2 Elements of classical computer science

Good references for the material of this section are [3], Chap. 3, [5], Secs. 2 and 3, [8], Sec. 6.1, and [13] as a nice readable account of complexity with some more details than we will treat (and need) here.

2.1 Bits of History

Charles Babbage – The inventor of the first programmable computer is probably Charles Babbage (1791-1871). 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 Babbage's unpublished notebooks were discovered in 1937 and the 31-digit accuracy "Difference Engine No. 2" was built to Babbage's specifications in 1991. (Babbage was also Lucasian professor of mathematics in Cambridge, like Newton, Stokes, Dirac, and Hawking, and he invented the locomotive cowcatcher.)

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

George Boole – 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.

Alan Turing – Alan Turing (1912-1954) invented the Turing machine in 1936 in the context of the decidability problem posed by David Hilbert: Is it always possible to decide whether a given mathematical statement is true or not? (It is not, and Turing's machine helped show that.)

2.2 Boolean algebra and logic gates

Computers manipulate bit strings. Any transformation on n bits can be decomposed into one- and two-bit operations. There are only two one-bit operations: identity and NOT.

Logic gates – A *logic gate* is a one-bit function of a two-bit argument:

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

The four possible inputs 00, 01, 10, 11 are be mapped to two possible outputs 0 and 1 each; there are $2^4 = 16$ possible output strings of 4 symbols and thus there are 16 possible logic gates or "Boolean functions". Note that these gates are *irreversible* since the output is shorter than the input.

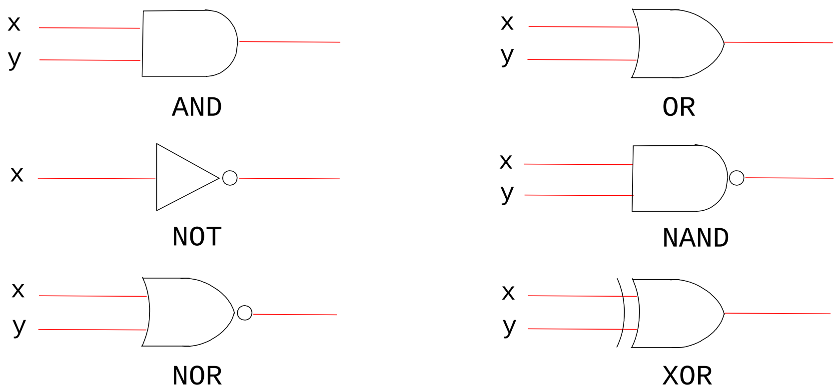

Figure 3: Basic classical logic gates. AND: $x \wedge y$, OR: $x \vee y$, NOT: $\bar{x}$, NAND: $\overline{x \wedge y}$, NOR: $\overline{x \vee y}$, XOR: $x \oplus y$.

Truth tables – Before we discuss these functions we will recall some elements of standard Boolean logic. Boolean logic knows two *truth values*, true and false, which we identify with 1 and 0, respectively, and the elementary operations NOT, OR, and AND from which all other operations and relations like IMPLIES or XOR can be constructed. The binary operations OR and AND are defined by their "truth tables":

x	y	x OR y	x AND y	x NAND y	x NOR y	x XOR y
		x+y -xy	xy	1-xy	(1-x)(1-y)	x+y-2xy
0	0	0	0	1	1	0
0	1	1	0	1	0	1
1	0	1	0	1	0	1
1	1	1	1	0	0	0

Note that, for example x NAND $y =$ NOT $(x$ AND $y)$ and

$$x \text{ XOR } y \;=\; (x \text{ OR } y) \text{ AND } (x \text{ NAND } y)\ .$$

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

The 16 boolean functions – 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 above truth table, read from top to bottom and interpreted as a binary number. Example: AND outputs 0001=1 and OR outputs 0111=7. We can thus characterize each gate (function) by a number between 0 and 15 and discuss them in order:

- 0: The absurdity, for example $(x$ AND $y)$ AND NOT $(x$ AND $y)$.
- 1: x AND y

- 2: x AND (NOT y)
- 4: (NOT x) AND y
- 8: (NOT x) AND (NOT y) = (x NOR y)
- 3: x, which can be written in a more complicated way: $x = x$ OR (y AND NOT y)
- 5: $y = ...$(see above)
- 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

- 15: The banality, for example (x AND y) OR NOT (x AND y).
- 14: NOT (x AND y) = x NAND y
- 13: NOT (x AND (NOT y)) = x IMPLIES y

NAND and NOT are complete – We have thus seen that all logic gates 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 and NOT to achieve any desired classical logic gate.

Logical networks – If we picture logic gates as elements with two input lines and one output line and connect these logic gates by wires through which we feed the input, we get what is called the *network model of computation*.

2.2.1 Minimum set of irreversible gates

NOT from NAND – We note that

$$x \text{ NAND } y \;=\; 1 - xy \;,$$

for $x, y = 0, 1$. 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 $x = 0, 1$. Or, if we can prepare a constant bit $y \equiv 1$:

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

AND and OR from NAND – If we can make copies form both x and y, then we can express AND and OR by NAND only:

$$(x \text{ NAND } y) \text{ NAND } (x \text{ NAND } y) \;=\; 1-(1-xy)^2 \;=$$

$$=\; 1-(1-2xy+x^2y^2) \;=\; 1-(1-xy) \;=\; xy \;\equiv\; x \text{ AND } y\ ,$$

and

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

$$=\; 1-(1-x)(1-y) \;=\; x+y-xy \;\equiv\; x \text{ OR } y\ .$$

The COPY gate – We introduce the new gate COPY. It is not a logical gate, which maps two bits onto one; it has two inputs and two outputs:

$$\text{COPY} \;:\quad (x,y) \;\longrightarrow\; (x,x)\ .$$

It copies x to another bit. NAND and COPY are a universal set of (irreversible) gates.

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

$$(x,y) \;\longrightarrow\; (1-x, 1-xy) \;=\; (\text{ NOT } x, x \text{ NAND } y)\ .$$

The NOT and NAND functions are obviously achieved by ignoring one output bit. For y=1 we obtain COPY, combined with a NOT which can be inverted.

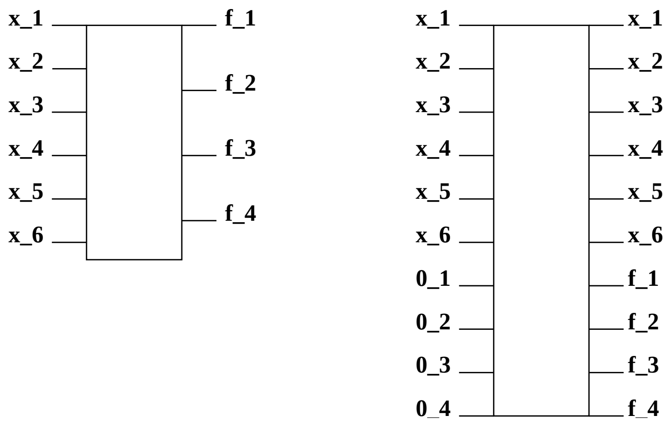

Figure 4: Left: A non reversible gate $(x_1,\ldots,x_6) \to (f_1,\ldots,f_4)$. Right: The same as a reversible gate.

2.2.2 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, it is interesting to discuss the possibility of reversible (dissipationless) computation. Here we discuss *classical* reversible gates.

Reversible gates – A reversible computer evaluates invertible n-bit function of n bits. Note that every irreversible function can be made reversible at the expense of additional bits, see Fig. 4 for an example:

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

is replaced by

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

The possible *reversible* n-bit functions are the *permutations* of the 2^n possible bit strings (permutations are a basis in the space of unitary transformations); there are $(2^n)!$ such functions. For comparison, the number of *arbitrary* n-bit functions is $(2^n)^{(2^n)}$. The universal gate set for reversible (classical) computation will have to include three-bit gates, as discussed in [8]. The number of reversible 1,- 2-, and 3-bit gates is 2, 24, and 40320, respectively.

The controlled NOT gate – One of the more interesting reversible two-bit gates is the CNOT, also known as "reversible XOR". The NOT-operation on the y-bit is controlled by the state of the x-bit:

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

that is

		CNOT		NOT/NAND	
x	y	x	x XOR y	1-x	1-xy
0	0	0	0	1	1
0	1	0	1	1	1
1	0	1	1	0	1
1	1	1	0	0	0

The reversibility is caused by the storage of x: Every input pair (x, y) shows up exactly once at the CNOT output, see table above. The square of this gate is IDENTITY. XOR is often symbolized by $\oplus$. The CNOT gate can be used to copy a bit, because it maps

$$(x, 0) \longrightarrow (x, x) \; .$$

Exchange of bits – The combination of three XOR gates in Fig. 5 achieves a SWAP. After the first XOR we have.

$$(x, y) \longrightarrow (x, x \text{ XOR } y) = (x, x + y - 2xy)$$

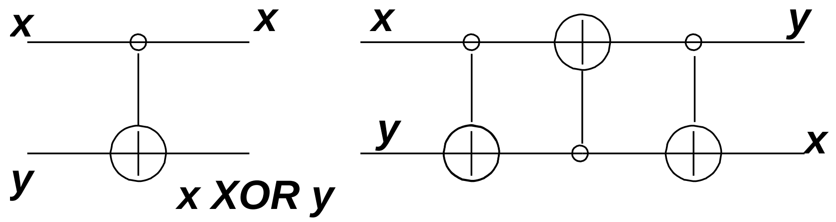

Figure 5: Left: Single CNOT gate. Right: SWAP gate

The second XOR yields

$$(x, x{+}y{-}2xy) \;\longrightarrow\; (x{+}(x{+}y{-}2xy){-}2x(x{+}y{-}2xy), x{+}y{-}2xy) \;=\; (y, x{+}y{-}2xy)\,.$$

The third XOR yields

$$(y, x + y - 2xy) \;\longrightarrow\; (y, y + (x + y - 2xy) - 2y(x + y - 2xy)) \;=\; (x, y)\;.$$

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

The Toffoli gate – We have see in Sec. 2.2.1 that NAND and COPY form a minimum set of irreversible gates and that they can be expressed by the two bit gate NOT/NAND. This function is, however not reversible, as can been seen in the table above: The combination $(1, 1)$ appears twice on the output! We need therefore a three-bit gate in order to construct a reversible version of the NOT/NAND gate the *Toffoli gate* $\theta^{(3)}$, also known as controlled-controlled-NOT, which maps

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

Exercise: Show that $\theta^{(3)}$ is reversible. The $\theta^{(3)}$-gate is universal, provided we can provide fixed input bits and ignore output bits:

- For $z = 1$ we have $(x, y, 1) \longrightarrow (x, y, 1 - xy) = (x, y, x \text{ NAND } y)$.
- For $x = 1$ we obtain z XOR y which can be used to copy, swap, etc.
- For $x = y = 1$ we obtain NOT.

Thus we can do any computation reversible. In fact it is even possible to avoid the dissipative step of memory clearing (in principle): 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. This may save some energy dissipation, but 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 $\propto T$ is needed.

However, there are ways [8] to split the computation up in a number of steps which are inverted individually, so that the additional storage grows only $\propto \log T$, but in that case more computing time is needed.

Conclusions –

- We have used the possibility to COPY bits throughout this section without much thought. In quantum computation we will have to do without COPYing due to the "no cloning theorem"!
- Classical reversible logics needs three bit gates. If this would be true quantum-mechanical, no quantum-computer would ever be built!

2.3 The Turing machine as a universal computer

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 binary symbol s from 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 α (left or right). The machine is completely specified by a *finite set of transition rules*

$$(s, G) \longrightarrow (s', G', \alpha) \ .$$

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.*

Universal Turing machine – 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 with

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

and the number of steps U needs to simulate each step of T is only a *polynomial* function of the length of $d[T]$. Thus we only have to supply the "description" $d[T]$ of T and the original input x on a tape to U and U will perform the same task as any machine T, with at most polynomial slowdown.

The Church-Turing thesis – Every function which would naturally be Other models of computation (for example the network model) are computationally equivalent to the Turing model: the same tasks can be performed with the same efficiency. Church [16] and Turing [17] stated the

*See http://www.strangehorizons.com/2001/20010402/turing.gif for a nice animated-gif Turing-machine.

Church-Turing thesis: Every function which would naturally be regarded as computable can be computed by the universal Turing machine.
("Computable functions" comprise an extremely broad range of tasks, such as text processing etc.) There is no proof of this thesis, but also no counterexample, despite many attempts to find one.

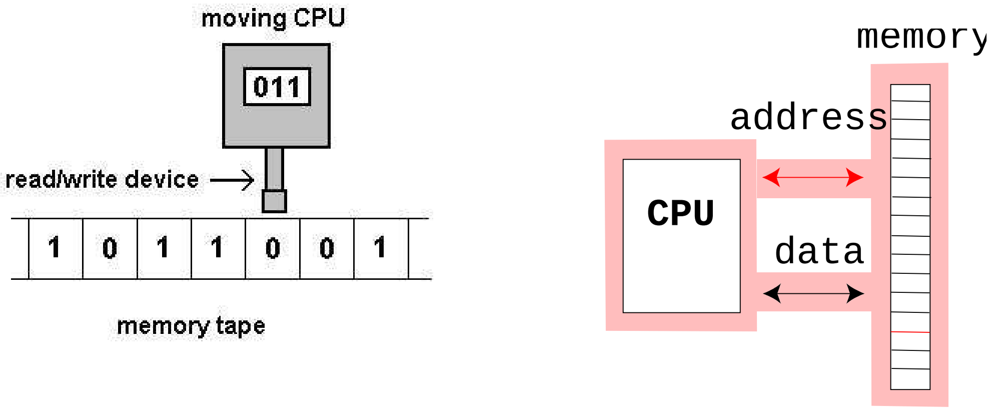

Figure 6: Left: A Turing machine operating on a tape with binary symbols and possessing several internal states, including the *halt* state. Right: A von Neumann machine which contains a processor (CPU), which works on the memory which contains the data and the programm (set of instructions).

2.4 The von Neumann machine

We can summarize the functioning of an ordinary computer by saying *one single thing at a time*. Von Neumann was the first to formalize the principles of a "program-registered calculator" based in the *sequential* execution of the programs registered in the memory of the computer. This is called a *von Neumann machine* (VNM). A VNM has the following parts which are depicted in Fig. 6:

Processor: The active part of the computer where the information contained in the programs is processed step by step.

Memory: The part of the computer devoted to the storage of the data and instructions to be processed. It is divided into individual cells which are accesible by means of a number called *address*.

Easy and universal – The functioning of a VNM is cyclic. One of these cycles contains the following operations: the control unit reads one program instruction from the memory, which is executed after being decoded. Depending on the type of instruction, a piece of data can either be read from or written in the memory, or an instruction be executed. In the next cycle to be performed, the control unit reads another program instruction which is precisely next in the memory to the one processed in the previous cycle. It is the simplicity of this sequentially operating model which makes it rather advantageous for many purposes because it facilitates the design of machines and programs.

2.5 Complexity and complexity classes

We will not discuss many examples of problems from the different complexity classes. The article by Mertens [13] gives several examples in easy-to-read style.

Computational complexity – Consider some task to be performed on an input (integer, for simplicity) 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 a Turing machine needs to solve the problem increases with L. 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 class P: there is an algorithm for which s is a *polynomial* function of L. 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 problems tractable!)

Verification of solutions – It is often much easier to verify a solution than to find it; think of factorizing large numbers. The class NP 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. By NP-hard one denotes the problems which to-date are hard problems (exponentially difficult) with solutions which can be verified in polynomial time.

NP means *non-deterministic polynomial.* A non-deterministic polynomial algorithm may *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.

Reducible problems – 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 is looking for paths with certain properties through a given graph (or network), see [13]. 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.* 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 clever people have in vain 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, say, the traveling salesman problem within reasonable time.

A famous example for a hard problem is the factoring problem already discussed in the preceding chapter. Some formulas, numbers, and references on this problem and its relation to cryptography can be found in section 3.2 of Steane's review [5]. Since the invention of Shor's [12] quantum factorization algorithm, 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.

Unsolvable problems – Some functions may be not just hard to compute but *uncomputable* because 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 $x = 6$.*

If this algorithm stops, we have found a counterexample to the famous Goldbach conjecture *"Every even integer n greater than two is the sum of two primes."*.

The halting problems – Another famous unsolvable problem is the *halting problem*, which is stated very easily: Is there a general algorithm to decide if any Turing machine T with description (transition rules) $d[T]$ will stop on a certain 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) does not halt:

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

(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 $T = T_H$

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

This contradiction shows that there is no algorithm that solves the halting problem. (Or that every Turing machine halts, fed its own description. That must be excluded somehow.) 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 Gdel's incompleteness theorem; a very interesting literary piece of work centered about the ideas of recursiveness and self-reference (in mathematics and other fields of culture) is the book "Gdel, Escher, Bach" [18] by the physicist/computer scientist Douglas R. Hofstadter.

References

[1] Michael Brooks. *Quantum Computing and Communications*. Springer, New York, 1999. 152 pp.

[2] Jozef Gruska. *Quantum computing*. McGraw-Hill, London, 2000. 439 pp.

[3] Michael A. Nielsen and Isaac L. Chuang. *Quantum computation and quantum information*. Cambridge Univ. Press, Cambridge, 2001. 676 pp.

[4] Colin P. Williams and Scott H. Clearwater. *Explorations in quantum computing*. Springer, New York, 1998. 307 pp, CD-ROM, DM 98.00.

[5] A. Steane. Quantum computing. *Rep. Prog. Phys.*, 61:117–173, 1998.

[6] Dorit Aharonov. Quantum computation. In Dietrich Stauffer, editor, *Annual Reviews of Computational Physics VI*, pages 259–346. World Scientific, 1999. Preprint quant-ph/9812037.

[7] A. Galindo and M. A. Martin-Delgado. Information and computation: Classical and quantum aspects, 2002. Rev. Mod. Phys., to appear. Preprint quant-ph/0112105.

[8] John Preskill. Quantum computation. http://theory.caltech.edu/ preskill/ph229/, 1997. 413 pp.

[9] Paul Benioff. Quantum mechanical Hamiltonian models of Turing machines. *J. Stat. Phys.*, 29:515, 1982.

[10] Y. Lecerf. Machines de Turing reversibles. Compters Rendus **257**, 1597, 1963.

[11] Richard P. Feynman. Simulating physics with computers. *Int. J. Theor. Phys.*, 21:467–488, 1982.

[12] Peter Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. In *Proceedings, 35th Annual Symposium on Foundations of Computer Science*, Piscataway, NJ, 1994. IEEE Press.

[13] Stephan Mertens. Computational complexity for physicists. Lecture given at the DPG school on "Quantum Computing", 2000. Preprint cond-mat/0012185.

[14] Barenco, A., C.H. Bennett, R. Cleve, D.P. DiVincenzo, N. Margolus, P. Shor, T. Sleator, J.A. Smolin, H. Weinfurter. *"Elementary gates for quantum computation"*, Phys. Rev. A **52**, 3457-3467 (1995).

[15] Barenco, A., D. Deutsch, A. Ekert and R. Jozsa. *"Conditional quantum dynamics and logic gates"*, Phys. Rev. Lett. **74**, 4083-6 (1995).

[16] Alonzo Church. An unsolvable problem of elementary number theory. *Am. J. Math.*, 58:345–363, 1936.

[17] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proc. Lond. Math. Soc. Ser.*, 42:230, 1936. See also ibidem 43, 544.

[18] Douglas R. Hofstadter. *Gdel, Escher, Bach: An Eternal Golden Braid*. Basic Books, New York, 1979. 777 pp.

[19] Leo Szilard. ber die Entropieverminderung in einem thermodynamischen System bei Eingriffen intelligenter Wesen. *Z. Physik*, 53:840–856, 1929.

[20] R. Landauer. Irreversibility and heat generation in the computing process. *IBM Journal Res Dev*, 5:183–191, 1961.

[21] Seth Lloyd. Ultimate physical limits to computation. *Nature*, 406:1047–1054, 2000.

[22] N. Margolus and L.B. Levitin. title. In T. Toffoli, M. Biafore, and J. Le?o, editors, *Proceedings of the Fourth Workshop on Physics and Computation PhysComp96 (New England Complex Systems Institute, Boston, MA, 1996)*. North-Holland, 1998. Physica D Vol. 120.

[23] Richard P. Feynman. *Feynman Lectures on Computing*, chapter 6, Quantum Mechanical Computers. Addison-Wesley, 1996. Originally published in Optics News, February 1985, pp. 11-20; see also Found. Phys. **16**, 507-532 (1986).

[24] P. Jordan and E. Wigner. Über das Paulische Äquivalenzverbot. *Z. Physik*, 47:631–651, 1928.

[25] Jian-Wei Pan, Dik Bouwmeester, Matthew Daniell, Harald Weinfurter, and Anton Zeilinger. Experimental test of quantum nonlocality in three-photon Greenberger-Horne-Zeilinger entanglement. *Nature*, 403:515–519, 2000.

[26] C. A. Sackett, D. Kielpinski, B. E. King, C. Langer, V. Meyer, C. J. Myatt, M. Rowe, Q. A. Turchette, W. M. Itano, D. J. Wineland, and C. Monroe. Experimental entanglement of four particles. *Nature*, 404:256–259, 2001.

[27] Rainer Blatt. Push-button entanglement. *Nature*, 404:231–232, 2001.

[28] Brian Julsgaard, Alexander Kozhekin, and Eugene S. Polzik. Experimental long-lived entanglement of two macroscopic objects. *Nature*, 413:400–403, 2001.

[29] J. Ignacio Cirac. Quantum physics: Entangled atomic samples. *Nature*, 413:375–377, 2001.

[30] L. Grover. (title). In *Proc. 28th Annual ACM Symposium on the Theory of Computation*, pages 212–219, New York, 1996. ACM Press.

[31] L.K. Grover. Quantum mechanics helps in searching for a needle in a haystack. *Phys. Rev. Lett.*, 79:325–328, 1997. Preprint quant-ph/9706033.

[32] P. Oscar Boykin, Tal Mor, Matthew Pulver, Vwani Roychowdhury, and Farrokh Vatan. On universal and fault-tolerant quantum computing, 1999. Preprint quant-ph/9906054.

[33] D.P. DiVincenzo. The physical implementation of quantum computation. *Fortschr. Phys.*, 48:771–783, 2000. Preprint quant-ph/0002077.

[34] C.E. Hughes and S. Wimperis. Bounds on spin dynamics and the design of multiple-pulse NMR experiments. *J. Chem. Phys.*, 106:2105–2117, 1997.

[35] J. Stoustrup, O. Schedletzky, S.J. Glaser, C. Griesinger, N.C. Nielsen, and O.W. Sorensen. Generalized bound on quantum dynamics: efficiency of unitary transformations between non-hermitian states. *Phys. Rev. Lett.*, 74:2921–2924, 1995.

[36] J. H. Smet, R. A. Deutschmann, F. Ertl, W. Wegscheider, G. Abstreiter, and K. von Klitzing. Gate-voltage control of spin interactions between electrons and nuclei in a semiconductor. *Nature*, 415:281–286, 2002.

[37] J. Kikkawa. Eavesdropping on spin talk. *Nature*, 415:273–274, 2002.

[38] W.K. Wootters and W.H. Zurek. A single quantum cannot be cloned. *Nature*, 299:802–803, 1982.

[39] D. Dieks. Communication by EPR devices. *Phys. Lett. A*, 92:271–271, 1982.

[40] C.H. Bennett, G. Brassard, C. Crépeau, R. Jozsa, A. Peres, and W. Wootters. Teleporting an unknown quantum state via dual classical and EPR channels. *Phys. Rev. Lett.*, 70:1895–1899, 1993.

[41] C.H. Bennettr. Logical reversability of computation. *IBM J. Res. Dev.* **17**, 525-532, 1973.

[42] C.H. Bennett and S.J. Wiesner. Communication via one- and two-particle operators on Einstein-Podolsky-Rosen states. *Phys. Rev. Lett.*, 69:2881–2884, 1992.

[43] C.H. Bennett and G. Brassard. Quantum cryptography: Public key distribution and coin tossing. In *Proceedings of IEEE International Conference on Computers, Systems, and Signal Processing (Bangalore, India, December 1984)*, pages 175–179, New York, 1984. IEEE.

[44] H. Zbinden, J.D. Gautier, N. Gisin, B. Huttner, A. Muller, and W. Tittle. Interferometry with Faraday mirrors and quantum cryptography. *Electron. Lett.*, 33:586–588, 1997.

[45] C.H. Bennett, G. Brassard, and N.D. Mermin. Quantum cryptography without Bell's theorem. *Phys. Rev. Lett.*, 68:557–559, 1992.

[46] G.M. Palma, K.-A. Suominen, and A.K. Ekert. Quantum computers and dissipation. *Proc. Roy. Soc. A*, 452:567–584, 1996.

[47] D.A. Lidar, I.L. Chuang, and K.B. Whaley. Decoherence-free subspaces for quantum computation. *Phys. Rev. Lett.*, 81:2594–2597, 1998.

[48] E. Knill, R. Laflamme, and L. Viola. Theory of quantum error correction for general noise. *Phys. Rev. Lett.*, 84:2525–2528, 2000.

[49] L. Viola, E.M. Fortunato, M.A. Pravia, E. Knill, R. Laflamme, and D.G. Cory. Experimental realization of noiseless subsystems for quantum information processing. *Science*, 293:2059–2063, 2001.