Chapter 1. Foundations

We use computer algorithms to solve problems, e.g., to compute the maximum of a set of real numbers or to compute the product of two integers. A problem P consists of infinitely many problem instances. An instance of the maximum problem is e.g., to compute the maximum of the following five numbers $2, 7, 3, 9, 8$. An instance of the multiplication problem is, e.g., to compute the product of 257 and 123. We associate with every problem instance $p \in P$ a natural number $g(p)$, its size. Sometimes, the size will be a tuple of natural numbers; e.g., we measure the size of a graph by a pair consisting of the number of nodes and the number of edges. In the maximum problem we can define the size as the cardinality of the input set (5 in our example), in the multiplication problem we can define the size as the sum of the lengths of the decimal representations of the factors (6 in our example). Although the definition of size is arbitrary, there is usually a natural choice.

Execution of a program on a machine requires resources, e.g., time and space. Resource requirements depend on the input. We use $T_A(p)$ to denote the running time of algorithm A on problem instance p. We can determine $T_A(p)$ by experiment and measure it in milliseconds.

Global information about the resource requirements of an algorithm is in general more expressive than information about resource requirements on particular instances. Global information such as maximal running time on an input of size n cannot be determined by experiment. Two abstractions are generally used: worst case and average case behavior.

Worst case behavior is the maximal running time on any input of a particular size. We use $T_A(n)$ to denote the worst case running time (or simply running time) of algorithm A on an input of size n, i.e.,

$$T_A(n) = \sup\{T_A(p);\ p \in P \text{ and } g(p) = n\}.$$

Worst case behavior considers algorithms from a pessimistic point of view. For every n we single out the input with maximal running time.

Sometimes, we are given a probability distribution on the set of problem instances. We can then talk about **average case behavior** (or expected behavior); it is defined as the expectation of the running time for problems of a particular size, i.e.,

$$T_A^{av}(n) = E\big(\{T_A(p);\ p \in P \text{ and } g(p) = n\}\big).$$

In this book (Chapters 2 and 3), computing expectations is always reduced to computing finite sums. Of course, average case running time is never larger than worst case running time and sometimes much smaller. However, an average case analysis always poses the following question: does the actual use of the algorithm conform to the probability distribution on which our analysis is based?

We can now formulate one goal of this book. Determine $T_A(n)$ for important algorithms A. More generally, develop methods for determining $T_A(n)$. Unfortunately, this goal is beyond our reach for many algorithms at the moment. We have to confine ourselves to determine upper and lower bounds for $T_A(n)$, i.e., to asymptotic analysis. A typical claim will be: $T(n)$ is bounded above by some quadratic function. We write $T(n) = O(n^2)$ which means that $T(n) \le c \cdot n^2$ for constants $c > 0$ and n_0 and all $n \ge n_0$. Or we claim that $T(n)$ grows at least as fast as $n \log n$. We write $T(n) = \Omega(n \log n)$ which means that there are constants $c > 0$ and n_0 such that $T(n) \ge c \cdot n \log n$ for all $n \ge n_0$ (log denotes log to base two throughout this book). We come back to this notation in Section 1.6.

We can also compare two algorithms A_1 and A_2 for the same problem. We say that A_1 is **faster** than A_2 if $T_{A_1}(n) \le T_{A_2}(n)$ for all n and that A_1 is **asymptotically faster** than A_2 if $\lim_{n\to\infty} T_{A_1}(n)/T_{A_2}(n) = 0$. Of course, if A_1 is asymptotically faster than A_2 then A_2 may still be more efficient than A_1 on instances of small size. This trivial observation is worth being exemplified.

Let us assume that we have 4 algorithms A, B, C, D for solving problem P with running times $T_A(n) = 1000n$, $T_B(n) = 200n \log n$, $T_C(n) = 10n^2$ and $T_D(n) = 2^n$ milliseconds. Then D is fastest for $1 \le n \le 9$, C is fastest for $10 \le n \le 100$ and A is fastest for $n \ge 101$. Algorithm B is never the most efficient. How large is the maximal problem instance which we can solve in one hour of computing time? The answer is 3600 (1600, 600, 21) for algorithm A (B, C, D). If the maximal solvable problem size is too small we can do either one of two things. Buy a larger machine or switch to a more efficient algorithm. Assume first that we buy a machine which is ten times as fast as the present one, or alternatively that we are willing to spend 10 hours of computing time. Then the maximal solvable problem size increases to 36000 (13500, 1900, 25) for algorithms A (B, C, D). We infer from this example that buying a faster machine hardly helps if we use a very inefficient algorithm (algorithm D) and that switching to a faster algorithm has a more drastic effect on the maximally solvable problem size. More generally, we infer from this example that asymptotic analysis is a useful concept and that special considerations are required for small instances (cf. Sections 2.1.5 and 5.4).

So far, we discussed the complexity of algorithms, sometimes, we will also talk about the **complexity of problems**. An upper bound on the complexity of a problem is established by devising and analyzing an algorithm; i.e., a problem P has complexity $O(n^2)$ if there is an algorithm for P whose running time is bounded by a quadratic function. Lower bounds are more difficult to obtain. A problem P has complexity $\Omega(n^2)$ if *every* algorithm for P has running time at least $\Omega(n^2)$. Lower bound proofs require the discussion of an entire class of algorithms and are usually very difficult to obtain. Lower bounds are only available in very rare circumstances (cf. Sections 2.1.6, 2.3, 3.4 and 5.7).

We will next define running time and storage space in precise terms. To do so we have to introduce a machine model. We want this machine model to abstract the most important features of existing computers, so as to make our analysis meaningful for every-day computing, and to make it simple enough, to make analysis possible.

1.1. Machine Models: RAM and RASP

A **random access machine** (**RAM**) consists of 4 registers, the accumulator α and the index registers γ_1, γ_2, γ_3 (the choice of three index registers is arbitrary), and an infinite set of storage locations numbered $0, 1, 2, \ldots$, cf. Figure 1.

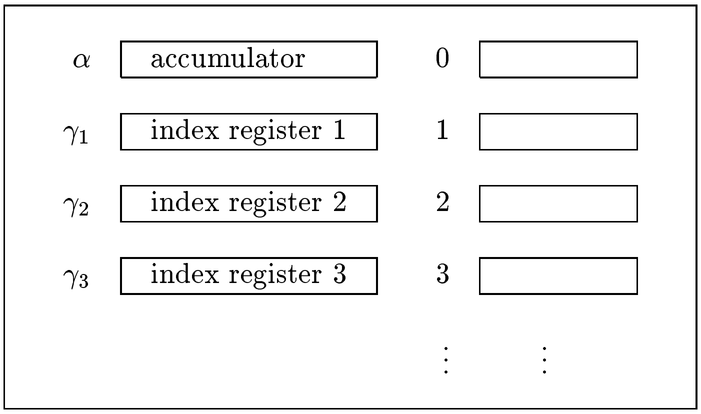

Figure 1. A RAM

The instruction set of a RAM consists of the following list of one address instructions. We use *reg* to denote an arbitrary register α, γ_1, γ_2, γ_3, i to denote a non-negative integer, *op* to denote an operand of the form i, $\rho(i)$ or *reg*, and *mop* to denote a modified operand of the form $\rho(i + \gamma_j)$. In applied position operand i evaluates to number i, $\rho(i)$ evaluates to the content of location i, *reg* evaluates to the content of *reg* and $\rho(i + \gamma_j)$ evaluates to the content of location numbered ($i +$ content of γ_j). Modified operands are the only means of address calculation in RAMs. We discuss other possibilities at the end of this section. The instruction set consists of four groups:

Load and store instructions:

$$\begin{array}{rclll}
reg & \leftarrow & op & \text{, e.g.,} & \gamma_1 \leftarrow \rho(2), \\
\alpha & \leftarrow & mop & \text{, e.g.,} & \alpha \leftarrow \rho(\gamma_2 + 3), \\
op & \leftarrow & reg & \text{, e.g.,} & \rho(3) \leftarrow \alpha, \\
mop & \leftarrow & \alpha & \text{, e.g.,} & \rho(\gamma_2 + 3) \leftarrow \alpha.
\end{array}$$

Jump instructions:

goto k , $k \in \mathbb{N}_0$,
if *reg* π 0 **then goto** k , $k \in \mathbb{N}_0$,
where $\pi \in \{=,\neq,<,\leq,>,\geq\}$ is a comparison operator.

Arithmetic instructions:

$\alpha \leftarrow \alpha\ \tau\ op$, $\alpha \leftarrow \alpha\ \tau\ mop$,
where $\tau \in \{+,-,\times,\text{div},\text{mod}\}$ is an arithmetic operator.

Indexregister instructions:

$\gamma_j \leftarrow \gamma_j \pm i \quad , \quad 1 \le j \le 3 \quad , \quad i \in \mathbb{N}_0.$

A RAM program is a sequence of instructions numbered $0, 1, 2, \ldots$. Integer k in jump instructions refers to this numbering. Flow of control runs through the program according to this numbering except for jump instructions.

Example: Program 1 shows a RAM program for computing 2^n. We assume that n is initially stored in location 0. The output is stored in location 1. The right column shows the number of executions of each instruction on input n. ∎

0:	$\gamma_1 \leftarrow \rho(0)$	1
1:	$\alpha \leftarrow 1$	1
2:	**if** $\gamma_1 = 0$ **then goto** 6	$n+1$
3:	$\alpha \leftarrow \alpha \times 2$	n
4:	$\gamma_1 \leftarrow \gamma_1 - 1$	n
5:	**goto** 2	n
6:	$\rho(1) \leftarrow \alpha$	1

Program 1

In our RAMs there is only one data type: integer. It is straightforward to extend RAMs to other data types such as boolean and reals; but no additional insight is gained by doing so. Registers and locations can store arbitrary integers, an unrealistic assumption. We balance this unrealistic assumption by a careful definition of execution time. Execution time of an instruction consists of two parts: storage access time and execution time of the instruction proper. We distinguish two cost measures: unit cost and logarithmic cost.

In the **unit cost measure** we abstract from the size of the operands and charge one time unit for each storage access and instruction execution. The unit cost measure is reasonable whenever algorithms use only numbers which fit into single locations of real computers. All algorithms in this book (except Chapter 6) are of this kind for practical problem sizes and we will therefore always use the unit cost measure outside Chapter 6. However, the reader should be warned. Whenever he analyzes an algorithm in the unit cost measure, he should give careful thought to the size of the operands involved.

In the **logarithmic cost measure** we explicitely account for the size of the operands and calculate the costs according to their length L. If binary representation is used then

$$L(n) = \begin{cases} 1 & \text{if } n = 0; \\ \lfloor \log n \rfloor + 1 & \text{otherwise.} \end{cases}$$

This explains the name "logarithmic cost measure". The logarithmic cost measure has to be used if the numbers involved do not fit into single storage locations anymore. In the following table we use m to denote the number moved in load and store instructions, and m_1 and m_2 to denote the numbers operated on in an arithmetic instruction. The meaning of all other quantities is obvious from the instruction format.

Costs for Storage Access:

Operand	Unit Cost	Logarithmic Cost
i	0	0
reg	0	0
$\rho(i)$	1	$L(i)$
$\rho(i+\gamma_j)$	1	$L(i)+L(\gamma_j)$

Cost for Executing the Instruction Proper:

	Unit Cost	Logarithmic Cost
Load and Stores	1	$1+L(m)$
Jumps	1	$1+L(k)$
Arithmetic	1	$1+L(m_1)+L(m_2)$
Index	1	$1+L(\gamma_j)+L(i)$

The cost of a conditional jump **if** $reg\ \pi\ 0$ **then goto** k is independent of the content of reg because all comparison operators require only that one checks some few bits of the binary representation of reg.

Under the unit cost measure the cost of an instruction is $1 + \#$ of storage accesses (we use the symbol $\#$ with the meaning of "number"), under the logarithmic cost measure it is $1+$ sum of the lengths of the addresses and numbers involved. Thus the execution time of an instruction is independent of the data in the unit cost measure, but it depends on the data in the logarithmic cost measure.

Example (continued): The instructions of program 1 have the following costs:

Instruction	Unit Cost	Logarithmic Cost
0	2	$L(0)+1+L(\rho(0))$
1	1	$1+L(1)$
2	1	$1+L(6)$
3	1	$1+L(\alpha)+L(2)$
4	1	$1+L(\gamma_1)+L(1)$
5	1	$1+L(2)$
6	2	$L(1)+1+L(\alpha)$

We thus have total cost $4n+6$ under the unit cost measure and $\Theta(n^2)$ under the logarithmic cost measure (compare Section 1.6 for a definition of Θ). Note that the cost of line 3 is $\sum_{i=0}^{n-1}(1+L(2^i)+L(2)) = \Theta(n^2)$. Note also, that one can reduce the cost of computing 2^n to $\Theta(\log n)$ under the unit cost measure and to $\Theta(n)$ under the logarithmic cost measure (cf. Exercise 1). ∎

We infer from this example that the cost of a program can differ drastically under the two measures. It is therefore important to always check whether the unit cost measure can be reasonably used. This will be the case in Chapters 2, 3, 4, 5, 7and 8.

Analogously, we use unit and logarithmic cost measure for storage space also. In the unit cost measure we count the number of storage locations and registers which are used in the computation and forget about the actual contents, in the logarithmic cost measure we sum the lengths of the binary representations of the contents of registers and storage locations and maximize over time.

Example (continued)**:** Program 1 for computing 2^n uses registers α and γ_1 and locations 0 and 1. Hence its space complexity is 4 under the unit cost measure. The content of all 4 cells is bounded by 2^n, two of them actually achieve that value. Hence space complexity is $\Theta(L(2^n)) = \Theta(n)$ under the logarithmic cost measure. ∎

Address modification by index registers is the only means of address modification in RAMs. Most realistic computers allow two other techniques for address modification: general address substitution and direct address calculation. **General address substitution** allows us to use any location as an index register, i.e., modified operands of the form $\rho(i + \rho(j))$ can be used also. The cost of fetching such an operand is 2 in the unit cost and $L(i) + L(j) + L(\rho(j))$ in the logarithmic cost measure. It is not too hard to simulate the enlarged instruction set by our original instruction set with only a constant increase in cost. Let us for example consider the instruction $\alpha \leftarrow \rho(i + \rho(j))$. It is simulated by

$$\begin{aligned} \gamma_1 &\leftarrow \rho(j) \\ \alpha &\leftarrow \rho(i + \gamma_1) \end{aligned}$$

However, the content of γ_1 is destroyed by this piece of code. We therefore have to save the content of γ_1 before executing it. Let us assume that location 0 is not used (Exercise 2 discusses this assumption in detail) in the program which is to be simulated. Then we only have to bracket the above piece of code by $\rho(0) \leftarrow \gamma_1$ and $\gamma_1 \leftarrow \rho(0)$. We obtain

New Instruction	Unit Cost	Logarithmic Cost
$\alpha \leftarrow \rho(i + \rho(j))$	3	$1 + L(i) + L(j) + L(\rho(j)) +$ $L(\rho(i + \rho(j)))$
Simulating Program		
$\rho(0) \leftarrow \gamma_1$ $\gamma_1 \leftarrow \rho(j)$ $\alpha \leftarrow \rho(i + \gamma_1)$ $\gamma_1 \leftarrow \rho(0)$	8	$6 + L(i) + L(j) + 2L(\rho(j)) +$ $L(\rho(i + \rho(j))) + 2L(\gamma_1)$

The cost of the simulating program is only larger by a constant factor in the unit cost measure. We thus have

Lemma 1. *General address substitution reduces the cost of RAM programs by only a constant factor in the unit cost measure.* ▮

The situation is slightly more complicated in the logarithmic cost measure. Factor $L(\gamma_1)$ cannot be estimated in a simple way. We therefore change the simulation method and arrive at a simple connection between the cost of the original program and the cost of the simulating program. We want location 0 to always contain the content of γ_1. We achieve this goal by inserting $\rho(0) \leftarrow \gamma_1$ after every instruction which modifies γ_1 and by inserting $\gamma_1 \leftarrow \rho(0)$ before every instruction which uses γ_1. For example, we replace $\alpha \leftarrow \rho(i+\gamma_1)$ by $\gamma_1 \leftarrow \rho(0)$; $\alpha \leftarrow \rho(i+\gamma_1)$ and $\gamma_1 \leftarrow \rho(i)$ by $\gamma_1 \leftarrow \rho(i)$; $\rho(0) \leftarrow \gamma_1$. This modification increases the cost only by a constant factor under *both* measures. Finally, we replace the instructions using general address substitution as described above, i.e., we replace, e.g., $\alpha \leftarrow \rho(i + \rho(j))$ by $\gamma_1 \leftarrow \rho(j)$; $\alpha \leftarrow \rho(i + \gamma_1)$. Note that we do not have to include this piece of code into brackets $\rho(0) \leftarrow \gamma_1$ and $\gamma_1 \leftarrow \rho(0)$ as before because we took care of saving γ_1 elsewhere. We thus have (details are left to Exercise 2)

Theorem 1. *General address substitution can reduce the time complexity of RAM programs by at most a constant factor in both cost measures.* ▮

Next we discuss **direct address calculation**. We extend the RAM model by a program store **PS** and call the extended model **RASP** (Random Access Stored Program Machine). The program store consists of infinitely many locations numbered $0, 1, 2, \ldots$. Each location has two parts. The first part contains the name of the instruction (the opcode), the second part contains the operand, i.e., either an address or the number of an instruction.

Example (continued)**:** The RASP-version of our example program is shown in Figure 2. We use the RAM-instruction as the opcode, data addresses are replaced by symbol a and instruction addresses are replaced by k. ▮

	Opcode	Address
0	$\gamma_1 \leftarrow \rho(a)$	0
1	$\alpha \leftarrow a$	1
2	**if** $\gamma_1 = 0$ **then goto** k	6
3	$\alpha \leftarrow \alpha \times a$	2
4	$\gamma_1 \leftarrow \gamma_1 - a$	1
5	**goto** k	2
6	$\rho(a) \leftarrow \alpha$	1

Figure 2. A RASP program for computing 2^n

The number of opcodes is finite because there are only four registers and only a finite number of instructions. For the sequel, we assume a fixed bijection between opcodes and some initial segment of the natural numbers. We use *Num* to denote that bijection.

In addition to the RAM instruction set, the RASP instruction set contains so-called π-instructions. π-instructions operate on the program store. They are

$$\left.\begin{aligned} \alpha &\leftarrow \pi_h(i) \\ \alpha &\leftarrow \pi_h(i+\gamma_j) \\ \pi_h(i) &\leftarrow \alpha \\ \pi_h(i+\gamma_j) &\leftarrow \alpha \end{aligned}\right\} \qquad \begin{aligned} &i \in \mathbb{N}_0, \\ &h \in \{1,2\}, \\ &j \in \{1,2,3\}. \end{aligned}$$

Instruction $\alpha \leftarrow \pi_h(i)$ loads the h-th component of location i of PS into the accumulator α. If $h = 1$ then mapping *Num* is applied additionally. The semantics of all other instructions is defined similarly.

Execution times of RASP instructions are defined as in the RAM case except one change. RASP programs can grow during execution and therefore the time required to modify the instruction counter cannot be neglected any longer. We therefore add $L(k)$ to the cost of an instruction stored in cell k in the logarithmic cost measure. We have the following relations in both cost measures.

Theorem 2. *Executing a RAM program of time complexity $T(n)$ on a RASP takes $\leq c \cdot T(n)$ time units, where $c \in \mathbb{R}$ is a constant depending on the RAM program but not on the input.* ∎

Theorem 3. *There is a $c > 0$ such that every RASP program of time complexity $T(n)$ can be simulated in $\leq c \cdot T(n)$ time units on a RAM.* ∎

Theorem 2 follows immediately from the observation, that a RAM program uses only a fixed number of storage locations of the program store and that therefore the additive factor $L(k)$ (k being the content of the program counter) can be bounded by a constant which is independent of the particular input. Thus the "RASP cost" of a RAM instruction is at most c times the "RAM cost" where $c = 1 + L$(length of RAM program to be executed on a RASP).

Theorem 3 is more difficult to prove. One has to write a RAM program which interprets RASP programs. Data store, program store and registers of the RASP are stored in the data store of the RAM, more precisely, we use location 1 for the accumulator, locations 2, 3 and 4 for the index registers, locations 5, 8, 11, 14, ... for the data store, and locations 6, 7, 9, 10, 12, 13, 15, 16, ... for the program store. Two adjacent cells are used to hold the two components of a location of the program store of the RASP. Location 0 is used as an instruction counter; it always contains the number of the RASP instruction to be executed next. The interpreter has the following structure:

(1) loop: load the opcode of the RASP-instruction to be executed into the accumulator;
(2) decode the opcode and transfer control to a modul which simulates the instruction;
(3) simulate the instruction and change the instruction counter.
(4) **goto** loop.

We leave the details to the reader (Exercise 3). According to Theorems 2 and 3, time complexities on RAMs and RASPs differ only by a constant factor. Since we will neglect constant factors anyway in most of what follows, the choice of the machine model is not crucial. We prefer the RAM model because of its simplicity.

So far, RAMs (and RASPs) have no ability to interact with their environment, i.e., there are no I/O-facilities. The details of the I/O-facilities are not important except for Chapter 6 and we therefore always assume that the input (and output) is stored in the memory in some natural way. For Chapter VI on NP-completeness we have to be more careful. We equip our machines with two semi-infinite tapes, a read only input tape and a write only output tape. The input tape contains a sequence of integers. There is one head on the input tape which is positioned initially on the first element of the input sequence. Execution of the instruction $\alpha \leftarrow$ Input transfers the integer under the input head into the accumulator and advances the input head by one position. The cost of instruction $\alpha \leftarrow$ Input is 1 in the unit cost measure and $1 + L(n)$ in the logarithmic cost measure where n is the integer to be read in. Similarly, the statement Output $\leftarrow \alpha$ transfers the content of α onto the output tape. Whenever a RAM attempts to read from the input tape and there is left no element on the input tape, the computation blocks. We will then say that the output is undefined and that the time complexity of that particular computation is the number of time units consumed until blocking occurred.

1.2. Randomized Computations

There are two important extensions of RAMs which we have to discuss: randomized RAMs and nondeterministic RAMs. We discuss randomized RAMs now and put off the discussion of nondeterministic RAMs to Chapter 6.

A randomized RAM (**RRAM**) has the ability to toss a perfect coin and to make further computation dependent on the outcome of the coin toss, i.e., there is an additional instruction

$$\alpha \leftarrow \text{random}$$

which assigns to α either 0 or 1 with probability 1/2 each. The cost of this instruction is 1 in both measures. We illustrate this new concept by a very simple example, an RRAM which computes constant 0.

1: $\alpha \leftarrow$ random
2: **if** $\alpha \neq 0$ **then goto** 1

Apparently, the content of α is 0 when the program stops. However, the running time of the algorithm depends on the outcome of the coin tosses. More precisely, if the random choice comes out 0 at the k-th toss for the first time, $k \geq 1$, then the running time is $2k$ in the unit cost measure. Since the coin is assumed to be fair, the probability of this case is 2^{-k} and therefore the average running time is $\sum_{k\geq 1} 2^{-k} \cdot 2k = 4$ (cf. appendix, formula S1). Note that the average running time is small, although there is a chance that the program never halts.

The notion of RRAM is most easily made precise by reducing it to ordinary RAMs with two input facilities. The first input facility records the actual input p for the randomized computation (as above, we leave the exact nature of that input facility unspecified), the second input facility is a read only input tape which contains a sequence of 0's and 1's. Execution of $\alpha \leftarrow$ random reads the next element (if there is one) from the input tape and transfers it into the accumulator α.

Let A be a RAM program. For s a sequence of 0's and 1's it thus makes sense to talk about $A(p, s)$, the output of A on input p and sequence s of coin tosses, and $T_A(p, s)$, the running time of A on input p and sequence s of coin tosses. Again, we leave it unspecified, whether the output is written into the memory or onto an output tape. The expected running time of randomized algorithm A on input p is then defined by

$$T_A(p) = \lim_{k\to\infty} 2^{-k} \sum_{s\in\{0,1\}^k} T_A(p, s).$$

$T_A(p)$ is well defined because of

Lemma 1. *For all k and p:*

$$2^{-k} \sum_{s\in\{0,1\}^k} T_A(p, s) \quad \leq \quad 2^{-k-1} \sum_{t\in\{0,1\}^{k+1}} T_A(p, t).$$

Proof: Let $s \in \{0,1\}^k$, and let $t = s0$ or $t = s1$. If the computation of A on input p and sequence s of coin tosses stops regularly, i.e., is not blocked because sequence s is exhausted, then $T_A(p, s) = T_A(p, t)$. If it is not blocked but never halts then $T_A(p, s) = \infty = T_A(p, t)$. If it is blocked then $T_A(p, s) \leq T_A(p, t)$. ∎

We can now define $T_A(n)$ and $T_A^{av}(n)$ as described above. Of course $T_A^{av}(n)$ is only defined with respect to a probability distribution on the inputs.

What do we understand by saying that a randomized algorithm A computes a function $f : P \mapsto Y$? The answer to this question is not evident since the output of A can depend on the particular sequence s of coin tosses used in the computation.

Definition: Let $f : P \mapsto Y$ and $\varepsilon : \mathbb{N} \mapsto \mathbb{R}$ be functions. The randomized algorithm A computes f with **error probability** at most ε if for all $p \in P$

$$\lim_{k\to\infty} \frac{|\{s \in \{0,1\}^k;\ f(p) = A(p, s)\}|}{2^k} \geq 1 - \varepsilon(g(p)),$$

where $g(p)$ is the size of input p. ∎

An argument similar to the one used in Lemma 1 shows that the limit in the definition above always exists. Of course, only the case $\varepsilon(n) < 1/2$ is interesting. Then A gives the desired output with probability larger than $1/2$. A randomized algorithm A is called **Las Vegas algorithm** for function f if it computes f with error probability 0 of error, i.e., $\varepsilon(n) = 0$ in the above definition. In particular, whenever $A(p,s)$ stops and is defined then $A(p,s) = f(p)$. Las Vegas algorithms are a particularly suitable class of randomized algorithms because the output is completely reliable. We will see examples of Las Vegas algorithms in Sections 2.1.3 and 3.1.2.

Of course, we want the error probability as small as possible. Suppose, that we have a randomized algorithm A which computes $f : P \mapsto Y$ with error probability at most $\varepsilon(n)$. If $\varepsilon(n)$ is too large we might just run A several times on the same input and then determine the output by a majority vote. This should strengthen our confidence in the output.

Lemma 2. *Let $\delta > 0$. If the randomized algorithm A computes $f : P \mapsto Y$ with error probability at most $\varepsilon(n) = \epsilon < \frac{1}{2}$ in time $T_A(n)$ and if $T_A \circ g$ is computable in time $O(T_A)$ then there is a randomized algorithm B which computes f with error probability at most δ in time $c \cdot m \cdot (\frac{1}{2} - \epsilon)^{-1} \cdot T_A(n)$. Here $m = 2\lceil (\log \delta)/\log(1-(\frac{1}{2} - \epsilon)^2)\rceil$ and c is a constant. Moreover, B always halts within $c \cdot m \cdot (\frac{1}{2} - \epsilon)^{-1} \cdot T_A(n)$ time units.*

Proof: Consider any $p \in P$. Let $n = g(p)$, $T = \lceil (4/(1-2\epsilon)) \cdot T_A(n)\rceil$ and $m = 2\lceil (\log \delta)/\log(1-(\frac{1}{2}-\epsilon)^2)\rceil$. On input p, B computes T, chooses m random sequences $s_1, s_2, \ldots, s_m$ of length T each, and simulates A on inputs $(p, s_1), \ldots, (p, s_m)$ for up to T time units each. It then outputs whatever the majority of the simulated runs of A outputs. Apparently B runs for at most $O((1+m)\cdot T) = O(m \cdot (\frac{1}{2} - \epsilon)^{-1} \cdot T_A(n))$ time units. Moreover, $f(p) \neq B(p, s_1, \ldots, s_m)$ iff $A(p, s_i) \neq f(p)$ for at least $m/2$ distinct i's. Next note that

$$
\begin{aligned}
&\frac{\left|\{s \in \{0,1\}^T;\ f(p) \neq A(p,s)\}\right|}{2^T} \\
&\leq \lim_{k\to\infty} \frac{|\{s \in \{0,1\}^k;\ f(p) \neq A(p,s)\}|}{2^k} + \frac{|\{s \in \{0,1\}^T;\ T_A(p,s) > T\}|}{2^T} \\
&\leq \epsilon + (\tfrac{1}{2} - \epsilon)/2 \\
&= \tfrac{1}{2} - \tfrac{1}{2}(\tfrac{1}{2} - \epsilon),
\end{aligned}
$$

since A computes f with error at most ϵ (and therefore the first term is bounded by ϵ) and since for $p \in P$ with $g(p) = n$ we have

$$\begin{aligned} T_A(n) &\geq T_A(p) \\ &\geq \sum \frac{|\{T_A(p,s);\ s \in \{0,1\}^T\}|}{2^T} \\ &\geq \sum \frac{|\{T_A(p,s);\ s \in \{0,1\}^T \text{ and } T_A(p,s) > T\}|}{2^T} \\ &> T \cdot \frac{\big|\{s \in \{0,1\}^T;\ T_A(p,s) \geq T\}\big|}{2^T}, \end{aligned}$$

and therefore the second term is bounded by $T_A(n)/T \leq (1-2\epsilon)/4$. Let $\gamma = \frac{1}{2} - \frac{1}{2}(\frac{1}{2} - \epsilon)$. Then

$$\begin{aligned} &\frac{\big|\{s_1 \cdots s_m \in \{0,1\}^{T\cdot m};\ B(p,s_1,\ldots,s_m) \neq f(p)\}\big|}{2^{T\cdot m}} \\ &\leq \sum_{i=m/2}^{m} \binom{m}{i} \gamma^i (1-\gamma)^{m-i} \\ &\leq \sum_{i=m/2}^{m} \binom{m}{i} \gamma^{m/2} (1-\gamma)^{m/2} && \text{(since } \gamma < 1/2) \\ &\leq 2^m \gamma^{m/2} (1-\gamma)^{m/2} \\ &= (4\gamma(1-\gamma))^{m/2} \\ &\leq \delta. && \text{(by definition of } m) \quad \blacksquare \end{aligned}$$

It is worth illustrating Lemma 2 by an example. Assume that $\epsilon = 0.49$ and $\delta = 0.03$. Then $m = 2\lceil(\log\delta/)\log(1-(\frac{1}{2}-\epsilon)^2)\rceil = 2\lceil\log 0.03/\log 0.9999\rceil = 70128$. Thus we have to repeat a computation which gives the corrct answer with probability 0.51 about 70000 times in order to raise the level of confidence to 0.97 . If we start with a more reliable machine, say $\epsilon = 0.25$, then m reduces to $2\lceil\log 0.03/\log 0.9375\rceil = 110$. By this example we see that bringing the error down from 0.49 to 0.25 is the difficult part, increasing the level of confidence further is easy.

Randomized algorithms have a fair coin available and deterministic algorithms have not. It is therefore important to know how well a randomized algorithm can be simulated by a deterministic algorithm. We approach this problem from two sides. First we show that for fixed problem size one can always replace the fair coin by a *fixed* sequence of 0's and 1's of reasonable length (Theorem 1), and then we show how to use good pseudo-random number generators in randomized computations.

Theorem 1. *Let $n \in \mathbb{N}$, $N = |\{p \in P;\ g(p) \leq n\}|$, $\delta = 1/(N+1)$ and let A, B, ϵ and f be defined as in Lemma 2. Then there is a sequence $s_0 \in \{0,1\}^{T\cdot m}$,*

m and T as in the proof of Lemma 2 (under the additional assumption that T_A is non-decreasing), such that $f(p) = B(p, s_0)$ for all $p \in P_n$.

Proof: B computes f with error probability at most δ. Let $P_n = \{p \in P;\ g(p) \leq n\}$. Then for all $p \in P_n$

$$\left|\{s \in \{0,1\}^{T\cdot m};\ B(p,s) \neq f(p)\}\right| \quad \leq \quad \delta \cdot 2^{T\cdot m} \quad = \quad 2^{T\cdot m}/(N+1),$$

and therefore

$$\begin{aligned} &\sum_{s\in\{0,1\}^{T\cdot m}} \ \sum_{p\in P_n} \textbf{if } B(p,s) \neq f(p) \textbf{ then } 1 \textbf{ else } 0 \\ &= \sum_{p\in P_n} \ \sum_{s\in\{0,1\}^{T\cdot m}} \textbf{if } B(p,s) \neq f(p) \textbf{ then } 1 \textbf{ else } 0 \\ &\leq N \cdot 2^{T\cdot m}/(N+1) \\ &< 2^{T\cdot m}. \end{aligned}$$

Thus there is at least one $s_0 \in \{0,1\}^{T\cdot m}$ such that

$$\sum_{p\in P_n} \textbf{if } B(p,s_0) \neq f(p) \textbf{ then } 1 \textbf{ else } 0 \quad < \quad 2^{T\cdot m}/2^{T\cdot m} \ = \ 1.$$

Hence $B(p, s_0) = f(p)$ for all $p \in P_n$. ∎

We illustrate Theorem 1 by an example. Assume $P = \{0,1\}^*$ and $g(p) = |p|$, the length of bit string p. Then $|P_n| \leq 2^{n+1}$. Assume also that we start with a randomized machine A with $\epsilon = 1/4$ and running time $T_A(n) = n^k$ for some k. Taking $\delta = 1/(2^{n+1}+1)$, Lemma 2 yields a machine B with worst case running time $T_B(n) = O((-\log\delta)\cdot T_A(n)) = O(n^{k+1})$ and error probability at most δ. Moreover, by Theorem 1, there is a fixed 0-1 sequence s_0 of length $O(n^{k+1})$ which can be used by B instead of a true random number generator. Unfortunately, the proof of Theorem 1 does not suggest an efficient method for finding a suitable s_0.

The question now arises whether we can use a pseudo-random number generator (say built-in procedure *Random* on your favorite computer) to generate coin toss sequences for randomized algorithms. A typical pseudo-random number generator works as follows. It consists of a function $T : \{0,1\}^m \mapsto \{0,1\}^m$ which is designed such that there is no "obvious" connection between argument x and value $T(x)$. The most popular choice of function T is

$$T(x) = (a \cdot x + c) \bmod 2^m$$

where the argument x is interpreted as a number between 0 and $2^m - 1$, and the numbers a and c are of the same range. The result is finally truncated to the last m bits. A user of a pseudo-random number generator provides a "seed"

$x_0 \in \{0,1\}^m$ and uses the transformation T to generate a sequence $x_1, x_2, \ldots, x_k$ with $x_{i+1} = T(x_i)$. Thus a pseudo-random number generator takes a bit sequence x_0 of length m and produces a bit sequence (take the concatenation of $x_1, \ldots, x_k$) of length $k \cdot m$ for some k.

We can therefore define a pseudo-random number generator as a mapping $\rho : \{0,1\}^m \mapsto \{0,1\}^{E(m)}$ where $E(m) \geq m$. It takes a seed $x \in \{0,1\}^m$ and produces a sequence $\rho(x)$ of length $E(m)$.

The choice of the seed is left to the user and we will not discuss it any further. He might use a physical device or actually toss a coin. However, we will discuss the desirable properties of mapping ρ in more detail. The mapping ρ takes a bit string of length m and produces a bit string of length $E(m)$. If $E(m) > m$ then $\rho(x)$ is certainly not a random string (in the sense that all strings of length $E(m)$ are equally likely) even if x is a random string of length m. After all, only 2^m out of the $2^{E(m)}$ possible strings of length $E(m)$ are in the range of ρ.

Suppose now we can generate random strings x of length m, is it then safe to use the pseudo-random strings $\rho(x)$ of length $E(m)$ in a randomized algorithm? At first glance the answer seems "No" because pseudo-random strings are *not* random strings. However, they might be "random enough" to be used anyway instead of a true random sequence. In order to make this precise we need to introduce a measure of quality for pseudo-random number generators. We do so by introducing the concept of a statistical test.

Consider for example the function $h : \{0,1\}^* \mapsto \{0,1\}$ which yields one if the number of zeroes and ones in the argument differs by at most 10%. Then h applied to a random bit string yields one with very high probability and we might require the same for a random element in the range of ρ. If this were the case then the statistical test h cannot distinguish between true random sequences and the sequences obtained by applying ρ to shorter random sequences. If this were true for all statistical tests (a notion which still needs to be defined) then the sequences generated by ρ are rightly called pseudo-random.

In general, we define a statistical test to be any function $h : \{0,1\}^* \mapsto \{0,1\}$ which yields a one for at least half of the arguments of any fixed length.

A randomized algorithm A can easily be turned into a statistical test h_A. The test h_A calls a sequence $s \in \{0,1\}^*$ "good" (i.e., yields a one) if the running time of algorithm A when using sequence s of coin tosses does not exceed its expected running time by more than a factor of, say, two. Then most random sequences of any fixed length are good, i.e., h_A is a statistical test and has polynomial running time if A has (we restrict our attention to polynomial time bounded algorithms because we saw in the beginning of this chapter that algorithms with exponential running time are hopelessly inefficient).

We say that a pseudo-random number generator ρ passes test h if "many" (a precise definition is given below) sequences in the range of ρ are good.

Suppose now that ρ passes *all* statistical tests of polynomial time complexity. Then ρ passes also test h_A if A is a polynomial time bounded algorithm and hence we can hope to use the pseudo-random sequences generated by ρ instead of true random sequences for operating algorithm A.

We will now make these concepts precise. Part c) of the definition below defines in precise terms what we mean by the phrase that the mapping ρ passes the statistical test h. We give two variants of the definition which are geared towards the two applications of pseudo-random number generators to be described later: fast simulation of randomized algorithms by deterministic algorithms and reduction of the number of coin tosses in randomized computations.

Definition:

a) A function $h : \{0,1\}^* \mapsto \{0,1\}$ is **polynomial time computable** if there is a deterministic algorithm computing h whose running time is bounded by a polynomial.

b) A **statistical test** is a function $h : \{0,1\}^* \mapsto \{0,1\}$ with

$$|\{x \in \{0,1\}^k;\ h(x) = 1\}| \ \geq \ 2^{k-1} \quad \text{for all } k.$$

c) Let $E : \mathbb{N} \mapsto \mathbb{N}$ be a function, let $\rho : \{0,1\}^* \mapsto \{0,1\}^*$ be such that $|\rho(x)| = E(m)$ for $|x| = m$ and let $m_0 : \mathbb{N} \mapsto \mathbb{N}$ be a function. Let h be a statistical test and let h be computable in time $t \cdot n^t$ for some t where n is the size of the input. Then ρ **passes** test h if for all $m \geq m_0(t)$

$$\left\{x \in \{0,1\}^{E(m)};\ x \in \text{range}(\rho) \text{ and } h(x) = 1\right\} \ \neq \ \emptyset.$$

Furthermore, ρ **passes** test h **well** if for all $m \geq m_0(t)$

$$\left|\{x \in \{0,1\}^{E(m)};\ x \in \text{range}\,(\rho) \text{ and } h(x) = 1\}\right| \ \geq \ 2^m/8.$$

Remark: If ρ passes test h well then a random element in the range of ρ satisfies h with probability exceeding $1/8$ while a true random element of $\{0,1\}^{E(m)}$ satisfies h with probability exceeding $1/2$. The choice of cut-points $a_1 = 1/2$ and $a_2 = 1/8$ is arbitrary; however $0 < a_2 \leq a_1$ is essential.

d) A mapping ρ is a good (very good) **pseudo-random number generator** if it passes all polynomial time computable statistical tests (well). ∎

The reader should pause at this point and should try to grasp the intuition behind this definition. We defined a statistical test to be any predicate on bit strings which at least half of the strings of any fixed length satisfy (part b)). Furthermore, we restrict our attention to simple (= polynomial time computable) predicates (part a) and c)). A pseudo-random number generator ρ passes all statistical tests if the range of ρ has no simple structure, i.e., if there is no large and computationally simple subset of $\{0,1\}^{E(m)}$, namely a set $\{x \in \{0,1\}^{E(m)};\ h(x) = 1\}$ for some statistical test h, which ρ either misses completely or does not hit with sufficiently high probability. In other words, the properties of a random element in range(ρ) are difficult to predict, and hence the elements produced by ρ are rightly called pseudo-random sequences. Note that "being random" is the same as "being difficult to predict".

It is not known whether (very) good random number generators in the sense of this definition exist. However, it can be shown that very good random number generators with $E(m) = m^k$ for any k computable in polynomial time exist if any one of the following number theoretic problems is hard: the discrete logarithm problem or the problem of factoring integers. We have to refer the reader to the literature for a discussion of these results (cf. A.C. Yao: "Theory and Applications of Trapdoor Functions", IEEE FOCS 1982, 80–91).

We proceed on the *assumption* that a (very) good polynomial time computable pseudo-random number generator ρ exists with, say, $E(m) = m^2$. We show that good pseudo-random number generators can be used to speed up the simulation of randomized algorithms by deterministic algorithms and that very good generators can be used to reduce the required number of true random choices. The latter consequence is important if generation of truely random bits ever became possible, yet would be expensive.

For concreteness and simplicity, let A be a Las Vegas algorithm with polynomial running time, i.e., $T_A(n) \leq t \cdot n^t$ for some $t \in \mathbb{N}$ and let ρ be a good pseudo-random number generator with $E(m) = m^2$. Let $p \in P$, $n = g(p)$, be such that $\sqrt{2t \cdot n^t} \geq m_0(t)$. Then $h_p : \{0,1\}^* \mapsto \{0,1\}$ with

$$h_p(s) = \begin{cases} 1 & \text{if } T_A(p,s) \leq 2t \cdot n^t; \\ 0 & \text{otherwise} \end{cases}$$

is computable in time $O(t \cdot n^t)$ and we have $h_p(s) = 1$ for at least fifty percent of the bit strings of length $2t \cdot n^t$. This follows from the fact that the running time of T_A on p and s can exceed twice the expected value for at most half of the sequences of coin tosses. Hence for all $m \geq \sqrt{2t \cdot n^t}$

$$\{s \in \{0,1\}^{E(m)};\ s \in \text{range}(\rho) \text{ and } h_p(s) = 1\} \ \neq\ \emptyset$$

or

$$\{s \in \{0,1\}^{E(m)};\ s \in \text{range}\,(\rho) \text{ und } T_A(p,s) \leq 2t \cdot n^t\} \ \neq\ \emptyset.$$

This relationship directly leads to a deterministic simulation of probabilistic machines which is more efficient than the naive one. Let $p \in P$, $g(p) = n$ and let $m = \sqrt{2t \cdot n^t}$. Consider Program 2.

Since A is a Las Vegas algorithm and since there is an $s \in \rho(\{0,1\}^m)$ with $T_A(p,s) \leq 2t \cdot n^t$, the simulation always produce the correct answer. Also the running time of the simulation is $O(2^{\sqrt{2t \cdot n^t}}(t \cdot n^t + q(m)))$, where q is the polynomial bound on the time needed for computing ρ. Thus the existence of good pseudo-random number generators leads to more efficient simulations of probabilistic machines by deterministic machines. Note that the naive simulation has running time $O(2^{t \cdot n^t} \cdot t \cdot n^t)$.

```
for all x ∈ {0,1}^m
do  s ← ρ(x);
    run A on p and s for up to 2t · n^t steps;
    if A halts within that number of steps
    then output, whatever A outputs and halt
    fi
od.
```

Program 2

Assume now that ρ is a very good pseudo-random number generator. Then

$$\left|\{s \in \{0,1\}^{E(m)};\ s \in \text{range}\,(\rho) \text{ and } T_A(p,s) \leq 2t \cdot n^t\}\right| \ \geq \ 2^m/8,$$

where m, n and p are defined as above. In other words, a random $x \in \{0,1\}^m$ produces an $s = \rho(x)$ such that $T_A(p,s) \leq 2t \cdot n^t$ with probability at least $1/8$. This observation leads to the following implementation of algorithm A in Program 3 which uses fewer coin tosses than A.

```
repeat generate a random sequence of m bits and call it x;
       s ← ρ(x)
until  A on p and s halts within 2t · n^t steps ;
output, whatever A outputs on p and s.
```

Program 3

Since a random $x \in \{0,1\}^m$ produces an $s = \rho(x) \in \{0,1\}^{E(m)}$ with $T_A(p,s) \leq 2t \cdot n^t$ having probability at least $1/8$ only 8 iterations of the loop are required on the average. Hence the algorithm above simulates A in time $O(T_A(n)) = O(n^t)$ and uses only $O(\sqrt{n^t})$ random bits. Thus if true random choices are possible but costly, this is a very significant improvement. This completes our discussion on the use of (very) good pseudo-random number generators in randomized computations.

We end this section with a remark on the relation between the expected time complexity of deterministic algorithms and the running time of probabilistic algorithms.

Let A be a Las Vegas algorithm which computes function $f : P \mapsto Y$. Let us assume for simplicity that A makes at most $a(n)$ coin tosses for every $n \in \mathbb{N}$ on any input $p \in P$ with $g(p) = n$. Then

$$T_A(p) \ = \sum_{s \in \{0,1\}^{a(n)}} T_A(p,s)/2^{a(n)}.$$

Suppose also that we are given a probability distribution μ on the set P of problem instances. Let B be a deterministic algorithm for f, whose expected running time on inputs of size n is minimal (B is certain to exist if P_n, the set of problem instances of size n is finite), i.e., for all deterministic algorithms C

$$E(\{T_B(p);\ p \in P \text{ and } g(p) = n\}) \ \leq\ E(\{T_C(p);\ p \in P \text{ and } g(p) = n\}),$$

where expectations are calculated with respect to probability distribution μ. For every fixed $s \in \{0,1\}^{a(n)}$ algorithm A with sequence s of coin tosses is a deterministic algorithm and hence

$$\sum_{p \in P_n} \mu(p) \cdot T_B(p) \ \leq\ \sum_{p \in P_n} \mu(p) \cdot T_A(p, s).$$

Since this inequality holds for every s we have

$$\begin{aligned}\sum_{p \in P_n} \mu(p) \cdot T_B(p) \ &\leq\ \sum_{s \in \{0,1\}^{a(n)}} \ \sum_{p \in P_n} \mu(p) \cdot T_A(p, s)/2^{a(n)} \\ &\leq \sum_{p \in P_n} \mu(p) \sum_{s \in \{0,1\}^{a(n)}} T_A(p, s)/2^{a(n)} \\ &\leq \sum_{p \in P_n} \mu(p) \cdot T_A(p).\end{aligned}$$

Thus the expected running time of Las Vegas algorithm A on inputs of size n can not be better than the expected running time of the best deterministic algorithm. We will use this fact to derive lower bounds on the randomized complexity of some sorting problems in Chapter 2. For sorting and some related problems we will derive $\Omega(n \log n)$ lower bounds on the expected running time of a large class of deterministic algorithms in Chapter 2. The inequality derived above immediately extends that lower bound to the corresponding class of Las Vegas algorithms.

1.3. A High Level Programming Language

Only a few algorithms in this book are formulated in RAM code, most algorithms are formulated in a high level ALGOL-like programming language. We feel free to introduce additional statements into the language, whenever the need arises and whenever translation into RAM code is obvious. Also we make frequent use of complex data structures such as lists, stacks, queues, trees and graphs. We choose this very high level description for many algorithms because it allows us to emphasize the principles more clearly. Most statements of our programming language are known from ALGOL-like languages. In particular, we use the conditional statement:

if ⟨`condition`⟩ **then** ⟨`statement`⟩ **else** ⟨`statement`⟩ **fi**;

the iterative statement:

while ⟨`condition`⟩ **do** ⟨`statement`⟩ **od**;

the **for**-loop:

for i **from** ⟨`expression`⟩ **step** ⟨`expression`⟩ **to** ⟨`expression`⟩
  do ⟨`statement`⟩ **od**

If the step size is unspecified then it is 1 by default. We also use a second form of the **for**-loop:

for $i \in$ ⟨`set`⟩ **do** ⟨`statement`⟩ **od**

with the following semantics. The statement is executed |⟨`set`⟩|-times; i runs through the members of the set in some *unspecified* order. Assignments are written in the form ⟨`variable`⟩ $\leftarrow$ ⟨`expression`⟩. Translation of all statements above into RAM code is simple. We describe the translation in the case of a while-statement

while B **do** S **od**.

Let P_1 be a RAM program for B, i.e., P_1 evaluates expression B and leaves a 0 (1) in the accumulator, if B evaluates to false (true). Let P_2 be a RAM program for S. Then the RAM code of Program 4 realizes the while-loop.

  P_1
  if $\alpha = 0$ **then goto** exit;
  P_2
  goto first instruction of P_1;
exit:

Program 4

Program 4 also defines the complexity of the while-loop; it is the sum of the time units spent on the repeated testing of the condition and the execution of the body.

Variables in our programs contain unstructered elementary values or structured values. The **elementary data types** are **integer**, **real**, **boolean**, **pointer** and an additional unspecified data type. We use this additional data type in sorting algorithms; the data to be sorted will be of the unspecified type. The operations defined on this additional data type are given on a case by case basis. **Structured data types** are strings over some alphabet, records, arrays, lists, stacks, queues, trees and graphs. Strings are treated in 2.2, graphs in 4.1, all other structured types are treated in 1.4. The type of a variable will be obvious from the context in most cases; in this case we will not declare variables explicitely.

Procedures play a major role in our programs. Parameters are restricted to elementary types. Then parameter passing takes constant time (cf. Section I.5). Non-recursive procedures are easily reduced to RAM code; one only has to substitute the procedure body at the place of the call. The situation is more complicated for recursive procedures. We treat recursive procedures in 1.5.

Comments are bracketed by **co** and **oc**.

1.4. Structured Data Types

Records and arrays are the most simple ways of structuring data.

An **array** A of n elements consists of n variables $A[1], \ldots, A[n]$ of the same type. An array is stored in n consecutive storage locations, e.g., in locations $BA+1$, $BA+2, \ldots, BA+n$. Here BA stands for base address. If x is a variable stored in location i then accessing array element $A[x]$ is realized by means of index registers. The following piece of code

$$\gamma_1 \leftarrow \rho(i);$$
$$\alpha \leftarrow \rho(BA + \gamma_1)$$

loads $A[x]$ into the accumulator for a cost of 4 in the unit cost measure and $2+L(i)+L(BA) + 2L(x) + L(A[x])$ in the logarithmic cost measure. Again the logarithmic cost measure is proportional to the length of the numbers involved.

Records are fixed size collections of variables of different type, e.g.,

record age : integer; income : real **end**.

A variable x of this record type is easily simulated by two simple variables, a variable x.age of type integer and a variable x.income of type real.

Queues, stacks, lists and trees are treated in the sections below. They are all reduced to arrays.

1.4.1. Queues and Stacks

Queues and stacks are used to represent sequences of elements which can be modified by insertions and deletions. In the case of queues insertions are restricted to the end of the sequence and deletions are restricted to the front of the sequence. A typical example is a waiting line in a student cafeteria. Queues are also known under the name FIFO store (first in - first out). In the case of stacks, insertions and deletions are restricted to the end of the sequence: LIFO store (last in - first out). Very often, the names *Push* and *Pop* are used instead of insertion into and deletion from a stack.

A stack K is most easily realized by an infinite array $K[1]$, $K[2]$, ... and an index *Top* of type integer. The stack consists of elements $K[1], \ldots, K[Top]$; $K[Top]$ is the top element of the stack. The following piece of code realizes operation $Push(K, a)$

$Top \leftarrow Top + 1;$
$K[Top] \leftarrow a.$

The next piece of code deletes an element from the stack and assigns it to variable x, i.e., it realizes $x \leftarrow Pop(K)$

if $Top = 0$ **then** error **fi**;
$x \leftarrow K[Top];$
$Top \leftarrow Top - 1.$

Of course, infinite arrays are rarely available. Instead we have to use a finite array of, say, n elements. In this case a push-operation should also check whether overflow occurs. In either case the stack operations *Push* and *Pop* take constant time in the unit cost measure.

A queue S is also realized by an array. We immediately treat the case of a finite array $S[1..n]$. We conceptually think of array S as a closed line, i.e., $S[1]$ follows $S[n]$, and use two indices *Front* and *End* to denote the borders of the queue. More precisely, if $Front < End$ then the queue consists of $S[Front], \ldots, S[End-1]$, if $Front > End$ then the queue consists of $S[Front], \ldots, S[N], S[1], \ldots, S[End-1]$, and if $Front = End$ then the queue is empty. Then deleting an element from S and assigning it to x is realized by

if $Front = End$ **then** error **fi**;
$x \leftarrow S[Front];$
$Front \leftarrow 1 + (Front \bmod n)$

and inserting an element a into the queue is realized by

$S[End] \leftarrow a;$
$End \leftarrow 1 + (End \bmod n);$
if $Front = End$ **then** error **fi**.

Insertions into and deletions from queues take constant time in the unit cost measure.

1.4.2. Lists

Linear lists are used to represent sequences which can be modified anywhere. In linear lists the elements of a sequence are not stored in consecutive storage locations, rather each element explicitly points to its successor (cf. Fig. 3).

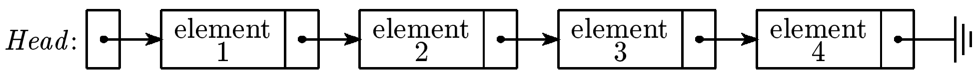

Figure 3. A linear list

There are many versions of linear lists: singly linked, doubly linked, circular, etc. We discuss singly linked lists here and leave the others for the exercises. In singly linked linear lists each element points to its successor. There are two realizations of linear lists: one by records and one by arrays. We discuss both, although internally the record representation boils down to the array representation. We use both representations throughout the book and always choose the one which is more convenient.

In the record representation an element of a linear list is a record of

$$\textbf{type}\ \text{element} = \textbf{record}\ \text{cont} : \text{real};\ \text{next} : \uparrow\!\text{element}\ \textbf{end}$$

and *Head* is a variable of type ↑element. *Head* always points to the first element of the list. The pictorial representation is as given in Figure 3.

The realization by two arrays is closer to RAM code. Real array $Content[1\,..\,n]$ contains the contents of the elements and integer array $next[1\,..\,n]$ contains the pointers. *Head* is an integer variable. Our example list can be stored as shown in Figure 4.

$Head = 2$

	Content	*next*
1	element 4	0
2	element 1	4
3	element 3	1
4	element 2	3

Figure 4. Realization by arrays

Here $Head = 2$ means that row 2 of the array contains the first element of the list, the second element is stored in row 4, etc. The last element of the list is stored in row 1; $next[1] = 0$ indicates that this element has no successor.

We describe now insertion into, deletion from and creation of linear lists. We give two versions of each program, one using records and one using arrays. In either case we assume that there is a supply of unused elements and that a call of procedure $Newr(\textbf{var}\ p :\uparrow\!\text{element})$ resp. $Newa(\textbf{var}\ p : \text{integer})$ takes a node from the supply and makes p point to it in the record (array) version and that a call of procedure $Disposer(\textbf{var}\ p :\uparrow\!\text{element})$ resp. $Disposea(\textbf{var}\ p : \text{integer}))$ takes the node pointed to by p and returns it to the supply. Suffixes r and a distinguish between the representations. We discuss later how the supply of elements is implemented. In our example a call $Newa(p)$ might assign 5 to p because the fifth row is unused and a call $Newr(p)$ results in the situation depicted in Figure 5.

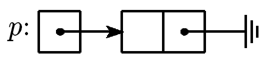

Figure 5. After call of *Newr*

Procedure *Create* takes one parameter and makes it the head of an empty list. Again we use suffixes r and a to distinguish the record and the array version.

```
procedure Creater(var Head : ↑element);     procedure Createa(var Head : integer);
Head ← nil                                  Head ← 0
end.                                        end.
```

Procedure *Insert* takes two parameters, a pointer to the element after which we want to insert and the content of the new element.

```
procedure Insertr(p:↑element, a:real);      procedure Inserta(p:integer, a:real);
var q : ↑element;                           var q : integer;
Newr(q);                                    Newa(q);
q↑.cont ← a;                                Content[q] ← a;
q↑.next ← p↑.next;                          next[q] ← next[p];
p↑.next ← q                                 next[p] ← q
end.                                        end.
```

This procedure fails for empty lists. (The following procedure *Delete* fails already for lists consisting of only one element.) A real implementation must take this into account and add tests for special cases. In general, the descriptions of our algorithms skip sometimes such special cases to enhance the readability of the book. Figure 6 illustrates one call of *Insertr*(p, a).

Procedure *Delete* takes one parameter, a pointer to the element which precedes the element we want to delete.

```
procedure Deleter(p :↑element);             procedure Deletea(p : integer);
var q :↑element;                            var q : integer;
q ← p↑.next;                                q ← next[p];
p↑.next ← q↑.next;                          next[p] ← next[q];
Disposer(q)                                 Disposea(q)
end.                                        end.
```

Finally, we have a function to test whether a list is empty.

```
function Emptyr(Head : ↑element);           function Emptya(Head : integer);
Emptyr ← (Head = nil)                       Emptya ← (Head = 0)
end.                                        end.
```

It remains to discuss the supply of unused nodes in more detail. Supply is again a linear list with head *Free*. We will only describe the array version of procedures *New* and *Dispose* because these procedures are usually built-in functions in programming languages which contain records. Internally, records are always realized by arrays and therefore *Newr* and *Disposer* are identical to *Newa* and *Disposea*. A supply of n elements is created by

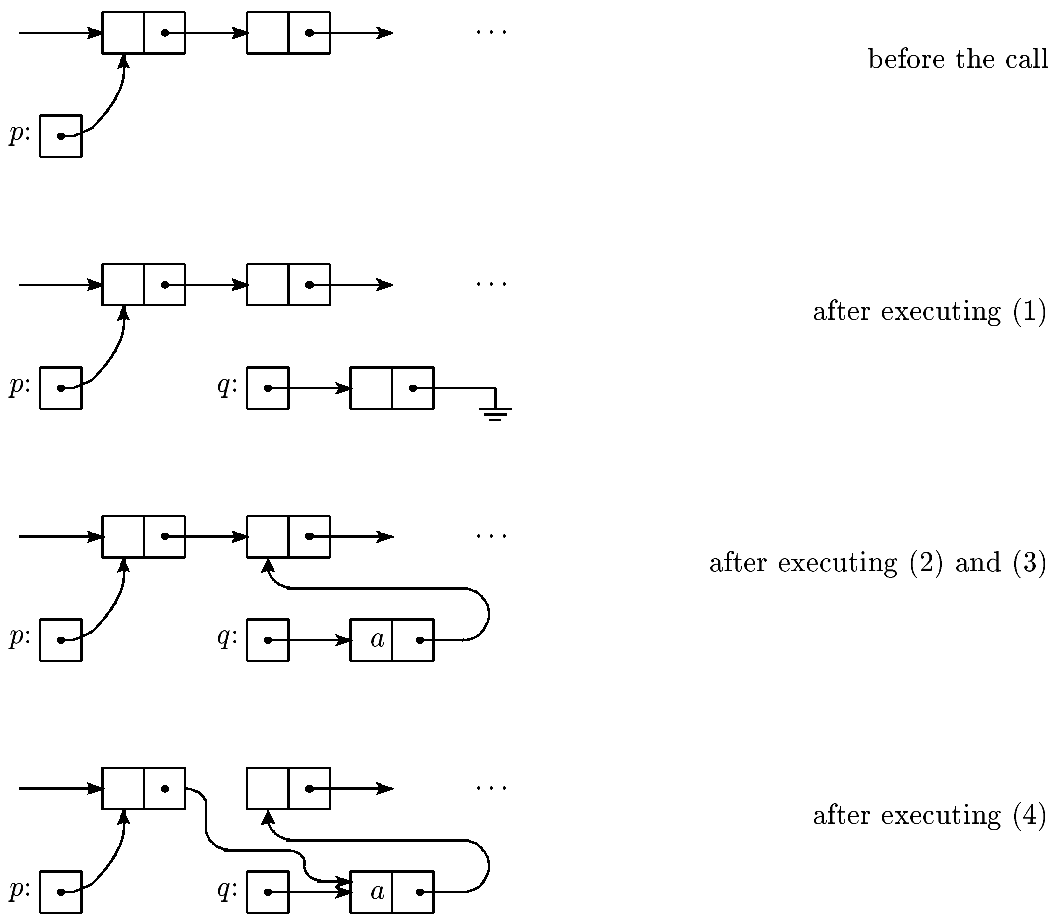

Figure 6. Snapshots during execution of *Insertr*(p, a)

procedure *Inita*$(n$: integer);
var i : integer;
Free $\leftarrow 1$;
for i **from** 1 **to** $n-1$
do $next[i] \leftarrow i+1$ **od**;
$next[n] \leftarrow 0$
end.

Newa and *Disposea* are realized by

> **procedure** *Newa*(q : integer);
> $q \leftarrow$ *Free*;
> **if** *Free* $= 0$ **then** supply exhausted **fi**;
> *Free* $\leftarrow$ *next*[*Free*];
> *next*[q] $\leftarrow 0$
> **end**.

and

> **procedure** *Disposea*(**var** q : integer);
> *next*[q] $\leftarrow$ *Free*;
> *Free* $\leftarrow q$;
> $q \leftarrow 0$
> **end**.

We summarize in

Theorem 1. *Creating a supply of n nodes takes time $O(n)$ in the unit cost measure, creation of an empty list, insertion into, deletion from a linear list given that the positions of insertion or deletion is known and testing for emptiness take time $O(1)$ in the unit cost measure.* ∎

One often uses linear lists to realize stacks and queues. In particular, several stacks and queues may share the same supply of unused nodes. This will guarantee high storage utilization if we have knowlegde of the total length of all stacks and queues but no knowlegde of individual length. Typical examples can be found in Chapter 6. We store a graph by listing the set of its successors for each node. In a graph of n nodes and m edges these lists have total length m, but nothing is known in advance about the length of individual lists.

1.4.3. Trees

Trees consist of nodes (branching points) and leaves. Let $V = \{v_1, v_2, \ldots\}$ be an infinite set of nodes and let $B = \{b_1, b_2, b_3, \ldots\}$ be an infinite set of leaves. We define the set of trees over V and B inductively.

Definition:

a) Each element $b_i \in B$ is a **tree**. Then b_i is also the **root** of the tree.

b) If $T_1, \ldots, T_m$ $(m \geq 1)$ are trees with pairwise disjoint sets of nodes and leaves and $v \in V$ is a new node then the $(m+1)$-tuple $T = \langle v, T_1, \ldots, T_m \rangle$ is a **tree**. Node v is the **root** of the tree, $\rho(v) = m$ is its **degree** and T_i is the i-th **subtree** of T. ∎

In the graph-theoretic literature trees as defined above are usually called **ordered rooted trees**. We always draw trees with the root at the top and the leaves at the bottom. As shown in the example tree of Figure 7 nodes are drawn as circles and leaves are drawn as rectangles.

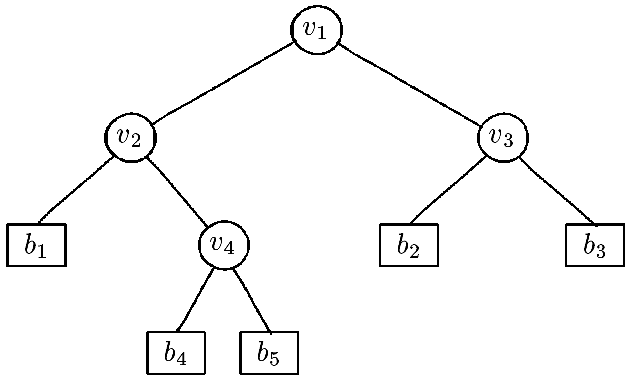

Figure 7. Tree T_{Ex} with nodes and leaves

We use the following terms when we talk about trees. Let T be a tree with root v and subtrees T_i, $1 \le i \le m$. Let $w_i = \text{root}(T_i)$. Then w_i is the i-th **son** of v and v is the **father** of w_i. **Descendant** (**ancestor**) denotes the reflexive, transitive closure of relation son (father). w_j is **brother** of w_i, $j \neq i$. In the tree of Figure 7 b_1 and v_4 are brothers, v_1 is father of v_3 and b_5 is descendant of v_2.

Definition (depth)**:** Let v be a node or leaf of tree T. If v is the root of T then $\text{depth}(v, T) = 0$. If v is not the root of T then v belongs to T_i for some i. Then $\text{depth}(v, T) = 1 + \text{depth}(v, T_i)$. We mostly drop the second argument of depth if it is clear from the context. ∎

Definition (height of a tree)**:** Let T be a tree. Then

$$\text{height}(T) = \max\{\text{depth}(b, T);\ b \text{ is leaf of } T\}.$$ ∎

In Figure 7 we have $\text{depth}(v_3) = 1$, $\text{depth}(v_4) = 2$ and $\text{height}(T_{Ex}) = 3$.

Definition: Tree T is a **binary tree** if all nodes of T have degree exactly 2. ∎

Our example tree T_{Ex} is a binary tree. A binary tree with n nodes has $n+1$ leaves. The 1st (2nd) subtree is also called left (right) subtree.

Information can be stored in the leaves and nodes of a tree. In some applications we use only one possibility. A binary tree is realized by three arrays *Lson*, *Rson* and *Content* or equivalently by records with three fields. Figure 8 gives the storage representation of our example tree T_{Ex}.

We have associated rows with nodes and leaves in some arbitrary way. If information is only stored in the nodes and not in the leaves then leaves do not have to be stored explicitly. All rows corresponding to leaves can be deleted and pointers pointing to leaves are set to 0. A 0-pointer then represents a subtree root = 5

	Content	*Lson*	*Rson*
1	Content of v_4	8	4
2	Content of v_2	3	1
3	Content of b_1	0	0
4	Content of b_5	0	0
5	Content of v_1	2	7
6	Content of b_2	0	0
7	Content of v_3	6	9
8	Content of b_4	0	0
9	Content of b_3	0	0

Figure 8. Realization of T_{Ex} by arrays

consisting of a single leaf. In the diagrams we will not draw leaves in this case (cf. Figures 9 and 10).

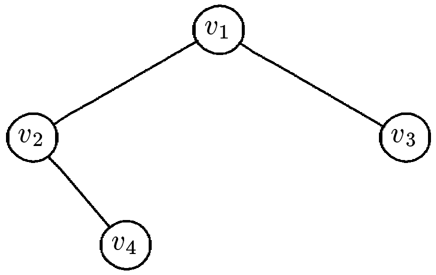

Figure 9. T_{Ex} without leaves

root = 3

	Content	*Lson*	*Rson*
1	Content of v_2	0	4
2	Content of v_3	0	0
3	Content of v_1	1	2
4	Content of v_4	0	0

Figure 10. Realization of T_{Ex} without leaves by arrays

Systematic exploration of a tree is needed frequently. A binary tree consists of three components: a root, a left subtree and a right subtree. Thus three methods of tree traversal come to mind naturally:

Preorder traversal: visit the root, traverse the left subtree, traverse the right subtree: root, L, R.

Postorder traversal: traverse the left subtree, traverse the right subtree, visit the root: L, R, root.

Symmetric traversal: traverse the left subtree, visit the root, traverse the right subtree: L, root, R.

Symmetrical variants are obtained by interchanging L and R. Procedure *Symord* of Program 5 traverses a tree in symmetrical order and prints the content of all nodes and leaves.

```
procedure Symord(v);
if v is leaf
then print(Content[v])
else  Symord(Lson[v]);
      print(Content[v]);
      Symord(Rson[v])
fi
end.
```

Program 5

1.5. Recursion

Recursive procedure *Symord* traverses a binary tree in symmetrical order. Before we can estimate time and space complexity of *Symord* we need to take a look at the implementation of recursive procedures in RAM or RASP code. Recursive procedures are realized by means of a stack. We associate with each call (incarnation, activation) of a procedure an element of the stack, called activation record. The activation record contains complete information about the call, i.e.,

a) the values of the actual parameters,
b) the return address,
c) the local variables.

If a procedure has n parameters and m local variables then the activation record consists of $n+1+m$ storage locations. The i-th cell, $1 \le i \le n$, contains the value of the i-th actual parameter, the $(n+1)$-st cell contains the return address, i.e., the address of the instruction which follows the procedure call in the calling program, and the $(n+1+j)$-th cell contains the j-th local variable. Parameters and local variables are addressed indirectly via the activation record. More precisely, if *Top* is the address of the storage location immediately preceding the activation record then location $Top+i$ contains the i-th actual parameter and cell $Top+n+1+j$ contains the j-th local variable. *Top* is best stored in an index register. After completion of the procedure call control is transfered to the address stored in location $Top+n+1$. Also *Top* is reduced after completion, i.e., the activation record is deleted from the stack. Parameters and return address are computed by the calling program, i.e., the first $n+1$ cells of an activation record are initialized by the calling program. We are now ready to give the non-recursive version of *Symord* by Program 6. Array $K[1\,..\,\infty]$ is used as the stack.

```
(1')             begin co the main program calls Symord(root) oc
(2')             Top ← 0;
(3')             K[1] ← root;
(4')             K[2] ← "HP";
(5')             goto Symord;
(6')       HP :  Halt;
(7')   Symord:   co here comes the code for Symord;
                    node v is stored in K[Top + 1],
                    and return address is stored in K[Top + 2]   oc
(8')             if Lson[K[Top + 1]] = Rson[K[Top + 1]] = 0
                 then co K[Top + 1] is a leaf oc
(9')                   print(Content[K[Top+1]]);
(10')                  goto Finish
                 else  co call Symord(Lson[v]) oc
(11')                  Top ← Top + 2;
(12')                  K[Top + 1] ← Lson[K[Top − 1]];
(13')                  K[Top + 2] ← "M1";
(14')                  goto Symord;
(15')      M1 :        Top ← Top − 2;
(16')                  print(Content[K[Top + 1]]);
                       co call Symord(Rson[v]) oc
(17')                  Top ← Top + 2;
(18')                  K[Top + 1] ← Rson[K[Top − 1]];
(19')                  K[Top + 2] ← "M2";
(20')                  goto Symord;
(21')      M2 :        Top ← Top − 2;
(22')                  goto Finish
                 fi;
(23')  Finish :  goto K[Top + 2]
                 end.
```

Program 6

Call *Symord*(*Lson*[v]) (line (3) of the recursive program) is simulated by lines (11')–(15') of the non-recursive program. In (11') storage space for the activation record is reserved. The activation record is initialized at lines (12') and (13'); at (12') the value of the actual parameter and at (13') the return address "M1" is stored. At line (14') control is transfered to *Symord* and the recursive call is started. Upon completion of the call control is transfered to label M1 (line (15')). The space for the activation record is released and execution of *Symord* is resumed. Analogously, one can associate the other instructions of the non-recursive program with the instructions of the recursive program.

Program 6 is practically a RASP program. Line (23') uses a π-instruction. It can be turned into a RAM program either by the method described in the proof of

Theorem 3 of Section 1 or more simply by replacing line (23') by Program 7.

```
if K[Top + 2] = "M1"
then goto M1;
if K[Top + 2] = "M2"
then goto M2;
goto HP
```

Program 7

We have described a simplified method for translating recursive programs into non-recursive ones. The method suffices if global variables can only come out of the main program. This will be the case throughout this book. In the general case a single register *Top* does not suffice; one has to replace it by a set of registers, one for each activation record which can contain global variables. We refer the reader to a book on compiler construction for details, e.g., Gries (71)).

We are now ready to analyze the time complexity of recursive procedures in the unit cost measure. A constant number of instructions is required to set up a recursive call, namely to increase *Top*, to store the n actual parameters and the return address. Note that n is fixed and in dependent of the input; n can be infered from the program text. Also, we allow only elementary types in parameter position and hence a single parameter can be passed in constant time. Upon return from a recursive procedure *Top* has to be decreased. Altogether, the administrative overhead for a procedure call is bounded by a constant.

The following method for computing the time complexity of recursive procedures is often useful. We associate with every procedure call the cost of executing the procedure body, including the administrative cost for initiating further recursive calls but *excluding* the time spent inside recursive calls. Then we sum over all calls and obtain the total cost in this way. We illustrate this method on procedure *Symord*. Each line in the body of *Symord* takes constant time. Recall that we only count the time required to set up the recursive calls at lines (3) and (5) of Program 5 but that we do not count the time needed to execute the recursive calls. *Symord* is called once for each leaf and node of a binary tree. Thus the total cost of *Symord* is $O(n)$ where n is the number of leaves and nodes of the binary tree.

The summation over all calls of a procedure can be done more formally by means of recursion equations. Let $T(v)$ be the running time of call $Symord(v)$. Then

$$T(v) = \begin{cases} c_1 & \text{if } v \text{ is a leaf;} \\ c_2 + T(Lson[v]) + T(Rson[v]) & \text{otherwise} \end{cases}$$

for suitable constants c_1 and c_2. We can now show by induction on the height of v (the tree with root v) that $T(v) = c_2 \cdot \#$ nodes in the tree with root $v + c_1 \cdot \#$ leaves in the tree with root v. Although the induction proof is very simple we include it for didactic purposes.

Height$(v) = 0$: Then v is a leaf and we have $T(v) = c_1$.
Height$(v) > 0$: Then v is a node and we have

$$\begin{aligned} T(v) &= c_2 + T(Lson[v]) + T(Rson[v]) \\ &= c_2 + c_2 \cdot \#\ \text{nodes}(Lson[v]) + c_1 \cdot \#\ \text{leaves}(Lson[v]) \\ &\qquad + c_2 \cdot \#\ \text{nodes}(Rson[v]) + c_1 \cdot \#\ \text{leaves}(Rson[v]) \\ &= c_2 \cdot \#\ \text{nodes}(v) + c_1 \cdot \#\ \text{leaves}(v). \end{aligned}$$

A more detailed discussion of recursion equations can be found in Section 2.1.3.

1.6. Order of Growth

In most cases we will not be able to derive exact expressions for the running time of algorithms; rather we have to be content with order of growth analysis. We use the following notation.

Definition: Let $f : \mathbb{N}_0 \mapsto \mathbb{N}_0$ be a function. Then $O(f)$, $\Omega(f)$ and $\Theta(f)$ denote the following sets of functions

$$\begin{aligned} O(f) &= \{g : \mathbb{N}_0 \to \mathbb{N}_0;\ \exists c > 0\ \exists n_0 : g(n) \le c \cdot f(n) \\ &\qquad\qquad \text{for all } n \ge n_0\} \\ \Omega(f) &= \{g : \mathbb{N}_0 \to \mathbb{N}_0;\ \exists c > 0\ \exists n_0 : g(n) \ge c \cdot f(n) \\ &\qquad\qquad \text{for all } n \ge n_0\} \\ \Theta(f) &= \{g : \mathbb{N}_0 \to \mathbb{N}_0;\ \exists c > 0\ \exists n_0 : (1/c) \cdot f(n) \le g(n) \le c \cdot f(n) \\ &\qquad\qquad \text{for all } n \ge n_0\} \end{aligned}$$ ∎

We use the O-notation in proofs of upper bounds, the Ω-notation in proofs of lower bounds and the Θ-notation whenever we can determine the order of growth exactly.

It is customary to use the notations above together with the equality sign instead of the symbols $\in$, $\subseteq$ for the relations 'element of' and 'subset of', i.e., we write $n^2 + 5n = n^2 + O(n) = O(n^2)$ instead of $n^2 + 5n \in n^2 + O(n) \subseteq O(n^2)$. More precisely, if $\circ$ is an n-ary operation on functions and $A_1, \ldots, A_n$ are sets of functions, then $\circ(A_1, \ldots, A_n)$ denotes the natural extension to sets of functions, e.g., $A_1 + A_2 = \{a_1 + a_2;\ a_1 \in A_1 \text{ and } a_2 \in A_2\}$. Singleton sets are denoted by their single member. Then expressions α and β which contain O-expressions denote sets of functions and $\alpha = \beta$ stands for $\alpha \subseteq \beta$. Equalities containing O-expressions can *only be read from left to right*. The terms of sequence $A_1 = A_2 = A_3 = \cdots = A_k$ represent larger and larger sets of functions; the bounds become coarser and coarser from left to right.

1.7. Secondary Storage

On a few occasions we consider algorithms which use secondary storage. We make the following assumption. Data is transported in blocks (pages) between main and secondary memory. A page consists of a fixed (say $2^{10} = 1024$) number of storage locations. It takes 5000 time units to transport a page from one memory to the other. This assumption approximates the behavior of modern disk memory.

1.8. Exercises

1) Develop a RAM program for computing 2^n which runs in $O(\log n)$ time units in the unit cost measure and $O(n)$ time units in the logarithm-cost measure. [Hint: Let $n = \sum_{i=0}^{k} a_i \cdot 2^i$, $a_i \in \{0,1\}$, be the binary representation of n. Note that $2^n = (\cdots((2^{a_k})^2 \cdot 2^{a_{k-1}})^2 \cdots 2^{a_1})^2 \cdot 2^{a_0}$.]

2) Give a detailed proof of Theorem 1 of Section 1.1. In particular, discuss the assumption that a RAM program does not use cell 0. Show that one can add 1 to all addresses dynamically and hence free cell 0.

3) Give a detailed proof of Theorem 3 of Section 1.1.

4) Show how to translate conditional and iterative statements into RAM code.

5) In doubly linked lists every list element points to its successor and its predecessor. Do Section 1.4.2. for doubly linked lists.

6) A tree is k-ary if every node has exactly k sons. Show that a k-ary tree with n nodes has exactly $k + (k-1)(n-1)$ leaves.

7) Prove

$$f(n) = O(f(n))$$
$$O(f(n)) + O(f(n)) = O(f(n))$$
$$c \cdot O(f(n)) = O(f(n))$$
$$O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n))$$

1.9. Bibliographic Notes

RAMs and RASPs were introduced by Shepherdson/Sturgis (63) and Elgot/Robinson (64). Our notation follows Hotz (72). The theorems in Section 1.1. are taken over from Cook/Reckhow (73). The discussion of randomized algorithms is based on Adleman (78), Reif (82), Yao (77) und Yao (82). A more detailed account of linear lists can be found in Maurer (74) and Knuth (68). For a detailed discussion of the implementation of recursive procedures we recommend Gries (71).