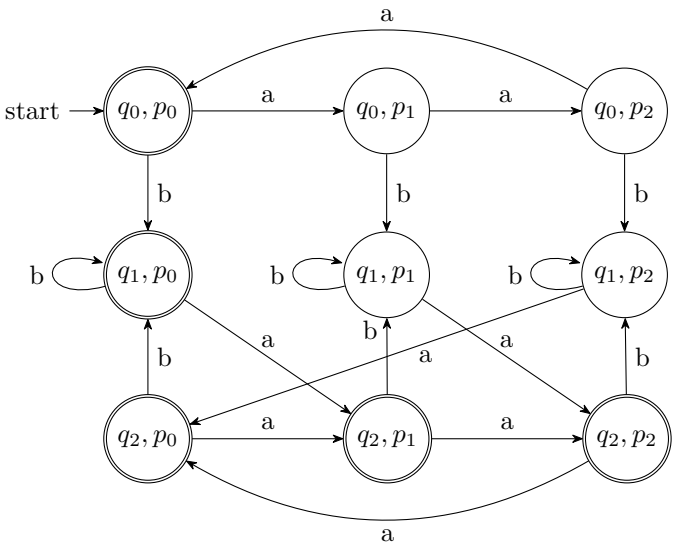
Theoretical Computer Science - Compact

Janis Hutz
https://janishutz.com

December 31, 2025



"*Sie können also alle C Programme in Kanonischer Ordnung aufzählen. Sollten Sie dies tun? Wahrscheinlich nicht. Was aber zählt ist, sie* ***können*** *es tun*"

- Prof. Dr. Dennis Komm, 2025

HS2025, ETHZ

Compact Summary of the book Theoretische Informatik

by Prof. Dr. Juraj Hromkovic

Contents

1 Introduction

This summary aims to provide a simple, easy to understand and short overview over the topics covered, with approaches for proofs, important theorems and lemmas, as well as definitions.

It does not aim to serve as a full replacement for the book or my main summary, but as a supplement to both of them.

It also lacks some formalism and is only intended to give some intuition, six pages are really not enough for a formal and complete overview of the topic.

As general recommendations, try to substitute possibly "weird" definitions in multiple choice to see a definition from the book.

All content up to Chapter 5.3 is relevant for the midterm directly.

The content for the endterm exam as of HS2025 starts in Chapter 5.3. All prior content is still relevent to the extent that you need an understanding of the concepts treated there

2 Alphabets, Words, etc

2.2 Alphabets, Words, Languages

Definition 2.1: *(Alphabet)* Set Σ. Important alphabets: Σ_{bool}, Σ_{lat} (all latin chars), Σ_{Keyboard} (all chars on keyboard), Σ_m (m-ary numbers)

Definition 2.2: *(Word)* Possibly empty (denoted λ) sequences of characters from Σ. $|w|$ is the length, Σ^* is the set of all words and $\Sigma^+ = \Sigma^* - \{\lambda\}$

Definition 2.3: *(Konkatenation)* $\text{Kon}(x, y) = xy$, (so like string concat). $(xy)^n$ is n-times repeated concat.

Definition 2.4: *(Reversal)* a^R, simply read the word backwards.

Definition 2.6: *(Prefix, Suffix, Subword)* v in $w = vy$; s in $w = xs$; Subword u in $w = xuy$; x, y possibly λ

Definition 2.7: *(Appearance)* $|x|_a$ is the number of times $a \in \Sigma$ appears in x

Definition 2.8: *(Canonical ordering)* Ordered by length and then by first non-common letter:

$$u < v \iff |u| < |v| \vee (|u| = |v| \wedge u = x \cdot s_i \cdot u' \wedge v = x \cdot s_j \cdot v') \text{ for any } x, u', v' \in \Sigma^* \text{ and } i < j$$

Definition 2.9: *(Language)* $L \subseteq \Sigma^*$, and we define $L^C = \Sigma^* - L$ as the complement, with $L_\emptyset$ being the empty language, whereas L_λ is the language with just the empty word in it.

Concatenation: $L_1 \cdot L_2 = \{vw | v \in L_1 \wedge w \in L_2\}$ and $L^{i+1} = L^i \cdot L \ \forall i \in \mathbb{N}$.

Cleen Star: $L^* = \bigcup_{i \in \mathbb{N}} L^i$ and $L^+ = L \cdot L^*$

Of note is that there are irregular languages whose Cleen Star is regular, most notably, the language $L = \{w \in \{0\}^* \mid |w| \text{ is prime}\}$'s Cleen Star is regular, due to the fact that the prime factorization is regular

Lemma 2.1: $L_1L_2 \cup L_1L_2 = L_1(L_2 \cup L_3)$ **Lemma 2.2:** $L_1(K_2 \cap L_3) \subseteq L_1L_2 \cap L_1L_3$

For multiple choice questions, really think of how the sets would look to determine if they fulfill a requirement.

2.4 Kolmogorov-Complexity

Definition 2.17: *(Kolmogorov-Complexity)* $K(x)$ for $x \in (\Sigma_{\text{bool}})^*$ is the minimum of all binary lengths of Pascal programs that output x, where the Program doesn't have to compile, i.e. we can describe processes informally

Lemma 2.4: For each word x exists constant d s.t. $K(x) \leq |x| + d$, for which we can use a program that simply includes a `write(x)` command

Definition 2.18: *(Of natural number)* $K(n) = K(\text{Bin}(x))$ with $|\text{Bin}(x)| = \lceil \log_2(x+1) \rceil$

Lemma 2.5: For each $n \in \mathbb{N} \exists w_n \in (\Sigma_{\text{bool}})^n$ s.t. $K(w_n) \geq |w_n| = n$, i.e. exists a non-compressible word.

Theorem 2.1: Kolmogorov-Complexity doesn't depend on programming language. It only differs in constant

Definition 2.19: *(Randomness)* $x \in (\Sigma_{\text{bool}})^*$ random if $K(x) \geq |x|$, thus for $n \in \mathbb{N}$, $K(n) \geq \lceil \log_2(n+1) \rceil - 1$

Theorem 2.3: *(Prime number)* $\lim_{n \to \infty} \frac{\text{Prime}(n)}{\frac{n}{\ln(n)}} = 1$ with $\text{Prime}(n)$ the number of prime numbers on $[0, n] \subseteq \mathbb{N}$

Proofs Proofs in which we need to show a lower bound for Kolmogorov-Complexity (almost) always work as follows: Assume for contradiction that there are no words with $K(w) > f$ for all $w \in W$. We count the number m of words in W and the number n of programs of length $\leq f$ (f being the given, lower bound). We will have $m - n > 0$, which means, there are more different words than there are Programs with Kolmogorov-Complexity $\leq f$, which is a contradiction to our assumption.

There are $\left\lfloor \frac{n}{k} \right\rfloor + 1$ numbers divisible by k in the set $\{0, 1, \ldots, n\}$.

Laws of logarithm

- $\log_a(x) + \log_a(y) = \log_a(x \cdot y)$
- $\log_a(x) - \log_a(y) = \log_a(x \div y)$
- $y \log_a(x) = \log_a(x^y)$
- $\log_a(x) = \frac{\ln(x)}{\ln(a)}$
- $\log_a(1) = 0$

3 Finite Automata

3.2 Representation

We can note the automata using graphical notation similar to graphs or as a series of instructions like this:

$$\begin{aligned} &\texttt{select } input = a_1 \texttt{ goto } i_1 \\ &\vdots \\ &input = a_k \texttt{ goto } i_k \end{aligned}$$

Definition 3.1: *(Finite Automaton)* $A = (Q, \Sigma, \delta, q_0, F)$ with

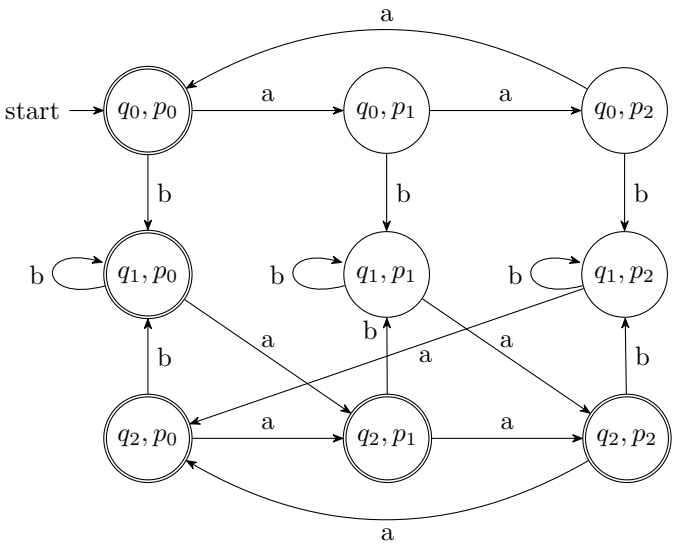
- Q set of states
- Σ input alphabet
- $\delta(q, a) = p$ transition from q on reading a to p
- q_0 initial state
- $F \subseteq Q$ accepting states
- $\mathcal{L}_{EA}$ regular languages (accepted by FA)

$\widehat{\delta}(q_0, w) = p$ is the end state reached when we process word w from state q_0, and $(q, w) \vdash_M^* (p, \lambda)$ is the formal definition, with $\vdash_M^*$ representing any number of steps $\vdash_M$ executed (transitive hull).

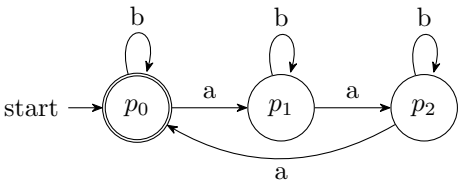
The class $\text{Cl}[q_i]$ represents all possible words for which the FA is in this state. Be cautious when defining them, make sure that no extra words from other classes could appear in the current class, if this is not intended.

Sometimes, we need to combine two (or more) FA to form one larger one. We can do this easily with product automata. To create one from two automata M_1 (states q_i) and M_2 (states p_j) we do the following steps:

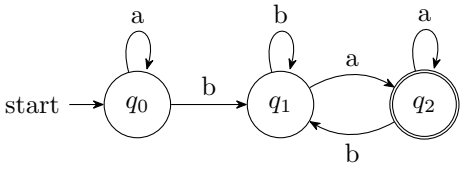
1. Write down the states as tuples of the form (q_i, p_j) (i.e. form a grid by writing down one of the automata vertically and the other horizontally)
2. From each state, the automata on the horizontal axis decides for the input symbol if we move left or right, whereas the automata on the vertical axis decides if we move up or down.



For the automata



(a) Module to compute $|w|_b \equiv |w| (\text{mod } 3)$. States $q \in Q_a$



(b) Module to compute w contains sub. ba and ends in a. States $p \in Q_b$

Figure 3.1: Graphical representation of the Finite Automaton of Task 9 in 2025

3.4 Proofs of nonexistence

We have three approaches to prove non-regularity of words. Below is an informal guide as to how to do proofs using each of the methods and possible pitfalls.

For all of them start by assuming that L is regular.

Lemma 3.3

Regular words — **Lemma 3.3**

Let A be a FA over Σ and let $x \neq y \in \Sigma^*$, such that $\widehat{\delta}_A(q_0, x) = \widehat{\delta}(q_0, y)$. Then for each $z \in \Sigma^*$ there exists an $r \in Q$, such that $xz, yz \in \text{Cl}[r]$, and we thus have

$$xz \in L(A) \iff yz \in L(A)$$

1. Pick a FA A over Σ and say that $L(A) = L$
2. Pick $|Q| + 1$ words x such that $xy = w \in L$ with $|y| > 0$.
3. State that via pigeonhole principle there exists w.l.o.g $i < j \in \{1, \dots, |Q| + 1\}$, s.t. $\widehat{\delta}_A(q_0, x_i) = \widehat{\delta}_A(q_0, x_j)$.
4. Build contradiction by picking z such that $x_i z \in L$.
5. Then, if z was picked properly, since $i < j$, we have that $x_j z \notin L$, since the lengths do not match

That is a contradiction, which concludes our proof

Pumping Lemma

Pumping-Lemma für reguläre Sprachen — **Lemma 3.4**

Let L be regular. Then there exists a constant $n_0 \in \mathbb{N}$, such that each word $w \in \Sigma^*$ with $|w| \geq n_0$ can be decomposed into $w = yxz$, with

(i) $|yx| \leq n_0$
(ii) $|x| \geq 1$
(iii) For $X = \{yx^k z \mid k \in \mathbb{N}\}$ *either* $X \subseteq L$ or $X \cap L = \emptyset$ applies

1. State that according to Lemma 3.4 there exists a constant n_0 such that $|w| \geq n_0$.
2. Choose a word $w \in L$ that is sufficiently long to enable a sensible decomposition for the next step.
3. Choose a decomposition, such that $|yx| = n_0$ (makes it quite easy later). Specify y and x in such a way that for $|y| = l$ and $|x| = m$ we have $l + m \leq n_0$
4. According to Lemma 3.4 (ii), $m \geq 1$ and thus $|x| \geq 1$. Fix z to be the suffix of $w = yxz$
5. Then according to Lemma 3.4 (iii), fill in for $X = \{yx^k z \mid k \in \mathbb{N}\}$ we have $X \subseteq L$.
6. This will lead to a contradiction commonly when setting $k = 0$, as for a language like $0^n 1^n$, we have $0^{(n_0 - m) + km} 1^{n_0}$ as the word (with $n_0 - m = l$), which for $k = 0$ is $u = 0^{n_0 - m} 1^{n_0}$ and since $m \geq 1$, $u \notin L$ and thus by Lemma 3.4, $X \cap L = \emptyset$, but that is also not true, as the intersection is not empty (for $k = 1$)

Kolmogorov Complexity

1. We first need to choose an x such that $L_x = \{y \mid xy \in L\}$. If not immediately apparent, choosing $x = a^{\alpha+1}$ for $a \in \Sigma$ and α being the exponent of the exponent of the words in the language after a variable rename. For example, for $\{0^{n^2+2n} \mid n \in \mathbb{N}\}$, $\alpha(m) = m^2 + 2m$. Another common way to do this is for languages of the form $\{a^n b^n \mid n \in \mathbb{N}\}$ to use $x = a^m$ and $L_{0^m} = \{y \mid 0^m y \in L\} = \{0^j 1^{m+j} \mid j \in \mathbb{N}\}$.
2. Find the first word $y_1 \in L_x$. In the first example, this word would be $y_1 = 0^{(m+1)^2 \cdot 2(m+1) - m^2 \cdot 2m + 1}$, or in general $a^{\alpha(m+1) - \alpha(m) + 1}$. For the second example, the word would be $y_1 = 1^m$, i.e. with $j = 0$
3. According to Theorem 3.1, there exists constant c such that $K(y_k) \leq \lceil \log_2(k+1) \rceil + c$. We often choose $k = 1$, so we have $K(y_1) \leq \lceil \log_2(1+1) \rceil + c = 1 + c$ and with $d = 1 + c$, $K(y_1) \leq d$
4. This however leads to a contradiction, since the number of programs with length $\leq d$ is at most 2^d and thus finite, but our set L_x is infinite.

Minimum number of states

To show that a language needs *at least* n states, use Lemma 3.3 and n words. We thus again do a proof by contradiction:

1. Assume that there exists FA with $|Q| < n$. We now choose n words (as short as possible), as we would for non-regularity proofs using Lemma 3.3 (i.e. find some prefixes). It is usually beneficial to choose prefixes with $|w|$ small (consider just one letter, λ, then two and more letter words). An "easy" way to find the prefixes is to construct a finite automaton and then picking a prefix from each class
2. Construct a table for the suffixes using the n chosen words such that one of the words at entry x_{ij} is in the language and the other is not. ($n \times n$ matrix, see below in example)
3. Conclude that we have reached a contradiction as every field x_{ij} contains a suffix such that one of the two words is in the language and the other one is not.

Example 3.1: Let $L = \{x1y \mid x \in (\Sigma_{\text{bool}})^*, y \in \{0,1\}^2\}$. Show that any FA that accepts L needs at least four states.

Assume for contradiction that there exists EA $A = (Q, \Sigma_{\text{bool}}, \delta_A, q_0, F)$ with $|Q| < 4$. Let's take the 4 words $00, 01, 10, 11$. Then according to Lemma 3.3, there needs to exist a z such that $xz \in L(A) \iff yz \in L(A)$ with $\widehat{\delta}_A(q_0, x) = \widehat{\delta}_A(q_0, y)$ for $x, y \in \{00, 01, 10, 11\}$.

This however is a contradiction, as we can find a z for each of the pairs (x, y), such that $xz \in L(A)$, but $yz \notin L(A)$. See for reference the below table (it contains suffixes z fulfilling prior condition):

	00	01	10	11
00	-	00	0	0
01		-	0	0
10			-	00
11				-

Thus, all four words have to lay in pairwise distinct states and we thus need at least 4 states to detect this language.

3.5 Non-determinism

The most notable differences between deterministic and non-deterministic FA is that the transition function is different: $\delta : Q \times \Sigma \to \mathcal{P}(Q)$. I.e., there can be any number of transitions for one symbol of Σ for each state. This is (in graphical notation) represented by arrows that have the same label going to different nodes.

It is also possible for there to not be a transition function for a certain element of the input alphabet. In that case, regardless of state, the NFA rejects, as it "gets stuck" in a state and can't finish processing.

Additionally, the NFA accepts x if it has at least one accepting calculation on x.

Theorem 3.2: For every NFA M there exists a FA A such that $L(M) = L(A)$. They are then called ***equivalent***

Potenzmengenkonstruktion States are no now sets of states of the NFA in which the NFA could be in after processing the preceding input elements and we have a special state called q_{trash}.

For each state, the set of states $P = \widehat{\delta}(q_0, z)$ for $|z| = n$ represents all possible states that the NFA could be in after doing the first n calculations.

Correspondingly, we add new states if there is no other state that is in the same branch of the calculation tree $\mathcal{B}_M(x)$. So, in other words, we execute BFS on the calculation tree.

4 Turing Machines

4.3 Representation

Turing machines are much more capable than FA and NFA. A full definition of them can be found in the book on pages 96 - 98 (= pages 110 - 112 in the PDF).

For example, to detect a recursive language like $\{0^n1^n \mid n \in \mathbb{N}\}$ we simply replace the left and rightmost symbol with a different one and repeat until we only have the new symbol, at which point we accept, or there are no more 0s or 1s, at which point we reject.

The Turing Machines have an accepting q_{accept} and a rejecting state q_{reject} and a configuration is an element of $\{\{¢\} \cdot \Gamma^* \cdot Q \cdot \Gamma^+ \cup Q \cdot \{¢\} \cdot \Gamma^+\}$ with $\cdot$ being the concatenation and ¢ the marker of the start of the band.

$$\mathcal{L}_{RE} = \{L(M) \mid M \text{ is a TM}\}$$
$$\mathcal{L}_{R} = \{L(M) \mid M \text{ is a TM and it always halts}\}$$

4.4 Multi-tape TM and Church's Thesis

k-Tape Turing machines have k extra tapes that can be written to and read from, called memory tapes. They *cannot* write to the input tape. Initially the memory tapes are empty and we are in state q_0. All read/write-heads of the memory tapes can move in either direction, granted they have not reached the far left end, marked with ¢.

As with normal TMs, the Turing Machine M accepts w if and only if M reaches the state q_{accept} and rejects if it does not terminate or reaches the state q_{reject}

Lemma 4.1: There exists an equivalent 1-Tape-TM for every TM.

Lemma 4.2: There exists an equivalent TM for each Multi-tape TM.

Church's Thesis states that the Turing Machines are a formalization of the term "Algorithm". It is the only axiom specific to Computer Science.

All the words that can be accepted by a Turing Machine are elements of $\mathcal{L}_{RE}$ and are called ***recursively enumerable***.

4.5 Non-Deterministic Turing Machines

The same ideas as with NFA apply here. The transition function also maps into the power set:

$$\delta : (Q - \{q_{\text{accept}}, q_{\text{reject}}\}) \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R, N\})$$

Again, when constructing a normal TM from a NTM (which is not required at the Midterm, or any other exam for that matter in this course), we again apply BFS to the NTM's calculation tree.

Theorem 4.2: For an NTM M exists a TM A s.t. $L(M) = L(A)$ and if M doesn't contain infinite calculations on words of $(L(M))^C$, then A always stops.

5 Computability

5.2 Diagonalization

The ***set of binary encodings of all TMs*** is denoted KodTM and KodTM $\subseteq (\Sigma_{\text{bool}})^*$ and the upper bound of the cardinality is $|(\Sigma_{\text{bool}})^*|$, as there are infinitely many TMs.

Below is a list of countable objects. They all have corresponding Lemmas in the script, but omitted here:

- Σ^* for any Σ
- KodTM
- $\mathbb{N} \times \mathbb{N}$
- $\mathbb{Q}^+$

The following objects are uncountable: $[0, 1]$, $\mathbb{R}$, $\mathcal{P}((\Sigma_{\text{bool}})^*)$

Corollary 5.1: $|\text{KodTM}| < |\mathcal{P}((\Sigma_{\text{bool}})^*)|$ and thus there exist infinitely many not recursively enumerable languages over Σ_{bool}

Theorem 5.3: $L_{\text{diag}} \notin \mathcal{L}_{RE}$

Proof of L (not) recursively enumerable

Proving that a language *is* recursively enumerable is as difficult as providing a Turing Machine that accepts it.

Proving that a language is *not* recursively enumerable is likely easier. For it, let $d_{ij} = 1 \iff M_i$ accepts w_j.

Example 5.1: Assume towards contradiction that $L_{\text{diag}} \in \mathcal{L}_{RE}$. Let

$$\begin{aligned} L_{\text{diag}} &= \{w \in (\Sigma_{\text{bool}})^* \mid w = w_i \text{ for an } i \in \mathbb{N} - \{0\} \text{ and } M_i \text{ does not accept } w_i\} \\ &= \{w \in (\Sigma_{\text{bool}})^* \mid w = w_i \text{ for an } i \in \mathbb{N} - \{0\} \text{ and } d_{ii} = 0\} \end{aligned}$$

Thus assume that, $L_{\text{diag}} = L(M)$ for a Turing Machine M. Since M is a Turing Machine in the canonical ordering of all Turing Machines, so there exists an $i \in \mathbb{N} - \{0\}$, such that $M = M_i$.

This however leads to a contradiction, as $w_i \in L_{\text{diag}} \iff d_{ii} = 0 \iff w_i \notin L(M_i)$.

In other words, w_i is in L_{diag} if and only if w_i is not in $L(M_i)$, which contradicts our statement above, in which we assumed that $L_{\text{diag}} \in \mathcal{L}_{RE}$.

In other, more different, words, w_i being in L_{diag} implies (from the definition) that $d_{ii} = 0$, which from its definition implies that $w_i \notin L(M_i)$. □

Another result (not formally proven in the script, but there is a proof by intimidation) that can come in useful, especially when trying to show $L \notin \mathcal{L}_{RE}$ is:

$$L, L^C \in \mathcal{L}_{RE} \iff L \in \mathcal{L}_R$$

Additionally, as a reminder, $\mathcal{L}_{RE} = \{L(M) \mid M \text{ is a TM}\}$, so to prove that a language $L \notin \mathcal{L}_{RE}$, we only need to show that there exists no TM M, for which $L(M) \in \mathcal{L}_{RE}$.

5.3 Reductions

This is the start of the topics that are explicitly part of the endterm.

For the reductions, it is important to get the order right.
To show that a language L_1 is not part of e.g. $\mathcal{L}_R$, show that there exists a reduction into a language $L_2 \notin \mathcal{L}_R$, i.e. e.g. show $L_2 \leq_R L_1$.
To show that a language L_1 is part of e.g. $\mathcal{L}_R$, show that there exists a reduction into a language $L_2 \in \mathcal{L}_R$, i.e. e.g. show $L_1 \leq_R L_2$.

For a language to be in $\mathcal{L}_R$, in contrast to $L \in \mathcal{L}_{RE}$, the TM has to halt also for `no` instances, i.e. it has to be an algorithm. In other words: A TM A can enumerate all valid strings of a *recursively enumerable language* ($L \in \mathcal{L}_{RE}$), where for *recursive languages*, it has to be able to difinitively answer for both `yes` and `no` and thus halt in finite time for both.

First off, a list of important languages for this and the next section:

- $L_U = \{\text{Kod}(M)\#w \mid w \in (\Sigma_{\text{bool}})^* \text{ and TM } M \text{ accepts } w\}$ ($\in \mathcal{L}_{RE}$, but $\notin \mathcal{L}_R$)
- $L_H = \{\text{Kod}(M)\#x \mid x \in (\Sigma_{\text{bool}})^* \text{ and TM } M \text{ halts on } x\}$ ($\in \mathcal{L}_{RE}$, but $\notin \mathcal{L}_R$)
- $L_{\text{diag}} = \{w \in (\Sigma_{\text{bool}})^* \mid w = w_i \text{ for an } i \in \mathbb{N} - \{0\} \text{ and } M_i \text{ does not accept } w_i\}$ ($\notin \mathcal{L}_{RE}$ and thus $\notin \mathcal{L}_R$)
- $(L_{\text{diag}})^C$ ($\in \mathcal{L}_{RE}$, but $\notin \mathcal{L}_R$)
- $L_{EQ} = \{\text{Kod}(M)\#\text{Kod}(\overline{M}) \mid L(M) = L(\overline{M})\}$ ($\notin \mathcal{L}_{RE}$, and thus $\notin \mathcal{L}_R$)
- $(L_{EQ})^C = \{\text{Kod}(M)\#\text{Kod}(\overline{M}) \mid L(M) \neq L(\overline{M})\}$ ($\notin \mathcal{L}_{RE}$, and thus $\notin \mathcal{L}_R$)
- $L_{\text{empty}} = \{\text{Kod}(M) \mid L(M) = \emptyset\}$ ($\in \mathcal{L}_{RE}$, but $\notin \mathcal{L}_R$)
- $(L_{\text{empty}})^C = \{x \in (\Sigma_{\text{bool}})^* \mid x \notin \text{Kod}(\overline{M}) \forall \text{ TM } \overline{M} \text{ or } x = \text{Kod}(M) \text{ and } L(M) \neq \emptyset\}$ ($\in \mathcal{L}_{RE}$, but $\notin \mathcal{L}_R$)
- $L_{H,\lambda} = \{\text{Kod}(M) \mid M \text{ halts on } \lambda\}$ ($\in \mathcal{L}_{RE}$, but $\notin \mathcal{L}_R$)

An important consequence of the fact that both L_{EQ} and its complement are $\notin \mathcal{L}_{RE}$ is that it is not guaranteed for a language's complement to *necessarily* be in $\mathcal{L}_{RE}$, if the language is not.

Definition 5.3: *(Recursively reducible languages)* $L_1 \leq_R L_2$ (L_1 reducible into L_2), if $L_2 \in \mathcal{L}_R \Rightarrow L_1 \in \mathcal{L}_R$

Definition 5.4: *(EE-Reductions)* $L_1 \leq_{EE} L_2$ if there exists a TM M that implements image $f_M : \Sigma_1^* \to \Sigma_2^*$, for which we have $x \in L_1 \Leftrightarrow f_M(x) \in L_2$ for all $x \in (\Sigma_{\text{bool}})_1^*$

Lemma 5.3: If $L_1 \leq_{EE} L_2$ then also $L_1 \leq_R L_2$

Lemma 5.4: For each language $L \subseteq \Sigma^*$ we have: $L \leq_R L^C$ and $L^C \leq_R L$

Theorem 5.6: *(Universal TM)* A TM U, such that $L(U) = L_U$

Showing reductions First, a general guide to reductions and below what else we need to keep in mind for specific reductions:

1. We construct a TM A that:
   (a) Checks if the input has the right form and if it does not, returns some output that is $\notin L_2$
   (b) Applies the transformation to all remaining input
2. We show $x \in L_1 \Leftrightarrow A(x) \in L_2$ by showing the implications:
   (a) For $\Rightarrow$, we show it directly, by assuming that $x \in L_1$ (obviously) and we can ignore the invalid input (as that $\notin L_1$ anyway)
   (b) For $\Leftarrow$, we have two options (mention what happens to invalid input here):
   - We show $A(x) \in L_2 \Rightarrow x \in L_1$ directly (usually harder)
   - We show $x \notin L_1 \Rightarrow A(x) \notin L_2$ (contraposition)
3. Show that the TM always halts (for P, EE and R reductions at least)

EE-reductions They follow the above scheme exactly

R-reductions It is usually a good idea to draw the setup here. We have a TM C that basically executes an EE-reduction and we have a TM A that can check L_2. Then, we have a TM B that wraps the whole thing: It first executes TM C, which will either output an transformation of L_1 for L_2 (i.e. execute an EE-reduction) or output some encoding for *invalid word*. If it outputs the encoding for *invalid word*, B will output $x \notin L_1$.

If C does not output an encoding for *invalid word*, then B will execute A on the output of C and then use the output of A (either accepting or rejecting) to output the same (i.e. if A accepts, then B will output $x \in L_1$ and if A rejects, B outputs $x \notin L_1$)

Intuition: In R-reductions, we construct a full verifier for L_1 using the verifier for L_2, i.e. we can use TM B directly to check if a given word is in L_1 given that the transformed word is also in L_2.

P-reductions (Used in Chapter 6). We need to also show that A terminates in polynomial time.

Tips & Tricks:

- The TM A has to terminate always
- Check the input for the correct form first
- For the correctness, show $x \in L_1 \Leftrightarrow A(x) \in L_2$
- The following tricks can be useful:
  - Transitions into q_{accept} and q_{reject} can be redirected to $q_{\text{accept}}/q_{\text{reject}}$ or into an infinite loop
  - Construct TM M' that ignores input and does the same, regardless of input
- Generate encoding of a TM with special properties (e.g. accepts all input, never halts, ...)

5.4 Rice's Theorem

Definition 5.7: L is called a ***semantically non-trivial decision problem***, if these conditions apply:

(i) There exists a TM M_1, such that $\text{Kod}(M_1) \in L$ (i.e. $L \neq \emptyset$)

(ii) There exists a TM M_2, such that $\text{Kod}(M_2) \notin L$ (not all encodings are in L)

(iii) For two TM A and B: $L(A) = L(B) \Rightarrow \text{Kod}(A) \in L \Rightarrow \text{Kod}(B) \in L$

Theorem 5.9: *(Rice's Theorem)* Every semantically non-trivial decision problem over TMs is undecidable

Using Rice's Theorem We only need to show that a language is semantically non-trivial, which we do by checking the above conditions. For the third condition, intuitively, we only need to check if in the definition of L only $L(M)$ appears and nowhere M directly (except of course, to say that M has to be a TM), or the condition can be restated such that only $L(M)$ is described by it.

For a more formal proof of that condition, simply show that the implication holds

5.6 The method of the Kolmogorov-Complexity

Theorem 5.10: The problem of computing the Kolmogorov-Complexity $K(x)$ for each x is algorithmically unsolvable.

Lemma 5.5: If $L_H \in \mathcal{L}_R$, then there exists an algorithm to compute the Kolmogorov-Complexity $K(x)$ for each $x \in (\Sigma_{\text{bool}})^*$

As of HS2025, chapters 5.5 and 5.7 are not relevant for the Endterm or Session exam, so they are omitted here

6 Complexity

6.2 Measurements of Complexity

D 6.1: *(Time complexity)* For a computation $D = C_1, \dots, C_k$ of M on x is defined by $\text{Time}_M(x) = k - 1$. For the TM M itself, we have $\text{Time}_M(n) = \max\{\text{Time}_M(x) \mid x \in \Sigma^n\}$

Space complexity — **Definition 6.2**

Let $C = (q, x, i, \alpha_1, i_1, \dots, \alpha_k, i_k)$, with $0 \leq i \leq |x| + 1$ and $0 \leq i_j \leq |\alpha_j|$ for $j = 1, \dots, k$ be a configuration.
The space complexity of configuration C is $\text{Space}_M(C) = \max\{|\alpha_i| \mid i = 1, \dots, k\}$.
The space complexity of a calculation $D = C_1, \dots, C_l$ on x is $\text{Space}_M(x) = \max\{\text{Space}_M(C_i) \mid i = 1, \dots, l\}$
The space complexity of a TM M is $\text{Space}_M(n) = \max\{\text{Space}_M(x) \mid x \in \Sigma^n\}$

Lemma 6.1: For every k-tape-TM A, there exists an equivalent 1-tape-TM B such that $\text{Space}_B(n) \leq \text{Space}_A(n)$

Lemma 6.2: For every k-tape-TM A, $\exists$ a k-tape-TM such that $L(A) = L(B)$ and $\text{Space}_B(n) \leq \frac{\text{Space}_A(n)}{2} + 2$

Definition 6.3: The big-O-notation is defined as in A&D, we however write $\text{Time}_A(n) \in \mathcal{O}(g(n))$, etc

Theorem 6.1: There exists decision problem $(\Sigma_{\text{bool}}, L)$, such that for each MTM A that decides it, there exists an MTM B that also decides it and for which $\text{Time}_B(n) \leq \log_2(\text{Time}_A(n))$

Definition 6.4: An MTM C is ***optimal*** for L, if $\text{Time}_C(n) \in \mathcal{O}(f(n))$ and $\Omega(()\, f(n))$ is a lower bound for the time complexity of L

6.3 Complexity classes

Below is a list of complexity classes

Complexity classes — **Definition 6.5**

$$\begin{aligned}
\text{TIME}(f) &= \{L(B) \mid B \text{ is an MTM with } \text{Time}_B(n) \in \mathcal{O}(f(n))\} \\
\text{SPACE}(g) &= \{L(A) \mid A \text{ is an MTM with } \text{Space}_A(n) \in \mathcal{O}(g(n))\} \\
\text{DLOG} &= \text{SPACE}(\log_2(n)) \\
\text{P} &= \bigcup_{c \in \mathbb{N}} \text{TIME}(n^c) \\
\text{PSPACE} &= \bigcup_{c \in \mathbb{N}} \text{SPACE}(n^c) \\
\text{EXPTIME} &= \bigcup_{d \in \mathbb{N}} \text{TIME}(2^{n^d})
\end{aligned}$$

Lemma 6.3: For any function $t : \mathbb{N} \to \mathbb{R}^+$, we have $\text{TIME}(t(n)) \subseteq \text{SPACE}(t(n))$.

A list of relationships for these classes:

- $P \subseteq \text{PSPACE}$
- $\text{DLOG} \subseteq P$
- $\text{PSPACE} \subseteq \text{EXPTIME}$
- $\text{DLOG} \subseteq P \subseteq \text{PSPACE} \subseteq \text{EXPTIME}$

Space- and time-constructible — **Definition 6.6**

Let $s, t : \mathbb{N} \to \mathbb{N}$. s is called ***space-constructible*** if there exists 1-Band-TM M, such that
1. $\text{Space}_M(n) \leq s(n)\ \forall n \in \mathbb{N}$
2. for each input 0^n with $n \in \mathbb{N}$, M generates the word $0^{s(n)}$ on its memory tape and stops in q_{accept}

t is called ***time-constructible***, if there exists an MTM A, such that
1. $\text{Time}_A(n) \in \mathcal{O}(t(n))$
2. For each input 0^n with $n \in \mathbb{N}$, A generates $0^{t(n)}$ on its first memory tape and stops in q_{accept}

Lemma 6.4: Let s be space-constructible, M an MTM with $\text{Space}_M(x) \leq s(|x|)\ \forall x \in L(M)$. Then exists MTM A with $L(A) = L(M)$ and $\text{Space}_A(n) \leq s(n)$, i.e. we have $\text{Space}_A(y) \leq s(|y|)\ \forall y \in \Sigma_M$.

Lemma 6.5: Let t be time-constructible, M an MTM with $\text{Time}_M(x) \leq t(|x|)\ \forall x \in L(M)$. Then exists MTM A with $L(A) = L(M)$ and $\text{Time}_A(n) \in \mathcal{O}(t(n))$

Theorem 6.2: $\forall s : \mathbb{N} \to \mathbb{N}$ with $s(n) \geq \log_2(n)$, we have $\text{SPACE}(s(n)) \subseteq \bigcup_{c\in\mathbb{N}} \text{TIME}(c^{s(n)})$

Theorem 6.3: Given $s_1, s_2 : \mathbb{N} \to \mathbb{N}$ with properties $s_2(n) \geq \log_2(n)$, s_2 is space-constructible and $s_1(n) = o(s_2(n))$ ($s_2(n)$ grows asymptotically faster than s_1). Then we have $\text{SPACE}(s_1) \subsetneq \text{SPACE}(s_2)$

Theorem 6.4: Given $t_1, t_2 : \mathbb{N} \to \mathbb{N}$ with properties t_2 is time-constructible and $t_1(n) \cdot \log_2(t_1(n)) = o(t_2(n))$ Then we have $\text{TIME}(s_1) \subsetneq \text{TIME}(s_2)$

6.4 Non-deterministic measurements of complexity

Definition 6.7: For NMTM or NTM, the time complexity is the length of the shortest accepting calculation of M on x and the same applies to space complexity as well. The rest of the definition is equivalent to the one for deterministic TM and MTM.

Complexity classes — **Definition 6.8**

For all $f, g : \mathbb{N} \to \mathbb{R}^+$, we define

$$\begin{aligned}
\text{NTIME}(f) &= \{L(M) \mid M \text{ is an NMTM with } \text{Time}_M(n) \in \mathcal{O}(f(n))\} \\
\text{NSPACE}(g) &= \{L(M) \mid M \text{ is an NMTM with } \text{Space}_M(n) \in \mathcal{O}(g(n))\} \\
\text{NLOG} &= \text{NSPACE}(\log_2(n)) \\
\text{NP} &= \bigcup_{c\in\mathbb{N}} \text{NTIME}(n^c) \\
\text{NPSPACE} &= \bigcup_{c\in\mathbb{N}} \text{NSPACE}(n^c)
\end{aligned}$$

Lemma 6.6: For all t and s with $s(n) \geq \log_2(n)$: $\text{NTIME} \subseteq \text{NSPACE}(t)$ and $\text{NSPACE}(s) \subseteq \bigcup_{c\in\mathbb{N}} \text{NTIME}(c^{s(n)})$

For $t : \mathbb{N} \to \mathbb{R}^+$ and every space-constructible s with $s(n) \geq \log_2(n)$, we have:

1. $\text{TIME}(t) \subseteq \text{NTIME}(t)$
2. $\text{SPACE}(t) \subseteq \text{NSPACE}(t)$
3. $\text{NTIME}(s(n)) \subseteq \text{SPACE}(s(n)) \subseteq \bigcup_{c\in\mathbb{N}} \text{TIME}(c^{s(n)})$
4. $\text{NP} \subseteq \text{PSPACE}$
5. $\text{NSPACE}(s(n)) \subseteq \bigcup_{c\in\mathbb{N}} \text{TIME}(c^{s(n)})$
6. $\text{NLOG} \subseteq P$
7. $\text{NPSPACE} \subseteq \text{EXPTIME}$
8. $\text{NSPACE}(s(n)) \subseteq \text{SPACE}(s(n)^2)$ (Savitch)
9. $\text{PSPACE} = \text{NPSPACE}$

If we combine some of the above results, we get:

$$\text{DLOG} \subseteq \text{NLOG} \subseteq P \subseteq NP \subseteq \text{PSPACE} \subseteq \text{EXPTIME}$$

6.5 Proof verification

***p*-Verifier** — **Definition 6.9**

And MTM A is a p-Verifier (for $p : \mathbb{N} \to \mathbb{N}$) and $V(A) = L$ for $L \subseteq \Sigma^*$, if A has the following properties and works on all inputs $\Sigma^* \times (\Sigma_{\text{bool}})^*$:

(i) $\text{Time}_A(w, x) \leq p(|w|)$ for each input $(w, x) \in \Sigma^* \times (\Sigma_{\text{bool}})^*$

(ii) $\forall w \in L, \exists x \in (\Sigma_{\text{bool}})^*$, such that $|x| \leq p(|w|)$ and $(w, x) \in L(A)$. x is ***proof*** of the claim $w \in L$

(iii) $\forall y \notin L$ we have $(y, z) \notin L(A)$ for all $z \in (\Sigma_{\text{bool}})^*$

(iv) If $p(n) \in \mathcal{O}(n^k), k \in \mathbb{N}$, then p is a polynomial time verifier. The class of polynomial time verifiers is

$$VP = \{V(A) \mid A \text{ is polynomial time verifier }\}$$

Theorem 6.5: $VP = NP$

6.6 NP-Completeness

Definition 6.10: *(Polynomial Reduction)* $L_1 \leq_p L_2$, if there exists polynomial TM A with $x \in L_1 \Leftrightarrow A(x) \in L_2$. A is called a polynomial reduction of L_1 into L_2

NP-Hard and NP-Complete — **Definition 6.11**

A language L is called NP-hard, if for all $L' \in NP$, we have $L' \leq_p L$
A language L is called NP-complete, if it is NP-hard and and $L \in NP$

Lemma 6.7: If $L \in P$ and L is NP-hard, then $P = NP$

Definition 6.12: *(Cook)* SAT is NP-complete

Lemma 6.8: If $L_1 \leq_p L_2$ and L_1 is NP-hard, then L_2 is NP-hard

A few languages commonly used to show NP-completeness:

- $SAT = \{\Phi \mid \Phi \text{ is a satisfiable formula in CNF}\}$
- $3SAT = \{\Phi \mid \Phi \text{ is a satisfiable formula in CNF with all clauses containing } \textit{at most} \text{ three literals}\}$
- $\text{CLIQUE} = \{(G, k) \mid G \text{ is an undirected graph that contains a } k\text{-clique }\}$
- $VC = \{(G, k) \mid G \text{ is an undirected graph with a vertex cover of size } \leq k \}$
- $SCP = \{(X, \mathcal{S}, k) \mid X \text{ has a set cover } \mathcal{C} \subseteq \mathcal{S} \text{ such that } |\mathcal{C}| \leq k \}$
- $DS = \{(G, k) \mid G \text{ has a dominating set } D \text{ such that } |D| \leq k\}$

where a k-clique is a complete subgraph consisting of k vertices in G, with $k \leq |V|$; where a subset $\mathcal{C} \subseteq \mathcal{S}$ is a *set cover* of X if $X = \bigcup_{S \in \mathcal{C}}$; where a *dominating* set is is a set $D \subseteq V$ such that for every vertex $v \in V$, $v \in D$ or exists $w \in D$ such that $\{v, w\} \in E$ and where a vertex cover is any set $U \subseteq V$ where all edges $\{u, v\} \in E$ have at least one endpoint $u, v \in U$

We have $SAT \leq_p \text{CLIQUE}$, $SAT \leq_p 3SAT$, $\text{CLIQUE} \leq_p VC$, $VC \leq_p SCP$ and $SCP \leq_p DS$. Logically, we also have $SAT \leq_p DS$, etc, since $\leq_p$ is transitive (in fact, all reductions that we covered are transitive)

Additionally, MAX-SAT and MAX-CL, the problem to determine the maximum number of fulfillable clauses in a formula Φ and the problem to determine the maximum clique, respectively, are NP-hard

7 Grammars

Definition 7.1: *(Grammar)* $G := (\Sigma_N, \Sigma_T, P, S)$

1. ***Non-Terminals*** Σ_N (Are used for the rules)
2. ***Terminals*** Σ_T (The symbols at the end (i.e. only they can be remaining after the last derivation))
3. ***Start symbol*** $S \in \Sigma_N$
4. ***Derivation rules*** $P \subseteq \Sigma^*\Sigma_N\Sigma^* \times \Sigma^*$

where $\Sigma_N \cap \Sigma_T = \emptyset$ and $\Sigma := \Sigma_N \cup \Sigma_T$

Types of grammars **Definition 7.2**

1. G is a ***Type-0-Grammar*** if it has no further restrictions.
2. G is a ***Type-1-Grammar*** (or ***context-sensitive*** (= Kontextsensitiv) Grammar) if we cannot replace a subword α with a shorter subword β.
3. G is a ***Type-2-Grammar*** (or ***context-free*** (= Kontextfrei) Grammar) if all rules have the form $X \to \beta$ for a non-terminal X.
4. G is a ***Type-3-Grammar*** (or ***regular*** (= regulär) Grammar) if all rules have the form $X \to u$ or $X \to uY$

A few examples to highlight what kind of derivation rules are allowed. The rules disallowed in n are also disallowed in $n+1$:

1. All kind of rules are allowed
2. Rules like $X \to \lambda$ or $0Y1 \to 00$ are not allowed (they shorten the output)
3. Rules like $aA \to Sb$ are not allowed, as they are not context-free (i.e. all rules have to be of form $X \to \ldots$)
4. Rules like $S \to abbAB$ are not allowed, as two non-terminals appear