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 an 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. You can also skip the parts on parallel processing when you are only considering sequential algorithms. On the 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 machine models in Sect. 2.2 that allow us to abstract from the highly variable complications introduced by real hardware. The models are concrete enough to have predictive value and abstract enough to permit 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. 2.6. Section 2.7 gives the first comprehensive example: binary search in a sorted array. In Sect. 2.8, we introduce mathematical techniques

[1] The above illustration of Stonehenge is from [254].

for analyzing the complexity of programs, in particular, for analyzing nested loops and recursive procedure calls. Additional analysis techniques are needed for average-case analysis and parallel algorithm analysis; these are covered in Sects. 2.9 and 2.10, respectively. Randomized algorithms, discussed in Sect. 2.11, use coin tosses in their execution. Section 2.12 is devoted to graphs, a concept that will play an important role throughout the book. In Sect. 2.13, we discuss the question of when an algorithm should be called efficient, and introduce the complexity classes **P** and **NP** and the concept of **NP**-completeness. Finally, as in most chapters of this book, we close with implementation notes (Sect. 2.14) and historical notes and further findings (Sect. 2.15).

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.

Hence, we group the set of all inputs into classes of "similar" inputs and summarize the performance on all instances in the same class in 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 that 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 set of 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]

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

We are most interested in the worst-case execution time, since it gives us the strongest performance guarantee. A comparison of the best and the worst case tells us how much the execution time varies for different inputs in the same class. If the

[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.

discrepancy is big, the average case may give more insight into the true performance of the algorithm. Section 2.9 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 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. 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. A remark about notation is in order here. In the definitions above, we use "$f(n)$" and "$g(n)$" with two different meanings. In "$\mathrm{O}(f(n))$" and "$\{g(n) : \ldots\}$", they denote the functions f and g and the "n" emphasizes that these are functions of the argument n, and in "$g(n) \leq c \cdot f(n)$", they denote the values of the functions at the argument n.

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 0 as n goes to infinity. Here "1" stands for the function $n \mapsto 1$, which is 1 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* $p(n) \in \Theta(n^k)$.

Proof. It suffices to show that $p(n) \in \mathrm{O}(n^k)$ and $p(n) \in \Omega(n^k)$. First observe that for $n \geq 1$,

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

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

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

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

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

Exercise 2.1. Right or wrong? (a) $n^2 + 10^6 n \in \mathrm{O}(n^2)$, (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 $\mathrm{O}(n^2)$) 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,

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

Never forget that sequences of "equalities" involving O-notation are really membership and inclusion relations and, as such, can 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 1 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:*

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

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 Sequential Machine Model

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

In 1945, John von Neumann (Fig. 2.1) introduced a computer architecture [242] which was simple, yet powerful. The limited hardware technology of the time forced him to come up with a 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.

The variant of von Neumann's model used in algorithmic analysis is called the *RAM* (random access machine) model. It was introduced by Shepherdson and Sturgis [293] in 1963. 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 (main) memory, or *store*, consists of infinitely many cells $S[0]$, $S[1]$, $S[2]$, ...; at any point in time, only a finite number of them will be in use. In addition to the main memory, there are a small number of *registers* $R_1, \ldots, R_k$.

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 whose absolute value is bounded by a polynomial in the size of the input 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.

A RAM can execute (machine) programs. A program is simply a sequence of machine instructions, numbered from 0 to some number ℓ. The elements of the sequence are called *program lines*. The program is stored in a program store. Our RAM supports 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* the contents of register R_i in the memory cell indexed by the contents of R_j.
- $R_i := R_j \odot R_h$ executes the binary operation $\odot$ on the contents of registers R_j and R_h and stores the result in register R_i. Here, $\odot$ is a placeholder for a variety of operations. The *arithmetic* operations are the usual $+$, $-$, and $*$; they interpret the contents of the registers as integers. The operations **div** and **mod** stand for integer division and the remainder, respectively. The *comparison* operations $\leq$, $<$, $>$, and $\geq$ for integers return *truth values*, i.e., either *true* (= 1) or *false* (= 0). The *logical* operations $\wedge$ and $\vee$ manipulate the truth values 0 and 1. We also have bitwise Boolean operations `|` (OR), `&` (AND), and $\oplus$ (exclusive OR, XOR). They interpret contents as bit strings. The shift operators >> (shift right) and << (shift left) interpret the first argument as a bit string and the second argument as a nonnegative integer. 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.
- $R_i := \odot R_j$ executes the *unary* operation $\odot$ on the contents of register R_j and stores the result in register R_i. The operators $-$, $\neg$ (logical NOT), and ~ (bitwise NOT) are available.
- $R_i := C$ assigns the *constant* value C to R_i.
- `JZ` k, R_i continues execution at program line k, if register R_i is 0, and at the next program line otherwise (*conditional branch*). There is also the variant `JZ` R_j, R_i, where the target of the jump is the program line stored in R_j.
- `J` k continues execution at program line k (*unconditional branch*). Similarly to `JZ`, the program line can also be specified by the content of a register.

A program is executed on a given input step by step. The input for a computation is stored in memory cells $S[1]$ to $S[R_1]$ and execution starts with program line 1. With the exception of the branch instructions `JZ` and `J`, the next instruction to be executed is always the instruction in the next program line. The execution of a program terminates if a program line is to be executed whose number is outside the range $1..\ell$. Recall that ℓ is the number of the last program line.

We define the execution time of a program on an input in the most simple way: *Each instruction takes one time step to execute. The total execution time of a program is the number of instructions executed.*

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. This statement was also true when 32-bit words were introduced.

Our complexity model is 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. App. B discusses hardware architecture in more detail.

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 shortcomings of the model. Two important shortcomings of the RAM model, namely the lack of a memory hierarchy and the limited parallelism ,are discussed in the next two subsections.

2.2.1 External Memory

The organization of the memory is a major difference between an RAM and a real machine: a uniform flat memory in a RAM and a complex memory hierarchy in a real machine. In Sects. 5.12, 6.3, 7.7, and 11.5 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 is limited 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. The reason for transferring a block of B words instead of a single word is that the memory access time is large for a slow memory in comparison with the transfer time for a single word. The value of B is chosen such that the transfer time for B words is approximately equal to the access time. 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, $M = 8\,\text{GB}$ and $B = 2\,\text{MB}$ 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, one can model the smaller access time difference between a hardware cache and main memory.

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#, Rust, Swift, Python, 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 for understanding the timing model. There is no need to worry about compiler optimization techniques, since constant factors are ignored in asymptotic analysis anyway. 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 [165], 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 to the value x. For example, "$\mathit{answer} = 42 : \mathbb{N}$" introduces a variable *answer* assuming nonnegative 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 to 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 an array $a[0..k]$, the elements have unit size, and R_i contains the integer 42, the instruction sequence "$R_1 := R_a + R_i$; $R_2 := S[R_1]$" 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. 3.4, 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 a handle to c, i.e., the address of 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 to 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 maximum 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 contain only 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 of representing 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 will 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

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

where $eval(C)$ is a sequence of instructions that evaluate the expression C and leave its value in register R_c, $trans(I)$ is a sequence of instructions that implement statement I, $trans(J)$ implements J, *sElse* is the address of the first instruction in $trans(J)$, and *sEnd* is the address of the first instruction after $trans(J)$. 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 $trans(I)$; $eval(C)$; `JZ` sI, R_c, where sI is the address of the first instruction in $trans(I)$. We shall also use many other types of loops that can be viewed as shorthands for various 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** $i \leq b$ **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 to 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

```
Function factorial(n) : ℤ
    if n = 1 then return 1 else return n · factorial(n − 1)
```

```
factorial:                                              // the first instruction of factorial
R_n := S[R_r − 1]        // load n into register R_n. Abbreviation of R_tmp := R_r − 1; R_n := S[R_tmp]
JZ thenCase, R_n                                        // jump to then case, if n is 0
S[R_r] = aRecCall                                       // else case; return address for recursive call
S[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 := S[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 S[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 the register R_{result}. To keep the presentation short, we take the liberty of directly using subexpressions, where, strictly speaking, sequences of assignments using temporary registers would be needed.

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. Register R_r always points to the first free entry in this stack. The stack contains a sequence of *activation records*, one for each active procedure call. 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* 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 ith for-

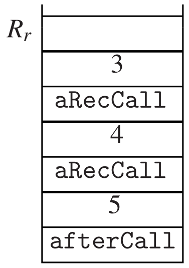

Fig. 2.3. The recursion stack of a call *factorial*(5) when the recursion has reached *factorial*(3).

mal parameter ($0 \leq i < k$) is an access to $S[R_r - \ell - k + i]$ and any access to the ith local variable ($0 \leq i < \ell$) is an access to $S[R_r - \ell + i]$. 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 $S[R_r]$). 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. There is no need to translate the output command, in which the value in the box is output as a number. 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                 // nonprimes $\leq n$ have a factor $\leq \lfloor\sqrt{n}\rfloor$
    if $a[i]$ then                                                  // $i$ is prime
        for $j := 2i$ to $n$ step $i$ do $a[j] := 0$     // all multiples of $i$ are nonprime
    for $i := 2$ to $n$ do if $a[i]$ then output("$\boxed{i}$ 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 data structures. We introduce our notation by way of example. The definition

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

gives a (partial) implementation of a complex number type that can use arbitrary numeric types such as $\mathbb{Z}$, $\mathbb{Q}$, and $\mathbb{R}$ for the real and imaginary parts. Our class names (here "*Complex*") will usually begin with capital letters. The real and imaginary parts are stored in the *member variables re* and *im*, respectively. Now, the declaration "$c : Complex(2,3)$ **of** $\mathbb{R}$" declares a complex number c initialized to $2 + 3i$, where i is the imaginary unit. The expression $c.im$ evaluates to the imaginary part of c, and $c.abs$ returns the absolute value of c, a real number.

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 Parallel Machine Models

We classify parallel machine models into two broad classes: shared-memory machines and distributed-memory machines. In both cases, we have *p processing elements* (PEs). In the former case, these PEs share a common memory and all communication between PEs is through the shared memory. In the latter case, each PE has its own private memory, the PEs are connected by a communication network, and all communication is through the network. We introduce shared-memory machines in Sect. 2.4.1 and discuss distribute- memory machines in Sect. 2.4.2.

2.4.1 Shared-Memory Parallel Computing

In a shared-memory machine, the PEs share a common memory (Fig. 2.4). Each PE knows its number i_{proc} (usually from $1..p$ or $0..p-1$). The theoretical variant of this model is known as the *PRAM (parallel random access machine)*. PRAMs come in several flavors. The main distinction is whether concurrent access to the same memory cell is allowed. This leads to the submodels *EREW-PRAM*, *CREW-PRAM*, and *CRCW-PRAM* where "C"stands for "concurrent" (concurrent access allowed), "E" stands for "exclusive" (concurrent access forbidden), "R" stands for "read" and "W" stands for "write". Thus a CREW-PRAM supports concurrent reads but forbids concurrent writes. Real-world shared-memory machines support something resembling concurrent read, so that we do not need to bother with the complications introduced by exclusive reads. We therefore concentrate on the CREW and CRCW. Concurrent writing makes the model more powerful, but we have be careful with the semantics of concurrent writing. To illustrate the pitfalls of concurrent memory access, let us consider a simple example: Two PEs a and b share the same counter variable c, say because they want to count how often a certain event happens. Suppose the current value of c is 41 and both a and b want to increment c at the same time. Incrementing means first loading the old value into a register, and then incrementing the register and storing it back in memory. Suppose both PEs read the value 41, increment this value to 42 and then write it back – all at the same time. Afterwards, $c = 42$, although the programmer probably intended $c = 43$. Different semantics of concurrent writing lead to several subflavors of CRCW-PRAMs. The two most widely used ones

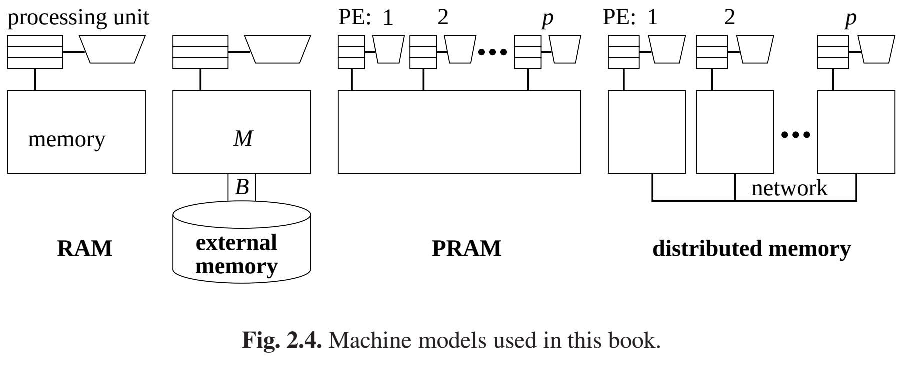

Fig. 2.4. Machine models used in this book.

are “common” and “arbitrary”. The *common CRCW-PRAM* only allows concurrent write access if all PEs writing to the same memory cell are writing the same value. There is no such restriction in *arbitrary CRCW-PRAMs*. If several different values are written to the same cell in the same time step, one of these values is chosen arbitrarily. We shall try to avoid algorithms requiring the CRCW model, since concurrent write access to the same memory cells causes problems in practice. In particular, it becomes difficult to ensure correctness, and performance may suffer.

PRAM models assume globally synchronized time steps – every PE executes exactly one machine instruction in every time step. This makes it relatively easy to write and understand PRAM programs. Unfortunately, this assumption is untenable in practice – the execution of instructions (in particular memory access instructions) happens in several stages and it depends on the current state of the computation how long it takes to finish the instruction. Moreover, even instructions accessing memory at the same time may see different values in the same memory cell owing to the behavior of caches. More realistic models of shared memory therefore introduce additional mechanisms for explicitly controlling concurrent memory access and for synchronizing PEs.

A very general such mechanism is *transactions* (or *critical sections*). A transaction t consists of a piece of code that is executed *atomically*. Atomic means indivisible – during the execution of t, no other PE writes to the memory cells accessed by t. For example, if the PEs a and b in the example above were to execute transactions

```
**begin transaction** $c := c + 1$ **end transaction**
```

then the hardware or runtime system would make sure that the two transactions were executed one after the other, resulting in the correct value $c = 43$ after executing both transactions. Some processor architectures support transactions in hardware, while others only support certain atomically executed instructions that can be used to implement higher-level concepts. In principle, these can be used to support general transactions in software. However, this is often considered too expensive, so one often works with less general concepts. See Sect. B.5 for some details of actual hardware implementations of transactional memory.

Perhaps the most important *atomic instruction* for concurrent memory access is *compare-and-swap* (CAS).

```
Function $CAS(i, expected, desired) : \{1, 0\}$
    begin transaction
        if $S[i] = expected$ then    $S[i] := desired$; return 1        // success
        else                         $expected := S[i]$; return 0       // failure
    end transaction
```

A call of CAS specifies a value expected to be present in memory cell $S[i]$ and the value that it wants to write. If the expectation is true, the operation writes the desired value into the memory cell and succeeds (returns 1). If not, usually because some other PE has modified $S[i]$ in the mean time, the operation writes the actual value of $S[i]$ into the variable *expected* and fails (returns 0). CAS can be used to implement transactions acting on a single memory cell. For example, atomically adding a value to a memory cell can be done as follows:

```
**Function** $fetchAndAdd(i, \Delta)$
    $expected := S[i]$
    **repeat** $desired := expected + \Delta$ **until** $CAS(i, expected, desired)$
    **return** $desired$
```

The function reads the value of $S[i]$ and stores the old and the incremented value in *expected* and *desired*, respectively. It then calls $\mathrm{CAS}(i, expected, desired)$. If the value of $S[i]$ has not changed since it was stored in *expected*, the call succeeds, and the incremented value is stored in $S[i]$ and returned. Otherwise, the current value of $S[i]$ is stored in *expected* and the call fails. Then another attempt to increment the variable is made.

Regardless of whether hardware transactions or atomic instructions are used, when many PEs try to write to the same memory cell at once, some kind of serialization will take place and performance will suffer. This effect is called *write contention*. Asymptotically speaking, it will take time $\Omega(p)$ if all PEs are involved. Note that this is far from the behavior of a CRCW-PRAM, where concurrent writing is assumed to work in constant time.

Let us look at a simple example where this makes a difference: Assume each PE has a Boolean value and we want to compute the logical OR of all these values. On a common CRCW-PRAM, we simply initialize a global variable g to false, and each PE with a local value of true writes true to g. Within the theoretical model, this works in constant time. However, when we try to do this on a real-world machine using transactions, we may observe time $\Omega(p)$ when all local values are true. We shall later see more complicated algorithms achieving time $\mathrm{O}(\log p)$ on CREW machines.

In order to get closer to the real world, additional models of PRAMs have been proposed that assume that the cost of memory access is proportional to the number of PEs concurrently accessing the same memory cell. For example, QRQW (queue-read-queue-write) means that contention has to be taken into account for both reading and writing [126]. Since modern machines support concurrent reading by placing copies of the accessed data in the machine caches, it also makes sense to consider CRQW (concurrent-read-queue-write) models. In this book, we shall sometimes use the aCRQW-PRAM model, where the “a” stands for “asynchronous”, i.e., there is no step-by-step synchronization between the PEs.[3] For the cost model, this means that all instructions take constant time, except for write operations, whose execution time is proportional to the number of PEs trying to access that memory cell concurrently.

[3] There is previous work on asynchronous PRAMs [125] that is somewhat different, however, in that it subdivides computations into synchronized phases. Our aCRQW model performs only local synchronization. Global synchronization requires a separate subroutine that can be implemented using $\mathrm{O}(\log p)$ local synchronizations (see Sect. 13.4.2).

2.4.2 Distributed-Memory Parallel Computing

Another simple way to extend the RAM model is to connect several RAMs with a communication network (Fig. 2.4). The network is used to exchange messages. We assume that messages have exactly one sender and one receiver (*point-to-point communication*) and that exchanging a message of length ℓ takes time $\alpha + \ell\beta$ regardless which PEs are communicating. In particular, several messages can be exchanged at once except that no PE may send several messages at the same time or receive several messages at the same time. However, a PE is allowed to send one message and to receive another message at the same time. This mode of communication is called (full-duplex) single-ported communication. The function call $send(i,m)$ sends a message m to PE i and $receive(i,m)$ receives a message from PE i. When the parameter i is dropped, a message from any PE can be received and the number of the actual sender is the return value of the *receive* function. For every call $send(i,m)$, PE i must eventually execute a matching receive operation. The *send* operation will only complete when the matching *receive* has completed, i.e., PE i now has a copy of message m.[4] Thus sends and receives *synchronize* the sender and the receiver. The integration of data exchange and synchronization is an important difference with respect to shared-memory parallel programming. We will see that the message-passing style of parallel programming often leads to more transparent programs despite the fact that data exchange is so simple in shared-memory programming. Moreover, message-passing programs are often easier to debug than shared-memory programs.

Let us consider a concrete example. Suppose each PE has stored a number x in a local variable and we want to compute the sum of all these values so that afterwards PE 0 knows this sum. The basic idea is to build a binary tree on top of the PEs and to add up the values by layer. Assume that PEs are numbered from 0 to $p-1$ and that p is a power of two. In the first round, each odd numbered PE sends his value to the PE numbered one smaller and then stops. The even numbered PEs sum the number received to the number they hold. In the second round, the even numbered PEs whose number is not divisible by four send to the PE whose number is smaller by two. Continuing in this way, the total sum is formed in time $O(\log p)$. The following lines of pseudocode formalize this idea. They work for arbitrary p.

```
Function reduceAdd(x)                          // let $i$ denote the local PE number
    for ($d := 1$;  $d < p$;  $d *= 2$)
        if ($i$ bitand $d$) $= 0$ then
            if $i + d < p$ then receive($i + d, x'$);   $x += x'$
        else send($i - d, x$);   return
    return $x$                                 // only reached by PE 0
```

Initially all PEs are active. The layer counter d is a power of two and is also interpreted as a bit string. It starts at $d = 1 = 2^0$. Let us first assume that p is a power of two. PEs with an odd PE number (i **bitand** $d \neq 0$) exit from the for-loop and send

[4] Most algorithms in this book use synchronous communication. However, *asynchronous* send operations can also have advantages since they allow us do decouple communication and cooperation. An example can be found in Sect. 6.4 on parallel priority queues.

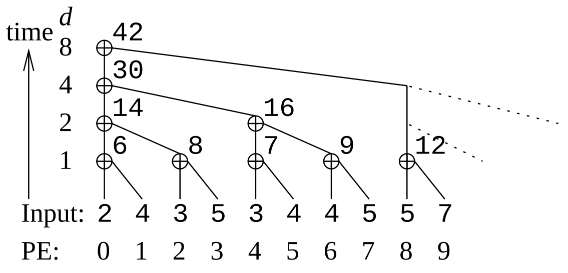

Fig. 2.5. Parallel summation using a tree

their value to PE $i-1$. PEs with an even PE number (i **bitand** $d = 0$) issue a receive request *receive*$(i+d, x')$ (note that $i+d < p$ for all even PEs) and add the value received to their own value. At this point, the odd-numbered PEs have terminated and the even-numbered PEs have increased the value of d to 2. In the second round, exactly the same reduction happens on the next to last bit, i.e., PEs whose number ends with 00 issue a *receive* and PEs whose number ends with 10 issue a *send* and terminate. Generally, in a round with $d = 2^k$, $k \geq 0$, the PEs whose PE numbers end with 10^k send their value to PE $i-d$ and terminate. The PEs whose PE numbers end with 0^{k+1} receive and add the received value to their current value. When $d = p$, the for-loop terminates and only PE 0 remains active. It returns x, which now contains the desired global sum.

The program is also correct when p is not a power of two since we have made sure that no PE tries to receive from a nonexistent PE and all sending PEs have a matching receiving PE. Figure 2.5 shows an example computation. In this example, processor 8 issues a *receive*$(9, x')$ in round 1, sits idle in rounds 2 and 3, and sends its value to PE 0 in round 4.

On a (synchronous) PRAM, we can do something very similar – it is even slightly easier. Assume the inputs are stored in an array $x[0..p-1]$. We replace "*receive*$(i+d, x')$; $x \mathrel{+}= x'$" by "$x[i] \mathrel{+}= x[i+d]$", drop lines 4 and 5, and obtain the final result in $x[0]$. However, if we use this code on an asynchronous shared-memory machine (e.g., in our aCRQW PRAM model), it is incorrect. We have to add additional synchronization code to make sure that an addition is only performed when its input values are available. The resulting code will be at least as complex as the message-passing code. Of course, we could use the function *fetchAndAdd* mentioned in from the preceding section to realize correct concurrent access to a global counter. However, this will take time $O(p)$ owing to write contention, and, indeed, it is likely that on a large machine it would be faster to perform one global synchronization followed by adding up the values in $x[0..p-1]$ sequentially.

2.4.3 Parallel Memory Hierarchies

The models presented in Sects. 2.2.1–2.4.2 each address an important aspect of real-world machines not present in the RAM model. However, they are still a gross simplification of reality. Modern machines have a memory hierarchy with multiple levels and use many forms of parallel processing (see Fig. 2.6). Appendix B describes a concrete machine that we used for the experiments reported in this book. We next briefly discuss some important features found in real-world machines.

Many processors have 128–512-bit *SIMD* registers that allow the parallel execution of a *s*ingle *i*nstruction on *m*ultiple *d*ata objects (*SIMD*).

They are *superscalar*, i.e., they can execute multiple independent instructions from a sequential instruction stream in parallel.

Simultaneous multithreading allows processors to better utilize parallel execution units by running multiple threads of activity on a single processor core sharing the same first-level (L1) cache.

Even mobile devices nowadays have *multicore* processors, i.e., multiple processor cores, that can independently execute programs. There are further levels of on-chip cache. The further up in this hierarchy, the more PEs share the same cache. For example, each PE may have its own L1 and L2 caches but eight cores on one chip might share a rather large L3 cache. Most servers have several multicore processors accessing the same shared memory. Accessing the memory chips directly connected to the processor chip is usually faster than accessing memory connected to other processors. This effect is called *nonuniform memory access* (NUMA).

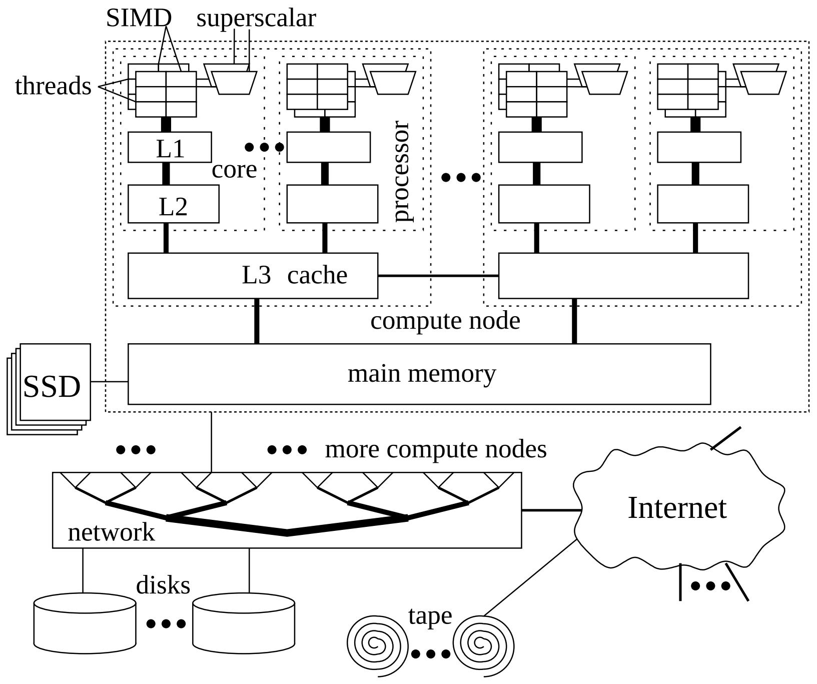

Fig. 2.6. Example of a parallel memory hierarchy

Coprocessors, in particular *graphics processing units* (GPUs), have even more parallelism on a single chip. GPUs have their own peculiarities complicating the model. In particular, large groups of threads running on the same piece of hardware can only execute in SIMD mode, i.e., in the same time step, they all have to execute the same instruction.

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.

Storage devices such as solid state disks (SSDs), hard disks, or tapes may be connected to different levels of this hierarchy of processors. For example, a high-performance computer might have solid state disks connected to each multiprocessor board while its main communication network has ports connected to a large array of hard disks. Over the internet, it might do data archiving to a remote tape library.

Attempts to grasp this complex situation using a single overarching model can easily fall into a complexity trap. It is not very difficult to define more general and accurate models. However, the more accurate the model gets, the more complicated it gets to design and analyze algorithms for it. A specialist may succeed in doing all this (at least for selected simple problems), but it would still then be very difficult to understand and apply the results. In this book, we avoid this trap by flexibly and informally combining simple models such as those shown in Fig. 2.4 in a case-by-case fashion. For example, suppose we have many shared-memory machines connected by a fast network (as is typical for a high-performance computer). Then we could first design a distributed-memory algorithm and then replace its local computations by shared-memory parallel algorithms. The sequential computations of a thread could then be made cache-efficient by viewing them as a computation in the external-memory model. These isolated considerations can be made to work on the overall system by making appropriate modifications to the model parameters. For example, the external-memory algorithm may use only $1/16$ of the L3 cache if there are 16 threads running in parallel on each processor sharing the same L3 cache.

2.5 Parallel Pseudocode

We now introduce two different ways of writing parallel pseudocode.

2.5.1 Single-Program Multiple-Data (SPMD)

Most of our parallel algorithms will look very similar to sequential algorithms. We design a single program that is executed on all PEs. This does not mean that all PEs execute the same sequence of instructions. The program may refer to the local ID i_{proc} and the total number p of PEs. In this way, each incarnation of the program may behave differently. In particular, the different PEs may work on different parts of the data. In the simplest case, each PE performs the same sequence of instructions on a disjoint part of the data and produces a part of the overall result. No synchronization is needed between the PEs. Such an algorithm is called *embarrasingly parallel*. Making large parts of a computation embarassingly parallel is a major design goal. Of course, some kind of interaction between the PEs is needed in most algorithms. The PEs may interact using primitives from the shared memory model, such as CAS, or from the distributed-memory model such as *send/receive*. Often, we are even able to abstract from the concrete model by using constructs that can be implemented in both models.

In our SPMD programs each PE has its private, local version of every variable. We write $v@j$ to express a remote access to the copy of v at PE j. We shall even use this notation for distributed memory algorithms when it is clear how to translate it into a pair of procedure calls $\mathit{send}(i,v)$ on PE j and $\mathit{receive}(j,\ldots)$ on PE i.

2.5.2 Explicitly Parallel Programs

Sometimes it is more elegant to augment a sequential program with explicit parallel constructs. In particular, writing **do**$\|$ in a for-loop indicates that the computations for each loop index can be done independently and in parallel. The runtime system or the programming language can then distribute the loop iterations over the PEs. If all these operations take the same time, each PE will be responsible for n/p loop iterations. Otherwise, one of the load-balancing algorithms presented in Chap. 14 can be used to distribute the loop iterations. This declarative style of parallelization also extends to initializing arrays.

Threads and Tasks. So far, we have implicitly assumed that there is a one-to-one correspondence between computational activities (*threads*) and pieces of hardware (processor cores or hardware-supported threads). Sometimes, however, a more flexible approach is appropriate. For example, allowing more threads than PEs can increase efficiency when some of the threads are waiting for resources. Moreover, when threads represent pieces of work (*tasks*) of different size, we can leave it to the runtime system of our programming language to do the load balancing.

In our pseudocode, we separate statements by the separator " $\|$ " to indicate that all these statements can be executed in parallel; it is up to the runtime system how it exploits this parallelism. We refer to Sect. 14.5 for load balancing algorithms that make this decision. A parallel statement finishes when all its constituent subtasks have finished. For example, for a parallel implementation of the recursive multiplication algorithm presented in Sect. 1.4, we could write

$$a0b0 := a_0 \cdot b_0 \;\|\; a0b1 := a_0 \cdot b_1 \;\|\; a1b0 := a_1 \cdot b_0 \;\|\; a1b1 := a_1 \cdot b_1$$

to compute all the partial products in parallel. Note that in a recursive multiplication algorithm for n-digit numbers, this would result in up to n^2 tasks. Our most frequent use of the $\|$ operator is of the form $\mathit{send}(i,a) \,\|\, \mathit{receive}(j,b)$ to indicate concurrent send and receive operations exploiting the full-duplex capability of our distributed-memory model (see Sect. 2.4.2).

Locks. These are an easy way to manage concurrent data access. A lock protects a set S of memory cells from concurrent access. Typically S is the state of a data structure or a piece of it. We discuss the most simple *binary lock* first. A lock can be held by at most one thread, and only the thread holding the lock can access the memory cells in S. For each lock, there is a corresponding global *lock variable* $\ell_S \in \{0,1\}$; it has the value 1 if some thread holds the lock and has the value 0 otherwise. This variable can be modified only by the procedures *lock* and *unlock*. A thread u can set (or *acquire*) lock ℓ_S by calling $lock(\ell_S)$. This call will wait for any other thread currently holding the lock and returns when it has managed to acquire the lock for thread u. A thread holding a lock may *release* it by calling $unlock(\ell_S)$, which resets ℓ_S to 0. We refer to Sect. 13.4.1 for possible implementations. There are also more general locks distinguishing between read and write access – multiple readers are allowed but a writer needs exclusive access. Section C.2 gives more information.

The simplicity of locks is deceptive. It is easy to create situations of cyclic waiting. Consider a situation with two threads. Thread 1 first locks variable A and then variable B. Thread 2 first locks variable B and then variable A. Suppose both threads reach their first lock operation at the same time. Then thread 1 acquires the lock on A and thread 2 acquires the lock on B. Then both of them proceed to their second lock operation and both will start to wait. Thread 1 waits for thread 2 to release the lock on B, and thread 2 waits for thread 1 to release the lock on A. Hence, they will both wait forever. This situation of cyclic waiting is called a *deadlock*.

Indeed, an interesting area of research are algorithms and data structures that do *not* need locks at all (non-blocking, lock-free, wait-free) and thus avoid many problems connected to locks. The hash table presented in Sects. 4.6.2–4.6.3 is lock-free as long as it does not need to grow.

2.6 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 in the design of correct algorithms and programs.

2.6.1 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 2.7 shows an example of the use of assertions and invariants

```
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
    $p = a : \mathbb{R}$;   $r = 1 : \mathbb{R}$;   $n = n_0 : \mathbb{N}$                        // 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. 2.7. An algorithm that computes integer powers of real numbers

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 a and n_0 are not allowed to be both 0.[5] 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. A property that holds whenever control passes a certain location in the program is called an *invariant*. Loop invariants and data structure invariants are of particular importance.

2.6.2 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

[5] The usual convention is $0^0 = 1$. The program is then also correct for $a = 0$ and $n_0 = 0$.

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. 2.7 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.

2.6.3 Data Structure Invariants

More complex programs encapsulate their state in objects and offer the user an abstract view of the state. The connection between the abstraction and the concrete representation is made by an invariant. 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. Here, set is the abstraction and sorted array is the concrete representation. The data structure invariant will state that the data structure uses an array a and an integer n, that n is the size of the set stored, 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.

2.6.4 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 the algorithm 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. 2.12. 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 (as a sequence e_1 to e_k of edges). 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 check that the returned sequence of edges is indeed an odd-length cycle in G. 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.

2.7 An Example – Binary Search

Binary search is a very useful technique for searching in an ordered set of elements. 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. We want to find the index k with $a[k-1] < x \le a[k]$; here, $a[0]$ and $a[n+1]$ should be interpreted as virtual elements with values $-\infty$ and $+\infty$, respectively. We can use these virtual 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 $k = m$. 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 subarray. We have two indices ℓ and r and have restricted the search for x to the subarray $a[\ell+1]$ to $a[r-1]$. More precisely, we maintain the invariant

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

This is true initially, with $\ell = 0$ and $r = n+1$. Once ℓ and r become consecutive indices, we may conclude that x is not contained in the array. Figure 2.8 shows the complete program.

We now prove correctness of the program. We shall first show that the loop invariant holds whenever the loop condition "$\ell+1 < r$" is checked. We do so by induction on the number of iterations. We have already established that the invariant holds initially, i.e., if $\ell = 0$ and $r = n+1$. This is the basis of the induction. For the induction step, we have to show that if the invariant holds before the loop condition is checked and the loop condition evaluates to true, then the invariant holds at the end of the loop body. So, assume that the invariant holds before the loop condition is checked and that $\ell+1 < r$. Then we enter the loop, and $\ell+2 \le r$ since ℓ and r are integral. We compute m as $\lfloor (r+\ell)/2 \rfloor$. Since $\ell+2 \le r$, we have $\ell < m < r$. 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 $\ell < r$ and $a[\ell] < x < a[r]$. Thus the invariant holds at the end of the loop body and therefore before the next test of the loop condition. Hence (I) holds whenever the loop condition is checked.

```
Function binarySearch(x : Element, a : Array [1..n] of Element) : 1..n+1
    $(\ell, r) := (0, n+1)$
    assert (I)                                                    // (I) holds here.
    while $\ell + 1 < r$ do
        invariant (I): $0 \le \ell < r \le n+1 \wedge a[\ell] < x < a[r]$      // (I) is the loop invariant.
        assert (I) and $\ell + 1 < r$                 // Invariant (I) holds here. Also $\ell + 1 < r$.
        $m := \lfloor (r+\ell)/2 \rfloor$                                        // $\ell < m < r$
        $s := compare(x, a[m])$               // $-1$ if $x < a[m]$, 0 if $x = a[m]$, $+1$ if $x > a[m]$
        if $s = 0$ then return $m$                                            // $x = a[m]$
        if $s < 0$ then $r := m$                                   // $a[\ell] < x < a[m] = a[r]$
        if $s > 0$ then $\ell := m$                                   // $a[\ell] = a[m] < x < a[r]$
        assert (I)                                               // Invariant (I) holds here.
    assert (I) and $\ell + 1 = r$                     // Invariant (I) holds here. Also $\ell + 1 = r$.
    return $r$                                                      // $a[r-1] < x < a[r]$
```

Fig. 2.8. Binary search for x in a sorted array a. Returns an index k with $a[k-1] < x \le a[k]$.

It is now easy to complete the correctness proof. If we do not enter the loop, we have $\ell + 1 \ge r$. Since $\ell < r$ by the invariant and ℓ and r are integral, we have $\ell + 1 = r$. Thus $a[r-1] < x < a[r]$ by the second part of the invariant. We have now established correctness: The program returns either an index k with $a[k] = x$ or an index k with $a[k-1] < x < a[k]$.

We next argue termination. We observe first that if an iteration is not the last one, then we either increase ℓ or decrease r. 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. We therefore study the quantity $r - \ell - 1$. 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 at least halves the size of the problem. Indeed, in a round $r - \ell - 1$ 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 is at least halved. We start with $r - \ell - 1 = n + 1 - 0 - 1 = n$, and hence have $r - \ell - 1 \le \lfloor n/2^h \rfloor$ after h iterations.

Let us use k to denote the number of times the comparison between x and $a[m]$ is performed If x occurs in the array, the k-th comparison yields that $x = a[m]$, which ends the search. Otherwise testing the loop condition in the $k+1$-th iteration yields that $r \le \ell + 1$, and the search ends with this test. So when the loop condition is tested for the k-th time we must have $\ell + 1 < r$. Thus $r - \ell - 1 \ge 1$ after the $k-1$-th iteration, and hence $1 \le n/2^{k-1}$, which means $k \le 1 + \log n$. We conclude that, at most, $1 + \log n$ comparisons are performed. Since the number of comparisons is a natural number, we can sharpen the bound to $1 + \lfloor \log n \rfloor$.

Theorem 2.3. *Binary search locates an element in a sorted array of size n in at most* $1 + \lfloor \log n \rfloor$ *comparisons between elements. The computation time is* $\mathrm{O}(\log n)$.

Exercise 2.6. Show that the above bound is sharp, i.e., for every n, there are instances where exactly $1 + \lfloor \log n \rfloor$ comparisons are needed.

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

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 increasing function f and arguments i and j with $f(i) < x \leq f(j)$, we can use binary search to find $k \in i+1..j$ such that $f(k-1) < x \leq f(k)$. 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 $a[1..\infty]$ and some x, and we want to find the smallest k such that $x \leq a[k]$. If x is larger than all elements in the array, the procedure is allowed to diverge. We proceed as follows. We compare x with $a[2^0]$, $a[2^1]$, $a[2^2]$, $a[2^3]$, ..., until the first i with $x \leq a[2^i]$ is found. This is called an *exponential search*. If $x = a[2^i]$ or $i \leq 1$ (note that in the latter case, either $x \leq a[1]$ or $a[1] < x < a[2]$ or $x = a[2]$), we are done. Otherwise, $i > 1$ and $a[2^{i-1}] < x < a[2^i]$, and we complete the task by binary search on the subarray $a[2^{i-1}+1..2^i-1]$. This subarray contains $2^i - 2^{i-1} - 1 = 2^{i-1} - 1$ elements. Note that one comparison is carried out if $x \leq a[1]$.

Theorem 2.4. *The combination of exponential and binary search finds* $x > a[1]$ *in an unbounded sorted array in at most* $2\lceil \log k \rceil$ *comparisons, where* $a[k-1] < x \leq a[k]$.

Proof. If $a[1] < x \leq a[2]$, two comparisons are needed. So, we may assume $k > 2$, and hence the exponential search ends with $i > 1$. We need $i+1$ comparisons to find the smallest i such that $x \leq a[2^i]$ and $\lfloor \log(2^i - 2^{i-1} - 1) \rfloor + 1 = \lfloor \log(2^{i-1} - 1) \rfloor + 1$ $= i - 1$ comparisons for the binary search. This gives a total of $2i$ comparisons. Since $k > 2^{i-1}$, we have $i < 1 + \log k$, and the claim follows. Note that $i \leq \lceil \log k \rceil$ since i is integral. □

Binary search is certifying. It returns an index k with $a[k-1] < x \leq a[k]$. If $x = a[k]$, the index proves that x is stored in the array. If $a[k-1] < x < a[k]$ 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.

We have described binary search as an iterative program. It can also, and maybe even more naturally, be described as a recursive procedure; see Fig. 2.9. As above, we assume that we have two indices ℓ and r into an array a with index set $1..n$ such that $0 \leq \ell < r \leq n+1$ and $a[\ell] < x < a[r]$. If $r = \ell + 1$, we stop. This is correct by the assertion $a[\ell] < x < a[r]$. Otherwise, we compute $m = \lfloor (\ell + r)/2 \rfloor$. Then $\ell < m < r$. Hence we may access $a[m]$ and compare x with this entry (in a three-way fashion). If $x = a[m]$, we found x and return m. This is obviously correct. If $x < a[m]$, we make

```
**Function** binSearch(x : Element, $\ell, r$ : 0..n+1, a : Array $[1..n]$ of Element) : $1..n+1$
    assert $0 \leq \ell < r \leq n+1 \wedge a[\ell] < x < a[r]$                // The precondition
    if $\ell + 1 = r$ then return $r$                   // $x$ is not in the array and $a[r-1] < x < a[r]$
    $m := \lfloor (r+\ell)/2 \rfloor$                                          // $\ell < m < r$
    $s := compare(x, a[m])$                // $-1$ if $x < a[m]$, 0 if $x = a[m]$, $+1$ if $x > a[m]$
    if $s = 0$ then return $m$                                              // $x = a[m]$
    if $s < 0$ then return $binSearch(x, l, m, a)$
    if $s > 0$ then return $binSearch(x, m, r, a)$
```

Fig. 2.9. A recursive function for binary search

the recursive call for the index pair (ℓ, m). Note that $a[\ell] < x < a[m]$ and hence the precondition of the recursive call is satisfied. If $x > a[m]$, we make the recursive call for the index pair (m, r).

Observe that at most one recursive call is generated, and that the answer to the recursive call is also the overall answer. This situation is called *tail recursion*. Tail recursive procedures are easily turned into loops. The body of the recursive procedure becomes the loop body. Each iteration of the loop corresponds to a recursive call; going to the next recursive call with new parameters is realized by going to the next round in the loop, after changing variables. The resulting program is our iterative version of binary search.

2.8 Basic Algorithm Analysis

In this section, we introduce a set of simple rules for determining the running time of pseudocode. We start with a summary of the principles of algorithm analysis as we established them in the preceding sections. 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 now come to the set of 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:

- *Sequential composition*: $T(I;I') = T(I) + T(I')$.
- *Conditional instructions*: $T(\textbf{if } C \textbf{ then } I \textbf{ else } I') = \mathrm{O}(T(C) + \max(T(I), T(I')))$.

- *Loops*: $T(\textbf{repeat } I \textbf{ until } C) = \mathrm{O}\left(\sum_{i=1}^{k(n)} T(I,C,i)\right)$, where $k(n)$ is the maximum number of loop iterations on inputs of length n, and $T(I,C,i)$ is the time needed in the ith iteration of the loop, including the test C.

We postpone the treatment of subroutine calls to Sect. 2.8.2. Of the rules above, only the rule for loops is nontrivial to apply; it requires evaluating sums.

2.8.1 "Doing Sums"

We 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 loop variable i of the outer loop runs from from 2 to n. For any i, the inner loop performs at most $i-1$ iterations. Hence, the total number of iterations of the inner loop is at most

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

where the second equality comes from (A.12). Since the time for one execution of the inner loop is $\Theta(1)$, 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 currently 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.4. 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:

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

2.8.2 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 $T(n) = 4T(\lceil n/2 \rceil) + 4n$ for the number of primitive operations. For the Karatsuba algorithm, the corresponding equations were $T(n) = 3n^2$ for $n \leq 3$ and $T(n) = 3T(\lceil n/2 \rceil + 1) + 8n$ for $n > 3$. In general, a *recurrence relation* (or recurrence) defines a function in terms of the values of the same function on smaller arguments. Explicit definitions for small parameter values (the base case) complete the definition. Solving recurrences, i.e., finding nonrecursive, closed-form expressions for the functions defined by 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 $k \geq 1$, we invest linear work cn dividing the problem into d subproblems of size n/b and in combining the results. If $k = 0$, there are no recursive calls, we invest work a in computing the result directly, and are done.

Theorem 2.5 (master theorem (simple form)). *For positive constants a, b, c, and d, and integers n that are nonnegative powers of b, 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 2.10 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, i.e., at the zeroth level of the recursion we have one problem of size n. At the first level of the recursion, we have d subproblems of size n/b. From one level of the recursion to the next, the number of subproblems is multiplied by a factor of d and the size of the subproblems shrinks by a factor of b. Therefore, at the ith level of the recursion, we have d^i problems, each of size n/b^i. Thus the total size of the problems at the ith level is equal to

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

The work performed for a problem (excluding the time spent in recursive calls) 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 *topmost* 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.

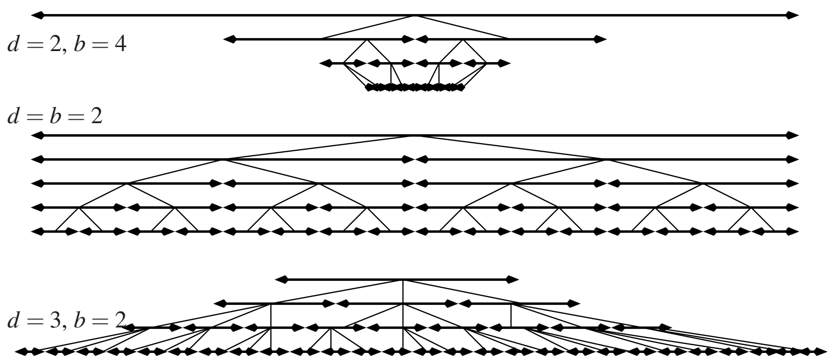

Fig. 2.10. Examples of the three cases of the master theorem. Problems are indicated by horizontal line segments with arrows at both ends. The length of a segment represents the size of the problem, and the subproblems resulting from a problem are shown in the line below it. The topmost part of figure 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 $n = b^k$. We call this level zero of the recursion.[6] 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 of 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 0 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 of $ad^k = ab^k = an = \Theta(n)$ for the bottom of the recursion and cost of $cnk = cn \log_b n = \Theta(n \log n)$ for the divide-and-conquer steps.

Case $d < b$. We have a cost of $ad^k < ab^k = an = \mathrm{O}(n)$ for the bottom of the recursion. For the cost of the divide-and-conquer steps, we use the summation formula (A.14) for a geometric series, namely $\sum_{0 \le i < k} q^i = (1 - q^k)/(1 - q)$ for $q > 0$ and $q \ne 1$, and obtain

[6] In this proof, we use the terminology of recursive programs in order to provide intuition. However, our mathematical arguments apply to any recurrence relation of the right form, even if it does not stem from a recursive program.

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

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

$$d^k = (b^{\log_b d})^k = b^{k \log_b d} = (b^k)^{\log_b d} = n^{\log_b d}.$$

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

$$cb^k \frac{(d/b)^k - 1}{d/b - 1} = c\frac{d^k - b^k}{d/b-1} = cd^k \frac{1-(b/d)^k}{d/b-1} = \Theta\Big(d^k\Big) = \Theta\Big(n^{\log_b d}\Big)\,. \qquad \square$$

There are many generalizations of the master theorem: We might break the recursion earlier, the cost of dividing and conquering might be nonlinear, the size of the subproblems might vary within certain bounds or vary stochastically, the number of subproblems might depend on the input size, etc. We refer the reader to the books [137, 288] and the papers [13, 100, 266] for further information. The recurrence $T(n) = 3n^2$ for $n \le 3$ and $T(n) \le 3T(\lceil n/2 \rceil + 1) + 8n$ for $n \ge 4$, governing Karatsuba's algorithm, is not covered by our master theorem, which neglects rounding issues. We shall now state, without proof, a more general version of the master theorem. Let $r(n)$ satisfy

$$r(n) \le \begin{cases} a & \text{if } n \le n_0, \\ cn^s + d \cdot r(\lceil n/b \rceil + e_n) & \text{if } n > n_0, \end{cases} \tag{2.1}$$

where $a > 0$, $b > 1$, $c > 0$, $d > 0$, and $s \ge 0$ are real constants, and the e_n, for $n > n_0$, are integers such that $-\lceil n/b \rceil < e_n \le e$ for some integer $e \ge 0$. In the recurrence governing Karatsuba's algorithm, we have $n_0 = 3$, $a = 27$, $c = 8$, $s = 1$, $d = 3$, $b = 2$, and $e_n = 1$ for $n \ge 4$.

Theorem 2.6 (master theorem (general form)).
If $r(n)$ satisfies the recurrence (2.1), then

$$r(n) = \begin{cases} \mathrm{O}(n^s) & \text{for } d < b^s, \text{ i.e., } \log_b d < s, \\ \mathrm{O}(n^s \log n) & \text{for } d = b^s, \text{ i.e., } \log_b d = s, \\ \mathrm{O}(n^{\log_b d}) & \text{for } d > b^s, \text{ i.e., } \log_b d > s. \end{cases} \tag{2.2}$$

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

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

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 $T(n) = \mathrm{O}(\log n)$.

2.8.3 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 quite 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 1 initially and at most n at termination, and hence the number of executions is bounded logarithmically.

2.9 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. The use of the language and tools of probability theory suggests the following approach to average case analysis. We view the inputs as coming from a probability space, e.g., all inputs from a certain size with the uniform distribution, and determine the expexted complexity for an instance sampled randomly from this space. Section A.3 reviews the basic probability theory.

2.9.1 Incrementing a Counter

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

```
$i := 0$
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 0 or 1 with probability $1/2$, and different digits are independent. Alternatively, we may say that all bit strings of length n are equally likely. The loop body is executed k times, if either $k < n$ and $a[0] = a[1] = \ldots = a[k-1] = 1$ and $a[k] = 0$ or if $k = n$ and all digits of a are equal to 1. 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

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

where the last equality is the same as (A.15).

2.9.2 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, it 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 1 for a permutation π if the kth 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.3) and the third equality follows from the I_k's being indicator variables. It remains to determine the probability that $I_k = 1$. The kth element of a random permutation is a left-to-right maximum if and only if the kth 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 $\text{prob}(I_k = 1) = 1/k$ and hence

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

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 "nth 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.13). We conclude that the average number of left-to-right maxima is much smaller than their maximum number.

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 $A_n = H_n$.

2.9.3 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 fixed probabilities; say, we shall 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 2.7. *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

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

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 item i is placed after item 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 the item in position 1 and move the items in positions 1 to $\ell_i - 1$ by 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.

We assume for the analysis that we start with an arbitrary, but fixed, initial arrangement of the n items and then perform search rounds. In each round, we access item i with probability p_i *independently of what happened in the preceding rounds*. Since the cost of the first access to any item is essentially determined by the initial configuration, we shall ignore it and assign a cost of 1 to it.[7] We now compute the expected cost in round t. We use C_{MTF} to denote this expected cost. Let ℓ_i be the position of item i at the beginning of round t. The quantities $\ell_1, \ldots, \ell_n$ are random variables that depend only on the accesses in the first $t-1$ rounds; recall that we assume a fixed initial arrangement. If we access item i in round t, we incur a cost of $1 + Z_i$, where[8]

[7] The cost ignored in this way is at most $n(n-1)$. One can show that the expected cost in round t ignored in this way is no more than n^2/t.

[8] We define the cost as $1 + Z_i$, so that $Z_i = 0$ is the second case.

$$Z_i = \begin{cases} \ell_i - 1 & \text{if } i \text{ was accessed before round } t, \\ 0 & \text{otherwise.} \end{cases}$$

Of course, the random variables $Z_1, \dots, Z_n$ also depend only on the sequence of accesses in the first $t-1$ rounds. Thus

$$C_{\text{MTF}} = \sum_i p_i (1 + \mathrm{E}[Z_i]) = 1 + \sum_i p_i \mathrm{E}[Z_i].$$

We next estimate the expectation $\mathrm{E}[Z_i]$. For this purpose, we define for each $j \neq i$ an indicator variable

$$I_{ij} = \begin{cases} 1 & \text{if } j \text{ is located before } i \text{ at the beginning of round } t \\ & \text{and at least one of the two items was accessed before round } t, \\ 0 & \text{otherwise.} \end{cases}$$

Then $Z_i \leq \sum_{j;\, j \neq i} I_{ij}$. Indeed, if i is accessed for the first time in round t, $Z_i = 0$. If i was accessed before round t, then $I_{ij} = 1$ for every j that precedes i in the list, and hence $Z_i = \sum_{j;\, j \neq i} I_{ij}$. Thus $\mathrm{E}[Z_i] \leq \sum_{j;\, j \neq i} \mathrm{E}[I_{ij}]$. We are now left with the task of estimating the expectations $\mathrm{E}[I_{ij}]$.

If there was no access to either i or j before round t, $I_{ij} = 0$. Otherwise, consider the last round before round t in which either i or j was accessed. The (conditional) probability that this access was to item j and not to item i is $p_j/(p_i + p_j)$. Therefore, $\mathrm{E}[I_{ij}] = \mathrm{prob}(I_{ij} = 1) \leq p_j/(p_i + p_j)$, and hence $\mathrm{E}[Z_i] \leq \sum_{j;\, j \neq i} p_j/(p_i + p_j)$. Summation over i yields

$$C_{\text{MTF}} = 1 + \sum_i p_i \mathrm{E}[Z_i] \leq 1 + \sum_{i,j;\, i \neq j} \frac{p_i p_j}{p_i + p_j}.$$

Observe that for each i and j with $i \neq j$, the term $p_i p_j/(p_i + p_j) = p_j p_i/(p_j + p_i)$ appears twice in the sum above. In order to proceed with the analysis, we assume $p_1 \geq p_2 \geq \cdots \geq p_n$. We use this assumption in the analysis, but the algorithm has no knowledge of this. With $\sum_i p_i = 1$, we obtain

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

Theorem 2.8. *If the cost of the first access to each item is ignored, the expected search cost of the move-to-front-heuristic is at most twice the cost of the optimal fixed arrangement.*

2.10 Parallel-Algorithm Analysis

Analyzing a sequential algorithm amounts to estimating the execution time $T_{\text{seq}}(I)$ of a program for a given input instance I on a RAM. Now, we want to find the execution time $T_{\text{par}}(I,p)$ as a function of both the input instance and the number of available processors p of some parallel machine. As we are now studying a function of two variables, we should expect some complications. However, the basic tools introduced above – evaluating sums and recurrences, using asymptotics, ... – will be equally useful for the analysis of parallel programs. We begin with some quantities derived from T_{seq} and T_{par} that will help us to understand the results of the analysis.

The (absolute) *speedup*

$$S(I,p) := \frac{T_{\text{seq}}(I)}{T_{\text{par}}(I,p)} \tag{2.3}$$

gives the factor of speed improvement compared with the best known sequential program for the same problem. Sometimes the relative speedup $T_{\text{par}}(I,1)/T_{\text{par}}(I,p)$ is considered, but this is problematic because it does not tell us anything about how useful parallelization was. Of course, we would like to have a large speedup. But how large is good? Ideally, we would like to have $S = p$ – perfect speedup[9]. Even $S = \Theta(p)$ – *linear speedup* – is good. Since speedup $\Theta(p)$ means "good", it makes sense to normalize speedup to the *efficiency*

$$E(I,p) := \frac{S(I,p)}{p}, \tag{2.4}$$

so that we are now looking for constant efficiency. When do we call a parallel algorithm good or efficient? A common definition is to say that a parallel algorithm is efficient if it achieves constant efficiency for all sufficiently large inputs. The input size for which it achieves constant efficiency may grow with the number of processors. The *isoefficiency function* $I(p)$ measures this growth [191]. Let c be a constant. For any number p of processors, let $I(p)$ be the smallest n such that $E(I,p) \geq c$ for all instances I of size at least n. The isoefficiency function measures the scalability of an algorithm – the more slowly it grows as a function of p, the more scalable the algorithm is.

Brent's Principle. Brent's principle is a general method for converting inefficient parallel programs into efficient parallel programs for a smaller number of processors. It is best illustrated by an example. Assume we want to sum n numbers that are given in a global array. Clearly, $T_{\text{seq}}(n) = \Theta(n)$. With $n = p$ and the fast parallel sum algorithm presented in Sect. 2.4.2, we obtain a parallel execution time $O(\log p)$ and efficiency $O(1/\log p)$ – this algorithm is inefficient. However, when $n \gg p$ we can use a simple trick to get an efficient parallelization. We first use p copies of the sequential algorithm on subproblems of size n/p, and then

[9] There are situations where we can do even better. $S > p$ can happen if parallel execution mobilizes more resources. For example, on the parallel machine the input might fit into the aggregate cache of all PEs while a single PE needs to access the main memory a lot.

use the parallel algorithm with p processors to sum the p partial sums. Each PE adds n/p numbers sequentially and requires time $\Theta(n/p)$. The summation of the p partial results takes time $\Theta(\log p)$. Thus the overall parallel execution time is $T_{\text{par}} = \Theta(n/p + \log p)$. We obtain speedup $S = \Theta(n/(n/p + \log p))$ and efficiency $E = \Theta(n/(p(n/p + \log p))) = \Theta(1/(1 + (p \log p)/n))$. For E to be constant, we need $p \log(p)/n = \mathrm{O}(1)$, i.e., $n = \Omega(p \log p)$. Hence, the isoefficiency of the algorithm is $I(p) = \Theta(p \log p)$.

Work and Span. The work, $\mathrm{work}(I)$, of a parallel algorithm performed on an instance I is the number of operations executed by the algorithm. Its span, $\mathrm{span}(I)$, is the execution time $T(I, \infty)$ if an unlimited number of processors is available. These two quantities allow us to obtain a rather abstract view of the efficiency and scalability of a parallel algorithm. Clearly, $T(I, p) \geq \mathrm{span}(I)$ and $T(I, p) \geq \mathrm{work}(I)/p$ are obvious lower bounds on the parallel execution time. So even with an infinite number of processors, the speed-up cannot be better than $\mathrm{span}(I)/T_{\text{seq}}(I)$ (Amdahl's law). On the other hand, we can achieve $T(I, p) = \mathrm{O}(\mathrm{work}(I)/p + \mathrm{span}(I))$ if we manage to schedule the computations in such a way that no PE is unnecessarily idle. In Sect. 14.5 we shall see that this is often possible. In this case, the algorithm is efficient provided that $\mathrm{work}(I) = \mathrm{O}\big(T_{\text{seq}}(I)\big)$ and $\mathrm{span}(I) = \mathrm{O}(\mathrm{work}(I)/p)$. Indeed,

$$\begin{aligned} T(I, p) &= \mathrm{O}(\mathrm{work}(I)/p + \mathrm{span}(I)) \\ &= \mathrm{O}(\mathrm{work}(I)/p) && \text{since } \mathrm{span}(I) = \mathrm{O}(\mathrm{work}(I)/p) \\ &= \mathrm{O}\big(T_{\text{seq}}(I)\big) && \text{since } \mathrm{work}(I) = \mathrm{O}\big(T_{\text{seq}}(I)\big). \end{aligned}$$

For the array-sum example discussed above, we have $\mathrm{work}(n) = \Theta(n)$ and $\mathrm{span}(n) = \Theta(\log n)$, i.e., the algorithm is efficient if $\log n = \mathrm{O}(n/p)$, i.e., when $p = \mathrm{O}(n/\log n)$ or $n = \Omega(p \log p)$. Analyzing work and span allows us to consider scalability in a way similar to what we can with the isoefficiency function.

2.11 Randomized Algorithms

Suppose you are offered 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 amount in the box is larger than the amount in all previously opened boxes, you have to hand back a token.[10]
- 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.

[10] The amount in the first box opened is larger than the amount in all previously opened boxes, and hence the first token goes back to the presenter in the first round.

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 presenters'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 the presenterr 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 nth harmonic number. For $n = 100$, $H_n < 6$. So if the presenter were to point to the boxes 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 will 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.

We give a second example. Suppose that you are given an urn with white and red balls and your goal is to get a white ball. You know that at least half of the balls in the urn are white. Any deterministic strategy may be unlucky and look at all the red balls before it finds a white ball. It is much better to consider the balls in random order. Then the probability of picking a white ball is at least $1/2$ and hence an expected number of two draws suffices to get a white ball. Note that as long as you draw a red ball, the percentage of white balls in the urn is at least 50% and hence the probability of drawing a white ball stays at least $1/2$. The second example is an instance of the following scenario. You are given a large set of candidates and want to find a candidate that is good in some sense. You know that half of the candidates are good, but you have no idea which ones. Then you should examine the candidates in random order.

We come to a third example. Suppose, Alice and Bob are connected over a slow telephone line. Alice has an integer x and Bob has an integer y, each with six decimal digits. 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 6 digits. Alternatively, Alice could send the number digit by digit, and Bob would check for equality as the digits arrive and announce the result as soon as he knew it, i.e., as soon as corresponding digits differ or all digits had been transmitted. In the worst case, all 6 digits have to be transmitted. We shall now show how to use randomization for this task.

There are 21 two digit prime numbers, namely

$$11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97.$$

The protocol is now as follows. Alice chooses a random prime p from the list and sends p and $x \bmod p$ to Bob. Note that $x \bmod p$ is a 2-digit number, and hence Alice sends four digits. Bob computes $y \bmod p$ and compares it to $x \bmod p$. If the two remainders are different, he declares that x and y are different, otherwise he declares that x and y are the same. What is the probability that Bob declares an incorrect result? If x and y are equal, his answer is always correct. If x and y are different, their difference is a non-zero number which is less than 1 million in absolute value and hence is divided by at most six numbers on the list, since the product of any six numbers on the list exeeds 1 million. Thus the probability that Bob declares an incorrect result is at most $6/21$. Alice and Bob can reduce the probability of an incorrect result by computing the remainders with respect to more primes. Of course, this would also increase the number of bits sent. Can one live with incorrect answers? Yes, if the probability of an incorrect answer is sufficiently small. We continue this example in Section 2.11.2.

Exercise 2.12. Find out how many 3-digit primes there are? You should be able to find the answer in Wikipedia. Assume that x and y are less than 10^{20}. How many 3-digit primes can divide $x - y$, if $x \neq y$? What is the probability that Bob declares an incorrect result in the protocol above?

2.11.1 The Formal Model

Formally, we equip our RAM with an additional instruction: $R_i := \mathit{randInt}(C)$ which assigns a *random* integer between 0 and $C - 1$ to R_i. All C of these values are equally likely, and the value assigned is independent of the outcome of all previous random choices. In pseudocode, we write $v := \mathit{randInt}(C)$, 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.

2.11.2 *An Advanced Example

We continue with the third example from the introduction. Suppose, Alice and Bob are connected over a slow telephone line. Alice has an integer x and Bob has an integer y, 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 $\mathrm{O}(\log n)$ 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 L smallest primes $p_1, p_2, \ldots, p_L$ with a value of at least 2^k. We shall say more about the choice of L and k below. Clearly, Alice and Bob generate the same list. Then Alice chooses an index i, $1 \leq i \leq L$, at random and sends i and $x \bmod p_i$ to Bob. Bob computes $y \bmod p_i$. If $x \bmod p_i \neq y \bmod p_i$, 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 $x \bmod p_i \neq y \bmod p_i$, he will declare them different. However, if $x \neq y$ and yet $x \bmod p_i = y \bmod p_i$, he will erroneously declare the numbers equal. What is the probability of an error?

An error occurs if $x \neq y$ but $x \equiv y \bmod p_i$. The latter condition is equivalent to p_i dividing the difference $D = x - y$. 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[11] 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 $0.000001 = 10^{-6}$, we choose $L = 10^6(n/k)$.

What is the appropriate choice of k? For sufficiently large k, about $2^k/\ln(2^k) = 1.4477 \cdot 2^k/k$ primes[12] are contained in the interval $[2^k..2^{k+1}-1]$. Hence, if $2^k/k \geq 10^6 n/k$, the list will contain only $k+1$-bit integers. The condition $2^k \geq 10^6 n$ is equivalent to $k \geq \log n + 6\log 10$. 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 $L = 10^{12}(n/k)$. 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 2.13. Compare the efficiency of the two approaches for obtaining an error probability of 10^{-12}.

Exercise 2.14. In the protocol described above, Alice and Bob have to prepare ridiculously long lists of prime numbers. Discuss the following modified protocol. Alice chooses a random $k+1$-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 $x \bmod p$ to Bob.

Exercise 2.15. 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.

2.11.3 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 (if the player is provided with enough tokens); it always finds the box containing the maximum; however, the number of tokens to be returned (the number of left-to-right maxima) is a random variable. A *Monte Carlo* algorithm always has the same running time, but there is a nonzero probability that it will give an incorrect answer. The probability that the answer is incorrect is at most 1/4. Our algorithm

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

[12] 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 $x \geq 55$. See the Wikipedia entry "Prime numbers" for more information.

for comparing two numbers over a telephone line is a Monte Carlo algorithm. In Exercise 2.15, it is shown that the error probability of a Monte Carlo algorithm can be made arbitrarily small by repeated execution.

Exercise 2.16. 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 allotted time, this answer is returned, otherwise an arbitrary answer is returned. Show that the resulting algorithm is a Monte Carlo algorithm.

Exercise 2.17. 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 now 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 $c \geq 1$, $\text{prob}(X \geq c \cdot \text{E}[X]) \leq 1/c$; see (A.5) for additional information. We apply the inequality with $X = M_n$ and $c = 11/6$. We obtain

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

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

2.12 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* or *arcs*. Some obvious applications are 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 small disks 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. 2.11 shows a graph G with node set $\{s,t,u,v,w,x,y,z\}$ and edges (s,t), (t,u), (u,v), (v,w), (w,x), (x,y), (y,z), (z,s), (s,v), (z,w), (y,t), and (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* to u and v and that v and u are *adjacent*. The special case of a *self-loop* (v,v) is disallowed unless specifically mentioned otherwise. Modeling E as a set of edges also excludes multiple parallel edges between the same two nodes. However, sometimes it is useful to allow parallel edge, i.e., in a multigraph, E is a multiset where elements can appear multiple times.

The *outdegree* of a node v is the number of edges leaving it, and its *indegree* is the number of edges entering it. Formally, *outdegree*$(v) = |\{(v,u) \in E\}|$ and *indegree*$(v) = |\{(u,v) \in E\}|$. For example, node w in graph G in Fig. 2.11 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 2.11 includes a three-node undirected graph and its bidirected counterpart. Most graph-theoretic terms for undirected graphs have the same definition as for 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. 2.11, 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').

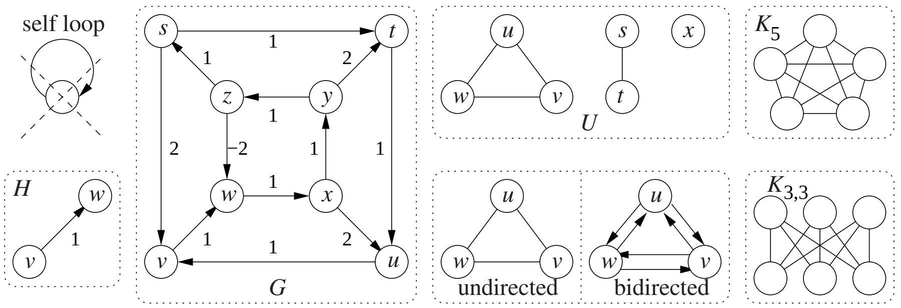

Fig. 2.11. Some graphs.

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. 2.11 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 rather many 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., $(v_0,v_1) \in E$, $(v_1,v_2) \in E$, $\ldots$, $(v_{k-1},v_k) \in E$; 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. 2.11. A sequence $p = \langle v_0, v_1, \ldots, v_k \rangle$ is a path in an undirected graph if it is a path in the corresponding bidirected graph and $v_{i-1} \neq v_{i+1}$ for $1 \leq i < k$, i.e., it is not allowed to use an edge and then immediately go back along the same edge. The sequence $\langle u,w,v,u,w,v \rangle$ is a path of length 5 in the graph U in Fig. 2.11. A path is *simple* if its nodes, except maybe for v_0 and v_k, are pairwise distinct. In Fig. 2.11, $\langle z,w,x,u,v,w,x,y \rangle$ is a nonsimple path in graph G. Clearly, if there is a path from u to v in some graph, there is also a simple path from u to v.

Cycles are paths of length at least 1 with a common first and last node. Cycles in undirected graphs have a length of at least three since consecutive edges must be distinct in a path in an undirected graph. In Fig. 2.11, the sequences $\langle u,v,w,x,y,z,w,x,u \rangle$ and $\langle u,w,v,u,w,v,u \rangle$ are cycles in G and U respectively. 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. 2.11 is Hamiltonian. The cycle $\langle w,u,v,w \rangle$ in U is also Hamiltonian.

The concepts of paths and cycles allow 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. 2.11 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. 2.11, we obtain a digraph without any 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. Any two nodes in the same connected component are connected by a path, and there are no edges connecting nodes in distinct connected components. For example, graph U in Fig. 2.11 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 is not a component.

Exercise 2.18. 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 2.19. 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 $K_{3,3}$ in Fig. 2.11 are not planar.

2.12.1 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 0 cannot lie on 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^* in which every node has an outdegree of at least 1. 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. 2.11, graph G has no node with outdegree 0. 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 $\mathrm{O}(|V|+|E|)$; see also Exercise 8.3. We can easily make our algorithm certifying. If the algorithm finds a cycle, the graph is certainly cyclic. Also it is easily checked whether the returned sequence of nodes is indeed a cycle. 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 0 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. Again this property is easily checked.

Exercise 2.20. Exhibit a DAG with n nodes and $n(n-1)/2$ edges for every n.

2.12.2 Trees

An undirected graph is a *tree* if there is *exactly* one path between any pair of nodes; see Fig. 2.12 for an example. An undirected graph is a *forest* if there is *at most* one path between any pair of nodes. Note that each connected component of a forest is a tree.

Lemma 2.9. *The following properties of an undirected graph G are equivalent:*

(a) G is a tree;
(b) G is connected and has exactly $n-1$ edges;
(c) G is connected and contains no cycles.

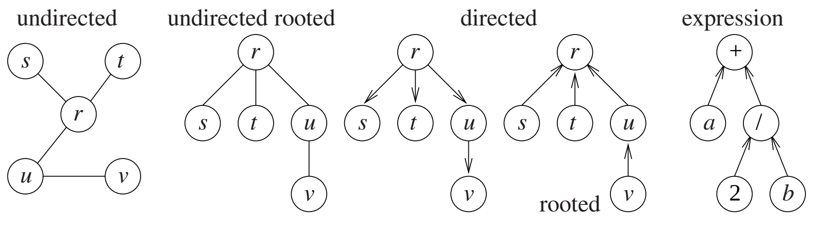

Fig. 2.12. Different kinds of trees. From *left* to *right*, we see an undirected tree, an undirected rooted tree, a directed out-tree, a directed in-tree, and an arithmetic expression.

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. Consider distinct paths p and q connecting the same pair of nodes. If the first edge of p is equal to the first edge of q, delete these edges from p and q to obtain p' and q'. The paths p' and q' are distinct and connect the same pair of nodes. Continuing in this way and also applying the argument to last edges, we end up with two paths connecting the same pair of nodes and having distinct first and last edges. Thus the concatenation of the first path with the reversal of the second forms a cycle. We have now shown that (a) and (c) are equivalent.

We next show the equivalence of (b) and (c). 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 (c) implies (b). Assume next that G is connected and has exactly $n-1$ edges. Again, add the edges one by one and observe that every addition must reduce the number of connected components by one, as otherwise we would not end up with a single component after $n-1$ additions. Thus no addition can close a cycle, as such an addition would not reduce the number of connected components. Thus (b) implies (c). □

Lemma 2.9 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 2.12 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 of the depths of its nodes.

We can 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 the *subtree* rooted at u. For example, in Fig. 2.12, 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.

2.12.3 Ordered Trees

Trees are ideally suited for representing hierarchies. For example, consider the expression $a+2/b$. We know that this expression means that a and $2/b$ are to be added. But deriving this from the sequence of characters $\langle a,+,2,/,b\rangle$ is difficult. For example, the rule that division binds more tightly than addition has to be applied. 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. 2.12. Such trees are directed and, in contrast to graph-theoretic trees, they are *ordered*, i.e., the children of each node 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 2.13 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 Chap. 12 uses them to systematically explore solution spaces.

```
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(first child of r)
        $v_2$ := eval(second child of r)
        return $v_1$ operator(r) $v_2$                    // apply the operator stored in r
```

Fig. 2.13. Recursive evaluation of an expression tree rooted at r

2.13 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 $\mathrm{O}(p(n))$. 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 $\mathrm{O}(n^3)$ algorithm by a factor of n. All chapters of this book, except for Chap. 12, are about efficient algorithms. We use **P** to denote the class of problems solvable in polynomial time.

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. A graph is *complete* if every pair of nodes is connected by an edge. The graph K_5 in Fig. 2.11 is an example.
- *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 $\sum_{i \in I} p_i \geq P$. Informally, item i has volume w_i and value p_i and we want to know whether we can pack a knapsack of volume M such that its value is at least P. This problem will be heavily used as an example in Chap. 12.
- *The traveling salesman problem*: Given an edge-weighted undirected graph and an integer C, decide whether the graph contains a Hamiltonian cycle of cost at most C. See Sect. 11.7.2 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. This problem will also be used as an example in Chap. 12.

The fact that we know no efficient algorithms for these problems does not imply that none exist. It is simply not known whether efficient algorithms exist 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. 5.5. 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.

*More on NP-Completeness

We shall now give formal definitions of the class **NP** and the class of **NP**-complete problems. We refer the reader to books about the theory of computation and complexity theory [22, 121, 298, 328] for a thorough treatment.

We assume, as is customary in complexity theory, that inputs are encoded in some fixed finite alphabet Σ. Think of the ASCII or Unicode alphabet or their binary encodings. In the latter case, $\Sigma = \{0,1\}$. We use Σ^* to denote all words (sequences of characters) over the alphabet Σ. The size of a word $x = a_1 \ldots a_n \in \Sigma^*$ is its length n. A *decision problem* is a subset $L \subseteq \Sigma^*$. We use χ_L (read "chi") 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 if and only if 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** if and only if there is a predicate $Q(x,y)$ (a subset $Q \subseteq (\Sigma^*)^2$) and a polynomial p such that

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

We call y satisfying (a) a *witness* for x or a *proof* of membership for x. For our example problems, it is easy to show that they belong to **NP**. In the case of the Hamiltonian cycle problem, the witness is a Hamiltonian cycle in the input graph. A witness for a Boolean formula is an assignment 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 2.21. Prove that the clique problem, the traveling salesman problem, and the graph-coloring problem are in **NP**.

It is widely believed that **P** is a proper subset of **NP**. There are good arguments for this belief, as we shall see in a moment; however, there is no proof. In fact, the problem of whether **P** is equal to **NP** or properly contained in it is considered one of the major open problems in computer science and mathematics. A proof that the two classes are equal would have dramatic consequences: Thousands of problems which are currently believed to have no efficient algorithm would suddenly have one. A proof that the two classes are not equal would probably have no dramatic effect on computing, as most algorithmicists work under the assumption that these classes are distinct, but it would probably have a dramatic effect on theoretical computer science, logic, and mathematics, as the proof would probably introduce a new kind of argument. If **P** is properly contained in **NP**, **NP**-complete problems have no efficient algorithm.

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 $x \in L$ if and only if $g(x) \in L'$. If L is reducible to L' and $L' \in \mathbf{P}$, then $L \in \mathbf{P}$. Assume we have an algorithm for the reduction g with a polynomial time bound $p(n)$ and an algorithm for $\chi_{L'}$ with a polynomial time bound $q(n)$. An algorithm for χ_L operates as follows. On input of x, it first computes $g(x)$ using the first algorithm and then tests $g(x) \in L'$ using the second algorithm. The running time is at most $p(|x|) + q(|g(x)|)$. Since Turing machines can write at most one symbol in each step, we have $|g(x)| \le |x| + p(|x|)$. Thus the running time is bounded by $p(|x|) + q(|x| + p(|x|))$; this is polynomial in $|x|$. A similar argument shows that 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 [79, 202]. 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 **NP**-complete problem, it becomes easier to show that other problems are **NP**-complete. There is a Wikipedia page for the list of **NP**-complete problems. We next give one example of a reduction.

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

Proof. Let $F = C_1 \wedge \ldots \wedge C_m$, where $C_i = \ell_{i1} \vee \ldots \vee \ell_{ih_i}$ and $\ell_{ik} = x_{ik}^{\beta_{ik}}$, be a formula in conjunctive form. Here, x_{ik} is a variable and $\beta_{ik} \in \{0,1\}$. A superscript 0 indicates a negated variable. Consider the following graph G. Its nodes V represent the literals in our formula, i.e., $V = \{(i,k) : 1 \le i \le m \text{ and } 1 \le k \le h_i\}$. Two nodes (i,k) and (j,k') are connected by an edge if and only if $i \ne j$ and either $x_{ik} \ne x_{jk'}$ or $\beta_{ik} = \beta_{jk'}$. 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 if and only if G has a clique of size m.

Assume first that there is a satisfying assignment α. The assignment must satisfy at least one literal in every clause, say literal ℓ_{ik_i} in clause C_i. Consider the subgraph of G induced by the node set $\{(i,k_i) : 1 \le i \le m\}$. This is a clique of size m. Assume otherwise; say, (i,k_i) and (j,k_j) are not connected by an edge. Then, $x_{ik_i} = x_{jk_j}$ and $\beta_{ik_i} \ne \beta_{jk_j}$. But then the literals ℓ_{ik_i} and ℓ_{jk_j} are complements of each other, and α cannot satisfy them both.

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

Exercise 2.22. Show that the Hamiltonian cycle problem is polynomial-time reducible to the traveling salesman problem.

Exercise 2.23. 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 better than or equal to k, where k is an additional input. Here, better means greater in a maximization problem and smaller in a minimization problem. Conversely, if we have an algorithm to decide whether there is a solution with a value better than or equal to k, we can use a combination of exponential and binary search (see Sect. 2.7) 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 to instances derived from the original instance. 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 first test whether G has a clique of size k. If not, there is no clique of size k. Otherwise, we select an arbitrary node v and ask whether $G' = G \setminus v$ has a clique of size k. If so, we search recursively for a clique of size k in G'. If not, we know that v must be part of the clique. Let V' be the set of neighbors of v. We search recursively 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.

2.14 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 [225] 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, we 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 pseudorandom numbers. We refer the reader to the Wikipedia page "Random number" for more information.

There has been a lot of research on parallel programming languages and software libraries for sequential languages. However, most users are conservative and use only a small number of tools that are firmly established and have wide industrial support in order to achieve high performance in a portable way. We shall take the same attitude in the implementation notes in this book. Moreover, some advanced, recently introduced, or rarely used features of firmly established tools may not deliver the performance you might expect, and should be used with care. However, we would like to point out that higher-level tools or features may be worth considering if you can validate your expectation that they will achieve sufficient performance and portability for your application.

2.14.1 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 arbitrary values (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 [255], the Boost [50] C++ libraries, and the LEDA [194, 217] library of efficient algorithms and data structures.

C++ (together with C) is perhaps the most widely used programming language for nonnumerical[13] parallel computing because it has good, widely portable compilers and allows low-level tuning. However, only the recent C++11 standard begins to define some support for parallel programming. In Appendix C, we give a short introduction to the parallel aspects of C++11. We also say a few words about shared-memory parallel-programming tools used together with C++ such as OpenMP, Intel TBB, and Cilk.

In Appendix D we introduce MPI, a widely used software library for message-passing-based programming. It supports a wide variety of message-passing routines, including collective communication operations (see also Chap. 13).

[13] For *numerical* parallel computing, Fortran was traditionally the most widely used language. But even that is changing.

2.14.2 Graphics Processing Units (GPUs)

GPUs are often used for general-purpose parallel processing (general-purpose computing on graphics processing units, GPGPU). GPUs can be an order of magnitude more efficient than classical multicore processors with a comparable number of transistors. This is achieved using massive parallelism – the number of (very lightweight) threads used can be three orders of magnitude larger than for a comparable multicore processor. GPU programs therefore need highly scalable parallel algorithms. Further complications are coordination with the host CPU (heterogeneity), explicit management of several types of memory, and threads working in SIMD mode. In this book, we focus on simpler hardware but many of the algorithms discussed are also relevant for GPUs. For NVIDIA GPUs there is a C++ extension (part of the Compute Unified Device Architecture, CUDA) that allows rather high-level programming. A more portable but lower-level system is the C extension OpenCL (Open Computing Language).

2.14.3 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.

Implementations for many data structures and algorithms are available in the package *java.util*.

Java supports multithreaded programming of a shared-memory machine, including locks, support for atomic instructions, and data structures supporting concurrent access; see the documentation of the libraries beginning with `java.util.concurrent`. However, some high-level concepts such as parallel loops, collective operations, and efficient task-oriented programming are missing. There is also no direct support for message-passing programming. There are several software libraries and compilers addressing these deficits; see [304] for an overview. However, it is perhaps too early to say whether any of these techniques will gain a wide user base with efficient, widely portable implementations. More fundamentally, when an application is sufficiently performance-sensitive for one to consider parallelization, it is worth remembering that Java often incurs a significant performance penalty compared with using C++. This overhead can be worse for a multithreaded code than for a sequential code, since additional cache faults may expose a bottleneck in the memory subsystem and garbage collection incurs additional overheads and complications in a parallel setting.

2.15 Historical Notes and Further Findings

Shepherdson and Sturgis [293] 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 [145]. Knuth [185] has described a more detailed abstract machine model.

A huge number of algorithms have been developed for the PRAM model. Jájá's textbook [163] is a good introduction.

There are many variants of distributed-memory models. One can take a more detailed look at concrete network topologies and differentiate between nearby and faraway PEs. For example, Leighton's textbook [197] describes many such algorithms. We avoid these complications here because many networks can actually support our simple model to a reasonable approximation and because we shy away from the complications and portability problems of detailed network models.

One can also take a more abstract view. The *bulk synchronous parallel* (BSP) model [318] divides the computation into globally synchronized phases of local computation on the one hand and of global message exchange on the other hand. During a local computation phase, each PE can post send requests. During a message exchange phase, all these messages are delivered. In the terminology of this book, BSP programs are message-passing programs that use only the nonuniform all-to-all method described in Sect. 13.6.3 for communication. Let h denote the maximum number of machine words sent or received by any PE during a message exchange. Then the BSP model assumes that this message exchange takes time $\ell + gh$, where ℓ and g are machine parameters. This assumption simplifies the analysis of BSP algorithms. We do not adopt the BSP model here, since we want to be able to describe asynchronous algorithms and because, with very little additional effort for the analysis, we get more precise results, for example, when other collective communication operations presented in Chap. 13 are used.

A further abstraction looks only at the *communication volume* of a parallel algorithm, for example, by summing the h-values occurring in the communication steps in the BSP model [274]. This makes sense on large parallel systems, where global communication becomes the bottleneck for processing large data sets.

For modeling cache effects in shared-memory systems, the *parallel external-memory* (PEM) model [19] is useful. The PEM is a combination of the PRAM model and the external-memory model. PEs have local caches of size M each and access the shared main memory in cache lines of size B.

Floyd [106] introduced the method of invariants to assign meaning to programs and Hoare [151, 152] systematized their use. The book [137] is a compendium of sums and recurrences and, more generally, discrete mathematics.

Books on compiler construction (e.g., [232, 330]) will tell you more about the compilation of high-level programming languages into machine code.