routine *called*. The called routine reserves space for its local variables by increasing R_r appropriately. Then the body of *called* is executed. During execution of the body, any access to the i-th formal parameter ($0 \leq i < k$) is an access to $RS[R_r - \ell - k + i]$ and any access to the i-th local variable ($0 \leq i < \ell$) is an access to $RS[R_r - \ell + i]$. When *called* executes a **return** statement, it decreases R_r by $1 + k + \ell$ (observe that *called* knows k and ℓ) and execution continues at the return address (which can be found at $RS[R_r]$). Thus control is returned to *caller*. Note that recursion is no problem with this scheme, since each incarnation of a routine will have its own stack area for its parameters and local variables. Figure 2.3 shows the contents of the recursion stack of a call *factorial*(5) when the recursion has reached *factorial*(3). The label `afterCall` is the address of the instruction following the call *factorial*(5), and `aRecCall` is defined in Fig. 2.2.

Exercise 2.5 (sieve of Eratosthenes). Translate the following pseudocode for finding all prime numbers up to n into RAM machine code. Argue correctness first.

```
$a = \langle 1, \ldots, 1 \rangle$ : Array $[2..n]$ **of** $\{0,1\}$ // if $a[i]$ is false, $i$ is known to be nonprime
**for** $i := 2$ **to** $\lfloor \sqrt{n} \rfloor$ **do**
    **if** $a[i]$ **then for** $j := 2i$ **to** $n$ **step** $i$ **do** $a[j] := 0$
                // if $a[i]$ is true, $i$ is prime and all multiples of $i$ are nonprime
**for** $i := 2$ **to** $n$ **do** **if** $a[i]$ **then** output "$i$ is prime"
```

2.3.4 Object Orientation

We also need a simple form of object-oriented programming so that we can separate the interface and the implementation of the data structures. We shall introduce our notation by way of example. The definition

```
**Class** $Complex(x, y : Element)$ **of** $Number$
    $Number\ r := x$
    $Number\ i := y$
    **Function** $abs : Number$ **return** $\sqrt{r^2 + i^2}$
    **Function** $add(c' : Complex) : Complex$    **return** $Complex(r + c'.r, i + c'.i)$
```

gives a (partial) implementation of a complex number type that can use arbitrary numeric types for the real and imaginary parts. Very often, our class names will begin with capital letters. The real and imaginary parts are stored in the *member variables* r and i, respectively. Now, the declaration "$c : Complex(2,3)$ **of** $\mathbb{R}$" declares a complex number c initialized to $2 + 3i$; $c.i$ is the imaginary part, and $c.abs$ returns the absolute value of c.

The type after the **of** allows us to parameterize classes with types in a way similar to the template mechanism of C++ or the generic types of Java. Note that in the light of this notation, the types "*Set* **of** *Element*" and "*Sequence* **of** *Element*" mentioned earlier are ordinary classes. Objects of a class are initialized by setting the member variables as specified in the class definition.

2.4 Designing Correct Algorithms and Programs

An algorithm is a general method for solving problems of a certain kind. We describe algorithms using natural language and mathematical notation. Algorithms, as such, cannot be executed by a computer. The formulation of an algorithm in a programming language is called a program. Designing correct algorithms and translating a correct algorithm into a correct program are nontrivial and error-prone tasks. In this section, we learn about assertions and invariants, two useful concepts for the design of correct algorithms and programs.

2.4.1 Assertions and Invariants

Assertions and *invariants* describe properties of the program state, i.e., properties of single variables and relations between the values of several variables. Typical properties are that a pointer has a defined value, an integer is nonnegative, a list is nonempty, or the value of an integer variable *length* is equal to the length of a certain list L. Figure 2.4 shows an example of the use of assertions and invariants in a function $power(a, n_0)$ that computes a^{n_0} for a real number a and a nonnegative integer n_0.

We start with the assertion **assert** $n_0 \geq 0$ and $\neg(a = 0 \wedge n_0 = 0)$. This states that the program expects a nonnegative integer n_0 and that not both a and n_0 are allowed to be zero. We make no claim about the behavior of our program for inputs that violate this assertion. This assertion is therefore called the *precondition* of the program. It is good programming practice to check the precondition of a program, i.e., to write code which checks the precondition and signals an error if it is violated. When the precondition holds (and the program is correct), a *postcondition* holds at the termination of the program. In our example, we assert that $r = a^{n_0}$. It is also good programming practice to verify the postcondition before returning from a program. We shall come back to this point at the end of this section.

One can view preconditions and postconditions as a *contract* between the caller and the called routine: if the caller passes parameters satisfying the precondition, the routine produces a result satisfying the postcondition.

For conciseness, we shall use assertions sparingly, assuming that certain "obvious" conditions are implicit from the textual description of the algorithm. Much more elaborate assertions may be required for safety-critical programs or for formal verification.

Preconditions and postconditions are assertions that describe the initial and the final state of a program or function. We also need to describe properties of intermediate states. Some particularly important consistency properties should hold at many places in a program. These properties are called *invariants*. Loop invariants and data structure invariants are of particular importance.

```
**Function** $power(a : \mathbb{R}; n_0 : \mathbb{N}) : \mathbb{R}$
    **assert** $n_0 \geq 0$ *and* $\neg(a = 0 \wedge n_0 = 0)$          // It is not so clear what $0^0$ should be
    $p = a : \mathbb{R};\quad r = 1 : \mathbb{R};\quad n = n_0 : \mathbb{N}$              // we have: $p^n r = a^{n_0}$
    **while** $n > 0$ **do**
        **invariant** $p^n r = a^{n_0}$
        **if** $n$ *is odd* **then** $n$--; $r := r \cdot p$            // invariant violated between assignments
        **else** $(n, p) := (n/2, p \cdot p)$                // parallel assignment maintains invariant
    **assert** $r = a^{n_0}$                  // This is a consequence of the invariant and $n = 0$
    **return** $r$
```

Fig. 2.4. An algorithm that computes integer powers of real numbers.

2.4.2 Loop Invariants

A *loop invariant* holds before and after each loop iteration. In our example, we claim that $p^n r = a^{n_0}$ before each iteration. This is true before the first iteration. The initialization of the program variables takes care of this. In fact, an invariant frequently tells us how to initialize the variables. Assume that the invariant holds before execution of the loop body, and $n > 0$. If n is odd, we decrement n and multiply r by p. This reestablishes the invariant (note that the invariant is violated between the assignments). If n is even, we halve n and square p, and again reestablish the invariant. When the loop terminates, we have $p^n r = a^{n_0}$ by the invariant, and $n = 0$ by the condition of the loop. Thus $r = a^{n_0}$ and we have established the postcondition.

The algorithm in Fig. 2.4 and many more algorithms described in this book have a quite simple structure. A few variables are declared and initialized to establish the loop invariant. Then, a main loop manipulates the state of the program. When the loop terminates, the loop invariant together with the termination condition of the loop implies that the correct result has been computed. The loop invariant therefore plays a pivotal role in understanding why a program works correctly. Once we understand the loop invariant, it suffices to check that the loop invariant is true initially and after each loop iteration. This is particularly easy if the loop body consists of only a small number of statements, as in the example above.

2.4.3 Data Structure Invariants

More complex programs encapsulate their state in objects whose consistent representation is also governed by invariants. Such *data structure invariants* are declared together with the data type. They are true after an object is constructed, and they are preconditions and postconditions of all methods of a class. For example, we shall discuss the representation of sets by sorted arrays. The data structure invariant will state that the data structure uses an array a and an integer n, that n is the size of a, that the set S stored in the data structure is equal to $\{a[1], \ldots, a[n]\}$, and that $a[1] < a[2] < \ldots < a[n]$. The methods of the class have to maintain this invariant and they are allowed to leverage the invariant; for example, the search method may make use of the fact that the array is sorted.

2.4.4 Certifying Algorithms

We mentioned above that it is good programming practice to check assertions. It is not always clear how to do this efficiently; in our example program, it is easy to check the precondition, but there seems to be no easy way to check the postcondition. In many situations, however, *the task of checking assertions can be simplified by computing additional information*. This additional information is called a *certificate* or *witness*, and its purpose is to simplify the check of an assertion. When an algorithm computes a certificate for the postcondition, we call it a *certifying algorithm*. We shall illustrate the idea by an example. Consider a function whose input is a graph $G = (V, E)$. Graphs are defined in Sect. 2.9. The task is to test whether the graph is bipartite, i.e., whether there is a labeling of the nodes of G with the colors blue and red such that any edge of G connects nodes of different colors. As specified so far, the function returns true or false – true if G is bipartite, and false otherwise. With this rudimentary output, the postcondition cannot be checked. However, we may augment the program as follows. When the program declares G bipartite, it also returns a two-coloring of the graph. When the program declares G nonbipartite, it also returns a cycle of odd length in the graph. For the augmented program, the postcondition is easy to check. In the first case, we simply check whether all edges connect nodes of different colors, and in the second case, we do nothing. An odd-length cycle proves that the graph is nonbipartite. Most algorithms in this book can be made certifying without increasing the asymptotic running time.

2.5 An Example – Binary Search

Binary search is a very useful technique for searching in an ordered set of items. We shall use it over and over again in later chapters.

The simplest scenario is as follows. We are given a sorted array $a[1..n]$ of pairwise distinct elements, i.e., $a[1] < a[2] < \ldots < a[n]$, and an element x. Now we are required to find the index i with $a[i-1] < x \leq a[i]$; here, $a[0]$ and $a[n+1]$ should be interpreted as fictitious elements with values $-\infty$ and $+\infty$, respectively. We can use these fictitious elements in the invariants and the proofs, but cannot access them in the program.

Binary search is based on the principle of divide-and-conquer. We choose an index $m \in [1..n]$ and compare x with $a[m]$. If $x = a[m]$, we are done and return $i = m$. If $x < a[m]$, we restrict the search to the part of the array before $a[m]$, and if $x > a[m]$, we restrict the search to the part of the array after $a[m]$. We need to say more clearly what it means to restrict the search to a subinterval. We have two indices ℓ and r, and maintain the invariant

$$(I) \qquad 0 \leq \ell < r \leq n+1 \quad \text{and} \quad a[\ell] < x < a[r] \,.$$

This is true initially with $\ell = 0$ and $r = n+1$. If ℓ and r are consecutive indices, x is not contained in the array. Figure 2.5 shows the complete program.

The comments in the program show that the second part of the invariant is maintained. With respect to the first part, we observe that the loop is entered with $\ell < r$. If $\ell + 1 = r$, we stop and return. Otherwise, $\ell + 2 \leq r$ and hence $\ell < m < r$. Thus m is a legal array index, and we can access $a[m]$. If $x = a[m]$, we stop. Otherwise, we set either $r = m$ or $\ell = m$ and hence have $\ell < r$ at the end of the loop. Thus the invariant is maintained.

Let us argue for termination next. We observe first that if an iteration is not the last one, then we either increase ℓ or decrease r, and hence $r - \ell$ decreases. Thus the search terminates. We want to show more. We want to show that the search terminates in a logarithmic number of steps. To do this, we study the quantity $r - \ell - 1$. Note that this is the number of indices i with $\ell < i < r$, and hence a natural measure of the size of the current subproblem. We shall show that each iteration except the last at least halves the size of the problem. If an iteration is not the last, $r - \ell - 1$ decreases to something less than or equal to

$$\begin{aligned}
&\max\{r - \lfloor (r+\ell)/2 \rfloor - 1, \lfloor (r+\ell)/2 \rfloor - \ell - 1\} \\
&\quad \le \max\{r - ((r+\ell)/2 - 1/2) - 1, (r+\ell)/2 - \ell - 1\} \\
&\quad = \max\{(r - \ell - 1)/2, (r-\ell)/2 - 1\} = (r - \ell - 1)/2\,,
\end{aligned}$$

and hence it at least halved. We start with $r - \ell - 1 = n + 1 - 0 - 1 = n$, and hence have $r - \ell - 1 \le \lfloor n/2^k \rfloor$ after k iterations. The $(k+1)$-th iteration is certainly the last if we enter it with $r = \ell + 1$. This is guaranteed if $n/2^k < 1$ or $k > \log n$. We conclude that, at most, $2 + \log n$ iterations are performed. Since the number of comparisons is a natural number, we can sharpen the bound to $2 + \lfloor \log n \rfloor$.

Theorem 2.3. *Binary search finds an element in a sorted array of size n in $2 + \lfloor \log n \rfloor$ comparisons between elements.*

Exercise 2.6. Show that the above bound is sharp, i.e., for every n there are instances where exactly $2 + \lfloor \log n \rfloor$ comparisons are needed.

Exercise 2.7. Formulate binary search with two-way comparisons, i.e., distinguish between the cases $x < a[m]$, and $x \ge a[m]$.

We next discuss two important extensions of binary search. First, there is no need for the values $a[i]$ to be stored in an array. We only need the capability to compute $a[i]$, given i. For example, if we have a strictly monotonic function f and arguments i and j with $f(i) < x < f(j)$, we can use binary search to find m such that $f(m) \le x < f(m+1)$. In this context, binary search is often referred to as the *bisection method*.

Second, we can extend binary search to the case where the array is infinite. Assume we have an infinite array $a[1..\infty]$ with $a[1] \le x$ and want to find m such that $a[m] \le x < a[m+1]$. If x is larger than all elements in the array, the procedure is allowed to diverge. We proceed as follows. We compare x with $a[2^1]$, $a[2^2]$, $a[2^3]$, ..., until the first i with $x < a[2^i]$ is found. This is called an *exponential search*. Then we complete the search by binary search on the array $a[2^{i-1}..2^i]$.

```
(ℓ,r) := (0,n+1)
while true do
    invariant I                                         // i.e., invariant (I) holds here
    if ℓ+1 = r then return "a[ℓ] < x < a[ℓ+1]"
    m := ⌊(r+ℓ)/2⌋                                      // ℓ < m < r
    s := compare(x, a[m])               // −1 if x < a[m], 0 if x = a[m], +1 if x > a[m]
    if s = 0 then return "x is equal to a[m]";
    if s < 0
        then r := m                                     // a[ℓ] < x < a[m] = a[r]
        else ℓ := m                                     // a[ℓ] = a[m] < x < a[r]
```

Fig. 2.5. Binary Search for x in a sorted array $a[1..n]$.

Theorem 2.4. *The combination of exponential and binary search finds x in an unbounded sorted array in at most $2\log m+3$ comparisons, where $a[m] \le x < a[m+1]$.*

Proof. We need i comparisons to find the first i such that $x < a[2^i]$, followed by $\log(2^i - 2^{i-1})+2$ comparisons for the binary search. This gives a total of $2i+1$ comparisons. Since $m \ge 2^{i-1}$, we have $i \le 1+\log m$ and the claim follows. □

Binary search is certifying. It returns an index m with $a[m] \le x < a[m+1]$. If $x = a[m]$, the index proves that x is stored in the array. If $a[m] < x < a[m+1]$ and the array is sorted, the index proves that x is not stored in the array. Of course, if the array violates the precondition and is not sorted, we know nothing. There is no way to check the precondition in logarithmic time.

2.6 Basic Algorithm Analysis

Let us summarize the principles of algorithm analysis. We abstract from the complications of a real machine to the simplified RAM model. In the RAM model, running time is measured by the number of instructions executed. We simplify the analysis further by grouping inputs by size and focusing on the worst case. The use of asymptotic notation allows us to ignore constant factors and lower-order terms. This coarsening of our view also allows us to look at upper bounds on the execution time rather than the exact worst case, as long as the asymptotic result remains unchanged. The total effect of these simplifications is that the running time of pseudocode can be analyzed directly. There is no need to translate the program into machine code first.

We shall next introduce a set of simple rules for analyzing pseudocode. Let $T(I)$ denote the worst-case execution time of a piece of program I. The following rules then tell us how to estimate the running time for larger programs, given that we know the running times of their constituents:

- $T(I;I') = T(I) + T(I')$.
- $T(\textbf{if } C \textbf{ then } I \textbf{ else } I') = \mathrm{O}(T(C) + \max(T(I), T(I')))$.
- $T(\textbf{repeat } I \textbf{ until } C) = \mathrm{O}\left(\sum_{i=1}^{k} T(i)\right)$, where k is the number of loop iterations, and $T(i)$ is the time needed in the i-th iteration of the loop, including the test C.

We postpone the treatment of subroutine calls to Sect. 2.6.2. Of the rules above, only the rule for loops is nontrivial to apply; it requires evaluating sums.

2.6.1 "Doing Sums"

We now introduce some basic techniques for evaluating sums. Sums arise in the analysis of loops, in average-case analysis, and also in the analysis of randomized algorithms.

For example, the insertion sort algorithm introduced in Sect. 5.1 has two nested loops. The outer loop counts i, from 2 to n. The inner loop performs at most $i-1$ iterations. Hence, the total number of iterations of the inner loop is at most

$$\sum_{i=2}^{n}(i-1) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \mathrm{O}\left(n^2\right) \ ,$$

where the second equality comes from (A.11). Since the time for one execution of the inner loop is $\mathrm{O}(1)$, we get a worst-case execution time of $\Theta\left(n^2\right)$. All nested loops with an easily predictable number of iterations can be analyzed in an analogous fashion: work your way outwards by repeatedly finding a closed-form expression for the innermost loop. Using simple manipulations such as $\sum_i ca_i = c\sum_i a_i$, $\sum_i(a_i+b_i) = \sum_i a_i + \sum_i b_i$, or $\sum_{i=2}^{n} a_i = -a_1 + \sum_{i=1}^{n} a_i$, one can often reduce the sums to simple forms that can be looked up in a catalog of sums. A small sample of such formulae can be found in Appendix A. Since we are usually interested only in the asymptotic behavior, we can frequently avoid doing sums exactly and resort to estimates. For example, instead of evaluating the sum above exactly, we may argue more simply as follows:

$$\sum_{i=2}^{n}(i-1) \le \sum_{i=1}^{n} n = n^2 = \mathrm{O}\left(n^2\right) \ ,$$

$$\sum_{i=2}^{n}(i-1) \ge \sum_{i=\lceil n/2\rceil}^{n} n/2 = \lfloor n/2\rfloor \cdot n/2 = \Omega\left(n^2\right) \ .$$

2.6.2 Recurrences

In our rules for analyzing programs, we have so far neglected subroutine calls. Nonrecursive subroutines are easy to handle, since we can analyze the subroutine separately and then substitute the bound obtained into the expression for the running time of the calling routine. For recursive programs, however, this approach does not lead to a closed formula, but to a recurrence relation.

For example, for the recursive variant of the school method of multiplication, we obtained $T(1) = 1$ and $T(n) = 6n + 4T(\lceil n/2\rceil)$ for the number of primitive operations. For the Karatsuba algorithm, the corresponding expression was $T(n) = 3n^2 + 2n$ for $n \le 3$ and $T(n) = 12n + 3T(\lceil n/2\rceil + 1)$ otherwise. In general, a *recurrence relation* defines a function in terms of the same function using smaller arguments. Explicit definitions for small parameter values make the function well defined. Solving recurrences, i.e., finding nonrecursive, closed-form expressions for them, is an interesting subject in mathematics. Here we focus on the recurrence relations that typically emerge from divide-and-conquer algorithms. We begin with a simple case that will suffice for the purpose of understanding the main ideas. We have a problem of size $n = b^k$ for some integer k. If $k > 1$, we invest linear work cn in dividing the problem into d subproblems of size n/b and combining the results. If $k = 0$, there are no recursive calls, we invest work a, and are done.

Theorem 2.5 (master theorem (simple form)). *For positive constants a, b, c, and d, and $n = b^k$ for some integer k, consider the recurrence*

$$r(n) = \begin{cases} a & \text{if } n = 1 \ , \\ cn + d \cdot r(n/b) & \text{if } n > 1 \ . \end{cases}$$

Then

$$r(n) = \begin{cases} \Theta(n) & \text{if } d < b\,, \\ \Theta(n \log n) & \text{if } d = b\,, \\ \Theta\left(n^{\log_b d}\right) & \text{if } d > b\,. \end{cases}$$

Figure 2.6 illustrates the main insight behind Theorem 2.5. We consider the amount of work done at each level of recursion. We start with a problem of size n. At the i-th level of the recursion, we have d^i problems, each of size n/b^i. Thus the total size of the problems at the i-th level is equal to

$$d^i \frac{n}{b^i} = n \left(\frac{d}{b}\right)^i .$$

The work performed for a problem is c times the problem size, and hence the work performed at any level of the recursion is proportional to the total problem size at that level. Depending on whether d/b is less than, equal to, or larger than 1, we have different kinds of behavior.

If $d < b$, the work *decreases geometrically* with the level of recursion and the *first* level of recursion accounts for a constant fraction of the total execution time.

If $d = b$, we have the same amount of work at *every* level of recursion. Since there are logarithmically many levels, the total amount of work is $\Theta(n \log n)$.

Finally, if $d > b$, we have a geometrically *growing* amount of work at each level of recursion so that the *last* level accounts for a constant fraction of the total running time. We formalize this reasoning next.

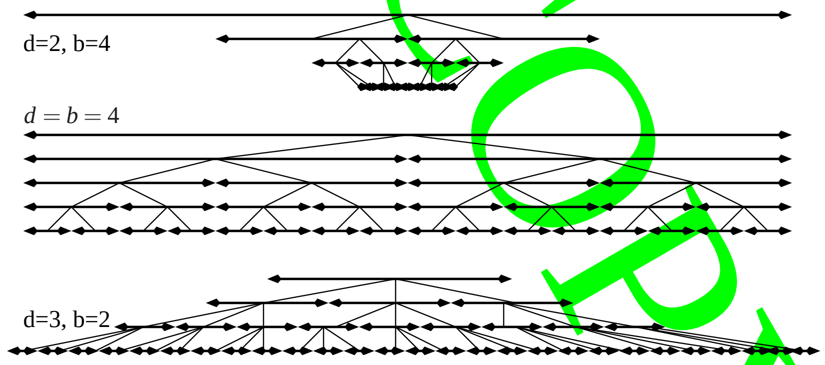

Fig. 2.6. Examples of the three cases of the master theorem. Problems are indicated by horizontal line segments with arrows at both ends. The length of a segment represents the size of the problem, and the subproblems resulting from a problem are shown in the line below it. The topmost part of figure corresponds to the case $d = 2$ and $b = 4$, i.e., each problem generates two subproblems of one-fourth the size. Thus the total size of the subproblems is only half of the original size. The middle part of the figure illustrates the case $d = b = 2$, and the bottommost part illustrates the case $d = 3$ and $b = 2$

Proof. We start with a single problem of size $n = b^k$. W call this level zero of the recursion.[3] At level 1, we have d problems, each of size $n/b = b^{k-1}$. At level 2, we have d^2 problems, each of size $n/b^2 = b^{k-2}$. At level i, we have d^i problems, each of size $n/b^i = b^{k-i}$. At level k, we have d^k problems, each of size $n/b^k = b^{k-k} = 1$. Each such problem has a cost a, and hence the total cost at level k is ad^k.

Let us next compute the total cost of the divide-and-conquer steps at levels 1 to $k-1$. At level i, we have d^i recursive calls each for subproblems of size b^{k-i}. Each call contributes a cost of $c \cdot b^{k-i}$, and hence the cost at level i is $d^i \cdot c \cdot b^{k-i}$. Thus the combined cost over all levels is

$$\sum_{i=0}^{k-1} d^i \cdot c \cdot b^{k-i} = c \cdot b^k \cdot \sum_{i=0}^{k-1} \left(\frac{d}{b}\right)^i = cn \cdot \sum_{i=0}^{k-1} \left(\frac{d}{b}\right)^i .$$

We now distinguish cases according to the relative sizes of d and b.

Case $d = b$. We have a cost $ad^k = ab^k = an = \Theta(n)$ for the bottom of the recursion and $cnk = cn\log_b n = \Theta(n\log n)$ for the divide-and-conquer steps.

Case $d < b$. We have a cost $ad^k < ab^k = an = \mathrm{O}(n)$ for the bottom of the recursion. For the cost of the divide-and-conquer steps, we use (A.13) for a geometric series, namely $\sum_{0 \le i < k} x^i = (1-x^k)/(1-x)$ for $x > 0$ and $x \neq 1$, and obtain

$$cn \cdot \sum_{i=0}^{k-1} \left(\frac{d}{b}\right)^i = cn \cdot \frac{1-(d/b)^k}{1-d/b} < cn \cdot \frac{1}{1-d/b} = \mathrm{O}(n)$$

and

$$cn \cdot \sum_{i=0}^{k-1} \left(\frac{d}{b}\right)^i = cn \cdot \frac{1-(d/b)^k}{1-d/b} > cn = \Omega(n) .$$

Case $d > b$. First, note that

$$d^k = 2^{k\log d} = 2^{k\frac{\log b}{\log b}\log d} = b^{k\frac{\log d}{\log b}} = b^{k\log_b d} = n^{\log_b d} .$$

Hence the bottom of the recursion has a cost of $an^{\log_b d} = \Theta\left(n^{\log_b d}\right)$. For the divide-and-conquer steps we use the geometric series again and obtain

$$cb^k \frac{(d/b)^k - 1}{d/b - 1} = c\frac{d^k - b^k}{d/b - 1} = cd^k \frac{1-(b/d)^k}{d/b - 1} = \Theta\left(d^k\right) = \Theta\left(n^{\log_b d}\right) .$$

□

We shall use the master theorem many times in this book. Unfortunately, the recurrence $T(n) = 3n^2 + 2n$ for $n \le 3$ and $T(n) \le 12n + 3T(\lceil n/2 \rceil + 1)$, governing

[3] In this proof, we use the terminology of recursive programs in order to give an intuitive idea of what we are doing. However, our mathematical arguments apply to any recurrence relation of the right form, even if it does not stem from a recursive program.

Karatsuba's algorithm, is not covered by our master theorem, which neglects rounding issues. We shall now show how to extend the master theorem to the following recurrence:

$$r(n) \le \begin{cases} a & \text{if } n \le n_0, \\ cn + d \cdot r(\lceil n/b \rceil + e) & \text{if } n > n_0, \end{cases}$$

where a, b, c, d, and e are constants, and n_0 is such that $\lceil n/b \rceil + e < n$ for $n > n_0$. We proceed in two steps. We first concentrate on n of the form $b^k + z$, where z is such that $\lceil z/b \rceil + e = z$. For example, for $b = 2$ and $e = 3$, we would choose $z = 6$. Note that for n of this form, we have $\lceil n/b \rceil + e = \lceil (b^k + z)/b \rceil + e = b^{k-1} + \lceil z/b \rceil + e = b^{k-1} + z$, i.e., the reduced problem size has the same form. For the n's in this special form, we then argue exactly as in Theorem 2.5.

How do we generalize to arbitrary n? The simplest way is semantic reasoning. It is clear[4] that the cost grows with the problem size, and hence the cost for an input of size n will be no larger than the cost for an input whose size is equal to the next input size of special form. Since this input is at most b times larger and b is a constant, the bound derived for special n is affected only by a constant factor.

The formal reasoning is as follows (you may want to skip this paragraph and come back to it when the need arises). We define a function $R(n)$ by the same recurrence, with $\le$ replaced by equality: $R(n) = a$ for $n \le n_0$ and $R(n) = cn + dR(\lceil n/b \rceil + e)$ for $n > n_0$. Obviously, $r(n) \le R(n)$. We derive a bound for $R(n)$ and n of special form as described above. Finally, we argue by induction that $R(n) \le R(s(n))$, where $s(n)$ is the smallest number of the form $b^k + z$ with $b^k + z \ge n$. The induction step is as follows:

$$R(n) = cn + dR(\lceil n/b \rceil + e) \le cs(n) + dR(s(\lceil n/b \rceil + e)) = R(s(n)) ,$$

where the inequality uses the induction hypothesis and $n \le s(n)$. The last equality uses the fact that for $s(n) = b^k + z$ (and hence $b^{k-1} + z < n$), we have $b^{k-2} + z < \lceil n/b \rceil + e \le b^{k-1} + z$ and hence $s(\lceil n/b \rceil + e) = b^{k-1} + z = \lceil s(n)/b \rceil + e$.

There are many generalizations of the master theorem: we might break the recursion earlier, the cost for dividing and conquering may be nonlinear, the size of the subproblems might vary within certain bounds, the number of subproblems may depend on the input size, etc. We refer the reader to the books [81, 175] for further information.

Exercise 2.8. Consider the recurrence

$$C(n) = \begin{cases} 1 & \text{if } n = 1, \\ C(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil) + cn & \text{if } n > 1. \end{cases}$$

Show that $C(n) = \mathrm{O}(n \log n)$.

[4] Be aware that most errors in mathematical arguments are near occurrences of the word "clearly".

***Exercise 2.9.** Suppose you have a divide-and-conquer algorithm whose running time is governed by the recurrence $T(1) = a$, $T(n) = cn + \lceil\sqrt{n}\,\rceil T(\lceil n/\lceil\sqrt{n}\,\rceil\rceil)$. Show that the running time of the program is $\mathrm{O}(n\log\log n)$.

Exercise 2.10. Access to data structures is often governed by the following recurrence: $T(1) = a$, $T(n) = c + T(n/2)$. Show that $T(n) = \mathrm{O}(\log n)$.

2.6.3 Global Arguments

The algorithm analysis techniques introduced so far are syntax-oriented in the following sense: in order to analyze a large program, we first analyze its parts and then combine the analyses of the parts into an analysis of the large program. The combination step involves sums and recurrences.

We shall also use a completely different approach which one might call semantics-oriented. In this approach we associate parts of the execution with parts of a combinatorial structure and then argue about the combinatorial structure. For example, we might argue that a certain piece of program is executed at most once for each edge of a graph or that the execution of a certain piece of program at least doubles the size of a certain structure, that the size is one initially, and at most n at termination, and hence the number of executions is bounded logarithmically.

2.7 Average-Case Analysis

In this section we shall introduce you to average-case analysis. We shall do so by way of three examples of increasing complexity. We assume that you are familiar with basic concepts of probability theory such as discrete probability distributions, expected values, indicator variables, and the linearity of expectations. Section A.3 reviews the basics.

2.7.1 Incrementing a Counter

We begin with a very simple example. Our input is an array $a[0..n-1]$ filled with digits zero and one. We want to increment the number represented by the array by one.

```
$i := 0$
**while** $(i < n$ *and* $a[i] = 1)$ **do** $a[i] = 0$; $i$++;
**if** $i < n$ **then** $a[i] = 1$
```

How often is the body of the while loop executed? Clearly, n times in the worst case and 0 times in the best case. What is the average case? The first step in an average-case analysis is always to define the model of randomness, i.e., to define the underlying probability space. We postulate the following model of randomness: each digit is zero or one with probability $1/2$, and different digits are independent. The loop body is executed k times, $0 \le k \le n$, iff the last $k+1$ digits of a are 01^k or k

is equal to n and all digits of a are equal to one. The former event has probability $2^{-(k+1)}$, and the latter event has probability 2^{-n}. Therefore, the average number of executions is equal to

$$\sum_{0 \le k < n} k2^{-(k+1)} + n2^{-n} \le \sum_{k \ge 0} k2^{-k} = 2 ,$$

where the last equality is the same as (A.14).

2.7.2 Left-to-Right Maxima

Our second example is slightly more demanding. Consider the following simple program that determines the maximum element in an array $a[1..n]$:

```
$m := a[1];$    for $i := 2$ to $n$ do if $a[i] > m$ then $m := a[i]$
```

How often is the assignment $m := a[i]$ executed? In the worst case, it is executed in every iteration of the loop and hence $n-1$ times. In the best case, it is not executed at all. What is the average case? Again, we start by defining the probability space. We assume that the array contains n distinct elements and that any order of these elements is equally likely. In other words, our probability space consists of the $n!$ permutations of the array elements. Each permutation is equally likely and therefore has probability $1/n!$. Since the exact nature of the array elements is unimportant, we may assume that the array contains the numbers 1 to n in some order. We are interested in the average number of *left-to-right maxima*. A left-to-right maximum in a sequence is an element which is larger than all preceding elements. So, $(1,2,4,3)$ has three left-to-right-maxima and $(3,1,2,4)$ has two left-to-right-maxima. For a permutation π of the integers 1 to n, let $M_n(\pi)$ be the number of left-to-right-maxima. What is $\mathrm{E}[M_n]$? We shall describe two ways to determine the expectation. For small n, is easy to determine $\mathrm{E}[M_n]$ by direct calculation. For $n = 1$, there is only one permutation, namely (1), and it has one maximum. So $\mathrm{E}[M_1] = 1$. For $n = 2$, there are two permutations, namely $(1,2)$ and $(2,1)$. The former has two maxima and the latter has one maximum. So $\mathrm{E}[M_2] = 1.5$. For larger n, we argue as follows.

We write M_n as a sum of indicator variables I_1 to I_n, i.e., $M_n = I_1 + \ldots + I_n$, where I_k is equal to one for a permutation π if the k-th element of π is a left-to-right maximum. For example, $I_3((3,1,2,4)) = 0$ and $I_4((3,1,2,4)) = 1$. We have

$$\begin{aligned}\mathrm{E}[M_n] &= \mathrm{E}[I_1 + I_2 + \ldots + I_n] \\ &= \mathrm{E}[I_1] + \mathrm{E}[I_2] + \ldots + \mathrm{E}[I_n] \\ &= \mathrm{prob}(I_1 = 1) + \mathrm{prob}(I_2 = 1) + \ldots + \mathrm{prob}(I_n = 1) ,\end{aligned}$$

where the second equality is the linearity of expectations (A.2) and the third equality follows from the I_k's being indicator variables. It remains to determine the probability that $I_k = 1$. The k-th element of a random permutation is a left-to-right maximum if and only if the k-th element is the largest of the first k elements. In a random permutation, any position is equally likely to hold the maximum, so that the probability we are looking for is $\mathrm{prob}(I_k = 1) = 1/k$ and hence

$$\mathrm{E}[M_n] = \sum_{1\le k\le n} \mathrm{prob}(I_k = 1) = \sum_{1\le k\le n} \frac{1}{k}\,.$$

So, $\mathrm{E}[M_4] = 1+1/2+1/3+1/4 = (12+6+4+3)/12 = 25/12$. The sum $\sum_{1\le k\le n} 1/k$ will appear several times in this book. It is known under the name "n-th harmonic number" and is denoted by H_n. It is known that $\ln n \le H_n \le 1+\ln n$, i.e., $H_n \approx \ln n$; see (A.12). We conclude that the average number of left-to-right maxima is much smaller than in the worst case.

Exercise 2.11. Show that $\sum_{k=1}^{n} \frac{1}{k} \le \ln n + 1$. Hint: show first that $\sum_{k=2}^{n} \frac{1}{k} \le \int_1^n \frac{1}{x}\,dx$.

We now describe an alternative analysis. We introduce A_n as a shorthand for $\mathrm{E}[M_n]$ and set $A_0 = 0$. The first element is always a left-to-right maximum, and each number is equally likely as the first element. If the first element is equal to i, then only the numbers $i+1$ to n can be further left-to-right maxima. They appear in random order in the remaining sequence, and hence we shall see an expected number of A_{n-i} further maxima. Thus

$$A_n = 1 + \left(\sum_{1\le i\le n} A_{n-i}\right)/n \qquad \text{or} \qquad nA_n = n + \sum_{0\le i\le n-1} A_i\,.$$

A simple trick simplifies this recurrence. The corresponding equation for $n-1$ instead of n is $(n-1)A_{n-1} = n-1+\sum_{1\le i\le n-2} A_i$. Subtracting the equation for $n-1$ from the equation for n yields

$$nA_n - (n-1)A_{n-1} = 1 + A_{n-1} \qquad \text{or} \qquad A_n = 1/n + A_{n-1}\,,$$

and hence $A_n = H_n$.

2.7.3 Linear Search

We come now to our third example; this example is even more demanding. Consider the following search problem. We have items 1 to n, which we are required to arrange linearly in some order; say, we put item i in position ℓ_i. Once we have arranged the items, we perform searches. In order to search for an item x, we go through the sequence from left to right until we encounter x. In this way, it will take ℓ_i steps to access item i.

Suppose now that we also know that we shall access the items with different probabilities; say, we search for item i with probability p_i, where $p_i \ge 0$ for all i, $1 \le i \le n$, and $\sum_i p_i = 1$. In this situation, the *expected* or *average* cost of a search is equal to $\sum_i p_i \ell_i$, since we search for item i with probability p_i and the cost of the search is ℓ_i.

What is the best way of arranging the items? Intuition tells us that we should arrange the items in order of decreasing probability. Let us prove this.

Lemma 2.6. *An arrangement is optimal with respect to the expected search cost if it has the property that $p_i > p_j$ implies $\ell_i < \ell_j$. If $p_1 \geq p_2 \geq \ldots \geq p_n$, the placement $\ell_i = i$ results in the optimal expected search cost $Opt = \sum_i p_i i$.*

Proof. Consider an arrangement in which, for some i and j, we have $p_i > p_j$ and $\ell_i > \ell_j$, i.e., item i is more probable than item j and yet placed after it. Interchanging items i and j changes the search cost by

$$-(p_i\ell_i + p_j\ell_j) + (p_i\ell_j + p_j\ell_i) = (p_i - p_j)(\ell_i - \ell_j) < 0 \,,$$

i.e., the new arrangement is better and hence the old arrangement is not optimal.

Let us now consider the case $p_1 > p_2 > \ldots > p_n$. Since there are only $n!$ possible arrangements, there is an optimal arrangement. Also, if $i < j$ and i is placed after j, the arrangement is not optimal by the argument in the preceding paragraph. Thus the optimal arrangement puts item i in position $\ell_i = i$ and its expected search cost is $\sum_i p_i i$.

If $p_1 \geq p_2 \geq \ldots \geq p_n$, the arrangement $\ell_i = i$ for all i is still optimal. However, if some probabilities are equal, we have more than one optimal arrangement. Within blocks of equal probabilities, the order is irrelevant. □

Can we still do something intelligent if the probabilities p_i are not known to us? The answer is yes, and a very simple heuristic does the job. It is called the *move-to-front heuristic*. Suppose we access item i and find it in position ℓ_i. If $\ell_i = 1$, we are happy and do nothing. Otherwise, we place it in position 1 and move the items in positions 1 to $\ell_i - 1$ one position to the rear. The hope is that, in this way, frequently accessed items tend to stay near the front of the arrangement and infrequently accessed items move to the rear. We shall now analyze the expected behavior of the move-to-front heuristic.

Consider two items i and j and suppose that both of them were accessed in the past. Item i will be accessed before item j if the last access to item i occurred after the last access to item j. Thus the probability that item i is before item j is $p_i/(p_i + p_j)$. With probability $p_j/(p_i + p_j)$, item j stands before item i.

Now, ℓ_i is simply one plus the number of elements before i in the list. Thus the expected value of ℓ_i is equal to $1 + \sum_{j;\, j \neq i} p_j/(p_i + p_j)$, and hence the expected search cost in the move-to-front heuristic is

$$C_{MTF} = \sum_i p_i \left(1 + \sum_{j;\, j\neq i} \frac{p_j}{p_i + p_j} \right) = \sum_i p_i + \sum_{i,j;\, i\neq j} \frac{p_i p_j}{p_i + p_j} \,.$$

Observe that for each i and j with $i \neq j$, the term $p_i p_j/(p_i + p_j)$ appears twice in the sum above. In order to proceed with the analysis, we assume $p_1 \geq p_2 \geq \ldots \geq p_n$. This is an assumption used in the analysis, the algorithm has no knowledge of this. Then

$$C_{MTF} = \sum_i p_i + 2 \sum_{j;\ j<i} \frac{p_i p_j}{p_i + p_j} = \sum_i p_i \left(1 + 2 \sum_{j;\ j<i} \frac{p_j}{p_i + p_j}\right)$$
$$\leq \sum_i p_i \left(1 + 2 \sum_{j;\ j<i} 1\right) < \sum_i p_i 2i = 2\sum_i p_i i = 2Opt\ .$$

Theorem 2.7. *The move-to-front heuristic achieves an expected search time which is at most twice the optimum.*

2.8 Randomized Algorithms

Suppose you are offered the chance to participate in a TV game show. There are 100 boxes that you can open in an order of your choice. Box i contains an amount m_i of money. This amount is unknown to you but becomes known once the box is opened. No two boxes contain the same amount of money. The rules of the game are very simple:

- At the beginning of the game, the presenter gives you 10 tokens.
- When you open a box and the contents of the box are larger than the contents of all previously opened boxes, you have to hand back a token.[5]
- When you have to hand back a token but have no tokens, the game ends and you lose.
- When you manage to open all of the boxes, you win and can keep all the money.

There are strange pictures on the boxes, and the presenter gives hints by suggesting the box to be opened next. Your aunt, who is addicted to this show, tells you that only a few candidates win. Now, you ask yourself whether it is worth participating in this game. Is there a strategy that gives you a good chance of winning? Are the presenter's hints useful?

Let us first analyze the obvious algorithm – you always follow the presenter. The worst case is that he makes you open the boxes in order of increasing value. Whenever you open a box, you have to hand back a token, and when you open the 11th box you are dead. The candidates and viewers would hate the presenter and he would soon be fired. Worst-case analysis does not give us the right information in this situation. The best case is that the presenter immediately tells you the best box. You would be happy, but there would be no time to place advertisements, so that the presenter would again be fired. Best-case analysis also does not give us the right information in this situation. We next observe that the game is really the left-to-right maxima question of the preceding section in disguise. You have to hand back a token whenever a new maximum shows up. We saw in the preceding section that the expected number of left-to-right maxima in a random permutation is H_n, the n-th harmonic number. For $n = 100$, $H_n < 6$. So if the presenter were to point to the boxes

[5] The contents of the first box opened are larger than the contents of all previously opened boxes, and hence the first token goes back to the presenter in the first round.

in random order, you would have to hand back only 6 tokens on average. But why should the presenter offer you the boxes in random order? He has no incentive to have too many winners.

The solution is to take your fate into your own hands: *open the boxes in random order*. You select one of the boxes at random, open it, then choose a random box from the remaining ones, and so on. How do you choose a random box? When there are k boxes left, you choose a random box by tossing a die with k sides or by choosing a random number in the range 1 to k. In this way, you generate a random permutation of the boxes and hence the analysis in the previous section still applies. On average you will have to return fewer than 6 tokens and hence your 10 tokens suffice. You have just seen a *randomized algorithm*. We want to stress that, although the mathematical analysis is the same, the conclusions are very different. In the average-case scenario, you are at the mercy of the presenter. If he opens the boxes in random order, the analysis applies; if he does not, it does not. You have no way to tell, except after many shows and with hindsight. In other words, the presenter controls the dice and it is up to him whether he uses fair dice. The situation is completely different in the randomized-algorithms scenario. You control the dice, and you generate the random permutation. The analysis is valid no matter what the presenter does.

2.8.1 The Formal Model

Formally, we equip our RAM with an additional instruction: $R_i := randInt(C)$ assigns a *random* integer between 0 and $C-1$ to R_i. In pseudocode, we write $v := randInt(C)$, where v is an integer variable. The cost of making a random choice is one time unit. Algorithms *not* using randomization are called *deterministic*.

The running time of a randomized algorithm will generally depend on the random choices made by the algorithm. So the running time on an instance i is no longer a number, but a random variable depending on the random choices. We may eliminate the dependency of the running time on random choices by equipping our machine with a timer. At the beginning of the execution, we set the timer to a value $T(n)$, which may depend on the size n of the problem instance, and stop the machine once the timer goes off. In this way, we can guarantee that the running time is bounded by $T(n)$. However, if the algorithm runs out of time, it does not deliver an answer.

The output of a randomized algorithm may also depend on the random choices made. How can an algorithm be useful if the answer on an instance i may depend on the random choices made by the algorithm – if the answer may be “Yes” today and “No” tomorrow? If the two cases are equally probable, the answer given by the algorithm has no value. However, if the correct answer is much more likely than the incorrect answer, the answer does have value. Let us see an example.

Alice and Bob are connected over a slow telephone line. Alice has an integer x_A and Bob has an integer x_B, each with n bits. They want to determine whether they have the same number. As communication is slow, their goal is to minimize the amount of information exchanged. Local computation is not an issue.

In the obvious solution, Alice sends her number to Bob, and Bob checks whether the numbers are equal and announces the result. This requires them to transmit n

digits. Alternatively, Alice could send the number digit by digit, and Bob would check for equality as the digits arrived and announce the result as soon as he knew it, i.e., as soon as corresponding digits differed or all digits had been transmitted. In the worst case, all n digits have to be transmitted. We shall now show that randomization leads to a dramatic improvement. After transmission of only $O(\log n)$ bits, equality and inequality can be decided with high probability.

Alice and Bob follow the following protocol. Each of them prepares an ordered list of prime numbers. The list consists of the smallest L primes with k or more bits and leading bit 1. Each such prime has a value of at least 2^k. We shall say more about the choice of L and k below. In this way, it is guaranteed that both Alice and Bob generate the same list. Then Alice chooses an index i, $1 \le i \le L$, at random and sends i and $x_A \bmod p_i$ to Bob. Bob computes $x_B \bmod p_i$. If $x_A \bmod p_i \ne x_B \bmod p_i$, he declares that the numbers are different. Otherwise, he declares the numbers the same. Clearly, if the numbers are the same, Bob will say so. If the numbers are different and $x_A \bmod p_i \ne x_B \bmod p_i$, he will declare them different. However, if $x_A \ne x_B$ and yet $x_A \bmod p_i = x_B \bmod p_i$, he will erroneously declare the numbers equal. What is the probability of an error?

An error occurs if $x_A \ne x_B$ but $x_A \equiv x_B (\bmod p_i)$. The latter condition is equivalent to p_i dividing the difference $D = x_A - x_B$. This difference is at most 2^n in absolute value. Since each prime p_i has a value of at least 2^k, our list contains at most n/k primes that divide[6] the difference, and hence the probability of error is at most $(n/k)/L$. We can make this probability arbitrarily small by choosing L large enough. If, say, we want to make the probability less than $0.0000001 = 10^{-6}$, we choose $L = 10^6(n/k)$.

What is the appropriate choice of k? Out of the numbers with k bits, approximately $2^k/k$ are primes.[7] Hence, if $2^k/k \ge 10^6 n/k$, the list will contain only k-bit integers. The condition $2^k \ge 10^6 n$ is tantamount to $k \ge \log n + 6\log 10$. With this choice of k, the protocol transmits $\log L + k = \log n + 12\log 10$ bits. *This is exponentially better than the naive protocol.*

What can we do if we want an error probability less than 10^{-12}? We could redo the calculations above with $L = 10^{12}n$. Alternatively, we could run the protocol twice and declare the numbers different if at least one run declares them different. This two-stage protocol errs only if both runs err, and hence the probability of error is at most $10^{-6} \cdot 10^{-6} = 10^{-12}$.

Exercise 2.12. Compare the efficiency of the two approaches for obtaining an error probability of 10^{-12}.

Exercise 2.13. In the protocol described above, Alice and Bob have to prepare ridiculously long lists of prime numbers. Discuss the following modified protocol.

[6] Let d be the number of primes on our list that divide D. Then $2^n \ge |D| \ge (2^k)^d = 2^{kd}$ and hence $d \le n/k$.

[7] For any integer x, let $\pi(x)$ be the number of primes less than or equal to x. For example, $\pi(10) = 4$ because there are four prime numbers (2, 3, 5 and 7) less than or equal to 10. Then $x/(\ln x + 2) < \pi(x) < x/(\ln x - 4)$ for $x \ge 55$. See the Wikipedia entry on "prime numbers" for more information.

Alice chooses a random k-bit integer p (with leading bit 1) and tests it for primality. If p is not prime, she repeats the process. If p is prime, she sends p and x_A mod p to Bob.

Exercise 2.14. Assume you have an algorithm which errs with a probability of at most $1/4$ and that you run the algorithm k times and output the majority output. Derive a bound on the error probability as a function of k. Do a precise calculation for $k = 2$ and $k = 3$, and give a bound for large k. Finally, determine k such that the error probability is less than a given ε.

2.8.2 Las Vegas and Monte Carlo Algorithms

Randomized algorithms come in two main varieties, the Las Vegas and the Monte Carlo variety. A *Las Vegas algorithm* always computes the correct answer but its running time is a random variable. Our solution for the game show is a Las Vegas algorithm; it always finds the box containing the maximum; however, the number of left-to-right maxima is a random variable. A *Monte Carlo* algorithm always has the same run time, but there is a nonzero probability that it gives an incorrect answer. The probability that the answer is incorrect is at most 1/4. Our algorithm for comparing two numbers over a telephone line is a Monte Carlo algorithm. In Exercise 2.14, it is shown that the error probability can be made arbitrarily small.

Exercise 2.15. Suppose you have a Las Vegas algorithm with an expected execution time $t(n)$, and that you run it for $4t(n)$ steps. If it returns an answer within the alloted time, this answer is returned, otherwise, an arbitrary answer is returned. Show that the resulting algorithm is a Monte Carlo algorithm.

Exercise 2.16. Suppose you have a Monte Carlo algorithm with an execution time $m(n)$ that gives a correct answer with probability p and a deterministic algorithm that verifies in time $v(n)$ whether the Monte Carlo algorithm has given the correct answer. Explain how to use these two algorithms to obtain a Las Vegas algorithm with expected execution time $(m(n) + v(n))/(1 - p)$.

We come back to our game show example. You have 10 tokens available to you. The expected number of tokens required is less than 6. How sure should you be that you will go home a winner? We need to bound the probability that M_n is larger than 11, because you lose exactly if the sequence in which you order the boxes has 11 or more left-to-right maxima. *Markov's inequality* allows you to bound this probability. It states that, for a nonnegative random variable X and any constant $c \geq 1$, $\mathrm{prob}(X \geq c \cdot \mathrm{E}[X]) \leq 1/c$; see (A.4) for additional information. We apply the inequality with $X = M_n$ and $c = 11/6$. We obtain

$$\mathrm{prob}(M_n \geq 11) \leq \mathrm{prob}\left(M_n \geq \frac{11}{6}\mathrm{E}[M_n]\right) \leq \frac{6}{11}\,,$$

and hence the probability of winning is more than 5/11.

2.9 Graphs

Graphs are an extremely useful concept in algorithmics. We use them whenever we want to model objects and relations between them; in graph terminology, the objects are called *nodes*, and the relations between nodes are called *edges*. Some obvious applications are to road maps and communication networks, but there are also more abstract applications. For example, nodes could be tasks to be completed when building a house, such as “build the walls” or “put in the windows”, and edges could model precedence relations such as “the walls have to be built before the windows can be put in”. We shall also see many examples of data structures where it is natural to view objects as nodes and pointers as edges between the object storing the pointer and the object pointed to.

When humans think about graphs, they usually find it convenient to work with pictures showing nodes as bullets and edges as lines and arrows. To treat graphs algorithmically, a more mathematical notation is needed: a *directed graph* $G = (V,E)$ is a pair consisting of a *node set* (or *vertex set*) V and an *edge set* (or *arc set*) $E \subseteq V \times V$. We sometimes abbreviate “directed graph” to *digraph*. For example, Fig. 2.7 shows the graph $G = (\{s,t,u,v,w,x,y,z\}, \{(s,t),(t,u),(u,v),(v,w),(w,x),(x,y),(y,z),(z,s),(s,v),(z,w),(y,t),(x,u)\})$. Throughout this book, we use the convention $n = |V|$ and $m = |E|$ if no other definitions for n or m are given. An edge $e = (u,v) \in E$ represents a connection from u to v. We call u and v the *source* and *target*, respectively, of e. We say that e is *incident* on u and v and that v and u are *adjacent*. The special case of a *self-loop* (v,v) is disallowed unless specifically mentioned.

The *outdegree* of a node v is the number of edges leaving it, and its *indegree* is the number of edges ending at it, formally, $\textit{outdegree}(v) = |\{(v,u) \in E\}|$ and $\textit{indegree}(v) = |\{(u,v) \in E\}|$. For example, node w in graph G in Fig. 2.7 has indegree two and outdegree one.

A *bidirected graph* is a digraph where, for any edge (u,v), the reverse edge (v,u) is also present. An *undirected graph* can be viewed as a streamlined representation of a bidirected graph, where we write a pair of edges (u,v), (v,u) as the two-element set $\{u,v\}$. Figure 2.7 shows a three-node undirected graph and its bidirected counterpart. Most graph-theoretic terms for undirected graphs have the same definition as for

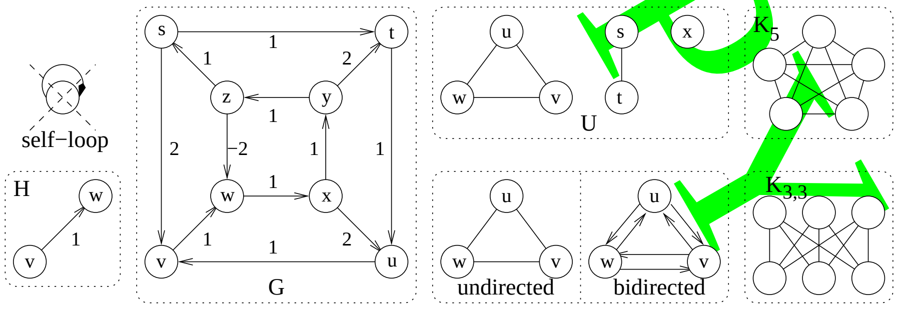

Fig. 2.7. Some graphs

their bidirected counterparts, and so this section will concentrate on directed graphs and only mention undirected graphs when there is something special about them. For example, the number of edges of an undirected graph is only half the number of edges of its bidirected counterpart. Nodes of an undirected graph have identical indegree and outdegree, and so we simply talk about their *degree*. Undirected graphs are important because directions often do not matter and because many problems are easier to solve (or even to define) for undirected graphs than for general digraphs.

A graph $G' = (V', E')$ is a *subgraph* of G if $V' \subseteq V$ and $E' \subseteq E$. Given $G = (V,E)$ and a subset $V' \subseteq V$, the subgraph *induced* by V' is defined as $G' = (V', E \cap (V' \times V'))$. In Fig. 2.7, the node set $\{v,w\}$ in G induces the subgraph $H = (\{v,w\}, \{(v,w)\})$. A subset $E' \subseteq E$ of edges induces the subgraph (V,E').

Often, additional information is associated with nodes or edges. In particular, we shall often need *edge weights* or *costs* $c : E \to \mathbb{R}$ that map edges to some numeric value. For example, the edge (z,w) in graph G in Fig. 2.7 has a weight $c((z,w)) = -2$. Note that an edge $\{u,v\}$ of an undirected graph has a unique edge weight, whereas, in a bidirected graph, we can have $c((u,v)) \neq c((v,u))$.

We have now seen quite a lot of definitions on one page of text. If you want to see them at work, you may jump to Chap. 8 to see algorithms operating on graphs. But things are also becoming more interesting here.

An important higher-level graph-theoretic concept is the notion of a path. A *path* $p = \langle v_0, \ldots, v_k \rangle$ is a sequence of nodes in which consecutive nodes are connected by edges in E, i.e., $(v_0,v_1) \in E$, $(v_1,v_2) \in E$, ..., $(v_{k-1},v_k) \in E$; p has length k and runs from v_0 to v_k. Sometimes a path is also represented by its sequence of edges. For example, $\langle u,v,w \rangle = \langle (u,v),(v,w) \rangle$ is a path of length 2 in Fig. 2.7. A path is *simple* if its nodes, except maybe for v_0 and v_k, are pairwise distinct. In Fig. 2.7, $\langle z,w,x,u,v,w,x,y \rangle$ is a nonsimple path.

Cycles are paths with a common first and last node. A simple cycle visiting all nodes of a graph is called a *Hamiltonian* cycle. For example, the cycle $\langle s,t,u,v,w,x,y,z,s \rangle$ in graph G in Fig. 2.7 is Hamiltonian. A simple undirected cycle contains at least three nodes, since we also do not allow edges to be used twice in simple undirected cycles.

The concepts of paths and cycles help us to define even higher-level concepts. A digraph is *strongly connected* if for any two nodes u and v there is a path from u to v. Graph G in Fig. 2.7 is strongly connected. A strongly connected component of a digraph is a maximal node-induced strongly connected subgraph. If we remove edge (w,x) from G in Fig. 2.7, we obtain a digraph without any directed cycles. A digraph without any cycles is called a *directed acyclic graph* (DAG). In a DAG, every strongly connected component consists of a single node. An undirected graph is *connected* if the corresponding bidirected graph is strongly connected. The connected components are the strongly connected components of the corresponding bidirected graph. For example, graph U in Fig. 2.7 has connected components $\{u,v,w\}$, $\{s,t\}$, and $\{x\}$. The node set $\{u,w\}$ induces a connected subgraph, but it is not maximal and hence not a component.

Exercise 2.17. Describe 10 substantially different applications that can be modeled using graphs; car and bicycle networks are not considered substantially different. At least five should be applications not mentioned in this book.

Exercise 2.18. A *planar graph* is a graph that can be drawn on a sheet of paper such that no two edges cross each other. Argue that street networks are *not* necessarily planar. Show that the graphs K_5 and K_{33} in Fig. 2.7 are not planar.

2.9.1 A First Graph Algorithm

It is time for an example algorithm. We shall describe an algorithm for testing whether a directed graph is acyclic. We use the simple observation that a node v with outdegree zero cannot appear in any cycle. Hence, by deleting v (and its incoming edges) from the graph, we obtain a new graph G' that is acyclic if and only if G is acyclic. By iterating this transformation, we either arrive at the empty graph, which is certainly acyclic, or obtain a graph G^* where every node has an outdegree of at least one. In the latter case, it is easy to find a cycle: start at any node v and construct a path by repeatedly choosing an arbitrary outgoing edge until you reach a node v' that you have seen before. The constructed path will have the form $(v,\ldots,v',\ldots,v')$, i.e., the part $(v',\ldots,v')$ forms a cycle. For example, in Fig. 2.7, graph G has no node with outdegree zero. To find a cycle, we might start at node z and follow the path $\langle z,w,x,u,v,w\rangle$ until we encounter w a second time. Hence, we have identified the cycle $\langle w,x,u,v,w\rangle$. In contrast, if the edge (w,x) is removed, there is no cycle. Indeed, our algorithm will remove all nodes in the order w, v, u, t, s, z, y, x. In Chap. 8, we shall see how to represent graphs such that this algorithm can be implemented to run in linear time. See also Exercise 8.3. We can easily make our algorithm certifying. If the algorithm finds a cycle, the graph is certainly cyclic. If the algorithm reduces the graph to the empty graph, we number the nodes in the order in which they are removed from G. Since we always remove a node v of outdegree zero from the current graph, any edge out of v in the original graph must go to a node that was removed previously and hence has received a smaller number. Thus the ordering proves acyclicity: along any edge, the node numbers decrease.

Exercise 2.19. Show an n-node DAG that has $n(n-1)/2$ edges.

2.9.2 Trees

An undirected graph is a *tree* if there is *exactly* one path between any pair of nodes; see Fig. 2.8 for an example. An undirected graph is a *forest* if there is *at most* one path between any pair of nodes. Note that each component of a forest is a tree.

Lemma 2.8. *The following properties of an undirected graph G are equivalent:*

1. *G is a tree.*
2. *G is connected and has exactly $n-1$ edges.*
3. *G is connected and contains no cycles.*

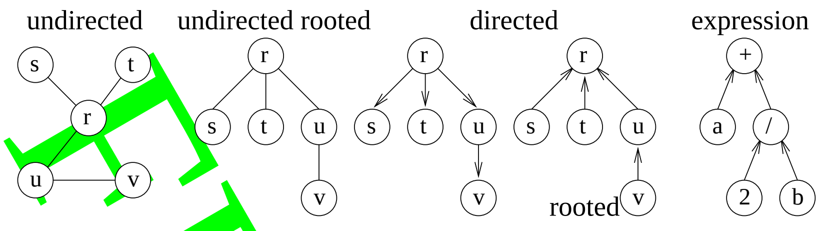

Fig. 2.8. Different kinds of trees. From *left* to *right*, we see an undirected tree, an undirected rooted tree, a directed out-tree, a directed in-tree, and an arithmetic expression

Proof. In a tree, there is a unique path between any two nodes. Hence the graph is connected and contains no cycles. Conversely, if there are two nodes that are connected by more than one path, the graph contains a cycle. Thus (1) and (3) are equivalent. We next show the equivalence of (2) and (3). Assume that $G = (V, E)$ is connected, and let $m = |E|$. We perform the following experiment: we start with the empty graph and add the edges in E one by one. Addition of an edge can reduce the number of connected components by at most one. We start with n components and must end up with one component. Thus $m \geq n - 1$. Assume now that there is an edge $e = \{u, v\}$ whose addition does not reduce the number of connected components. Then u and v are already connected by a path, and hence addition of e creates a cycle. If G is cycle-free, this case cannot occur, and hence $m = n - 1$. Thus (3) implies (2). Assume next that G is connected and has exactly $n - 1$ edges. Again, add the edges one by one and assume that adding $e = \{u, v\}$ creates a cycle. Then u and v are already connected, and hence e does not reduce the number of connected components. Thus (2) implies (3). □

Lemma 2.8 does not carry over to digraphs. For example, a DAG may have many more than $n - 1$ edges. A directed graph is an *out-tree* with a *root* node r, if there is exactly one path from r to any other node. It is an *in-tree* with a root node r if there is exactly one path from any other node to r. Figure 2.8 shows examples. The *depth* of a node in a rooted tree is the length of the path to the root. The *height* of a rooted tree is the maximum over the depths of its nodes.

We can also make an undirected tree rooted by declaring one of its nodes to be the root. Computer scientists have the peculiar habit of drawing rooted trees with the root at the top and all edges going downwards. For rooted trees, it is customary to denote relations between nodes by terms borrowed from family relations. Edges go between a unique *parent* and its *children*. Nodes with the same parent are *siblings*. Nodes without children are *leaves*. Nonroot, nonleaf nodes are *interior* nodes. Consider a path such that u is between the root and another node v. Then u is an *ancestor* of v, and v is a *descendant* of u. A node u and its descendants form a *subtree* rooted at u. For example, in Fig. 2.8, r is the root; s, t, and v are leaves; s, t, and u are siblings because they are children of the same parent r; u is an interior node; r and u are ancestors of v; s, t, u, and v are descendants of r; and v and u form a subtree rooted at u.

```
**Function** $eval(r) : \mathbb{R}$
    **if** $r$ *is a leaf* **then return** *the number stored in* $r$
    **else**                                                  // $r$ is an operator node
        $v_1 := eval(\textit{first child of } r)$
        $v_2 := eval(\textit{second child of } r)$
        **return** $v_1 \textit{operator}(r) v_2$              // apply the operator stored in $r$
```

Fig. 2.9. Recursive evaluation of an expression tree rooted at r.

2.9.3 Ordered Trees.

Trees are ideally suited to representing hierarchies. For example, consider the expression $a+2/b$. We have learned that this expression means that a and $2/b$ are added. But deriving this from the sequence of characters $\langle a,+,2,/,b\rangle$ is difficult. For example, it requires knowledge of the rule that division binds more tightly than addition. Therefore compilers isolate this syntactical knowledge in *parsers* that produce a more structured representation based on trees. Our example would be transformed into the expression tree given in Fig. 2.8. Such trees are directed and, in contrast to graph-theoretic trees, they are *ordered*. In our example, a is the first, or left, child of the root, and $/$ is the right, or second, child of the root.

Expression trees are easy to evaluate by a simple recursive algorithm. Figure 2.9 shows an algorithm for evaluating expression trees whose leaves are numbers and whose interior nodes are binary operators (say $+, -, \cdot, /$).

We shall see many more examples of ordered trees in this book. Chapters 6 and 7 use them to represent fundamental data structures, and Chapter 12 uses them to systematically explore solution spaces.

2.10 P and NP

When should we call an algorithm efficient? Are there problems for which there is no efficient algorithm? Of course, drawing the line between "efficient" and "inefficient" is a somewhat arbitrary business. The following distinction has proved useful: an algorithm $\mathscr{A}$ runs in *polynomial time*, or is a *polynomial-time algorithm*, if there is a polynomial $p(n)$ such that its execution time on inputs of size n is $O(p(n))$. If not otherwise mentioned, the size of the input will be measured in bits. A problem can be solved in *polynomial time* if there is a polynomial-time algorithm that solves it. We equate "efficiently solvable" with "polynomial-time solvable". A big advantage of this definition is that implementation details are usually not important. For example, it does not matter whether a clever data structure can accelerate an $O(n^3)$ algorithm by a factor of n. All chapters of this book, except for Chap. 12, are about efficient algorithms.

There are many problems for which no efficient algorithm is known. Here, we mention only six examples:

- The Hamiltonian cycle problem: given an undirected graph, decide whether it contains a Hamiltonian cycle.
- The Boolean satisfiability problem: given a Boolean expression in conjunctive form, decide whether it has a satisfying assignment. A Boolean expression in conjunctive form is a conjunction $C_1 \wedge C_2 \wedge \ldots \wedge C_k$ of clauses. A clause is a disjunction $\ell_1 \vee \ell_2 \vee \ldots \vee \ell_h$ of literals, and a literal is a variable or a negated variable. For example, $v_1 \vee \neg v_3 \vee \neg v_9$ is a clause.
- The clique problem: given an undirected graph and an integer k, decide whether the graph contains a complete subgraph (= a clique) on k nodes.
- The knapsack problem: given n pairs of integers (w_i, p_i) and integers M and P, decide whether there is a subset $I \subseteq [1..n]$ such that $\sum_{i \in I} w_i \leq M$ and $\sum_{i \in I} p_i \geq P$. «««< .mine
- The traveling salesman problem: given an edge-weighted undirected graph and an integer C, decide whether the graph contains a Hamiltonian cycle of length at most C. See Sect. 11.6.2 for more details.
- The graph coloring problem: given an undirected graph and an integer k, decide whether there is a coloring of the nodes with k colors such that any two adjacent nodes are colored differently.

The fact that we know no efficient algorithms for these problems does not imply that none exists. It is simply not known whether an efficient algorithm exists or not. In particular, we have no proof that such algorithms do not exist. In general, it is very hard to prove that a problem cannot be solved in a given time bound. We shall see some simple lower bounds in Sect. 5.3. Most algorithmicists believe that the six problems above have no efficient solution.

Complexity theory has found an interesting surrogate for the absence of lower-bound proofs. It clusters algorithmic problems into large groups that are equivalent with respect to some complexity measure. In particular, there is a large class of equivalent problems known as **NP**-*complete* problems. Here, **NP** is an abbreviation for "nondeterministic polynomial time". If the term "nondeterministic polynomial time" does not mean anything to you, ignore it and carry on. The six problems mentioned above are **NP**-complete, and so are many other natural problems. It is widely believed that **P** is a proper subset of **NP**. This would imply, in particular, that **NP**-complete problems have no efficient algorithm. In the remainder of this section, we shall give a formal definition of the class **NP**. We refer the reader to books about theory of computation and complexity theory [14, 72, 181, 205] for a thorough treatment.

We assume, as is customary in complexity theory, that inputs are encoded in some fixed finite alphabet Σ. A *decision problem* is a subset $L \subseteq \Sigma^*$. We use χ_L to denote the characteristic function of L, i.e., $\chi_L(x) = 1$ if $x \in L$ and $\chi_L(x) = 0$ if $x \notin L$. A decision problem is polynomial-time solvable iff its characteristic function is polynomial-time computable. We use **P** to denote the class of polynomial-time-solvable decision problems.

A decision problem L is in **NP** iff there is a predicate $Q(x,y)$ and a polynomial p such that

(1) for any $x \in \Sigma^*$, $x \in L$ iff there is a $y \in \Sigma^*$ with $|y| \leq p(|x|)$ and $Q(x,y)$, and
(2) Q is computable in polynomial time.

We call y a *witness* or *proof* of membership. For our example problems, it is easy to show that they belong to **NP**. In case of the Hamiltonian cycle problem, the witness is a Hamiltonian cycle in the input graph. A witness for a Boolean formula is an assignments of truth values to variables that make the formula true. The solvability of an instance of the knapsack problem is witnessed by a subset of elements that fit into the knapsack and achieve the profit bound P.

Exercise 2.9. Prove that the clique problem, the traveling salesman problem, and the graph coloring problem are in **NP**.

A decision problem L is *polynomial-time reducible* (or simply *reducible*) to a decision problem L' if there is a polynomial-time-computable function g such that for all $x \in \Sigma^*$, we have $x \in L$ iff $g(x) \in L'$. Clearly, if L is reducible to L' and $L' \in \mathbf{P}$, then $L \in \mathbf{P}$. Also, reducibility is transitive. A decision problem L is **NP**-*hard* if every problem in **NP** is polynomial-time reducible to it. A problem is **NP**-*complete* if it is **NP**-hard and in **NP**. At first glance, it might seem prohibitively difficult to prove any problem **NP**-complete – one would have to show that *every* problem in **NP** was polynomial-time reducible to it. However, in 1971, Cook and Levin independently managed to do this for the Boolean satisfiability problem [44, 120]. From that time on, it was "easy". Assume you want to show that a problem L is **NP**-complete. You need to show two things: (1) $L \in \mathbf{NP}$, and (2) there is *some* known **NP**-complete problem L' that can be reduced to it. Transitivity of the reducibility relation then implies that all problems in **NP** are reducible to L. With every new complete problem, it becomes easier to show that other problems are **NP**-complete. The website `http://www.nada.kth.se/~viggo/wwwcompendium/wwwcompendium.html` maintains a compendium of **NP**-complete problems. We give one example of a reduction.

Lemma 2.10. *The Boolean satisfiability problem is polynomial-time reducible to the clique problem.*

Proof. Let $F = C_1 \wedge \ldots \wedge C_k$, where $C_i = \ell_{i1} \vee \ldots \vee \ell_{ih_i}$ and $\ell_{ij} = x_{ij}^{\beta_{ij}}$, be a formula in conjunctive form. Here, x_{ij} is a variable and $\beta_{ij} \in \{0,1\}$. A superscript 0 indicates a negated variable. Consider the following graph G. Its nodes V represent the literals in our formula, i.e., $V = \{r_{ij} : 1 \leq i \leq k \text{ and } 1 \leq j \leq h_i\}$. Two nodes r_{ij} and $r_{i'j'}$ are connected by an edge iff $i \neq i'$ and either $x_{ij} \neq x_{i'j'}$ or $\beta_{ij} = \beta_{i'j'}$. In words, the representatives of two literals are connected by an edge if they belong to different clauses and an assignment can satisfy them simultaneously. We claim that F is satisfiable iff G has a clique of size k.

Assume first that there is a satisfying assignment α. The assignment must satisfy at least one literal in every clause, say literal ℓ_{ij_i} in clause C_i. Consider the subgraph of G spanned by the r_{ij_i}, $1 \leq i \leq k$. This is a clique of size k. Assume otherwise; say, r_{ij_i} and $r_{i'j_{i'}}$ are not connected by an edge. Then, $x_{ij_i} = x_{i'j_{i'}}$ and $\beta_{ij_i} \neq \beta_{i'j_{i'}}$. But then the literals ℓ_{ij_i} and $\ell_{i'j_{i'}}$ are complements of each other, and α cannot satisfy them both.

Conversely, assume that there is a clique K of size k in G. We can construct a satisfying assignment α. For each i, $1 \le i \le k$, K contains exactly one node r_{ij_i}. We construct a satisfying assignment α by setting $\alpha(x_{ij_i}) = \beta_{ij_i}$. Note that α is well defined because $x_{ij_i} = x_{i'j_{i'}}$ implies $\beta_{ij_i} = \beta_{i'j_{i'}}$; otherwise, r_{ij_i} and $r_{i'j_{i'}}$ would not be connected by an edge. α clearly satisfies F. □

Exercise 2.20. Show that the Hamiltonian cycle problem is polynomial-time reducible to the traveling salesman problem.

Exercise 2.21. Show that the clique problem is polynomial-time reducible to the graph-coloring problem.

All **NP**-complete problems have a common destiny. If anybody should find a polynomial time algorithm for *one* of them, then **NP** $=$ **P**. Since so many people have tried to find such solutions, it is becoming less and less likely that this will ever happen: The **NP**-complete problems are mutual witnesses of their hardness.

Does the theory of **NP**-completeness also apply to optimization problems? Optimization problems are easily turned into decision problems. Instead of asking for an optimal solution, we ask whether there is a solution with an objective value greater than or equal to k, where k is an additional input. Conversely, if we have an algorithm to decide whether there is a solution with a value greater than or equal to k, we can use a combination of exponential and binary search (see Sect. 2.5) to find the optimal objective value.

An algorithm for a decision problem returns yes or no, depending on whether the instance belongs to the problem or not. It does not return a witness. Frequently, witnesses can be constructed by applying the decision algorithm repeatedly. Assume we want to find a clique of size k, but have only an algorithm that decides whether a clique of size k exists. We select an arbitrary node v and ask whether $G' = G \setminus v$ has a clique of size k. If so, we recursively search for a clique in G'. If not, we know that v must be part of the clique. Let V' be the set of neighbors of v. We recursively search for a clique C_{k-1} of size $k-1$ in the subgraph spanned by V'. Then $v \cup C_{k-1}$ is a clique of size k in G.

2.11 Implementation Notes

Our pseudocode is easily converted into actual programs in any imperative programming language. We shall give more detailed comments for C++ and Java below. The Eiffel programming language [138] has extensive support for assertions, invariants, preconditions, and postconditions.

Our special values $\perp$, $-\infty$, and ∞ are available for floating-point numbers. For other data types, we have to emulate these values. For example, one could use the smallest and largest representable integers for $-\infty$ and ∞, respectively. Undefined pointers are often represented by a null pointer **null**. Sometimes we use special values for convenience only, and a robust implementation should avoid using them. You will find examples in later chapters.

Randomized algorithms need access to a random source. You have a choice between a hardware generator that generates true random numbers and an algorithmic generator that generates pseudo-random numbers. We refer the reader to the Wikipedia page on "random numbers" for more information.

2.11.1 C++

Our pseudocode can be viewed as a concise notation for a subset of C++. The memory management operations **allocate** and **dispose** are similar to the C++ operations *new* and *delete*. C++ calls the default constructor for each element of an array, i.e., allocating an array of n objects takes time $\Omega(n)$ whereas allocating an array n of *int*s takes constant time. In contrast, we assume that *all* arrays which are not explicitly initialized contain garbage. In C++, you can obtain this effect using the C functions *malloc* and *free*. However, this is a deprecated practice and should only be used when array initialization would be a severe performance bottleneck. If memory management of many small objects is performance-critical, you can customize it using the *allocator* class of the C++ standard library.

Our parameterizations of classes using **of** is a special case of the C++-template mechanism. The parameters added in brackets after a class name correspond to the parameters of a C++ constructor.

Assertions are implemented as C macros in the include file `assert.h`. By default, violated assertions trigger a runtime error and print their position in the program text. If the macro *NDEBUG* is defined, assertion checking is disabled.

For many of the data structures and algorithms discussed in this book, excellent implementations are available in software libraries. Good sources are the standard template library STL [157], the Boost [27] C++ libraries, and the LEDA [131, 118] library of efficient algorithms and data structures.

2.11.2 Java

Java has no explicit memory management. Rather, a *garbage collector* periodically recycles pieces of memory that are no longer referenced. While this simplifies programming enormously, it can be a performance problem. Remedies are beyond the scope of this book. Generic types provide parameterization of classes. Assertions are implemented with the *assert* statement.

Excellent implementations for many data structures and algorithms are available in the package *java.util* and in the JDSL [78] data structure library.

2.12 Historical Notes and Further Findings

Sheperdson and Sturgis [179] defined the RAM model for use in algorithmic analysis. The RAM model restricts cells to holding a logarithmic number of bits. Dropping this assumption has undesirable consequences; for example, the complexity classes **P** and **PSPACE** collapse [87]. Knuth [113] has described a more detailed abstract machine model.

Floyd [62] introduced the method of invariants to assign meaning to programs and Hoare [91, 92] systemized their use. The book [81] is a compendium on sums and recurrences and, more generally, discrete mathematics.

Books on compiler construction (e.g., [144, 207]) tell you more about the compilation of high-level programming languages into machine code.