COMPUTABILITY: Some history

David Hilbert (1862–1943) wanted to clarify the methods of mathematical reasoning by creating a *consistent* and *complete* formal axiomatic proof system for all of mathematics.

- A *formal axiomatic proof system* is a precisely defined set of rules consisting of postulates and methods of inference. Here "precisely defined" means that the rules can be defined similarly as the syntax of a programming language (for example, using a grammar).
- *Consistence* means that it is not possible to prove an assertion and the negation of the same assertion. (Note that in an inconsistent system you can prove any assertion! Why?)
- *Completeness* means that if you take any meaningful assertion A, within the system you can prove either A or (not$A$).

To the surprise of everyone, **Kurt Gödel** showed in 1931 that the so called "Hilbert's program" cannot be realised. In fact, Gödel showed that any formal axiomatic proof system dealing just with elementary number theory (the natural numbers with addition and multiplication) cannot be both consistent and complete. That is, in any consistent system there must be true statements that are not provable. Consistency is a minimum requirement since otherwise the system could prove anything!

The fundamental idea of Gödel's result is *self-reference.* Gödel's proof involves arithmetization of logical statements and is not easy to follow.

Alan Turing used in 1936 a completely different approach to prove unsolvability of certain algorithmic problems. However, the fundamental idea of self-reference remains the same. Turing's result is much more transparent, mainly because he uses a nice formal model for algorithms. This is the model later named as *Turing machines.* The impossibility of Hilbert's goal follows also from Turing's argument.

The Turing machine model is by no means the only theoretical "general purpose" model for algorithms. It is, perhaps, the most commonly used and, from a theoretical point of view, the most convenient model to be used both in computability theory and in complexity theory. The reasons for the above claims will be discussed during the course.

More information on the history can be found in

- G.J. Chaitin: *Thinking about Gödel and Turing, Essays on Complexity,* World Scientific, 2007.

Turing machines

Alan Turing developed the model now called *Turing machine* in 1935–36, that is, about 10 years *before* first electronic computers were built. Turing used the model to study the limits of what he called "mechanical computation".

In spite of it's simplicity, the Turing machine model is very powerful: according to the *Church-Turing thesis* every algorithmically computable function can be computed by a Turing machine. However, the Church-Turing thesis is a "thesis" and not a "theorem" because it is impossible to prove to be correct due to the fact that the notion "algorithmically computable function" is only intuitive and does not have a formal definition.[1]

TM's can be viewed as an extension of finite automata or pushdown automata:

1. The read-write head can move in either direction.

2. The head can change symbols ("write") on the tape.

Turing machines are used as a model for general-purpose computers. We will go through the formal definition of a TM in the coming week.

[1] When discussing the Church-Turing thesis we should be careful to restrict consideration to computational problems that are defined simply as a function that associates an output to a set of inputs that are all given at the start of the computation. We will return to this issue later during the course.

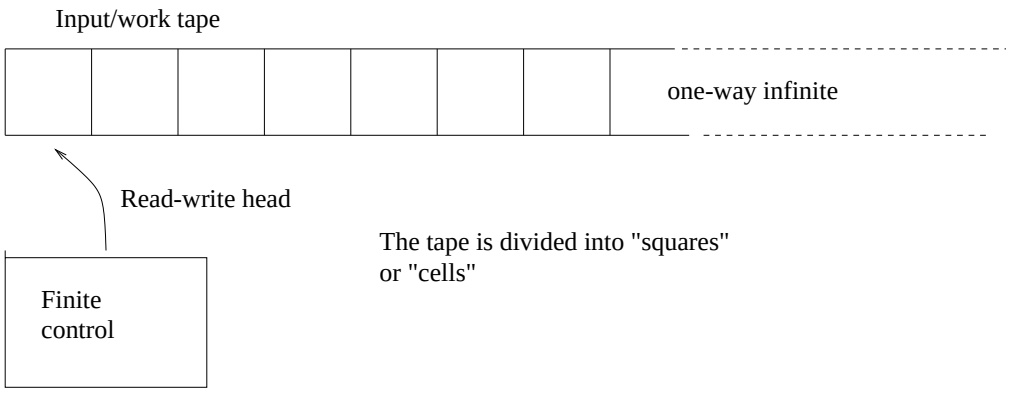

Figure 1: Turing machine (basic) model

On the other hand, TM's are quite clumsy as a practical model of an algorithm and it is cuite cumbersome to give precise descriptions of TM's that solve non-trivial problems. We will go through several TM implementations of various algorithms. This is done mainly to get ourselves acquainted with the TM model. The textbook has good and readable descriptions of TM implementations of algorithms that perform some (relatively) non-trivial tasks.

In some sense, TM's can be viewed as a very low-level programming language. The fact that the model is very simple makes it hard to explicitly construct machines that solve problems of interest, however, the simple definition makes the analysis of TMs more manageable.

COMPUTATIONAL COMPLEXITY

Computability theory wants to determine which algorithmic problems are solvable, in principle, on a computer. Computability is not concerned with time (or other resources) used by an algorithm. As long as we have an algorithm that eventually terminates with the correct answer, the problem is considered "solvable" in computability – it does not make any difference if running the algorithm already on inputs of small size to longer than the age of the universe. When we want a more realistic answer whether or not an algorithmic problem has a feasible solution, we need to consider the computational complexity of problems.

The goal of *complexity theory* is to determine the computational resources required to

solve important computational problems and to classify problems according to their inherent difficulty. The resources typically considered include computational time (that is, number of computation steps), space (that is, the amount of memory used) and circuit size (that is, available hardware).

When we are given a specific algorithm for a computational problem, this naturally gives an upper bound for the time or space complexity of the problem. The main challenge in computational complexity is to prove lower bounds, that is, to prove that certain problems cannot be solved without expending large amounts of resources. It is possible to prove that inherently difficult problems exist. That is, we can prove that certain problems have arbitrarily "bad" time/space complexity. However, the proof involves (resource bounded) diagonalization and the problems are artificially constructed. It turns out to be much more difficult to prove complexity lower bounds for naturally defined and useful problems.

In many cases we can provide *strong evidence* of intractability using notions of reducibility and completeness. Important computational models include *nondeterministic* and *probabilistic* machines. A nondeterministic machine can be viewed as a deterministic machine that only needs to verify the correctness of solutions, instead of finding the solutions. It seems reasonable to expect (at least when thinking of natural hard combinatorial search problems) that it is more difficult to find a solution to a problem than it is to check the correctness of a proposed solution. Or, when thinking of mathematical theorems the same question can be formulated as follows: is it more difficult to find a proof of a theorem than it is to verify that a given proof is correct. Showing nondeterministic algorithms to be more powerful than corresponding deterministic algorithms using the same amout of time/space, would confirm this "intuitive feeling" as a fact of life. The philosophical importance of the well known P vs. NP question is based on above types of considerations.

Algorithmic problems

An *algorithmic problem* consists of a (usually infinite) set of inputs each of which is associated with a unique output. An algorithmic problem can be viewed as a function

$$S_{\text{inputs}} \to S_{\text{outputs}}$$

from the set of inputs onto the set of outputs. The sets of inputs and outputs are typically encoded as sets of finite strings over an alphabet Σ.

As a special case we can consider *decision problems* where the set of outputs has only two values "yes" and "no" (or 0 and 1). Typically when dealing with computability or complexity questions it is possible to reduce the problem to a "corresponding" decision problem, i.e., we get essentially similar type results by restricting consideration to algorithmic problems with only "yes" and "no" outputs.

It makes life easier if we restrict consideration to decision problems because then, instead of talking about functions, we can identify the problem with *a language* (or set of strings). The language consists of inputs corresponding to the "yes" value.

Intuitively, by an *(informal) algorithm* we mean a set of instructions which can be used to solve a given problem in a systematic way. We typically would want such a set of instructions I_S to have at least the following properties in order to call it an algorithm.

Definition 1. `Informal definition of an algorithm.`

(A1) *I_S is finite (a finite number of atomic instructions)*

(A2) *The instructions of I_S are applied deterministically, i.e., at each stage of the computation the next instruction is uniquely determined.*

(A3) *The instructions are capable of solving any instance of the problem.*

(A4) *The instructions are* terminating*, that is, for each legal input the result is obtained after a finite number of applications of the operations.*

From (A4) a natural question arises: How many steps, or applications of the instructions are needed on a given input of given size? This is informally referred to as *time complexity* of the problem. Alternatively, in complexity considerations we can measure other resources such as the number of memory cells used by the computation (or *space complexity*), or the size of the algorithm itself.

Note that time/space complexity is typically measured as a function of the size of the input. We will be interested in *asymptotic complexity,* i.e., the question how the complexity function behaves for inputs of large size.

Furthermore, each type of complexity can be measured either for *worst-case* or *average-case* inputs. Usually researchers concentrate on worst-case complexity (this applies also to the present course). As we will see, already determining the precise relationships of the fundamental worst-case complexity classes is quite challenging. Questions dealing with average-case complexity are much harder and very little is known on this topic.

Finally, we can note that many (in fact "most", in a precisely quantifiable sense) decision problems, that is, functions $f_{dp} : \Sigma^* \to \{0, 1\}$ are, in fact, *uncomputable.* That is, such problems cannot be solved by any algorithm independently of the amount of resources available. We will go over this result, as well as other aspects of uncomputability, in the early part of the course.

Formalization of the notion of algorithm

Our intuitive definition of an algorithm (Definition 1) is sufficient e.g. when we want to show that

- a given problem has an algorithmic solution, or,
- a given problem can be solved in polynomial time (using a particular set of instructions).

On the other hand, the informally defined notion of algorithm is not precise enough if we

want to

- obtain *lower bounds* for the complexity of some problem, that is, show that the problem has no algorithmic solution using time (or space) less than function f, or,
- show that a given problem has no algorithmic solution.

To prove that no algorithm solves a problem we have to know *precisely what are all the algorithms!* In order to formalize the notion of algorithms we will use the *Turing machine (TM)* model discussed in the next subsection.

As is well known, all basic questions related to correctness of computations of the TM model (such as the halting problem, equality of accepted languages, etc.) are *unsolvable,* in particular, it is not possible to determine whether a TM is an algorithm in the sense of Definition 1.

We want to argue that the uncomputability issues are not caused by any *deficiency* in the TM model, and in fact, there does not exist any finitely specified formalization of algorithms that would satisfy conditions (A1)–(A4) of Definition 1. The notion of finitely specified formalization is defined below.

Definition 2. *A* finitely specified *formalization of algorithms encodes each algorithm effectively as a finite string. This means the following:*

- *for each algorithm the corresponding instruction set I_S is encoded as a string $w \in \Omega^*$, where given w the instruction set I_S can be constructed,*
- *there is an algorithm that for a given a string $w \in \Omega^*$ decides whether or not w encodes some algorithm (in our formalization).*

The above conditions seem quite reasonable if we think of representations of algorithms as programs. They just say that each program has finite length when written over a fixed character set and there is an algorithm that decides whether or not a given character string

encodes a legal program. (The syntactic correctness of programs written in any standard programming language is verified by a compiler.)

We have the following negative result:

Theorem 1. *There is* no *finitely specified formalization of the informal definition of algorithms.*

The method used to prove Theorem 1 is called *diagonalization* (which is a specific form of self-reference that was mentioned earlier.) We will discuss diagonalization in detail in the context of the Turing machine model.

Due to the diagonalization problem, when formalizing algorithms we need to drop the requirement that the algorithms must terminate (condition (A4)). Computations of the Turing machine model are not guaranteed to terminate and, in fact, it is an unsolvable problem to determine whether at Turing machine halts on a given input. Theorem 1 indicates that this is not caused by some "defect" in the Turing machine model – according to Theorem 1 there simply does not exist any finitely specified formalization of algorithms where the algorithms are additionally required to terminate.

In theoretical considerations it is sometimes useful to consider also *nondeterministic algorithms* where we additionally drop condition (A2).