Lecture 13: Computing

Computing deals with algorithms and the praxis of programming. While the subject intersects with computer science, information technology, the theory is by nature very mathematical. But there are new aspects: computers have opened the field of **experimental mathematics** and serve now as the **laboratory** for new mathematics. Computers are not only able to **simulate** more and more of our physical world, they allow us to **explore** new worlds.
A mathematician pioneering new grounds with computer experiments does similar work than an experimental physicist. This is ok, since much of theoretical physics is so remote from experiments that one has to judge it as part of mathematics. Computers have smeared the boundaries between physics and mathematics. According to Borwein and Bailey, experimental mathematics consists of:

Gain insight and intuition.	Explore possible new results
Find patterns and relations	Suggest approaches for proofs
Display mathematical principles	Automate lengthy hand derivations
Test and falsify conjectures	Confirm already existing proofs

When using computers to prove things, reading and verifying the computer program is part of the proof. If Goldbach's conjecture would be known to be true for all $n > 10^{18}$, the conjecture should be accepted because numerical verifications have been done until $2 \cdot 10^{18}$ until today. The first famous theorem proven with the help of a computer was the "4 color theorem" in 1976.
Here are some pointers in the history of computing:

-300	Early Abacus	1941	Zuse 3	1973	Windowed OS
1300	Modern Abacus	1943	Harvard Mark I	1977	Apple II
1600	Slide rule	1944	Colossus	1981	Windows I
1623	Schickard computer	1946	ENIAC	1983	IBM PC
1640	Pascal Calculator	1947	Transistor	1984	Macintosh
1672	Leibniz multiplier	1952	IBM 701	1988	Next
1801	Punch cards	1958	Integrated circuit	1989	HTTP
1820	Difference Engine	1969	Arpanet	1993	Webbrowser, PDA
1876	Mechanical integrator	1971	Microchip	1998	Google
1935	Zuse 1 programmable	1972	Email	2007	iPhone

We live in a time where technology explodes exponentially.**Moore's law** from 1965 predicted that semiconductor technology doubles in capacity and overall performance every 2 years. This has happened since. Some futurologists like Ray Kurzweil conclude from this technological singularity in which artificial intelligence might take over. Lets move to safer ground and discuss an important concept of computing, the question how to decide whether a computation is "easy" or "hard". In 1937, **Alan Turing** introduced the idea of a **Turing machine**, a theoretical model of a computer which allows to quantify complexity. It has finitely many states $S = \{s_1, ..., s_n, h\}$ and works on an tape of $0-1$ sequences. The state h is the "halt" state. If it is reached, the machine stops. The machine has rules which tells what it does if it is in state s and reads a letter a. Depending on s and a, it writes 1 or 0 or moves the tape to the left or right and moves into a new state. Turing showed that anything we know to compute today can be computed with Turing machines. For any known machine, there is a polynomial p so that a computation done in k steps with that computer can be done in $p(k)$ steps on a Turing machine. What can actually be computed? Church's thesis of 1934 states that everything which can be computed can be computed with Turing machines. Similarly as in mathematics itself, there are limitations of computing. Turing's setup allowed him to enumerate all possible Turing machine and use them as input of an other machine. Denote by TM the set of all pairs (T, x), where T is a Turing machine and x is a finite input. Let $H \subset TM$ denote the set of Turing machines (T, x) which halt with the tape x as input. Turing looked at the decision problem: is there a machine which decides whether a given machine (T, x) is in H or not. An ingenious Diagonal argument of Turing shows that the answer is "no". [Proof: assume there is a machine $HALT$ which returns from the input (T, x) the output HALT(T, x) = true, if T halts with the input x and otherwise returns HALT(T, x) = false. Turing constructs a Turing machine DIAGONAL, which does the following:

1) Read x. 2) Define Stop=HALT(x,x) 3) While Stop=True repeat Stop:=True; 4) Stop

Now, DIAGONAL is either in H or not. If DIAGONAL is in H, then the variable Stop is true which means that the machine DIAGONAL runs for ever and DIAGONAL is not in H. But if DIAGONAL is not in H, then the variable Stop is false which means that the loop 3) is never entered and the machine stops. The machine is in H.]

Lets go back to the problem of distinguishing "easy" and "hard" problems: One calls **P** the class of decision problems that are solvable in polynomial time and **NP** the class of decision problems which can efficiently be tested if the solution is given. These categories do not depend on the computing model used. The question

"N=NP?"

is the most important open problem in theoretical computer science. It is one of the seven **millenium problems** and it is widely believed that $P \neq NP$. If a problem is such that every other NP problem can be reduced to it, it is called **NP-complete**. Popular games like Minesweeper or Tetris are NP-complete. If $P \neq NP$, then there is no efficient algorithm to beat the game. The intersection of NP-hard and NP is the class of NP-complete problems. An example of an NP-complete problem is the **balanced number partitioning problem**: given n positive integers, divide them into two subsets A, B, so that the sum in A and the sum in B are as close as possible. A first shot: chose the largest remaining number and distribute it to alternatively to the two sets. We all feel that it is harder to **find a solution to a problem** rather than to **verify a solution**. If $N \neq NP$ there are one way functions, functions which are easy to compute but hard to verify. For some important problems, we do not even know whether they are in NP. Here are two examples: 1) **the integer factoring problem**: given n find the factors 2) **the merit factor problem**: minimize $\sum_{k=-n}^{n} c_k^2$, where $c_k = \sum_{j=0}^{n-k} a_j a_{j+k}$ An efficient algorithm for the first one would have enormous consequences for ou modern lives.
Finally, lets look at some mathematical problems in artificial intelligence AI:

problem solving	playing games like chess, performing algorithms, solving puzzles
pattern matching	speech, music, image, face, handwriting, plagiarism detection, spam
reconstruction	tomography, city reconstruction, body scanning
research	computer assisted proofs, discovering theorems, verifying proofs
data mining	knowledge acquisition, knowledge organization, learning
translation	language translation, porting applications to programming languages
creativity	writing poems, jokes, novels, music pieces, painting, sculpture
simulation	physics engines, evolution of bots, game development, aircraft design
inverse problems	earth quake location, oil depository, tomography
prediction	weather prediction, climate change, warming, epidemics, supplies

We had started with basic human activities defining mathematical fields, we end the course with mathematical activities defining some aspects of computing. Our journey through math is over.