1 Introduction to Complexity Theory

"Complexity theory" is the body of knowledge concerning fundamental principles of computation. Its beginnings can be traced way back in history to the use of asymptotic complexity and reducibility by the Babylonians. Modern complexity theory is the result of research activities in many different fields: biologists studying models for neuron nets or evolution, electrical engineers developing switching theory as a tool to hardware design, mathematicians working on the foundations of logic and arithmetics, linguists investigating grammars for natural languages, physicists studying the implications of building Quantum computers, and last but not least, computer scientists searching for efficient algorithms for hard problems. The course will give an introduction to some of these areas.

In this lecture we introduce the notation and models necessary to follow the rest of the course. First, we introduce some basic notation. Afterwards, we discuss the question "what is computation?", followed by definitions of various types of Turing machines. Afterwards, we introduce basic complexity classes for this machine and study their relationships. (??)

1.1 Basic notation

Set notation

Basically, a *set* is a collection of elements without repetition. Finite sets may be specified by listing their members between brackets. For example, $\{0, 1\}$ is a set. The most important sets are the set of all natural numbers $\mathbb{N} = \{1, 2, 3, \ldots\}$ and the set of real numbers $\mathbb{R}$. We also specify sets by a *set former*:

$$\{x : \; P(x)\} \quad \text{or} \quad \{x \in S : \; P(x)\} \; ,$$

where $P()$ is a predicate that can be either true or false. Thus, $\{x : \; P(x)\}$ is the set of all elements x such that $P(x)$ is true.

If x is a member of S, we write $x \in S$. If every member of A is a member of B, we write $A \subseteq B$ and say A is *contained* in B. In case that $A \subseteq B$ but $A \neq B$, we write $A \subset B$. Sets A and B are *equal* if and only if $A \subseteq B$ and $B \subseteq A$. The *cardinality* of a set S, denoted by $|S|$, is the number of members it contains. A set if *enumerable* if its elements can be enumerated in such a way that every element has a finite number of predecessors. Obviously, every finite set is enumerable.

The usual operations defined on sets are:

- $A \cup B = \{x : \; x \in A \text{ or } x \in B\}$, called the *union* of A and B
- $A \cap B = \{x : \; x \in A \text{ and } x \in B\}$, called the *intersection*) of A and B,
- $A - B = \{x : \; x \in A \text{ and } x \notin B\}$, called the *difference* between A and B,
- $A \times B = \{(x, y) : \; x \in A \text{ and } y \in B\}$, called the *Cartesian product* of A and B, and
- $2^A = \{B : \; B \subseteq A\}$, called the *power set* of A.

Strings, alphabets, and languages

A *symbol* is an abstract, atomic entity. Letters and digits are examples of frequently used symbols. A *string* is a finite sequence of symbols juxtaposed. For example, a, b, and c are symbols and $abcb$ is a string. The *length* of a string w, denoted $|w|$, is the number of symbols composing the string. The empty string, denoted by ϵ, is the string consisting of zero symbols. Thus $|\epsilon| = 0$. The *concatenation* of two strings is the string formed by writing the first, followed by the second, with no intervening space. For example, the concatenation of aaa and bb is $aaabb$. The juxtaposition is used as the concatenation operator. That is, if v and w are strings, then vw is the concatenation of these two strings. The empty string is the identity for the concatenation operator. That is, $\epsilon w = w\epsilon = w$ for each string w.

An *alphabet* is a finite set of symbols. A *language* is a set of strings of symbols from some alphabet. Examples for languages are the empty set, $\emptyset$, the set consisting of the empty string $\{\epsilon\}$, or the set $\{a, ab, aabc\}$. Note that the first two sets are distinct. Given an alphabet Σ, Σ^* denotes the set of all possible strings that can be composed from the symbols in Σ, and $\Sigma^+ = \Sigma^* - \{\epsilon\}$. For example, if $\Sigma = \{0, 1\}$ then $\Sigma^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, \ldots\}$.

Relations and functions

A (binary) *relation* is a set of pairs. The first component of each pair is chosen from a set called the *domain*, and the second component of each pair is chosen from a (possibly different) set called the *range*. Often, the domain and the range are the same set S. In that case we say the relation is *on* S. If R is a relation and (a, b) is a pair in R, then we often write aRb.

We say a relation R on set S is

- *reflexive* if aRa for all $a \in S$,
- *irreflexive* if aRa is false for all $a \in S$,
- *transitive* if aRb and bRc imply aRc,
- *symmetric* if aRb implies bRa,
- *asymmetic* if aRb implies that bRa is false, and
- *antisymmetric* if aRb and bRa implies that $a = b$.

A relation R is called an *equivalence relation* if R is reflexive, symmetric, and transitive. An example for an equivalence relation is the mathematical symbol "$=$". R is called an *order* if R is reflexive, antisymmetric, and transitive. An example for an order is the mathematical symbol "$\leq$".

A relation R is a *partial mapping* from M to N if for every $x \in M$ there is at most one $y \in N$ with xRy. R is a *mapping* if for every $x \in M$ there is exactly one $y \in N$ with xRy. Partial mappings or mappings R are often represented as functions of the form $f_R : M \to N$ with the property that for every $(x, y) \in R$, $f_R(x) = y$. We will only use this notation in the following.

A mapping $f : M \to N$ is called

- *injective* if for all $x_1, x_2 \in M$ with $x_1 \neq x_2$ we have $f(x_1) \neq f(x_2)$,

- *surjective* if for every $y \in N$ there is at least one $x \in M$ with $f(x) = y$, and
- *bijective* if f is injective and surjective (that is, for every $y \in N$ there is exactly one $x \in M$ with $f(x) = y$).

1.2 What is computation?

Before we start with defining any computational models, we will discuss the question "what is computation?".

From a logical point of view, computation is the process by which to produce an answer to a question:

$$\text{question} \xrightarrow{\text{computation}} \text{answer}$$

To ensure that all parts of this process (the question, the computation, and the answer) are finite, it must be possible to present the question in finite time, to do a computation in finite time, and to read the answer in finite time. Using human standards, all questions that can be presented in finite time must be pairwise distinguishable by a human in finite time. Or in other words, a human must be able to determine in finite time whether two questions are different or the same. Since the limits to human cognition are finite, it can be argued that the set of all possible questions consists of finite, enumerable elements. The same holds for the answers. Furthermore, it can be shown that any infinite set of enumerable elements is equal to the set of natural numbers (in a sense that there is a bijection between the set of enumerable elements and the set of natural numbers). Thus, a computation can be viewed as a transformation from a natural number q (the question) to a natural number a (the answer).

Computation always requires a "computer", i.e., a system that performes the computation. A *configuration* of a computer is a complete description of the state of such a system at some time point. The initial configuration of the computer is the question plus the initial part of its state that is independent of the question (this usually consists of the computational rules or algorithms the computer will apply in order to answer the question). The final configuration of the computer contains the answer. Note that a computer may not necessarily always reach a final configuration (that is, there may be questions for which it runs forever). Mathematically, a computer can be viewed as a (partial) mapping that maps the set of questions to the set of answers, i.e. a function of the form $f : Q \to A$, (or, using our arguments above, simply $f : \mathbb{N} \to \mathbb{N}$).

A fundamental problem has been: is there a "universal" model for computation? Or in other words, is there a model of a computer that ensures that for every function f that is "computable" there is a computer with an initial configuration (independent of the question) so that for every $q \in Q$ it outputs $f(q)$ in finite time (if $f(q)$ exists)? The main difficulty in answering this problem is that we simply do not know how powerful computation can be. Obviously, any realizable computer must be a finite physical system. The operation of any such system is limited by the physical laws. However, as long as we do not have a complete picture of the physical laws, we do not know what kind of computations, or changes in state, are possible.

All computational models developed so far (including Quantum computing models) have been shown to be computationally equal to a very simple model, the so-called Turing machine. This lead Church and Turing in 1936 to the conjecture that the limitations on what can be computed do not depend on the understanding of physics, but are universal. This is called the *Church-Turing hypothesis*:

> *Every computable function can be computed by a Turing machine.*

One can formulate a more restrictive version of this hypothesis:

> *Every efficiently computable function can be efficiently computed by a Turing machine.*

Whereas this was shown to be true for all classical computational models, it is most likely not true for the Quantum computational models, since there is strong evidence that Quantum effects cannot be simulated efficiently by a (classical) Turing machine. However, since we are still far away from building a Quantum computer, the hypothesis still holds for all existing systems.

1.3 Decision problems

An important class of problems form the so-called decision problems. A *decision problem* is a function $f : Q \to A$ where $A = \{\text{“yes”}, \text{“no”}\}$. If we have a decision problem, we can simplify its representation to a set S containing only those questions q for which $f(q) =$“yes”. Such a set is also called a language. If we assume that Q is enumerable, then any such language is obviously enumerable. We say that a computer *accepts* a language L if it accepts exactly those questions q with $q \in L$. (For the other questions, it may either reject or run forever.) With this definition one can show that if a decision problem is acceptable and Q can be enumerated by some computer, there is also a computer that can enumerate all the elements for which the answer is “yes”. (This is actually one of the assignments.) Therefore, acceptable decision problems have also been called *computably enumerable*. A language L is called *computable* if there is a computer that accepts L and halts for all questions. A fundamental problem for decision problems has been whether every enumerable decision problem is also computably enumerable. In order to be able to answer this question, we need a formal model for a computer that will be introduced now.

1.4 Turing machine models

In the following we present different variants of the Turing machine that is accepted as a universal model for the study of decision problems. (Although in the recent advent of Quantum computing this might change.) In particular, the Turing machine is equivalent in computing power to the digital computer as we know it and also to all (classical) mathematical notions of computation such as λ-calculus (Church, 1941), recursive functions (Kleene, 1952), and Post systems (Post, 1943).

A formal model for an effective procedure should possess certain properties. First, each procedure should be finitely describable. Second, the procedure should consist of discrete steps, each of which can be carried out mechanically. Such a model was introduced by Alan Turing in 1936. We present a variant of it here.

The basic Turing machine

The basic model, illustrated in Figure 1, has a finite control, an input tape that is divided into cells, and a tape head that scans one cell of the tape at a time. The tape is infinite in both directions. Each cell of the tape may hold exactly one of a finite number of tape symbols. Initially, only the input is stored on the tape, surrounded by an infinity of cells that hold the blank symbol B. This is a special symbol that is not an input symbol.

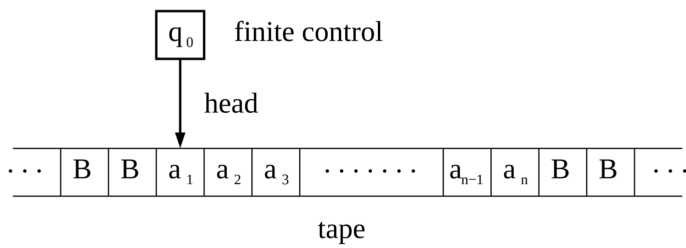

Figure 1: Initial state of the Turing machine.

In one move the Turing machine, depending upon the symbol scanned by the tape head and the state of the finite control,

1. changes state,
2. prints a symbol on the tape cell scanned, replacing what was written there, and
3. moves its head left or right one cell, or does no move.

Formally, a Turing machine (TM) is denoted

$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F) \ ,$$

where

- Q is the finite set of *states*,
- Γ is the finite set of allowable *tape symbols*,
- B, a symbol of Γ, is the blank symbol,
- Σ, s subset of Γ not including B, is the set of *input symbols*,
- δ is the *transition function*, a mapping from $Q \times \Gamma$ to $Q \times \Gamma \times \{L, N, R\}$ (δ may be undefined for some arguments),
- $q_0 \in Q$ is the *start state*, and
- $F \subseteq Q$ is the set of *final states*.

We denote the *configuration* of a Turing machine M by $(q, \alpha_1, \alpha_2) \in Q \times \Gamma^* \times \Gamma^*$. q is the current state of M, $\alpha_1\alpha_2$ is the string in Γ^* that is the contents of the tape up to the leftmost and rightmost nonblank symbol, and the first element in α_2 represents the current position of the head.

We define a *move* of M as follows. Let (p, α_1, α_2) be a configuration. Suppose that, for instance, $\alpha_2 = x\beta$ and $\delta(p, x) = (q, y, R)$. Then we write

$$(p, \alpha_1, \alpha_2) \stackrel{M}{\rightarrow} (q, \alpha_1 y, \beta) \ .$$

If configuration (p, α_1, α_2) yields configuration (q, β_1, β_2) in k steps, we write $\stackrel{M^k}{\rightarrow}$. Furthermore, if one configuration results from another by some finite number of moves, we write $\stackrel{M^*}{\rightarrow}$. In cases where there is no risk of confusion, we will simply write $\rightarrow$ instead of $\stackrel{M}{\rightarrow}$ or $\stackrel{*}{\rightarrow}$ instead of $\stackrel{M^*}{\rightarrow}$. If the Turing machine enters a state that has no following state, because δ is undefined for it, the Turing machine halts.

The *language accepted by M*, denoted $L(M)$, is the set of those words in Σ^* that cause M to enter a final state. Formally,

$$L(M) = \{w : \ w \in \Sigma^* \text{ and } (q_0, \epsilon, w) \stackrel{*}{\rightarrow} (p, \alpha_1, \alpha_2) \text{ for some } p \in F \text{ and } \alpha_1, \alpha_2 \in \Gamma^*\} \ .$$

Given a TM accepting a language L, we assume without loss of generality that the TM halts whenever the input is accepted. However, for words not accepted, it is possible that the TM will never halt. If a Turing machine M halts on all inputs, we say that M *decides* $L(M)$.

Multi-tape Turing machines

A multi-tape Turing machine consists of a finite control with k tape heads and k tapes; each tape infinite in both directions (see Figure 2). On a single move, depending on the state of the finite control and the symbol scanned by each of the tape heads, the machine can:

1. change state,

2. print a new symbol on each of the cells scanned by its tape heads, and

3. move each of its tape heads, independently, one cell to the left or right, or keep it stationary.

Initially, the input appears on the first tape, and the other tapes are blank. The rest is a straightforward generalization of the definition of a single-tape TM. In the following, if we simply speak about a Turing machine, we will always mean a multi-tape TM.

Recursively enumerable and recursive languages

A language that is accepted by a Turing machine is said to be *recursively enumerable* (r.e.). The term "enumerable" derives from the fact that it is precisely these languages whose strings can be enumerated (resp. listed) by a Turing machine. "Recursively" is a mathematical term predating the computer, and its meaning is similar to what the computer scientist would call "recursion". Since, in contrast to Section 1.3, the term computably enumerable is well-defined

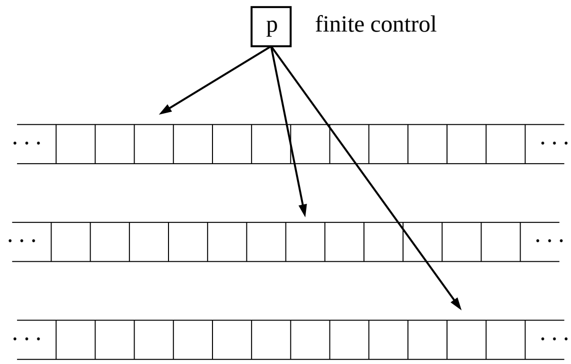

Figure 2: A multi-tape Turing machine.

in the context of Turing machines, we would be now able to study the problem whether all languages are recursively enumerable. This, however, will be postponed to a later lecture.

An important subclass of the r.e. sets are the so-called *recursive sets*, which are those languages accepted by at least one Turing machine that halts on all inputs (note that halting may or may not be preceded by acceptance). Within the recursive sets, languages are usually classified according to their time and/or space requirements.

Time complexity

Consider some TM M. If for every input word of length n, M makes at most $t(n)$ moves before halting, then M is said to be a $t(n)$ *time-bounded Turing machine*, or *of time complexity* $t(n)$, and the language accepted by M is said to be of *time complexity* $t(n)$. The family of languages of time complexity $O(t(n))$ is denoted by $\mathrm{DTIME}(t(n))$. Such a family forms a *complexity class*. The most important time complexity class is the class P that is defined as

$$\mathrm{P} = \bigcup_{k \geq 1} \mathrm{DTIME}(n^k) \ .$$

Intuitively, P can be viewed as the class of all languages that can be solved efficiently in a deterministic way. If a language is in P, we say that it can be decided in *(deterministic) polynomial time.*

Space complexity

If for every input word of length n, M visits at most $s(n)$ cells before halting, then M is said to be an $s(n)$ *space-bounded Turing machine*, or *of space complexity* $s(n)$, and the language accepted by M is said to be of *space complexity* $s(n)$. The family of languages of space complexity $O(s(n))$ is denoted by $\mathrm{DSPACE}(s(n))$. The most important space complexity class is the class PSPACE that is defined as

$$\mathrm{PSPACE} = \bigcup_{k \geq 1} \mathrm{DSPACE}(n^k) \ .$$

If a language is in PSPACE, we say that it can be decided in *(deterministic) polynomial space.*

1.5 The probabilistic Turing machine

Apart from the basic, so-called *deterministic Turing machine* (DTM), other Turing machine models have been constructed to include various characteristics. The most important of these models are the probabilistic Turing machine, the nondeterministic Turing machine, and the Quantum Turing machine

The single-tape *probabilistic Turing machine* (PTM) is defined similar to the single-tape DTM:

$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F) \ ,$$

The only difference is that δ is now a mapping from $Q \times \Gamma$ to $2^{Q \times \Gamma \times \{L,N,R\}}$. That is, a move does no longer have a unique outcome, but can have a set of outcomes that are equally likely to be selected. Or, in other words, the PTM can have many different computational paths for the same input. Thus, in contrast to the DTM presented previously, it can now happen that for some PTM M and input x there are computations for which M started on x halts, and computations for which M started on x does not hold. Furthermore, there may be computations for which M started on x does accept, and for which M started on x does not accept. Single-tape PTMs generalize to multi-tape PTMs in the same way as Single-tape DTMs generalize to multi-tape DTMs.

Several time complexity classes have been defined for (multi-tape) PTMs. We present the two most important of these.

- RP denotes the class of all languages L for which there is a polynomial time PTM so that
  - for every $x \in L$, Pr[M accepts x] $\geq 1/2$, and
  - for every $x \notin L$, Pr[M accepts x] $= 0$.
- BPP denotes the class of all languages L for which there is a polynomial time PTM so that
  - for every $x \in L$, Pr[M accepts x] $\geq 2/3$, and
  - for every $x \notin L$, Pr[M accepts x] $\leq 1/3$.

It is not difficult to show that P $\subseteq$ RP $\subseteq$ BPP and that BPP $\subseteq$ PSPACE. BPP can be viewed as the set of all languages that can be decided efficiently by todays computers in polynomial time. Important open problems are whether or not P = RP or RP = BPP.

1.6 The nondeterministic Turing machine

The (multi-tape) nondeterministic Turing machine (NTM) is defined in the same way as the (multi-tape) PTM. The only difference lies in how it accepts a word. We say that a NTM M accepts input x if at least one computation of M started on x leads to a final state. M *decides* a language L if all M halts on all inputs on all computational paths and accepts exactly those inputs x with $x \in L$.

Time complexity

Consider some NTM M. If for every input word of length n, M makes at most $t(n)$ moves before halting, no matter which computational path it selects, then M is said to be a $t(n)$ *time-bounded NTM*, and the language accepted by M is said to be of *nondeterministic time complexity* $t(n)$. The family of languages of nondeterministic time complexity $O(t(n))$ is denoted by NTIME$(t(n))$. The most important nondeterministic time complexity class is the class NP that is defined as

$$\mathrm{NP} = \bigcup_{k \geq 1} \mathrm{NTIME}(n^k) \ .$$

If a language is in NP, we say that it can be decided in *nondeterministic polynomial time*. It is not hard to see that RP $\subseteq$ NP and that NP $\subseteq$ PSPACE. Furthermore, it is widely believed that P $\neq$ NP and NP $\neq$ PSPACE, but nobody can prove it yet.

Space complexity

If for every input word of length n, M visits at most $s(n)$ cells before halting, no matter which computational path it selects, then M is said to be an $s(n)$ *space-bounded NTM*, and the language accepted by M is said to be of *nondeterministic space complexity* $s(n)$. The family of languages of nondeterministic space complexity $O(s(n))$ $s(n)$ is denoted by NSPACE$(s(n))$. The most important nondeterministic space complexity class is the class NSPACE that is defined as

$$\mathrm{NPSPACE} = \bigcup_{k \geq 1} \mathrm{NSPACE}(n^k) \ .$$

If a language is in NPSPACE, we say that it can be decided with *nondeterministic polynomial space*. Obviously, PSPACE $\subseteq$ NPSPACE.

1.7 The Quantum Turing machine

The Quantum Turing machine (QTM) is a Turing machine that can perform Quantum mechanical operations. It will be described in more detail later. We only mention here that QP is the class of all languages that can be by some Quantum Turing machine in polynomial time.

1.8 Turing machines as algorithms

Turing machines are ideal for solving string problems (such as, for instance, the problem given in the assignment). But how about developing algorithms capable of attacking problems whose instances are mathematical objects such as graphs, networks, and numbers? To solve such a problem by a Turing machine, we must decide how to *represent* by a string an instance of the problem. Once we have fixed this representation, an algorithm for a decision problem can be easily transformed into a Turing machine that decides the corresponding language. It should be clear that every finite mathematical object can be represented by a finite string over an appropriate alphabet. For example, elements of finite sets, such as the nodes of a graph, can be represented as integers in binary. Pairs and k-tuples of simpler mathematical objects are represented by using set brackets, and so on. Or, perhaps, a graph can be represented by its adjacency matrix.

There is a wide range of acceptable representations of integers, finite sets, graphs, and other elementary objects. All of these acceptable encodings have to be polynomially related. That is, if A and B are both reasonable representations, then $|A|$ should be at most polynomially larger than $|B|$. Representing numbers in unary, for instance, is not an acceptable representation, since it requires exponentially more symbols than the binary representation. For the mere question of computability, representation is not an important issue, but in order to determine whether a problem is *efficiently* computable, the encoding has to be acceptable.

Having this in mind, we will work in the following with mathematical objects instead of strings and with algorithms instead of Turing machines.

1.9 References

- R. Gandy. The confluence of ideas in 1936. In: *The Universal Turing Machine: A Half-Century Survey*, R. Herken (ed.), Oxford Press, Oxford, 1988, pp. 55-111.

- J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison Wesley, Reading, 1979.

- C.H. Papadimitriou. *Computational Complexity.* Addison Wesley, Reading, 1994.