Theoretical Computer Science - Compact

Janis Hutz
https://janishutz.com

November 1, 2025

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

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 $w = sx$; Subword is u in $w = xuy$

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 $L^+ = L \cdot L^*$

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

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 random if $K(x) \geq |x|$, thus for n, $K(n) \geq \lceil \log_2(n + 1) \rceil - 1$

Theorem 2.3: *(Prime number)* $\lim_{n\to\infty} \frac{\text{Prime}(n)}{\frac{n}{\ln(n)}}$

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:

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

Definition 3.1: *(Finite Automaton)*

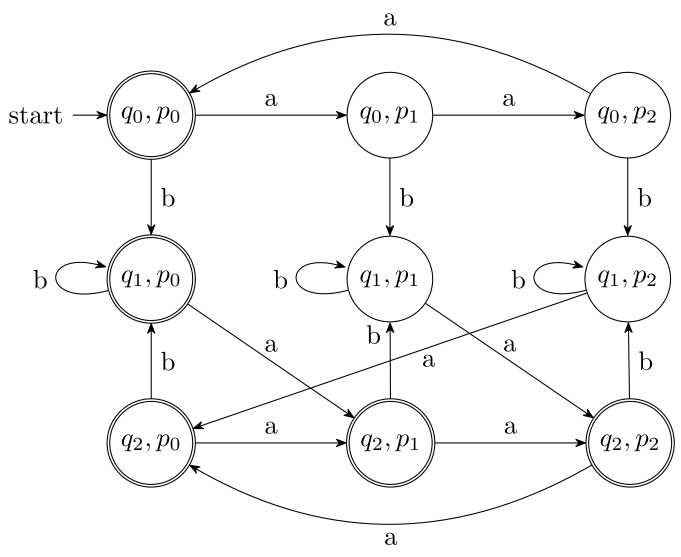
- Q set of states
- Σ input alphabet
- $\delta(q, a) = p$ transition from q on reading a to p
- q_0 initial state
- $F \subseteq Q$ accepting states
- $\mathcal{L}_{EA}$ regular languages (accepted by FA)

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

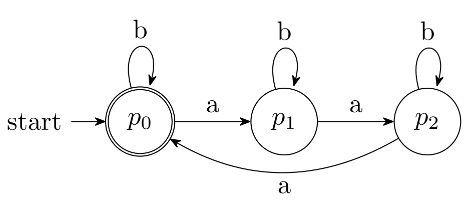
The class $\text{Cl}[q_i]$ represents all possible words for which the FA is in this state. Be cautious when defining them, make sure that no extra words from other classes could appear in the current class, if this is not intended.

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:

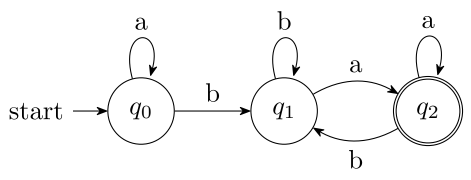
1. Write down the states as tuples of the form (q_i, p_j) (i.e. form a grid by writing down one of the automata vertically and the other horizontally)
2. From each state, the automata on the horizontal axis decides for the input symbol if we move left or right, whereas the automata on the vertical axis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

Unterscheidung von Wörtern **Lemma 3.3**

Sei A ein EA über Σ und $x \neq y \in \Sigma^*$ so dass $\widehat{\delta}_A(q_0, x) = \widehat{\delta}(q_0, y)$. Dann existiert für jedes $z \in \Sigma^*$ ein $r \in Q$, so dass $xz, yz \in \text{Cl}[r]$, also gilt insbesondere

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

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**

Sei L regulär. Dann existiert eine Konstante $n_0 \in \mathbb{N}$, so dass sich jedes Wort $w \in \Sigma^*$ mit $|w| \geq n_0$ in $w = yxz$ zerlegen lässt, wobei

(i) $|yx| \leq n_0$
(ii) $|x| \geq 1$
(iii) Für $X = \{yx^kz \mid k \in \mathbb{N}\}$ *entweder* $X \subseteq L$ oder $X \cap L = \emptyset$ gilt

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 $X = \{yx^kz \mid k \in \mathbb{N}\}$ we have $X \subseteq L$.
6. This will lead to a contradiction commonly when setting $k = 0$, as for a language like 0^n1^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, $X \cap L = \emptyset$

Kolmogorov Complexity

1. We first need to choose an x such that $L_x = \{y | xy \in L\}$. 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 $\{a^nb^n \mid n \in \mathbb{N}\}$ to use $x = a^m$ and $L_{0^m} = \{y | 0^m y \in L\} = \{0^j 1^{m+j} | j \in \mathbb{N}\}$.
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 and 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)
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 $\hat{\delta}_A(q_0, x) = \hat{\delta}_A(q_0, y)$ 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 maps is different: $\delta : Q \times \Sigma \to \mathcal{P}(Q)$. I.e., there can be any number of transitions for one symbol from Σ from 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 $P = \hat{\delta}(q_0, z)$ 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 $\{0^n 1^n \mid n \in \mathbb{N}\}$ 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.

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 and 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 Turin 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}

Proof of L (not) recursively enumerable

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

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

As an example, we'll use the following language

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.