3 Elements of Classical Computer Science

Computer science is a vast field, ranging from the very abstract and fundamental to the very applied and down-to-earth. It is impossible to summarize the status of the field for an audience of outsiders (such as physicists) on a few pages. The present chapter is intended to serve as an introduction to the most basic notions necessary to discuss logical operations, circuits, and algorithms. We will first introduce logic gates of two types: irreversible and reversible. Later we will discuss the Turing machine as a universal computer and the concept of complexity classes. All this will be done in an informal and highly non-rigorous style intended to provide our physicist readership with some rough idea about the subject.

3.1 Computer pioneers

3.1.1 19th century

Figure 3.1: Charles Babbage (1791–1871) and a difference engine built according to his design.

The inventor of the first programmable computer is probably Charles Babbage (1791–1871) (→ Fig. 3.1). He was interested in the automatic computation of mathematical tables and designed the mechanical "analytical engine" in the 1830s. The engine was to be controlled and programmed by punchcards, a technique already known from the automatic Jacquard loom, but was never actually built. Babbage's unpublished notebooks were discovered in 1937 and the 31-digit accuracy "Difference Engine No. 2" was built to Babbage's specifications in 1991, as shown in the right-hand side of Figure 3.1. Babbage was also Lucasian professor of mathematics in Cambridge, like Newton, Stokes, Dirac, and Hawking, and he invented important practical devices such as the locomotive cowcatcher.

The first computer programmer was probably Ada Augusta King, countess of Lovelace (1815–1852), daughter of the famous poet Lord Byron. She devised a program to compute Bernoulli numbers (recursively) with Babbage's engine. From this example we learn that the practice of devising algorithms for not-yet existing computers is considerably older than the quantum age.

Another important figure from 19th century Britain is George Boole (1815–1864) who in 1847 published his ideas for formalizing logical operations by using operations like AND, OR, and NOT on binary numbers. This will be discussed in detail in section 3.2.

3.1.2 20th century

Alan Turing (1912–1954) was a mathematician who became famous as a code breaker during WW II. As part of this activity, he built and used a computer. He also made many contributions to the foundations of computer science; this includes the design of a universal computer, now known as the Turing machine (→ section 3.3.1), the Church-Turing thesis and the Turing

test. He also was interested in the decidability problem posed by David Hilbert: Is it always possible to decide whether a given mathematical statement is true or not? His computer (Turing machine) helped to show that this is not possible.

Claude Elwood Shannon (1916 – 2001) was an American mathematician, electrical engineer, computer scientist, cryptographer and inventor known as the "father of information theory", with contributions to information theory as well as to the development of hardware for computers. In his master's thesis (MIT 1937), he discussed the implementation of Boolean logic by electric circuits and showed that electrical circuits could construct any logical numerical relationship.

This was the basis for the construction of programmable computers. The first universal (Turing-complete) computer was the Zuse Z3, which became operational in May 1941. The significantly more advanced ENIAC system became operational in 1945. It was based on electronic circuits like vacuum tubes, rather than mechanical components and therefore was approximately 1000 times faster than electromechanical machines. At the time of its inauguration, however, the required software was not yet available; developing it took significantly more time. Programming the device was done by six women and relied, to a significant part, on hardwired panels, rather than actual software.

3.2 Boolean algebra and logic gates

3.2.1 Bits and gates

Classical digital computers are based on Boolean logic. In this context, the "atoms" of information are the binary digits, or *bits*. Every bit can assume one of two values, which are usually labeled 0 and 1 or true and false. In the computing hardware, bits are represented by easily distinguishable physical states, such as high or low voltage or the presence or absence of a charge or current, or the direction of a magnetization. Figure

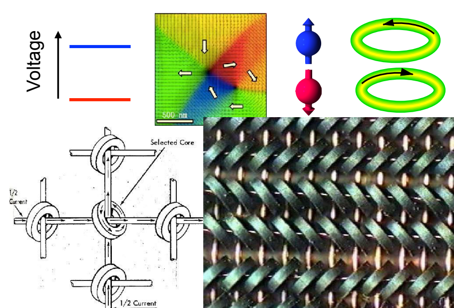

Figure 3.2: Collection of different physical systems for representing one bit of information.

3.2 represents schematically a few examples of bit representations. Generally, information is encoded in a string of bits, where the length of the string depends on the amount of information.

Information processing corresponds to manipulation of this information. Computations are defined by algorithms, i.e. sequences of elementary logical operations like NOT, OR, and AND, that act on (transform) strings of bits. Any transformation between two bit strings of finite length can be decomposed into one- and two-bit operations. (A proof of the quantum version of this important fact will be sketched in Chapter 5.)

Logic operations or gates can be characterized by the number of bits that they take as input and the number of bits they produce as output. Figure 3.3 shows the simplest example of a gate with one input bit and one output bit. This representation of logic gates, where wires represent bits, and boxes the gate operations leads naturally to what is called the *network model of computation* (often also called the *circuit model*).

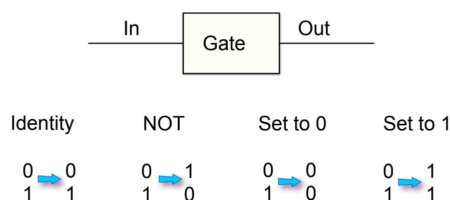

Figure 3.3: All possible one-bit gates.

The simplest type of logic gate operations are the

one bit gates, which act on a single input bit and produce a single output bit. Figure 3.3 shows the 4 possible operations that can be applied to a single bit: the bit may be left untouched (identity), it may be flipped (NOT), and it may be set to 0 or 1 unconditionally. The latter two operations are obviously irreversible.

3.2.2 2-bit logic gates

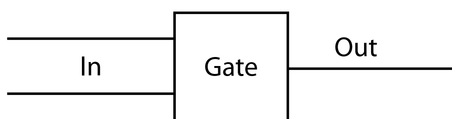

Figure 3.4: A logic gate with two input bits and one output bit.

At the next level of complexity are the 2-bit logic gates. We first discuss one-bit functions of a two-bit argument, as shown in Figure 3.4:

$$(x, y) \longrightarrow f(x, y) \text{ where } x, y, f = 0 \text{ or } 1.$$

Logic gates of this type are called *Boolean functions*. The four possible inputs 00, 01, 10, 11 can each be mapped to one of two possible outputs 0 and 1; the function is completely characterized by the string of four output bits $(f(00), f(01), f(10), f(11))$. Since there are $2^4 = 16$ different output strings, we have 16 possible Boolean functions of two binary variables. These gates are *irreversible* since the output is shorter than the input.

x	y	x OR y	x AND y
0	0	0	0
0	1	1	0
1	0	1	0
1	1	1	1

The binary operations OR and AND are defined by their *truth tables*, see the table above. Some elementary logic operations have useful algebraic representations:

- NOT $x \to 1 - x$
- x AND $y \to xy$
- x XOR $y \to x \oplus y$, i.e. addition modulo 2 .

All other operations, such as IMPLIES or XOR can be constructed from the elementary operations NOT, OR, and AND. As an example for the reduction of a logical operation to more elementary operations consider

$$x \text{ XOR } y = (x \text{ OR } y) \text{ AND } \text{ NOT } (x \text{ AND } y).$$

(XOR is also often denoted by $\oplus$, because it is equivalent to addition modulo 2.)

We now return to the 16 Boolean functions of two bits. We number them according to the four-bit output string as given in the truth table, read from top to bottom and interpreted as a binary number. For example AND outputs 0001=1 and OR outputs 0111=7. We can thus characterize each gate or function by a number from 0 to 15 and look at them in order. Some examples are: { AND }

AND

0: The absurdity, e.g.
$(x \text{ AND } y)$ AND NOT $(x \text{ AND } y)$.
$(x\{AND\}y)$ AND NOT $(x \text{ AND } y)$.

$$(x \text{ AND } y) \text{ AND } \text{ NOT } (x \text{ AND } y)$$

1: x AND y

2: x AND (NOT y)

3: x, which can be written in a more complicated way: $x = x$ OR $(y$ AND NOT $y)$

4: (NOT x) AND y

5: $y = ...$(see x above)

8: (NOT x) AND (NOT y) =: $(x$ NOR $y)$

9: ((NOT x) AND (NOT y)) OR $(x$ AND $y)$ = NOT $(x$ XOR $y)$ =: $(x$ EQUALS $y)$

All others can be obtained by negating the above; notable are

13: NOT $(x$ AND (NOT y)) =: x IMPLIES y

14: NOT (x AND y) $=: x$ NAND y

15: The banality, for example
(x AND y) OR NOT (x AND y).

We have thus seen that all Boolean functions can be constructed from the elementary Boolean operations. Furthermore, since

$$x \text{ OR } y = (\text{ NOT } x) \text{ NAND } (\text{ NOT } y),$$

we see that we *only* need NAND (as defined by line **14**) and NOT to achieve any desired classical logic gate with two input bits and one output bit.

In order to connect an arbitrary number n of input lines to m output lines we need, in addition to logic gates, the ability to COPY the contents of one bit to a different bit while keeping the original bit. This is usually symbolized by a branching line in a network diagram, which symbolizes a branching wire with equal voltage levels at the three terminals.

While copying a classical bit is thus a trivial operation, copying a quantum bit turns out to be impossible! This *no-cloning theorem* will be discussed in Chapter 4; it is at the heart of the schemes developed for *secure quantum communication* to be discussed in Chapter 13.

3.2.3 Minimal set of irreversible gates

For some problems, it is useful to reduce the number of gates needed to perform an arbitrary bit string operation to the absolute minimum. Being able to build a network using the smallest possible set of different elements is desirable from a theoretical point of view. On the other hand, such networks tend to become larger. In practice, it is therefore usually more advisable to employ a larger variety of gates in order to keep the total size of the network smaller.

Reduced sets of gates can be obtained by considering operations like

$$\begin{aligned} x \text{ NAND } y &= \text{ NOT } (x \text{ AND } y) \\ &= (\text{ NOT } x) \text{ OR } (\text{ NOT } y) \\ &= 1 - xy. \end{aligned}$$

If we can copy x to another bit, we can use NAND to achieve NOT:

$$x \text{ NAND } x = 1 - x^2 = 1 - x = \text{ NOT } x$$

(where we have used $x^2 = x$ for $x = 0, 1$). Alternatively, if we can prepare a constant bit 1:

$$x \text{ NAND } 1 = 1 - x = \text{ NOT } x.$$

We can also express AND and OR by NAND only:

$$\begin{aligned} & (x \text{ NAND } y) \text{ NAND } (x \text{ NAND } y) \\ =\ & 1 - (1 - xy)^2 \\ =\ & 1 - (1 - xy) = xy = x \text{ AND } y \end{aligned}$$

and

$$\begin{aligned} & (x \text{ NAND } x) \text{ NAND } (y \text{ NAND } y) \\ =\ & (\text{ NOT } x) \text{ NAND } (\text{ NOT } y) \\ =\ & 1 - (1 - x)(1 - y) = x \oplus y - xy \\ =\ & x \text{ OR } y. \end{aligned}$$

Thus the combination of the NAND gate and the COPY operation (which is not a gate in the strict sense defined above) forms a *universal set* of (irreversible) classical gates. A different universal set of two gates is given by NOR and COPY, for example.

The operations NAND and COPY can both be performed by a single two-bit to two-bit gate, if we can prepare a bit in state 1. This is the NAND/NOT gate:

$$\begin{aligned} (x, y) &\longrightarrow (1 - x, 1 - xy) \\ &= (\text{ NOT } x, x \text{ NAND } y). \end{aligned} \tag{3.1}$$

The truth table is

x	y	NOT x	x NAND y
0	0	1	1
0	1	1	1
1	0	0	1
1	1	0	0

The NOT and NAND functions are obviously achieved by ignoring the second and first output bit, respectively. For $y = 1$ we obtain COPY, combined with a NOT which can be inverted by the same gate.

3.2.4 Minimum set of reversible gates

Although we know how to construct a universal set of irreversible gates, there are good reasons to study the reversible alternative. Firstly, quantum gates *are* reversible, and secondly, reversible computation is in principle possible without dissipation of energy.

A reversible computer evaluates an invertible n-bit function of n bits. Note that every irreversible function can be made reversible at the expense of additional bits: the irreversible (for $m < n$) function mapping n bits to m bits

$$x(n \text{ bits}) \longrightarrow f(m \text{ bits})$$

is replaced by the reversible function mapping $n + m$ bits to $n + m$ bits

$$(x, m \text{ times } 0) \longrightarrow (x, f).$$

The *reversible* n-bit functions are *permutations* among the 2^n possible bit strings; there are $(2^n)!$ such functions. For comparison, the number of *arbitrary* n-bit functions is $(2^n)^{(2^n)}$ (Each of the 2^n input strings can be mapped to every possible output string). The number of reversible 1,- 2-, and 3-bit gates is 2, 24, and 40320, respectively.

While irreversible classical computation gets by with two-bit operations, reversible classical computation needs three-bit gates in order to be universal. This can be seen by observing that the 24 reversible two-bit gates are all linear, that is, they can be written in the form [37]

$$\begin{pmatrix} x \\ y \end{pmatrix} \longrightarrow \begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \alpha & \beta \\ \gamma & \delta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} a \\ b \end{pmatrix},$$

where all matrix and vector elements are bits and all additions are modulo 2. As the two reversible one-bit gates are also obviously linear, any combination of one- and two-bit operations applied to the components of a n-bit vector $\vec{x}$ can only yield a result linear in $\vec{x}$. On the other hand, for $n \geq 3$ there are invertible n-bit gates which are *not* linear, for example, the Toffoli gate to be discussed in section 3.2.6. In Chapter 5 we will see that quantum computing, although reversible too, does not need gates acting on three quantum bits to be universal. Furthermore all quantum gates will have to be strictly linear because quantum mechanics is a linear theory.

3.2.5 The CNOT gate

One of the more interesting reversible classical two-bit gates is the controlled NOT, or CNOT, also known as "reversible XOR", which makes the XOR operation reversible by storing one argument:

$$(x, y) \longrightarrow (x, x \text{ XOR } y). \tag{3.2}$$

The following table shows why (3.2) is called CNOT:

x	y	x	x XOR y
0	0	0	0
0	1	0	1
1	0	1	1
1	1	1	0

The target bit y is flipped if and only if the control bit $x = 1$. A second application of CNOT restores the initial state, so this gate is its own inverse.

The CNOT gate can be used to copy a classical bit, because it maps

$$(x, 0) \longrightarrow (x, x). \tag{3.3}$$

This is shown in the left-hand part of Figure 3.5.

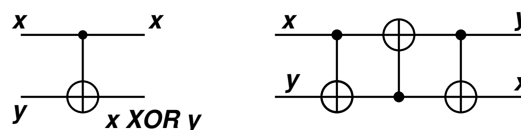

Figure 3.5: Left: Single CNOT gate. Right: SWAP gate from 3 CNOT gates.

The network combining three XOR gates in Figure 3.5 achieves a SWAP of the two input bits:

$$\begin{aligned} (x, y) &\longrightarrow (x, x \text{ XOR } y) \\ &\longrightarrow ((x \text{ XOR } y) \text{ XOR } x, x \text{ XOR } y) \\ &\longrightarrow (y, y \text{ XOR } (x \text{ XOR } y)) = (y, x) \end{aligned}$$

Thus the reversible XOR can be used to copy and move bits around.

3.2.6 The Toffoli gate

We will show now that the functionality of the universal NAND/NOT gate (3.1) can be achieved by adding a three-bit gate to our toolbox, the *Toffoli*[1] *gate* $\theta^{(3)}$, also known as controlled-controlled-NOT, (CCNOT) which maps

$$(x, y, z) \longrightarrow (x, y, xy \text{ XOR } z), \tag{3.4}$$

that is, z is flipped only if both x and y are 1. Table 3.1 shows the corresponding truth table.

Input			Output		
x	y	z	x	y	z
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	0
0	1	1	0	1	1
1	0	0	1	0	0
1	0	1	1	0	1
1	1	0	1	1	1
1	1	1	1	1	0

Table 3.1: Truth table of the Toffoli gate $\theta^{(3)}$=CCNOT.

The nonlinear nature of the Toffoli gate is evident from the presence of the product xy. This gate forms by itself a universal set, provided that we can prepare fixed input bits and ignore output bits:

- For $x = 1$ we obtain z XOR y which can be used to copy, swap, etc.
- For $x = y = 1$ we obtain NOT.

3.2.7 The Fredkin gate

Another reversible three-bit gate which can be used to build a universal set of gates is the Fredkin[2] gate [32]. While the Toffoli gate has two control bits and one target bit, the Fredkin gate has one control qubit and two target bits. The target bits are interchanged if the control bit is 1, otherwise they are left untouched. Table 3.2 shows the input and output of the Fredkin gate, where x is the control bit, and y and z are the target bits, respectively.

Input			Output		
x	y	z	x	y	z
1	1	1	1	1	1
1	1	0	1	0	1
1	0	1	1	1	0
1	0	0	1	0	0
0	1	1	0	1	1
0	1	0	0	1	0
0	0	1	0	0	1
0	0	0	0	0	0

Table 3.2: Truth table of the Fredkin gate.

To implement a reversible AND gate, for example, the z bit is set to 0 on input. On output z then contains x AND y, as can be seen in Table 3.2. If the other two bits x and y were discarded, this gate would be irreversible; keeping one of the input bits makes the operation reversible.

The NOT gate may also be embedded in the Fredkin gate: setting $y = 0$ and $z = 1$ on input we see that on output $z =$ NOT x, and $y = x$; thus we have implemented a COPY gate at the same time.

3.2.8 Reversible computation

These examples show that we can do any computation reversibly.

Figure 3.6 shows a schematic example of a resulting network. In general, one does not need all output bits, so they will be cleared before the next computation. In principle, it is even possible to avoid this dissipative step of memory

[1] Tommaso Toffoli (1934-)

[2] Edward Fredkin (1934 – 2023)

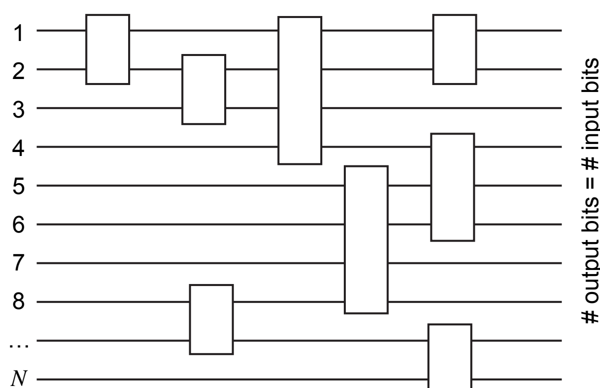

Figure 3.6: Principle of reversible computation in the network model. All gate operations have the same number of input- and output bits.

clearing: store all "garbage" which is generated during the reversible computation, copy the end result of the computation and then let the computation run backwards to clean up the garbage without dissipation. Though this may save some energy dissipation, it has a price as compared to reversible computation with final memory clearing:

- The time (number of steps) grows from T to roughly $2T$.
- Additional storage space, growing roughly proportional to T, is needed.

However, there are ways [37] to split the computation up in a number of steps which are inverted individually, so that the additional storage grows only proportional to $\log T$, but in that case more computing time is needed.

3.3 Universal computers

Computer science has developed some concepts that allow one to check for a given problem if an algorithm exists that terminates for all possible inputs. Such problems are called "computable", i.e. they can be solved on a computer. For existing algorithms, it is possible to determine their efficiency in a way that does not depend on hardware implementation. Computer science has developed some representations of computing machinery that have proved useful for this purpose, although nobody would actually build a computer according to this recipe. The most important example of such a "universal computer" is the Turing machine.

3.3.1 The Turing machine

The Turing machine has no importance as a practical computing device. However, according to the Church-Turing hypothesis (see next subsection) every task that can be performed by some computer can also be performed by a Turing machine, hence its importance in theoretical computer science as the simplest example for a *universal computer*.

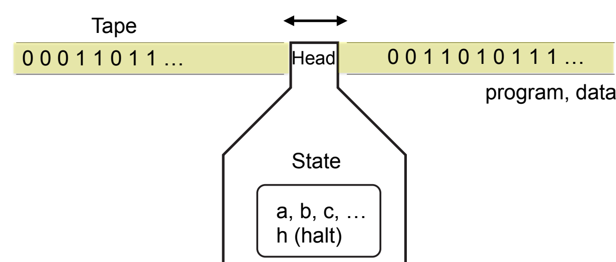

Figure 3.7: A Turing machine operating on a tape with binary symbols and possessing several internal states, including the *halt* state.

As shown in Figure 3.7, the Turing machine acts on a tape (or string of symbols) as an input/output medium. It has a finite number of internal states. If the machine reads the symbol s from the tape while being in state G, it will replace s by another symbol s', change its state to G' and move the tape one step in direction d (left or right). The machine is completely specified by a *finite set of transition rules*

$$(s, G) \longrightarrow (s', G', d)$$

The machine has one special internal state, the "halt" state, in which the machine stops all further activity. On input, the tape contains the

"program" and "input data"; on output, the result of the computation.

The (finite) set of transition rules for a given Turing machine T can be coded as a binary number $d[T]$ (the description of T). Let $T(x)$ be the output of T for a given input tape x. Turing showed that there exists a *universal Turing machine* U that can simulate every other Turing machine. If this universal machine is given a description $d[T]$ of any other machine T and the relevant input string x, it produces the same output that the simulated machine T would produce:

$$U(d[T], x) = T(x).$$

The number of steps that U needs to simulate each step of T is only a *polynomial* function of the length of $d[T]$.

3.3.2 Example

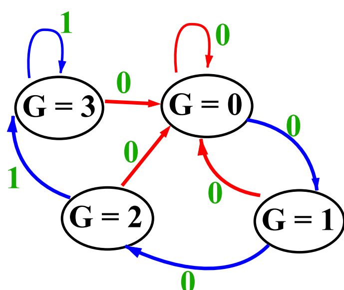

Figure 3.8: Example of the states and transitions in a Turing machine. G represents the state of the head, blue arrows the state change if 0 is read, red arrows the state change if 1 is read, and green numbers the output to be written.

Figure 3.8 shows an example for the possible states and transitions in a Turing machine. The machine has the task to analyze the contents of the tape and output a 1 if it finds 3 or more 1's in a row and 0 otherwise. The solution shown in the figure uses 4 states. Blue arrows indicate the transitions if a 1 is read, red arrows if a 0 is read and green numbers the output values.

With suitable sets of rules, Turing machines can be reversible. A reversible set of rules would be the set of operations represented in Table 3.3.

head state	bit read	change bit to	change state to	move to
A	1	0	A	left
A	0	1	B	right
B	1	1	A	left
B	0	0	B	right

Table 3.3: Example of a reversible Turing machine.

The information processing corresponds to a motion of the head. In the reversible Turing machine, the motion of the head is driven by thermal fluctuations and a small force defining the direction. The amount of energy dissipated in this computer decreases without limit as this external force is reduced, but at the same time the processing speed decreases. Overall the best picture to describe the operation of a reversible computer is that it is driven along a computational path. The same path may be retraced backward by changing some external parameter, thereby completely reversing the effect of the computation.

3.3.3 The Church–Turing hypothesis

Other models of computation (for example the network model) are computationally equivalent to the Turing model: Any task that can be computed in the network model can also be computed on a Turing machine (and vice versa). In 1936 Alonzo Church [24] and Alan Turing [25] independently stated the

Church–Turing hypothesis: Every function which would naturally be regarded as computable can be computed by the universal Turing machine.

The notion of a computable function here is meant to comprise an extremely broad range of

tasks. Any mapping of a finite string of bits to another finite string of bits falls into this range. The input string could come from a lengthy sequence of keystrokes where the output bit string is the print on this page. As another example, the input string could be some table containing numerical data and the output string a graphical representation of these data. There is no proof of the Church–Turing hypothesis, but also no counterexample has been found, despite decades of attempts to find one.

3.4 Complexity and algorithms

3.4.1 Complexity classes

Complexity has many aspects, and computational problems may be classified with respect to several measures of complexity. Here we will only treat very few examples from important complexity classes. The article by Mertens [38] gives more examples in easy-to-read style.

Consider some task to be performed on an integer input number x; for example, finding x^2 or determining if x is a prime. The number of bits needed to store x is

$$L = \log_2 x.$$

The *computational complexity* of the task characterizes, how fast the number s of steps that a Turing machine needs to solve the problem, increases with L. For example, the method by which most of us have computed squares of "large" numbers in primary school has roughly

$$s \propto L^2$$

(if you identify s with the number of digits you have to write on your sheet of paper). This is a typical problem from complexity class P: there is an algorithm for which s is a *polynomial* function of L. All polynomial algorithms are considered efficient. If s rises exponentially with L the problem is considered hard. Note, however, that it is not possible to exclude the discovery of new algorithms which make previously hard problems tractable!

In many cases, it is much easier to verify a solution than to find it. A good example for this is the factorization of large numbers. This type of problems forms the complexity class NP : It consists of problems for which solutions can be *verified* in polynomial time. Of course P is contained in NP, but *it is not known if NP is actually larger than P*, basically because revolutionary algorithms may be discovered any day. NP means *nondeterministic polynomial*. A nondeterministic polynomial algorithm may at any step *branch* into two paths which are both followed *in parallel*. Such a tree-like algorithm is able to perform an exponential number of calculational steps in polynomial time (at the expense of exponentially growing parallel computational capacity!). To verify a solution, however, one only has to follow "the right branch" of the tree and that is obviously possible in polynomial time.

Some problems may be *reduced* to other problems, that is, the solution of a problem P_1 may be used as a "step" or "subroutine" in an algorithm to solve another problem P_2. Often it can be shown that P_2 may be solved by applying the subroutine P_1 a polynomial number of times; then P_2 is *polynomially reducible* to P_1: $P_2 \leq P_1$. (Read: "P_2 cannot be harder than P_1.") Some nice examples are provided by problems from graph theory, where one searches paths with certain properties through a given graph (or network), see [38].

A problem is called *NP-complete* if any NP problem can be reduced to it. Hundreds of NP-complete problems are known, one of the most famous being the *traveling salesman problem* of finding the shortest route between a given number of cities that touches every city once and starts and ends at the same city. If somebody finds a polynomial solution for *any* NP-complete problem, then "P=NP" and one of the most fundamental problems of theoretical computer science is solved. This is, however, very unlikely, since many first-rate scientists have unsuccess-

fully tried to find such a solution.

It should be noted at this point that theoretical computer science bases its discussion of complexity classes on *worst case* complexity. In practical applications it is very often possible to find excellent approximations to the solution of, say, the traveling salesman problem within reasonable time.

3.4.2 Hard and impossible problems

A famous example for a hard problem is the factoring problem (finding the prime factors of a given large integer) already mentioned in Chapter 1. We will discuss this problem and its relation to cryptography in Chapter 8, where we will also treat Shor's [19] quantum factorization algorithm. Since Shor's discovery, suspicions have grown that the factoring problem may be in class *NPI* (*I* for intermediate), that is, harder than *P*, but not *NP*-complete. If this class exists, $P \neq NP$.

Some functions are not just hard to compute but *uncomputable*, i.e. the algorithm will never stop, or, nobody knows *if* it will ever stop. An example is the algorithm:

> *While x is equal to the sum of two primes, add 2 to x*
> *otherwise print x and halt*
> *beginning at* $x = 8$.

If this algorithm stops, we have found a counterexample to the famous Goldbach conjecture, that every even number is the sum of two primes.

Another famous unsolvable problem is the *halting problem*, which is stated very easily:

> *Is there a general algorithm to decide if a Turing machine T with description (transition rules) $d[T]$ will stop for a given input x?*

There is a nice argument by Turing showing that such an algorithm does not exist. Suppose such an algorithm existed. Then it would be possible to make a Turing machine T_H which halts if and only if $T(d[T])$ (that is, T, fed its own description as input) does not halt:

$$T_H(d[T]) \text{ halts } \Leftrightarrow T(d[T]) \text{ does not halt.}$$

This is possible since the description $d[T]$ contains sufficient information about the way T works. Now feed T_H the description of itself, that is, put $T = T_H$

$$T_H(d[T_H]) \text{ halts } \Leftrightarrow T_H(d[T_H]) \text{ does not halt.}$$

This contradiction shows that there is no algorithm that solves the halting problem. This is a nice recursive argument: let an algorithm find out something about its own structure. This kind of reasoning is typical of the field centered around Gödel's incompleteness theorem. A very interesting semi-literary piece of work centered about the ideas of recursiveness and self-reference in mathematics and other fields of culture is the book "Gödel, Escher, Bach" [39] by the physicist/computer scientist Douglas R. Hofstadter.

Further reading

More complete accounts of computer science aimed at the discussion of quantum computing can be found in [40], Chap. 3, [41], Secs. 2 and 3, [37], Sec. 6.1. These references also contain pointers to more rigorous mathematical treatments of the subject. Details on the history of computing can be found, for example, in the history section of the entry "Computers" in the *Encyclopedia Britannica*. A nice readable account of complexity with some more details than we treat (and need) here is [38].