Brute-force solution: $O(n!)$

Dynamic programming algorithms: $O(n^2 2^n)$

Selling on eBay: $O(1)$

Still working on your route?

Shut the hell up.
Organization & timetable

**Lectures:** prof. Tapio Elomaa, M.Sc. Timo Aho
- Tue and Thu 14–16 TB220
- Aug. 30 – Dec. 8, 2011
- **Exceptions:**
  - Thu Sept. 1 – lecture cancelled due to the illness of the lecturer
  - Tue Sept. 6 & Thu Sept. 8 – lectures given by M.Sc. Timo Aho
  - Thu Oct. 4 – lecture cancelled, ALT/DS 2011 in Espoo
  - Tue Oct. 18 & Thu Oct. 20 – no lectures, period break
  - Tue Dec. 6 – lecture cancelled, independence day

**Weekly exercises** start later →
- M.sc. Teemu Heinimäki,
Exam: Thu Dec. 15, 2011, 9–12 AM
Mon Jan. 30, 2012, 9–12 AM
Mon Mar. 19, 2012, 9–12 AM

Assessment: 30 + 6 = 36
{tapio.elomaa, timo.aho, teemu.heinimaki}@tut.fi
Weekly exercises

- It is **most advisable** to take part in the weekly exercises
- Being ready to present your solution to problem yields one mark
- Each weekly exercise session has approx. 6 problems ⇒ altogether you can collect approx. $6 \times 10 = 60$ marks

<table>
<thead>
<tr>
<th>Marks</th>
<th>Extra points</th>
</tr>
</thead>
<tbody>
<tr>
<td>25% (c. 15)</td>
<td>1</td>
</tr>
<tr>
<td>35% (c. 21)</td>
<td>2</td>
</tr>
<tr>
<td>45% (c. 27)</td>
<td>3</td>
</tr>
<tr>
<td>60% (c. 36)</td>
<td>4</td>
</tr>
<tr>
<td>75% (c. 45)</td>
<td>5</td>
</tr>
<tr>
<td>85% (c. 51)</td>
<td>6</td>
</tr>
</tbody>
</table>
Grading

- You can pass the course with grade 5/5 by taking the exam alone, but
  - The exam (max 30 p.) is not the easiest of them all
  - Therefore it is advisable to participate in the weekly exercises from which you can earn extra points (max 6 p.)
  - By independently solving the problems you learn the seemingly difficult material of the course
- The course grade will *most probably* be determined according to the following table:

<table>
<thead>
<tr>
<th>points</th>
<th>15</th>
<th>18</th>
<th>21</th>
<th>24</th>
<th>27</th>
</tr>
</thead>
<tbody>
<tr>
<td>grade</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>
Course Material

- The course text book in Fall 2011 is


- Almost any text book on this topic covers the material that we will go through
- The slides will appear into the Web weekly

- The exam is based on the lectures
# Timetable for Lectures

1. **Introduction** (1)
2. **Recap: automata, grammars and languages** (2–4)
   - 1. Regular languages
   - 2. Context-free grammars
3. **Computability theory** (4–8)
   - 1. Universal models of computation
   - 2. Solvability / decidability
   - 3. Reducibility
   - 4. Advanced topics
4. **Complexity theory** (9–14)
   - 1. Time complexity
   - 2. Space complexity
   - 3. Practical solvability
   - 4. Advanced topics
0. Introduction

- This course offers an introduction to the mathematical and theoretical background of computer science.
- Such information is an integral part of the general knowledge of a computer scientist.
- The aim is to gain a basic understanding of what kinds of problems can in principle be solved using a computer.
- Even more important is to observe which of the decidable problems can be solved efficiently by a computer program.
- The results that we cover in this course are fundamental in the sense that the increasing efficiency of computers in the years to come will not deem these results insignificant.
Future Computers? (from Wikipedia)

- **DNA computing** does not provide any new capabilities from the standpoint of computability theory.
  - For example, if the space required for the solution of a problem grows exponentially with the size of the problem (EXPSPACE problems) on von Neumann machines, it still grows exponentially with the size of the problem on DNA machines.

- **Quantum computing**, on the other hand, does provide some interesting new capabilities.
  - If large-scale quantum computers can be built, they will be able to solve certain problems much faster than any current classical computers (for example Shor's algorithm). Quantum computers don't allow the computations of functions that are not theoretically computable by classical computers, i.e. they do not alter the Church–Turing thesis. The gain is only in efficiency.
and the list goes on …

- Optical computing
- Chemical computing
- Parallel computing
- Cluster computing
- Massive parallel processing
- Distributed computing
- Grid computing
- Cloud computing
0.1 Computational Problems

- A computational problem can be modeled so that it can be solved using a computer;
  - e.g. arithmetic, lexicographic ordering, salary accounting, course maintenance, ...
- A representation of the problem that is more general than the solving program is easier to understand and makes it possible to analyze the problem
- A problem has instances (= inputs), its solution is an algorithm, which connects an answer (= output) to any instance
- An instance and its answer must be finitely represented (e.g. as bit strings). The number of instances, though, can be infinite
- A computational problem is a mapping from the set of finitely represented instances to the set of finitely represented answers
Strings and Languages

- Finite representation = a string over an alphabet
  - An *alphabet* is a nonempty finite set of *symbols*
  - E.g., *binary alphabet* \{ 0, 1 \} and
    the Latin alphabet \{ a, b, ..., z \}
- A *string* over an alphabet is a finite sequence of symbols from the alphabet
  - E.g., 01001 and *abracadabra*
- The *length* of a string \( w \), written \(|w|\), is the number of symbols that it contains
  - E.g. \(|01001| = 5\) ja \(|abracadabra| = 11\)
- The *empty string* \( \varepsilon \) has length \(|\varepsilon| = 0\)
Appending strings, the *concatenation*, is their basic operation.

- \texttt{abracadabra} \texttt{=} \texttt{abracadabra}
- \( x = 01, y = 10 \rightarrow xx = 0101, xy = 0110, yy = 1010 \) ja \( yx = 1001 \)
- For all \( w \) it holds that \( w\varepsilon = \varepsilon w = w \)
- For all \( x, y \) it holds that \( |xy| = |x| + |y| \)
- If \( w = xy \), then \( x \) is a *prefix* of \( w \) and \( y \) is its *suffix*
- All strings of the alphabet \( \Sigma \) are denoted by \( \Sigma^* \)
  - E.g. \( \Sigma = \{ 0, 1 \} \rightarrow \Sigma^* = \{ \varepsilon, 0, 1, 00, 01, 10, 11, \ldots \} \)
- Other notation: \( \Sigma^k \) and \( \Sigma^* = \bigcup_{k \in \mathbb{N}} \Sigma^k \)
Decision Problems

- A computational problem $\pi$ is hence a mapping
  $$\pi: \Sigma^* \rightarrow \Gamma^*,$$
  where $\Sigma$ and $\Gamma$ are alphabets

- *Decision problems* are an important subclass; in them the answer to an instance of the problem is simply *Yes* or *No*
  - I.e., they have the form $\pi: \Sigma^* \rightarrow \{0, 1\}$

- For every decision problem $\pi$ there is the set of those instances for which the answer is *Yes*:
  $$A_\pi = \{ x \in \Sigma^* \mid \pi(x) = 1 \}$$
• The other way around: For every set of strings $A$ there exists a decision problem $\pi_A : \Sigma^* \rightarrow \{0, 1\}$,

$$\pi_A(x) = \begin{cases} 1, & \text{if } x \in A \\ 0, & \text{if } x \notin A \end{cases}$$

• The set of strings $A \subseteq \Sigma^*$ is called a (formal) *language* of the alphabet $\Sigma$

• The respective decision problem $\pi_A$ is known as the recognition problem of the language $A$

• We can treat formal languages and decision problems as equals
Definition 4.12

- Let $A$ and $B$ be two sets and $f$ a function from $A$ to $B$
- We say that $f$ is **one-to-one** (injection) if it never maps two different elements to the same place – that is, $f(a) \neq f(b)$ whenever $a \neq b$
- We say that $f$ is **onto** (surjection) if it hits every element of $B$ – that is, if for every $b \in B$ there is an $a \in A$ such that $f(a) = b$
- We say that $A$ and $B$ are the **same size** if there is a one-to-one, onto function $f: A \rightarrow B$
- A function that is both one-to-one and onto is called a **correspondence** (bijection)
  - In a correspondence every element of $A$ maps to a unique element of $B$ and each element of $B$ has a unique element of $A$ mapping to it
  - A correspondence is simply a way of pairing the elements of $A$ with the elements of $B$
Solvability of Computational Problems

- We say that program \( P(x) \) solves the computational problem \( \pi \), if it outputs for each input \( x \) the value \( \pi(x) \).
- Can all possible computational problems be solved by programs (computers)?

**Definition 4.14:** A set \( A \) is **countable** if either it is finite or it has the same size as \( \mathbb{N} \).

- An infinite set that is not countable is **uncountable**.
Theorem 0.1 Let $\Sigma$ be an arbitrary alphabet. The set of strings over $\Sigma$, $\Sigma^*$, is countable.

Proof. Let $\Sigma = \{a_1, a_2, ..., a_n\}$. Let us fix an ”alphabetical ordering” for the symbols; let it be $a_1 < a_2 < ... < a_n$.

The strings belonging to $\Sigma^*$ can now be output in lexicographic (or canonical) order:

1. First output strings of length 0, then those of length 1, after which those of length 2, and so forth.
2. Within each length group the strings are given in the dictionary order as determined by the chosen alphabetical order.
Then the correspondence
\( f: \mathbb{N} \rightarrow \Sigma^* \) is

\[
\begin{align*}
0 & \mapsto \varepsilon \\
1 & \mapsto a_1 \\
2 & \mapsto a_2 \\
\vdots & \\
n & \mapsto a_n \\
n+1 & \mapsto a_1a_1 \\
n+2 & \mapsto a_1a_2 \\
\vdots & \\
2n & \mapsto a_1a_n \\
2n+1 & \mapsto a_2a_1 \\
\vdots & \\
3n & \mapsto a_2a_n \\
\vdots & \\
n^2+n & \mapsto a_na_n \\
n^2+n+1 & \mapsto a_1a_1a_1 \\
n^2+n+2 & \mapsto a_1a_1a_2 \\
\vdots & \\
\end{align*}
\]
Theorem 0.2  The set of decision problems over any alphabet $\Sigma$ is uncountable.

Proof. Let $\Pi$ denote the collection of all decision problems over $\Sigma$: $\Pi = \{ \pi \ | \ \pi \text{ is a mapping } \Sigma^* \rightarrow \{0, 1\} \}$.

Let us assume that $\Pi$ is countable; i.e., there exists an enumeration that covers all elements of $\Pi$:

$$\Pi = \{\pi_0, \pi_1, \pi_2, \ldots \}.$$

Let the strings belonging to $\Sigma^*$, given in the lexicographic ordering of Theorem 1.1, be $x_0, x_1, x_2, \ldots$
Let us compose a new decision problem $\xi: \Sigma^* \rightarrow \{0, 1\}$:

$$\xi(x_i) = \begin{cases} 1, & \text{if } \pi_i(x_i) = 0 \\ 0, & \text{if } \pi_i(x_i) = 1 \end{cases}$$

Because $\Pi$ covers all decision problems over $\Sigma$, it must be that $\xi \in \Pi$. Hence, $\xi = \pi_k$ for some $k \in \mathbb{N}$. Then

$$\xi(x_k) = \begin{cases} 1, & \text{if } \pi_k(x_k) = \xi(x_k) = 0 \\ 0, & \text{if } \pi_k(x_k) = \xi(x_k) = 1 \end{cases}$$

This is a contradiction. Thus, our assumption ($\Pi$ is countable) cannot hold. Hence, $\Pi$ must be uncountable.
• This type of proof is known as Cantor’s diagonalization method.
• In the end, e.g., Java programs are just strings over the alphabet ASCII. By Theorem 0.1 there exist only a countable set of them
• However, by Theorem 0.2 the set of computational problems is uncountable
• Therefore, out of all computational problems only a miniscule part can be solved using Java programs
• The problem is the same for all programming languages
• Unsolvable problems include also interesting and practical problems
Georg Cantor (1845–1918)

- Born in St. Petersburg
- To Frankfurt 1856
- Ph.D. (Berlin) 1867
- Halle 1869-
  - Privatdozent
  - Prof. 1872
- Countability of rational numbers 1873
- Set theory 1874
- Continuum hypothesis 1878
AUTOMATA AND LANGUAGES

1.1 Finite Automata

- A system of computation that only has a finite number of possible states can be modeled using a finite automaton.
- A finite automaton is often illustrated as a state diagram.

![Finite Automaton Diagram]
Definition 1.5: Finite Automaton

• A finite automaton is a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$, where
  • $Q$ is a finite set called the states,
  • $\Sigma$ is a finite set called the alphabet,
  • $\delta: Q \times \Sigma \rightarrow Q$ is the transition function,
  • $q_0 \in Q$ is the start state, and
  • $F \subseteq Q$ is the set of (accepting) final states.

• A machine $M$ accepts the string $w = w_1w_2...w_n \in \Sigma^n$ if a sequence of states $r_0, r_1, ..., r_n$ in $Q$ exists s.t.
  • $r_0 = q_0$,
  • $\delta(r_i, w_{i+1}) = r_{i+1}, i = 0, ..., n-1$,
  • $r_n \in F$. 
• The *language recognized* by $M$ is
  \[ L(M) = \{ w \in \Sigma^* \mid M \text{ accepts } w \} \]

• A language is called a *regular language*, if some finite automaton recognizes it

• Basic operations on languages $A$ and $B$ are
  • *Union* \[ A \cup B = \{ x \mid x \in A \lor x \in B \}, \]
  • *Concatenation* \[ A \circ B = \{ xy \mid x \in A \land y \in B \} \text{ and} \]
  • *(Kleene)* *Star (closure)* \[ A^* = \{ x_1x_2...x_k \mid k \geq 0 \land x_i \in A \ \forall i \} \]
Properties of Regular Languages

**Theorem 1.25** The class of regular languages is closed under the union operation.

In other words, if $A_1$ and $A_2$ are regular languages, so is $A_1 \cup A_2$.

**Theorem 1.26** The class of regular languages is closed under the concatenation operation.

DFA = Deterministic Finite Automaton
1.1.1 Minimization of DFAs

- Two automata that recognize exactly the same language are *equivalent* with each other.
- A finite automaton is *minimal* if it has the smallest number of states among equivalent automata.
- An automaton that has more states than in an equivalent minimal automaton is called *redundant*.
- Algorithms producing automata do not always generate a minimal automaton.
- Handling a minimal automaton is more efficient than that of a redundant automaton.
Algorithm MINIMIZE

Input: DFA $M = (Q, \Sigma, \delta, q_0, F)$.
1. Remove all states of $M$ that are unreachable from the start state.
2. Construct the following undirected graph $G$ whose nodes are the states of $M$.
3. Place an edge in $G$ connecting every accept state with every nonaccept state. Add additional edges as follows.
4. Repeat until no new edges are added to $G$:
   1. For every pair $q, r \in Q, q \neq r$, and every $a \in \Sigma$: add the edge $(q, r)$ to $G$ if $(\delta(q, a), \delta(r, a))$ is an edge of $G$.
   2. For each state $q \in Q$ let $[q]$ be the collection of states
      2. $[q] = \{ q \} \cup \{ r \in Q \mid \text{no edge joins } q \text{ and } r \text{ in } G \}$.
5. Form a new DFA $M' = (Q', \Sigma, \delta', q'_0, F')$, where
   • $Q' = \{ [q] \mid q \in Q \}$, (removing doubles)
   • $\delta'([q], a) = [\delta(q, a)]$, for every $q \in Q$ and $a \in \Sigma$,
   • $q'_0 = [q_0]$ and
   • $F' = \{ [q] \mid q \in F \}$.
6. Output $M'$. 
The End Result

- An automaton $M'$ that is equivalent with the input automaton $M'$, such that it has the minimum number of states.
- Automaton $M'$ is unique (up to the naming of the states).
1.2 Nondeterministic Finite Automata (NFAs)

- In an NFA a state can have many possible alternative transitions with the same symbol of the alphabet.

- Also $\varepsilon$-transitions are allowed.

- Implementing nondeterministic behavior is not straightforward (though possible), but as a modeling tool it is quite useful.

- Via NFAs we can connect DFAs and regular expressions.
• The definition of an automaton requires the transition function to be a function.  
• On the other hand, in an NFA the transition function should get mapped to a set of values.  
• An NFA accepts a string if a sequence of possible states leads to a final state.  
  • Only if no such sequence exists will the NFA reject the input string.  
• E.g. the previous NFA accepts the string 010110 because it can be processed as follows:  
  $$(q_0, 010110) \xrightarrow{} (q_0, 10110) \xrightarrow{} (q_1, 0110)$$  
  $$(q_2, 110) \xrightarrow{} (q_3, 10) \xrightarrow{} (q_3, 0) \xrightarrow{} (q_3, \varepsilon)$$
On the other hand, we can end up in a rejecting state:

\[
(q_0, 010110) \xrightarrow{0, 1} (q_0, 10110) \xrightarrow{0, \varepsilon} (q_0, 0110) \\
\xrightarrow{0, 1} (q_0, 110) \xrightarrow{0, 10} (q_1, 0) \xrightarrow{1} (q_2, \varepsilon)
\]
Definition of an NFA

- Let $P(A) = \{ B \mid B \subseteq A \}$ denote the power set of the set $A$ and for an alphabet $\Sigma: \Sigma_\varepsilon = \Sigma \cup \{ \varepsilon \}$

- A nondeterministic finite automaton is a 5-tuple $N = (Q, \Sigma, \delta, q_0, F)$
  - $Q$ is a finite set of states,
  - $\Sigma$ is a finite alphabet,
  - $\delta: Q \times \Sigma_\varepsilon \to P(Q)$ is the (set-valued) transition function, that also allows $\varepsilon$-transitions
  - $q_0 \in Q$ is the start state, and
  - $F \subseteq Q$ is the set of (accepting) final states
• The transition function of the previous automaton is

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>$\varepsilon$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td>${q_0}$</td>
<td>${q_0, q_1}$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$q_1$</td>
<td>${q_2}$</td>
<td>$\emptyset$</td>
<td>${q_2}$</td>
</tr>
<tr>
<td>$q_2$</td>
<td>$\emptyset$</td>
<td>${q_3}$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$q_3$</td>
<td>${q_3}$</td>
<td>${q_3}$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>

• Now we can easily express the error state as an empty set of possible next states
• An NFA $N = (Q, \Sigma, \delta, q_0, F)$ accepts the string $w$,
  • If we can write it as $w = y_1y_2...y_m \in \Sigma^m$ and a sequence of states $r_0, r_1, ..., r_m$ exists in $Q$ s.t.
    • $r_0 = q_0$,
    • $r_{i+1} \in \delta(r_i, y_{i+1}), i = 0, ..., m-1$, and
    • $r_m \in F$.
• DFAs are a special case of NFAs $\Rightarrow$
  all languages that can be recognized using the former can also be recognized using the latter
• Also the other way around: DFAs and NFAs recognize the same set of languages
Theorem 1.39  Let $A = L(N)$ be the language recognized by some NFA $N$. There exists a DFA $M$ such that $L(M) = A$

Proof. Let $N = (Q, \Sigma, \delta, q_0, F)$. We construct a DFA $M = (Q', \Sigma, \delta', q_0', F')$ that simulates the computation of $N$ in parallel in all its possible states at all times. Let us first consider the easier situation where $N$ has no $\varepsilon$ arrows.

Every state of $M$ is a set of states of $N$

$Q' = \mathcal{P}(Q)$

$q_0' = \{ q_0 \}$

$F' = \{ R \in Q' \mid R \text{ contains an accept state } r \in F \}$

$\delta'(R, a) = \bigcup_{r \in R} \delta(r, a)$
Without $\varepsilon$ arrows
After Minimization

\[ R_0 \xrightarrow{1} R_1 \xrightarrow{0} R_2 \xrightarrow{1} R_3 \]

\[ R_0 \xrightarrow{0} R_1 \xrightarrow{1} R_2 \xrightarrow{0} R_3 \]

\[ R_1 \xrightarrow{0} R_2 \xrightarrow{1} R_3 \]

\[ R_2 \xrightarrow{0} R_3 \]

\[ R_3 \xrightarrow{0, 1} R_0 \]
Let us check that $L(M) = L(N)$. The equivalence of the languages follows when we prove for all $x \in \Sigma^*$ and $r \in Q$ that

$$(q_0, x) \rightarrow N (r, \varepsilon) \iff (\{q_0\}, x) \rightarrow M (R, \varepsilon) \text{ and } r \in R,$$

where the notation $(q_0, x) \rightarrow N (r, \varepsilon)$ means that in automaton $N$ we can process the string $x$ starting from state $q_0$ so that we end up in state $r$ and there are no more symbols to process ($\varepsilon$).

We prove it using induction over the length of the string $x$:

1. **Basis**: $|x| = 0$: $(q_0, \varepsilon) \rightarrow N (r, \varepsilon) \iff r = q_0$.

   Similarly $(\{q_0\}, \varepsilon) \rightarrow M (R, \varepsilon) \iff R = \{q_0\}$
2. **Induction hypothesis**: the claim holds when $|x| \leq k$

3. $|x| = k + 1$: Then $x = ya$ for some $y$, $|y| = k$, for which the claim holds by the induction hypothesis. Now,

$$(q_0, x) = (q_0, ya) \implies_N (r, \varepsilon)$$

$$\iff \exists r' \in Q \text{ s.t. } (q_0, ya) \implies_N (r', a) \text{ and } (r', a) \implies_N (r, \varepsilon)$$

$$\iff \exists r' \in Q \text{ s.t. } (q_0, y) \implies_N (r', \varepsilon) \text{ and } (r', a) \implies_N (r, \varepsilon)$$

By induction hypothesis we get

$$\exists r' \in Q \text{ s.t. } (\{q_0\}, y) \implies_M (R', \varepsilon) \text{ and } r' \in R' \text{ and } r \in \delta(r', a)$$

Rearranging yields

$$\implies (\{q_0\}, y) \implies_M (R', \varepsilon) \text{ and } \exists r' \in R' \text{ s.t. } r \in \delta(r', a)$$

By the definition of the transition function $\delta'$
Let us return $a$ and name $\delta'(R', a)$

$$\Leftrightarrow (\{q_0\}, y) \gg M(R', \varepsilon) \text{ and } r \in \bigcup_{r' \in R'} \delta(r', a) = \delta'(R', a)$$

Let us return $a$ and name $\delta'(R', a)$

$$\Leftrightarrow (\{q_0\}, ya) \gg M(R', a) \text{ and } r \in \delta'(R', a) = R$$

$$\Leftrightarrow (\{q_0\}, ya) \gg M(R', a) \text{ and } (R', a) \geq M(R, \varepsilon) \text{ and } r \in R$$

Concluding

$$\Leftrightarrow (\{q_0\}, x) = (\{q_0\}, ya) \gg M(R, \varepsilon) \text{ and } r \in R$$

Which completes the proof of the claim.
• In order to take the $\varepsilon$ arrows into account, we compute for each state $R \subseteq Q$ of $M$ the collection of states that can be reached from $R$ by going only along $\varepsilon$ arrows:

$$E(R) = \{ q \mid q \text{ can be reached from } R \text{ by traveling along } 0 \text{ or more } \varepsilon \text{ arrows} \}$$

• It is enough to modify the transition function of $M$ and start state to take the $\varepsilon$ arrows into account:

$$\delta'(R, a) = \bigcup_{r \in R} E(\delta(r, a))$$

$$q_0' = E(\{q_0\})$$
With $\varepsilon$ arrows

\[ q_0 \xrightarrow{0,1} q_1 \xrightarrow{1} q_2 \xrightarrow{0,\varepsilon} q_3 \]

\[ \{q_0\} \xrightarrow{1} \{q_0, q_1, q_2\} \xrightarrow{1} \{q_0 - q_3\} \xrightarrow{1} \{q_0, q_2, q_3\} \]

\[ \{q_0, q_2\} \xrightarrow{0} \{q_0, q_3\} \xrightarrow{1} \{q_0, q_3\} \]

\[ \{q_0, q_2\} \xrightarrow{0} \{q_0, q_3\} \xrightarrow{1} \{q_0, q_3\} \]

\[ \{q_0, q_2\} \xrightarrow{0} \{q_0, q_3\} \xrightarrow{1} \{q_0, q_3\} \]

\[ \{q_0, q_2\} \xrightarrow{0} \{q_0, q_3\} \xrightarrow{1} \{q_0, q_3\} \]
After Minimization
**Theorem 1.45**  The class of regular languages is closed under the union operation.

**Proof.** Let the languages $A_1$ and $A_2$ be regular. Then, there exists (nondeterministic) finite automata $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ and $N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$, which recognize these two languages. Let us construct an automaton $N = (Q, \Sigma, \delta, q_0, F)$ for recognizing the language $A_1 \cup A_2$.

- $Q = \{q_0\} \cup Q_1 \cup Q_2$,
- The start state of $N$ is $q_0$,
- $F = F_1 \cup F_2$ and

$$\delta(q, a) = \begin{cases} 
\delta_1(q, a), & q \in Q_1 \\
\delta_2(q, a), & q \in Q_2 \\
\{q_1, q_2\}, & q = q_0 \text{ and } a = \varepsilon \\
\emptyset, & q = q_0 \text{ and } a \neq \varepsilon 
\end{cases}$$
Theorem 1.47  The class of regular languages is closed under the concatenation operation.

Proof. Let the languages $A_1$ and $A_2$ be regular. Then, there exists (nondeterministic) finite automata $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ and $N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$, which recognize these two languages. Let us construct an automaton $N = (Q, \Sigma, \delta, q_1, F_2)$ for recognizing $A_1 \circ A_2$.

- $Q = Q_1 \cup Q_2$,
- The start state of $N$ is $q_1$,
- The final states of $N$ are those in $F_2$ and

$$
\delta(q, a) = \begin{cases} 
\delta_1(q, a), & q \in Q_1 \text{ and } q \notin F_1 \\
\delta_1(q, a), & q \in F_1 \text{ and } a \neq \varepsilon \\
\delta_1(q, a) \cup \{q_2\}, & q \in F_1 \text{ and } a = \varepsilon \\
\delta_2(q, a), & q \in Q_2
\end{cases}
$$
Theorem 1.49. The class of regular languages is closed under the star operation.

Proof. Let the language $A$ be regular. Then, there exists a (nondeterministic) finite automaton $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$, which recognizes the language.

Let us construct an automaton $N = (Q, \Sigma, \delta, q_0, F)$ for recognizing $A^*$.

- $Q = \{ q_0 \} \cup Q_1$,
- The new start state of $N$ is $q_0$,
- $F = \{ q_0 \} \cup F_1$ and

$$
\delta(q, a) = \begin{cases} 
\delta_1(q, a), & q \in Q_1 \text{ and } q \notin F_1 \\
\delta_1(q, a), & q \in F_1 \text{ and } a \neq \varepsilon \\
\delta_1(q, a) \cup \{ q_1 \}, & q \in F_1 \text{ and } a = \varepsilon \\
\{ q_1 \}, & q = q_0 \text{ and } a = \varepsilon \\
\emptyset, & q = q_0 \text{ and } a \neq \varepsilon
\end{cases}
$$

$\Box$
1.3 Regular Expressions

- These have an important role in describing patterns in searching for strings in many applications (e.g. awk, grep, Perl, ...)

All regular expressions of alphabet $\Sigma$ are
1. $\emptyset$ and $\varepsilon$ are regular expressions,
2. $a$ is a regular expression of $\Sigma$ for all $a \in \Sigma$,
3. if $R_1$ and $R_2$ are regular expressions, then also
   - $(R_1 \cup R_2)$,
   - $(R_1 \circ R_2)$ and
   - $R_1^*$
   are regular expressions
Each regular expression $R$ of $\Sigma$ represents a language $L(R)$

1. $L(\emptyset) = \emptyset$, 
2. $L(\varepsilon) = \{\varepsilon\}$, 
3. $L(\{a\}) = \{a\}$ $\forall$ $a \in \Sigma$, 
4. $L((R_1 \cup R_2)) = L(R_1) \cup L(R_2)$, 
5. $L((R_1 \circ R_2)) = L(R_1) \circ L(R_2)$ and 
6. $L(R_1^*) = (L(R_1))^*$

*Proper closure*: $R^+$ is a shorthand for $RR^*$ (Kleene plus) 

Observe: $R^+ \cup \varepsilon = R^*$ 

- Let $R^k$ be shorthand for the concatenation of $k$ $R$’s with each other.
Examples

\[0^*10^* = \{ w \mid w \text{ contains a single } 1 \}\]
\[\Sigma^*001\Sigma^* = \{ w \mid w \text{ contains the string } 001 \text{ as a substring} \}\]
\[1^*(01^+)^* = \{ w \mid \text{every 0 in } w \text{ is followed by at least one } 1 \}\]
\[\Sigma\Sigma^* = \{ w \mid w \text{ is a string of even length} \}\]
\[01 \cup 10 = \{ 01, 10 \}\]
\[0\Sigma^*0 \cup 1\Sigma^*1 \cup 0 \cup 1 = \{ w \mid w \text{ starts and ends with the same symbol} \}\]
\[(0 \cup \varepsilon)1^* = 01^* \cup 1^*\]
\[(0 \cup \varepsilon)(1 \cup \varepsilon) = \{ \varepsilon, 0, 1, 01 \}\]
\[1^*\emptyset = \emptyset\]
\[\emptyset^* = \{ \varepsilon \}\]
• For any regular expression $R$
  
  
  For example, the unsigned real numbers that can be recognized using the previous automaton can be expressed with the regular expression

$$d^+ (.d^+ \cup \varepsilon)( \text{E} ( + \cup - \cup \varepsilon )d^+ \cup \varepsilon ),$$

where $d = ( 0 \cup ... \cup 9 )$
Theorem 1.54  A language is regular if and only if some regular expression describes it.

We state and prove both directions of this theorem separately.

Lemma 1.55  If a language is described by a regular expression, then it is regular.

Proof. Any regular expression can be converted into a finite automaton, which recognizes the same language as that described by the regular expression.

There are only six rules by which regular expressions can be composed. The following pictures illustrate the NFA for each of these cases.  □
\( r = \emptyset \)

\( r = \varepsilon \)

\( r = a \)
\[ r = s \cup t \]
$r = st$
$r = s^*$
$r = (a(b \cup bb))^*$
Lemma 1.60  If a language is regular, then it is described by a regular expression.

Proof. By definition a regular language can be recognized with a (nondeterministic) finite automaton, which can be converted into a generalized nondeterministic finite automaton (GNFA). The GNFA finally yields a regular expression that is equivalent with the original automaton.

Let $\text{RE}_\Sigma$ denote the set of regular expressions over $\Sigma$

- In a GNFA the transition function $\delta$ is a finite mapping
  $$\delta: Q \times \text{RE}_\Sigma \rightarrow \mathcal{P}(Q)$$
- $(q, w) \succ (q', w')$ if $q' \in \delta(q, r)$ for some $r \in \text{RE}_\Sigma$ s.t. $w = zw'$, $z \in L(r)$
A GNFA $M$ can be reduced into a regular expression which describes the language recognized by $M$

1. We compress $M$ into a GNFA with only 2 states (so that the language recognized remains equivalent)
   1. The accept states of $M$ are replaced by a single one ($\varepsilon$ arrows)
   2. We remove all other states $q$ except the start state and final state.
      Let $q_i$ and $q_j$ be the predecessor and successor of $q$ on some route passing through $q$.
      Now we can remove $q$ and rename the arrow between $q_i$ and $q_j$ with a new expression.

2. Eventually the GNFA contains at most two states. It is easy to convert the language recognized into a regular expression.
\[ \begin{align*}
q_i & \xrightarrow{r} q_j \\
r \cup s & \xrightarrow{r} q_j \\
q_i & \xrightarrow{r^*} q_j \\
r^* & \xrightarrow{s} q_j \\
r & \xrightarrow{s} q_j \\
t & \xrightarrow{s} q_j \\
r^*s(t^* \cup ur^*s)^* & \xrightarrow{s} q_j \\
\end{align*} \]
\[(ab \cup (aa \cup b)(ba) \ast (bb \cup a)) \ast \]
1.4 Nonregular Languages

- The number of formal languages over any alphabet (= decision/recognition problems) is uncountable.
- On the other hand, the number of regular expressions (= strings) is countable.
- Hence, all languages cannot be regular.
- Can we find an intuitive example of a nonregular language?
- The language of balanced pairs of parentheses:

\[ L_{\text{parenth}} = \{ (k)^k \mid k \geq 0 \} \]
Theorem 1.70 (Pumping lemma)

Let $A$ be a regular language. Then there exists $p \geq 1$ (the pumping length) s.t. any string $s \in A$, $|s| \geq p$, may be divided into three pieces, $s = xyz$, satisfying the following conditions:

- $|xy| \leq p$,
- $|y| \geq 1$ and
- $xy^iz \in A \quad \forall \ i = 0, 1, 2, \ldots$

Proof. Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA that recognizes $A$ s.t. $|Q| = p$. When the DFA is computing with input $s \in A$, $|s| \geq p$, it must pass through some state at least twice when processing the first $p$ characters of $s$. Let $q$ be the first such state.
Let us choose so that:

- \( x \) is the prefix of \( s \) that has been processed when \( M \) enters \( q \) for the first time,
- \( y \) is that part of the suffix \( s \) that gets processed by \( M \) before it re-enters state \( q \), and
- \( z \) is the rest of the string \( s \).

Obviously \( |xy| \leq p, |y| \geq 1 \) and \( xy^iz \in A \) for all \( i = 0, 1, 2, ... \)

\[ \square \]

**Observe:** The pumping lemma does not give us liberty to choose \( x \) and \( y \) as we please.
**Example**

Let us assume that \( L_{\text{parenth}} \) is a regular language. By the pumping lemma there exists some number \( p \) s.t. strings of \( L_{\text{parenth}} \) of length at least \( p \) can be pumped. Let us choose \( s = (p)^p \). Then \( |s| = 2p > p \).

By Lemma 1.70 \( s \) can be divided into three parts \( s = xyz \) s.t. \( |xy| \leq p \) and \( |y| \geq 1 \). Therefore, it must be that

- \( x = (i \ i \leq p-1) \),
- \( y = (j \ j \geq 1) \), and
- \( z = (p\cdot(i+j))p \).

By our assumption \( xy^kz \in L_{\text{parenth}} \) for all \( k = 0, 1, 2, ... \), but for example

\[
xy^0z = xz = (i \ (p\cdot(i+j))p = (p\cdot j)p \not\in L_{\text{parenth}},
\]

because \( p-j \neq p \) since \( j \geq 1 \).

Hence, \( L_{\text{parenth}} \) cannot be a regular language.
The main limitation that finite automata have is that they have no (external) means of keeping track of an unlimited number of possibilities; i.e., to count.

Consider the following two languages:

\[
C = \{ w \mid w \text{ has an equal number of 0s and 1s} \}
\]

\[
D = \{ w \mid w \text{ has an equal number of occurrences of 01 and 10 as substrings} \}
\]

At first glance a recognizing machine needs to count in each case.

The language \( C \) contains \( \{ 0^k 1^k \mid k \geq 0 \} \) as a subset and, hence, the nonregularity of \( L_{\text{parenth}} \) proves that of \( C \).

Surprisingly, \( D \) is regular.
Example 1.75

Let $F = \{ ww \mid w \in \{ 0, 1 \}^* \}$. We show that $F$ is not regular.

Assume that $F$ is regular. Let $p$ be the pumping length given by the pumping lemma. Let $s$ be the string $0^p10^p1$. Because $s$ is a member of $F$ and it has length more than $p$, the pumping lemma guarantees that $s$ can be split into pieces $s = xyz$, satisfying the three conditions of the lemma. We show that this outcome is impossible.

Because $|xy| \leq p$, $x$ and $y$ must consist only of 0s, so $xyyz \notin F$. More exactly, $x = 0^i$, $y = 0^j$, and $z = 0^{p-(i+j)}10^p1$.

Therefore, $xy^2z = xyyz = 0^{i+j+p-(i+j)}10^p1 = 0^{p+j}10^p1$ which does not belong to $F$ since $0^{p+j}1$ has more zeros than $0^p1$ since by pumping lemma $j \geq 1$. Hence, $F$ is not a regular language.
Example 1.77

Let \( E = \{ 0^i 1^j \mid i > j \} \). We show that \( E \) is not regular.

Assume that \( E \) is regular. Let \( p \) be the pumping length for \( E \) given by the pumping lemma. Let \( s \) be the string \( 0^{p+1} 1^p \). Then \( s \) can be split into \( xyz \) satisfying the conditions of the pumping lemma.

Because \( |xy| \leq p \), \( x \) and \( y \) must consist only of 0s: \( x = 0^i \) and \( y = 0^j \)

Let us examine the string \( xyyz \) to see whether it can be in \( E \).
Adding an extra copy of \( y \) increases the number of 0s. But \( E \) contains all strings in \( 0^* 1^* \) that have more 0s than 1s, so increasing the number of 0s will still give a string in \( E \).

We need to pump down: \( xy^0z = xz = 0^{i+p+1-(i+j)} 1^p = 0^{p+1-j} 1^p \notin E \) since \( p+1-j \leq p \) because by assumption \( j \geq 1 \). Hence, the claim follows.
2. Context-Free Languages

• The language of balanced pairs of parentheses is not a regular one
• On the other hand, it can be described using the following substitution rules
  1. $S \rightarrow \varepsilon$ and
  2. $S \rightarrow (S)$

• These productions generate the strings of the language $L_{\text{parenth}}$ starting from the start variable $S$

$$S^2 \Rightarrow (S)^2 \Rightarrow ((S))^2 \Rightarrow (((S)))^1 \Rightarrow (((\varepsilon))) = ((( )))$$
• The string being described is generated by substituting \textit{variables} one by one according to the given rules.
• The string surrounding a variable does not determine the chosen production ⇒ \textit{context-free grammar}.
• One often abbreviates
  \[ A \rightarrow w_1 \mid \ldots \mid w_k \]
  to describe the alternative productions associated with the variable \( A \)
  \[ A \rightarrow w_1, \ldots, A \rightarrow w_k \]

• \( S \rightarrow \varepsilon \mid (S) \)
Simple arithmetic expressions
($E = \text{expression, } T = \text{term and } F = \text{factor}$)

\[
\begin{align*}
E & \rightarrow E + T \mid T \\
T & \rightarrow T \times F \mid F \\
F & \rightarrow (E) \mid a
\end{align*}
\]

Generation the expression $(a + (a)) \times a$

\[
\begin{align*}
E & \Rightarrow T \Rightarrow T \times F \Rightarrow F \times F \Rightarrow (E) \times F \Rightarrow (E + T) \times F \Rightarrow \\
(T + T) \times F & \Rightarrow (F + T) \times F \Rightarrow (a + T) \times F \Rightarrow (a + F) \times F \Rightarrow \\
(a + (E)) \times F & \Rightarrow (a + (T)) \times F \Rightarrow (a + (F)) \times F \Rightarrow \\
(a + (a)) \times F & \Rightarrow (a + (a)) \times a
\end{align*}
\]
Definition 2.2 A context-free grammar is a 4-tuple
\[ G = (V, \Sigma, R, S) \], where
- \( V \) is a finite set called the variables,
- \( \Sigma \) is a finite set, disjoint from \( V \), called the terminals
- \( V \cup \Sigma \) is the alphabet of \( G \),
- \( R \subseteq V \times (V \cup \Sigma)^* \) is a finite set of rules, and
- \( S \in V \) is the start variable

\((A, w) \in R\) is usually denoted as \( A \rightarrow w \)
• Let $G = (V, \Sigma, R, S)$, strings $u, v, w \in (V \cup \Sigma)^*$, and $A \rightarrow w$ a production in $R$

• $uAv$ yields string $uwv$ in grammar $G$, written $uAv \Rightarrow_G uwv$

• String $u$ derives string $v$ in grammar $G$, written $u \Rightarrow_G v$,
if a sequence $u_1, u_2, \ldots, u_k \in (V \cup \Sigma)^*$ ($k \geq 0$) exists s.t. $u \Rightarrow_G u_1 \Rightarrow_G u_2 \Rightarrow_G \ldots \Rightarrow_G u_k \Rightarrow_G v$

• $k = 0$: $u \Rightarrow_G u$ for any $u \in (V \cup \Sigma)^*$
• \( u \in (V \cup \Sigma)^* \) is a sentential form of \( G \) if
  \[ S \Rightarrow_G u \]

• A sentential form consisting of only terminals \( w \in \Sigma^* \) is a sentence of \( G \)

• The language of the grammar \( G \) consists of sentences
  \[ L(G) = \{ w \in \Sigma^* \mid S \Rightarrow_G w \} \]

• A formal language \( L \subseteq \Sigma^* \) is context-free, if it can be generated using a context-free grammar
A context-free grammar is right-linear if all its productions are of type \( A \rightarrow \varepsilon \) or \( A \rightarrow aB \)

**Theorem** Any regular language can be generated using a right-linear context-free grammar.

**Theorem** Any right-linear context-free language is regular.

- Hence, right-linear grammars generate exactly regular languages
- However, there are context-free languages which are not regular; e.g., the language of balanced pairs of parentheses \( L_{\text{parenth}} \)
- Therefore, context-free languages are a proper superset of regular languages
Ambiguity

- The sequence of one-step derivations leading from the start variable $S$ to string $w$

$$S \Rightarrow w_1 \Rightarrow ... \Rightarrow w_k \Rightarrow w$$

is called the \textit{derivation} of $w$

In the grammar for arithmetic expressions the sentence $a+a$ can be derived in many different ways:

1. $E \Rightarrow E + T \Rightarrow T + T \Rightarrow F + T \Rightarrow a + T \Rightarrow a + F \Rightarrow a + a$
2. $E \Rightarrow E + T \Rightarrow E + F \Rightarrow T + F \Rightarrow F + F \Rightarrow F + a \Rightarrow a + a$
3. $E \Rightarrow E + T \Rightarrow E + F \Rightarrow E + a \Rightarrow T + a \Rightarrow F + a \Rightarrow a + a$

- The differences caused by varying substitution order of variables can be abstracted away by examining \textit{parse trees}
• Context-free grammar $G$ is *ambiguous* if some sentence of $G$ has two (or more) distinct parse trees
• Otherwise the grammar is *unambiguous*

• Language that has no unambiguous context-free grammar is *inherently ambiguous*

• E.g. language $\{ a^i b^j c^k \mid i = j \lor j = k \}$ is inherently ambiguous

• An alternative grammar for the simple arithmetic expressions:

$$E \rightarrow E + E \mid E \times E \mid (E) \mid a$$
\( a + a \times a \)
Chomsky Normal Form

Definition 2.8 A context-free grammar is in Chomsky normal form (CNF), if

- At most the start variable $S$ derives the empty string,
- Every rule is of the form $A \rightarrow BC$ or $A \rightarrow a$
  (except maybe $S \rightarrow \epsilon$),
- The start variable $S$ does not appear in the right-hand side of any rule.

Theorem 2.9 Any context-free language is generated by a context-free grammar in CNF.

Proof We convert any grammar into CNF. The conversion has three stages. First, we add a new start variable. Then, we eliminate all $\epsilon$ rules and unit rules.
Eliminating $\varepsilon$ rules

**Lemma**  Any context-free language can be converted into an equivalent grammar in which at most the start variable derives the empty string.

**Proof**
Let $G = (V, \Sigma, R, S)$. Computing the variables of $G$ that derive the empty string:

- **NULL** = $\{ A \in V \mid A \rightarrow \varepsilon \in R \}$
- Repeat until set **NULL** does not change any more:

  $$\text{NULL} += \{ A \in V \mid A \rightarrow B_1\ldots B_k \in R, \ B_i \in \text{NULL} \ \forall \ i = 1, \ldots, k \}$$
Each rule $A \rightarrow X_1 \ldots X_k$ in $G$ is replaced by the set of all such rules that are of form $A \rightarrow \alpha_1 \ldots \alpha_k$, where

$$\alpha_i = \begin{cases} X_i & \text{if } X_i \not\in \text{NULL} \\ X_i | \varepsilon & \text{if } X_i \in \text{NULL} \end{cases}$$

In the end we remove all rules that have the form $A \rightarrow \varepsilon$.

If $S \rightarrow \varepsilon$ belongs to the removed rules, we take a new start variable $S'$ for the grammar and give it rules $S' \rightarrow S | \varepsilon$. 

□
$S \rightarrow A \mid B$
$A \rightarrow aBa \mid \epsilon$
$B \rightarrow bAb \mid \epsilon$

$S \rightarrow A \mid B \mid \epsilon$
$A \rightarrow aBa \mid aa \mid \epsilon$
$B \rightarrow bAb \mid bb \mid \epsilon$

$S' \rightarrow S \mid \epsilon$
$S \rightarrow A \mid B$
$A \rightarrow aBa \mid aa$
$B \rightarrow bAb \mid bb$

NULL = { A, B, S }
Eliminating unit rules

A *unit rule* has the form $A \rightarrow B$, where $A$ and $B$ are variables.

**Lemma** *Any context-free language can be converted into an equivalent grammar which has no unit rules.*

**Proof** Let $G = (V, \Sigma, R, S)$.

Computing the unit followers for each variable in $G$:

1. $F(A) = \{ B \in V \mid A \rightarrow B \in R \}$
2. Until the $F$-sets do not change anymore
   \[ F(A) += \{ F(B) \mid A \rightarrow B \in R \} \]

In the end we remove all unit rules in $G$ and replace them by all rules of the form $A \rightarrow \Omega$, where $B \in F(A)$ and $B \rightarrow \Omega$. □
\[ S' \rightarrow S | \varepsilon \]
\[ S \rightarrow A | B \]
\[ A \rightarrow aBa | aa \]
\[ B \rightarrow bAb | bb \]

\[ F(S') = \{ S, A, B \} \]
\[ F(S) = \{ A, B \} \]
\[ F(A) = \emptyset \]
\[ F(B) = \emptyset \]

\[ S' \rightarrow aBa | aa | bAb | bb | \varepsilon \]
\[ S \rightarrow aBa | aa | bAb | bb \]
\[ A \rightarrow aBa | aa \]
\[ B \rightarrow bAb | bb \]
Once all $\varepsilon$ rules and unit rules have been eliminated, all rules have form $A \rightarrow a$, $A \rightarrow X_1...X_k$, $k \geq 2$, or $S \rightarrow \varepsilon$.

For every $a \in \Sigma$ we add to the grammar the variable $C_a$ and rule $C_a \rightarrow a$.

A rule $A \rightarrow X_1...X_k$, $k \geq 2$, is replaced by a set of rules

$$A \rightarrow X'_1 A_1$$
$$A_1 \rightarrow X'_2 A_2$$
$$\vdots$$
$$A_{k-2} \rightarrow X'_{k-2} A_{k-1}$$
$$A_{k-1} \rightarrow X'_{k-1} X'_k,$$

where

$$X'_i = \begin{cases} X_i & \text{if } X_i \in V \\ C_a & \text{if } X_i = a \in \Sigma \end{cases}$$
\[
\begin{align*}
S' & \rightarrow aBa \mid aa \mid bAb \mid bb \mid \varepsilon \\
S & \rightarrow aBa \mid aa \mid bAb \mid bb \\
A & \rightarrow aBa \mid aa \\
B & \rightarrow bAb \mid bb \\
S' & \rightarrow C_aS'_1 \\
S'_1 & \rightarrow BC_a \\
S' & \rightarrow C_aC_a \\
S' & \rightarrow C_bS'_2 \\
S'_2 & \rightarrow AC_b \\
S' & \rightarrow C_bC_b \\
S' & \rightarrow \varepsilon \\
S & \rightarrow C_aS_1 \\
S_1 & \rightarrow BC_a \\
S & \rightarrow C_aC_a \\
S & \rightarrow C_bS_2 \\
S_2 & \rightarrow AC_b \\
S & \rightarrow C_bC_b \\
A & \rightarrow C_aA_1 \\
A_1 & \rightarrow BC_a \\
A & \rightarrow C_aC_a \\
B & \rightarrow C_bB_1 \\
B_1 & \rightarrow AC_b \\
B & \rightarrow C_bC_b \\
C_a & \rightarrow a \\
C_b & \rightarrow b
\end{align*}
\]
Algorithm CYK

- The strings of a context-free grammar that has been converted into CNF can be parsed in $\theta(n^3)$ time using the Cocke-Younger-Kasami algorithm.
- In other words, context-free languages can be efficiently recognized.
- The operating principle of algorithm CYK is dynamic programming.
- For each substring we tabulate those variables from which the substring can be derived from.
- If in the end the start variable of the grammar belongs to the set of variables that derive the whole string, the string at hand belongs to the language.
2.2 Pushdown Automata

- Pushdown automata are like NFAs, but have an extra component: (an infinite) stack
- We can write a new symbol on the stack at the top by pushing it
- We can read and remove the top symbol from the stack by popping it
- In a pushdown automaton the transitions always also concern the stack

- The stack gives the automaton a "memory" by which we can avoid some of the limitations that finite automata have
Definition 2.13

A pushdown automaton is a 6-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$, where

- $Q$ is the finite set of states,
- $\Sigma$ is the input alphabet,
- $\Gamma$ is the stack alphabet,
- $q_0 \in Q$ is the start state,
- $F \subseteq Q$ is the set of accept states, and
- $\delta$ is the set-valued transition function:

$$
\delta: Q \times \Sigma \times \Gamma \rightarrow P(Q \times \Gamma)
$$
In general pushdown automata are nondeterministic:
\[ \delta(r, x, a) = \{ (r_1, b_1), \ldots, (r_k, b_k) \} \]

- By reading the input symbol \( x \) and stack symbol \( a \)
- The automaton may transfer from state \( r \) to one of the states \( r_1, \ldots, r_k \), and
- Simultaneously replace the top symbol of the stack by one of the symbols \( b_1, \ldots, b_k \).

1. If \( x = \varepsilon \), the automaton transfers without reading an input symbol;
2. If \( a = \varepsilon \), the automaton does not read a stack symbol, but writes a new symbol at the top of the stack, leaving the old top symbol as is (push);
3. If \( a \neq \varepsilon \) and \( b_i = \varepsilon \), the top symbol of the stack is read and removed, but no new symbol is written in its stead (pop)
A pushdown automaton $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ accepts the string $w \in \Sigma^*$ if

- it can be written as $w = w_1w_2...w_m$, where each $w_i \in \Sigma_\varepsilon$, and furthermore there exists
- a sequence of states $r_0, r_1, ..., r_m \in Q$ and
- strings $s_0, s_1, ..., s_m \in \Gamma^*$

satisfying the following three conditions.

1. In the start $M$ is in the start state and with an empty stack:
   
   $r_0 = q_0$ and $s_0 = \varepsilon$;

2. $(r_{i+1}, b) \in \delta( r_i, w_{i+1}, a)$ for all $i \in \{ 0, ..., m-1 \}$,
   where $s_i = at$ and $s_{i+1} = bt$ for some $a, b \in \Gamma_\varepsilon$ and $t \in \Gamma^*$;

3. $r_m \in F$. 

The language of balanced pairs of parentheses \( \{ (^k)^k \mid k \geq 0 \} \) is a context-free language that is not a regular language. It can be recognized with a pushdown automaton \( M = (Q, \Sigma, \Gamma, \delta, q_0, F) \):

- \( Q = \{ q_0, q_1, q_2, q_3 \} \),
- \( \Sigma = \{ (, ) \} \),
- \( \Gamma = \{ $, € \} \),
- \( q_0 \) is the start state,
- \( F = \{ q_0, q_3 \} \) and
- \( \delta \) is:

\[
\delta(q_0, (, \varepsilon) = \{ (q_1, €) \},
\delta(q_1, (, \varepsilon) = \{ (q_1, $) \},
\delta(q_1, , $) = \{ (q_2, \varepsilon) \},
\delta(q_1, , €) = \{ (q_3, \varepsilon) \},
\delta(q_2, , $) = \{ (q_2, \varepsilon) \},
\delta(q_2, , €) = \{ (q_3, \varepsilon) \},
\delta(q, \sigma, \gamma) = \emptyset \quad \text{for other triples } (q, \sigma, \gamma)
\]
\(\delta(q_0, (((()))), \varepsilon) \Rightarrow \delta(q_1, ())), \varepsilon) \Rightarrow \delta(q_1, ()), $$\varepsilon) \Rightarrow \delta(q_1, ))), $$\varepsilon) \Rightarrow \delta(q_2, ))), $$\varepsilon) \Rightarrow \delta(q_2, ), \varepsilon) \Rightarrow \delta(q_3, \varepsilon, \varepsilon)\)

\(q_3 \in F, \text{ and hence } (((()))) \in L(M)\)
Theorem 2.20  *A language is context free if and only if some pushdown automaton recognizes it.*

- A pushdown automaton $M$ is deterministic if every configuration $(r, x, a)$ has at most one possible successor $(r', x', a')$, for which $(r, x, a) \Rightarrow_M (r', x', a')$
- Nondeterministic pushdown automata are strictly more powerful than deterministic ones. For example, the language 
  \[ \{ w w^R \mid w \in \{ a, b \}^* \} \]
  cannot be recognized using a deterministic pushdown automaton
- Deterministic context-free languages can be parsed more efficiently than general context-free languages
Nondeterministic PDA for recognizing \( \{ ww^R \mid w \in \{0, 1\}^* \} \)
3. The Church-Turing Thesis

**Church-Turing thesis**  
*Any mechanically solvable problem can be solved using a Turing machine.*

Many formulations of mechanical computation have an equal computing power with Turing machines:

- RAM machines, simple random access computer models
- Programming languages (1950s)
- String rewriting systems (Post, 1936 and Markov, 1951)
- \(\lambda\)-calculus (Church, 1936)
- Recursively defined functions (Gödel and Kleene, 1936)
3.1 Turing Machines

- Alan Turing (1912-1954) 1935-36
- Turing machine (TM) is similar to a finite automaton but with an unlimited and unrestricted memory (tape)
- TM can read and write the memory cells of the tape
- We can move to either direction in the tape
- The input to a TM is given in the beginning of the tape
- A TM has both an accepting final state, accept, and a rejecting one, reject
- The remainder of the tape is filled with blank characters (\`\`\'), which cannot be a symbol in the input alphabet
• In one transition step a TM always reads one symbol and decides based on it and the state in which the machine finds itself
  • The new state,
  • The symbol that is written on the tape, and
  • The direction into which we move the tape head

• If the TM halts in the accepting final state, accept, the input string belongs to the language recognized by the TM
• If the TM halts in the rejecting final state, reject, or if it does not halt at all, then the input string does not belong to the to the language
Definition 3.3

A Turing machine is a 7-tuple

\[ M = (Q, \Sigma, \Gamma, \delta, q_0, \text{accept}, \text{reject}) \],

where

- \( Q \) is a finite set of states,
- \( \Sigma \) is the input alphabet, not containing the blank symbol, \( \Box \notin \Sigma \),
- \( \Gamma \) is the tape alphabet, where \( \Box \in \Gamma \) and \( \Sigma \subseteq \Gamma \),
- \( q_0 \in Q \) is the start state,
- \( \text{accept} \in Q \) is the accepting final state,
- \( \text{reject} \in Q \) is the rejecting final state, and
- \( \delta : Q' \times \Gamma \to Q \times \Gamma \times \{L, R\} \) is the transition function, where \( Q' = Q \setminus \{\text{accept}, \text{reject}\} \).
The value of transition function

\[ \delta(q, a) = (q', b, \Delta) \]

means that

- when the TM reads tape symbol \( a \) while in state \( q \),
- it transfers to state \( q' \), writes symbol \( b \) into the tape cell that was just read, and moves the tape head one position to direction \( \Delta \in \{ L, R \} \)

From the values of the transition function it is required that

1. You cannot move to the left of the leftmost cell in the tape. If that is ever tried, the tape head stays in the same place.
2. If \( b = \square \), then \( a = \square \) and \( \Delta = L \).
A Turing machine for recognizing $\{ a^{2^k} | k \geq 0 \}$:
The configuration of a TM is denoted by \( u q a v \), where

- \( q \in Q \),
- \( a \in \Gamma \cup \{ \varepsilon \} \)
- \( u, v \in \Gamma^* \)

Intuitively

- The TM is in state \( q \),
- The tape head is positioned at a cell containing symbol \( a \),
- The contents of the tape from the left-most cell to the left of the tape head is \( u \), and
- Its contents from the right of the tape head to the end of used space is \( v \)

- E.g., \( 1011 \, q_7 \, 01111 \)
  - \( u = 1011, \, q = q_7, \, a = 0 \), and \( v = 1111 \)
• The start configuration with input $w$ is $q_0w$

• By $ua q_i bv \Rightarrow_M u q_j acv$ we denote that configuration $ua q_i bv$ yields configuration $u q_j acv$

• It is defined as follows
  • If $\delta(q_i, b) = (q_j, c, L)$, then $ua q_i bv \Rightarrow_M u q_j acv$,
  • if $\delta(q_i, b) = (q_j, c, R)$, then $ua q_i bv \Rightarrow_M uac q_j v$,
• When the tape head is on the left-most cell, the configuration is $q_i b v$ and
  • transition $\delta(q_i, b) = (q_j, c, L)$ yields configuration $q_j c v$
  • transition $\delta(q_i, b) = (q_j, c, R)$ yields configuration $c q_j v$

• When the tape head is on the right-most non-blank cell, the configuration is $u a q_i \square$ and
  • transition $\delta(q_i, \square) = (q_j, c, L)$ yields configuration $u q_j a c$
  • transition $\delta(q_i, \square) = (q_j, c, R)$ yields configuration $u a c q_j \square$
  • only in the former situation can we have $c = \square$
• The value of the transition function is unspecified in final states accept and reject, and thus
  • configurations with accept or reject do not yield another configuration, instead the machine halts in them
• A Turing machine $M$ accepts input $w$ if a sequence of configurations $C_1, C_2, ..., C_k$ exists, where
  1. $C_1$ is the start configuration of $M$ on input $w$,
  2. each $C_i$ yields $C_{i+1}$, and
  3. $C_k$ is an accepting configuration.
• The collection of strings that $M$ accepts is the language of $M$, or the language recognized by $M$, denoted $L(M)$
A TM recognizing
the language
\( \{ a^k b^k c^k \mid k \geq 0 \} \):
3.2 Variants of Turing Machines

The following generalizations of Turing machines do not change the collection of languages recognized.

**Multitrack machines**

- The tape of a TM consists of $k$ parallel tracks.
- The TM still has only one tape head.
- The machine reads and writes each track in each computation step (transition).
- We fill the contents of each track to be of equal length with a special empty symbol (#).
- The input is given at the left end of track 1.
• A multitrack Turing machine is easy to simulate using a standard TM
• It is enough to have such a tape alphabet whose symbols can (if necessary) represent all possible $k$ symbols on top of each other for a $k$-track TM
• For example:

$$\delta(q,(a_1,\ldots,a_k)) \rightarrow \hat{\delta}\left(q,\begin{bmatrix} a_1 \\ \vdots \\ a_k \end{bmatrix}\right)$$
Doing the expansion of the alphabet systematically and changing all the symbols of the input

\[
\begin{array}{c}
a \\
\#
\end{array}
\rightarrow
\begin{array}{c}
a \\
\#
\end{array}
\]

into track 1, we can simulate the multitrack Turing machine with a standard one

**Theorem** If a formal language can be recognized using a $k$-track Turing machine, then it can also be recognized with a standard Turing machine.
Multitape Turing Machines

- Now a Turing machine may have \( k \) tapes, each with its own tape head.
- The TM reads and writes each tape in each step of computation (transition).
- The tape heads move independent of each other.
- The input is given at the left end of tape 1.
- It is easy to simulate a multitape Turing machine using a multitrack one.
- For each tape we have two tracks one of which corresponds to the tape and its contents and the other track holds a marker in the position of the tape head.
- The marker is moved by simulating the movement of the tape head in the multitape TM.
• By reading all tracks from left to right we find out all the symbols that are at the positions of the tape heads
• Then we know which symbols to write into the tapes and to which directions to move the tape heads
• We can implement/simulate the required changes in a right-to-left sweep over the tracks

**Theorem 3.13** Every multitape Turing machine has an equivalent single-tape (standard) Turing machine.
Nondeterministic Turing Machines

- By giving the transition function of a Turing machine the potential of "prediction" we can define a nondeterministic Turing machine.
- It is not a realistic model of mechanical computation, but it is useful in formal description of problems and to show their solvability.
- Nondeterministic Turing machines have equivalent power in recognizing languages with the deterministic TMs.
- Nevertheless, they have a central role in computational complexity theory.
- A nondeterministic TM accepts its input if some branch of the computation leads to the accept state.
A nondeterministic Turing machine is a 7-tuple
\[ M = (Q, \Sigma, \Gamma, \delta, q_0, \text{accept}, \text{reject}), \]
where
\[ \delta: Q' \times \Gamma \rightarrow P(Q \times \Gamma \times \{L, R\}), \]
In which \( Q' = Q \setminus \{\text{accept, reject}\} \)

The value of the transition function
\[ \delta(q, a) = \{ (q_1, b_1, \Delta_1), \ldots, (q_k, b_k, \Delta_k) \} \]
means that
- when reading symbol \( a \) while in state \( q \)
- the TM can choose the triple \( (q_i, b_i, \Delta_i) \) as it pleases
  (which ever of them is best for its task)
Composite Numbers

- Recognizing nonnegative *composite numbers*: 
  \[ n = pq? \ (p, q > 1) \]

- If a number is not composite, then it is a **prime number**

- All known deterministic tests for composite numbers end up to go through a large number of potential factors in the worst case (basis for encryption)

- However, using a nondeterministic Turing machine “recognizing” composite numbers is easy

- The nondeterministic Turing machine does not give us an algorithm; it is only a computational description of composite numbers
We need Turingin machines …

- **CHECK-MULT** recognizes the language
  \[ \{ n#p#q \mid n, p, q \in \{ 0, 1 \}^*, n = pq \} \]
- **GEN-INT** generates an arbitrary integer (>1) in binary to the end of the tape
- **GO-START** positions the tape head to the first memory cell of the tape

… to construct the TM
- **TEST-COMPOSITE** which recognizes the language
  \[ \{ n \in \{ 0, 1 \}^* \mid \text{n is a composite number} \} \]
Turing Machine GEN-INT

- $q_0$: $0 \rightarrow 0, R$
- $q_0$: $1 \rightarrow 1, R$
- $q_0$: $\square \rightarrow \# , R$
- $q_1$: $\square \rightarrow 0, R$
- $q_1$: $\square \rightarrow 1, R$
- $q_1$: $\square \rightarrow \# , R$
- $q_1$: $\square \rightarrow 1, R$
- $q_1$: $\square \rightarrow 0, R$
- $q_1$: $\square \rightarrow 1, R$
- End state
Turing machine TEST-COMPOSITE
Deterministic Primality Test

• In 2002 Agarwal, Kayal & Saxena from India managed to develop the first "efficient" \( (\log n)^{12} \) deterministic algorithm for recognizing prime numbers

• This problem has been studied for centuries (Sieve of Eratosthenes, from approximately 240 before common era)

• The algorithm of AKS borrows techniques from randomized primality test algorithms, which are still today the most useful (efficient) solution techniques

• Because recognizing composite numbers is a complementary problem to recognizing prime numbers, also it can be solved in polynomial time deterministically

• However, one does not know how to factorize a composite number (it is, though, believed to be possible)
Theorem 3.16  Every nondeterministic Turing machine has an equivalent deterministic Turing machine.

- A nondeterministic TM can be simulated with a 3-tape Turing machine, which goes systematically through the possible computations of the nondeterministic TM
- Tape 1 maintains the input
- Tape 2 simulates the tape of the nondeterministic TM
- Tape 3 keeps track of the situation with the possible computations
• The tree formed of the possible computation paths of the nondeterministic TM has to be examined using breadth-first algorithm, not in depth-first order
• Let \( b \) be the largest number of possible successor states for any state in the Turing machine
• Each node in the tree can be indexed with a string over the alphabet \{ 1, 2, ..., \( b \) \}
• For example, 231 is the node that is the first child of the third child of the second child of the root of the tree
• All strings do not correspond to legal computations (nodes of the tree) — those get rejected
• Going through the strings in the lexicographic order corresponds to examining all computation paths, and the search order in the tree is breadth-first search
Working idea:

- Copy the input from tape 1 to tape 2
- Tape 3 tells us which is (in the lexicographic order) the next computation alternative for the nondeterministic TM. We simulate that computation targeting the updates on the tape of the original TM into tape 2 of the simulating TM
- Observe that there is a finite number of transition possibilities (successor states)
- Systematic examination of the possible computations of the nondeterministic TM ends into the accepting final state only if the original TM has an accepting computation path
- If no accepting computation exists, the simulating TM never halts
UNRESTRICTED GRAMMARS

- Context-free grammar allows to substitute only variables with strings
- In an unrestricted grammar (or a rewriting system) one may substitute any non-empty string (containing variables and terminals) with another one (also with the empty string $\varepsilon$)

An unrestricted grammar is a 4-tuple $G = (V, \Sigma, R, S)$, where
- $V$ is the set of variables,
- $\Sigma$ is the set of terminals,
- $\Gamma = V \cup \Sigma$ is the alphabet of $G$,
- $R \subseteq \Gamma^+ \times \Gamma^*$ is the set of rules, and
- $S \in V$ is the start variable

$(w, w') \in R$ is denoted as $w \rightarrow w'$
Let

- $G = (V, \Sigma, R, S)$,
- strings $v \in \Gamma^+$ and $u, w, x \in \Gamma^*$ as well as
- $v \rightarrow x$ a rule in $R$

- $uvw$ yields string $uxw$ in grammar $G$,
  $$uvw \Rightarrow_G uxw$$

- String $v$ derives string $w$ in grammar $G$,
  $$v \Rightarrow_G w,$$
  if there exists a sequence $v_1, v_2, \ldots, v_k \in \Gamma^*$ ($k \geq 0$) s.t.
  $$v \Rightarrow_G v_1 \Rightarrow_G v_2 \Rightarrow_G \ldots \Rightarrow_G v_k \Rightarrow_G w$$

- $k = 0$: $v \Rightarrow_G v$ for any $v \in \Gamma^*$
• $u \in \Gamma^*$ is a sentential form of $G$ if $S \Rightarrow_G u$

• A sentential form consisting only of terminal symbols $w \in \Sigma^*$ is a sentence of $G$

• The language of the grammar $G$ consists of sentences

\[
L(G) = \{ w \in \Sigma^* | S \Rightarrow_G w \}
\]

The language $\{ a^k b^k c^k | k \geq 0 \}$ is not a context-free one; it can be generated with an unrestricted grammar, which

1. Generates the variable sequence $L(ABC)^k$ (or $\varepsilon$)
2. Orders the variables lexicographically $\Rightarrow LA^k B^k C^k$
3. Replaces the variables with terminals
\[ S \rightarrow LT | \varepsilon \quad \text{LA} \rightarrow a \]
\[ T \rightarrow ABCT | ABC \quad aA \rightarrow aa \]
\[ BA \rightarrow AB \quad aB \rightarrow ab \]
\[ CB \rightarrow BC \quad bB \rightarrow bb \]
\[ CA \rightarrow AC \quad bC \rightarrow bc \]
\[ cC \rightarrow cc \]
For example, we can generate the sentence $aabbcc$ as follows

$$S \Rightarrow LT \Rightarrow LABCT \Rightarrow LABCABC$$
$$\Rightarrow LABACBC \Rightarrow LAABCBC$$
$$\Rightarrow LAABBCC \Rightarrow aABBCC$$
$$\Rightarrow aaBBCC \Rightarrow aabBCC$$
$$\Rightarrow aabbCC \Rightarrow aabbcC$$
$$\Rightarrow aabbcc$$
**Theorem** A formal language $L$ generated by an unrestricted grammar can be recognized with a Turing machine.

**Proof.** Let $G = (V, \Sigma, R, S)$ be the unrestricted grammar generating language $L$. We devise a two-tape nondeterministic Turing machine $M_G$ for recognizing $L$.

$M_G$ maintains the input string on tape 1. On tape 2 there is some sentential form of $G$ which we try to rewrite as the input string.

At the beginning tape 2 contains the start variable $S$.

The computation of $M_G$ repeats the following stages
1. The tape head of tape 2 is (non-deterministically) moved to some location on the tape;
2. we choose (non-deterministically) some rule of $G$ and try to apply it to the chosen location of the tape;
3. if the symbols on the tape match the symbols on the left-hand side of the rule, $M_G$ replaces them on tape 2 with the symbols on the right-hand side of the rule;
4. we compare the strings in tapes 1 and 2 with each other;
   a) if they are equal, the Turing machine enters the accepting final state and halts,
   b) otherwise, we go back to step 1
Theorem  If a formal language $L$ can be recognized with a Turing machine, then it can be generated with an unrestricted grammar.

Proof. Let $M = (Q, \Sigma, \Gamma, \delta, q_0, \text{accept}, \text{reject})$ be a standard Turing machine recognizing language $L$.

Let us compose an unrestricted grammar $G_M$ that generates $L$.

As variables of the grammar we take symbols representing all states $q \in Q$ of $M$. The configuration of the TM $uqav$ is represented as string $[uqav]$.

By the transition function of $M$ we give $G_M$ rules so that

$$[uqav] \Rightarrow_{G_M} [u'q'a'v'] \iff uqav \Rightarrow_{M} u'q'a'v'$$
Then

\[ x \in L(M) \iff [q_0x] \Rightarrow_{G_M} [u \text{accept} v], \ u, v \in \Sigma^* \]

There are three types of rules in \( G_M \):

1. Those that generate any string \( x[q_0x], x \in \Sigma^* \) and \([, ], q_0 \in V \) from the start variable:

\[
S \rightarrow T[q_0] \\
T \rightarrow \varepsilon \\
T \rightarrow aTA_a \\
A_a[q_0 \rightarrow [q_0A_a] \\
A_a b \rightarrow bA_a \\
A_a] \rightarrow a]
\]
2. Those that simulate the transition function of the Turing machine:

\[
\begin{align*}
\delta(q, a) &= (q', b, R) & qa &\rightarrow bq' \\
\delta(q, a) &= (q', b, L) & cqa &\rightarrow q'cb \\
\delta(q, \square) &= (q', b, R) & q] &\rightarrow bq'] \\
\delta(q, \square) &= (q', b, L) & cq] &\rightarrow q'cb] \\
\delta(q, \square) &= (q', \square, L) & cq] &\rightarrow q'c] \\
\end{align*}
\]
3. Those that replace a string of the form \([u\text{accept}v]\) to an empty string

\[
\text{accept} \rightarrow E_L E_R \\
aE_L \rightarrow E_L \\
[E_L \rightarrow \varepsilon \\
E_R a \rightarrow E_R \\
E_R] \rightarrow \varepsilon
\]

Now a string \(x\) in \(L(M)\) can be generated as follows

\[
S \Rightarrow_1 x[q_0x] \Rightarrow_2 x[u\text{accept}v] \Rightarrow_3 x
\]
3.3 The Definition of Algorithm

- The formulations of computation by Alonzo Church and Alan Turing were given in response to Hilbert’s tenth problem which he posed in 1900 in his list of 23 challenges for the new century.
- What Hilbert essentially asked for was an algorithm for determining whether a polynomial has an integral root.
- Today we know that this problem is algorithmically unsolvable.
- It is possible to give algorithms without them being exactly defined, but it is not possible to show that such cannot exist without a proper definition.
- It was not until 1970 that Matijasevič showed that no algorithm exists for testing whether a polynomial has integral roots.
Expressed as a formal language Hilbert’s tenth problem is

\[ D = \{ p \mid p \text{ is a polynomial with an integral root} \} \]

Concentrating on single variable polynomials we can see how the language \( D \) could be recognized.

- In order to find the correct value of the only variable, we go through its possible integral values: 0, 1, -1, 2, -2, 3, -3, ...
- If the polynomial attains value 0 with any examined value of the variable, then we accept the input.
- A similar approach is possible when there are multiple variables in the polynomial.
For a single variable polynomial the roots must lie within 

$$\pm k(\frac{c_{\text{max}}}{c_1}),$$

- where $k$ is the number of terms in the polynomial,
- $c_{\text{max}}$ is the coefficient with largest absolute value, and
- $c_1$ is the coefficient of the highest order term

If a root is not found within these bounds, the machine rejects

Matijasevič's theorem shows that calculating such bounds for multivariable polynomials is impossible

The language $D$ can, thus, be recognized with a Turing machine, but cannot be decided with a Turing machine (may never halt)
Computability Theory

- We will examine the *algorithmic solvability* of problems
  - I.e. solvability using Turing machines

- We make a distinction between cases in which formal languages can be recognized with a Turing machine and those in which the Turing machine is required to halt with each input

- It turns out that there are many natural and interesting problems that cannot be solved using a Turing machine

- Hence, by Church-Turing thesis these problems are unsolvable by a computer!
**Definition 3.5** Call a language **Turing-recognizable** (or recursively enumerable, RE-language) if some Turing machine recognizes it.

**Definition 3.6** Call a language **Turing-decidable** (or decidable, or recursive) if some TM decides it (halts on every input, is total).

- The decision problem corresponding to language $A$ is **decidable** if $A$ is Turing-decidable.
- A problem that is not decidable is **undecidable**
- The decision problem corresponding to language $A$ is **semidecidable** if $A$ is Turing-recognizable
- Observe: an undecidable problem can be semidecidable.
Basic Properties of Turing-recognizable Languages

**Theorem A** Let $A, B \subseteq \Sigma^*$ be Turing-decidable languages. Then also languages

1. $\bar{A} = \Sigma^* \setminus A$,
2. $A \cup B$, and
3. $A \cap B$

are Turing-decidable.

**Proof.**

1. Let $M_A$ be the total TM recognizing language $A$. By exchanging the accepting and rejecting final state of $M_A$ with each other, we get a total Turing machine deciding the language $\bar{A}$.  


The deciding machine for the complement of $A$
2. Let $M_A$ and $M_B$, respectively, be the total Turing machines deciding $A$ and $B$.
   - Let us combine them so that we first check whether $M_A$ accepts the input.
   - If it does, so does the combined TM.
   - On the other hand, if $M_A$ rejects the input, we pass it on to TM $M_B$ check.
   - In this case $M_B$ decides whether the input will be accepted. $M_A$ must pass the original input to $M_B$.
   - It is clear that the combined TM is total and accepts the language

   \[
   A \cup B = \{ x \in \Sigma^* \mid x \in A \lor x \in B \}
   \]

3. The Turing-decidability of $(A \cap B)$ follows from the previous results because

   \[
   A \cap B = \overline{A} \cup \overline{B}
   \]
The deciding machine for the union of languages $A$ and $B$: 

![Diagram showing the deciding machine for the union of languages $A$ and $B$.]
Theorem B Let $A, B \subseteq \Sigma^*$ be Turing-recognizable languages. Then also languages $A \cup B$ and $A \cap B$ are Turing-recognizable.

Proof. Exercises.

- As a consequence of Theorem 4.22 we get a hold of languages that are not Turing-recognizable ($\bar{A}$ is the complement of $A$):

Theorem C Let $A \subseteq \Sigma^*$ be a Turing-recognizable language that is not Turing-decidable. Then $\bar{A}$ is not Turing-recognizable.
Theorem 4.22 Language $A \subseteq \Sigma^*$ is Turing-decidable $\iff$ $A$ and $\overline{A}$ are Turing-recognizable.

Proof.

$\Rightarrow$ If $A$ is Turing-decidable, then it is also Turing-recognizable. By Theorem A(1) the same holds also for $\overline{A}$.

$\Leftarrow$ Let $M_A$ and $M_{\overline{A}}$ be the TMs recognizing $A$ and $\overline{A}$.

For all $x \in \Sigma^*$ it holds that either $x \in A$ or $x \in \overline{A}$. In other words, either $M_A$ or $M_{\overline{A}}$ halts on input $x$.

Combining machines $M_A$ and $M_{\overline{A}}$ to run parallel gives a total Turing machine for recognizing $A$. \qed
Total Turing machine for recognizing language $A$:
Encoding Turing Machines

Standard Turing machine

\[ M = (Q, \Sigma, \Gamma, \delta, q_0, \text{accept}, \text{reject}) \]

where \( \Sigma = \{0, 1\} \), can be represented as a binary string as follows:

- \( Q = \{q_0, q_1, ..., q_n\} \), where \( q_{n-1} = \text{accept} \) and \( q_n = \text{reject} \)
- \( \Gamma = \{a_0, a_1, ..., a_m\} \), where \( a_0 = 0, a_1 = 1, a_2 = \square \)
- Let \( \Delta_0 = L \) and \( \Delta_1 = R \)
- The code for the transition function \( \delta \) rule

\[ \delta(q_i, a_j) = (q_r, a_s, \Delta_t) \]

is

\[ c_{ij} = 0^{i+1}10^{j+1}10^{r+1}10^{s+1}10^{t+1} \]
The code for the whole machine $M, \langle M \rangle$, is

$$111c_{00}11c_{01}11 \ldots 11c_{0m}11c_{10}11 \ldots 11c_{n-2,0}11 \ldots 11c_{n-2,m}111$$

For example, the code for the following TM is

$$\langle M \rangle = 1110101001010010100100100100100100100 \ldots 111$$

$$\delta(q_0, 0) = (q_1, 0, R) \quad \delta(q_0, 1) = (q_3, 1, R)$$

Thus, every standard Turing machine $M$ recognizing some language over the alphabet $\{0, 1\}$ has a binary code $\langle M \rangle$.

On the other hand, we can associate some Turing machine $M_b$ to each binary string $b$. 
A TM recognizing the language \{0^{2k} \mid k \geq 0\}:
• However, all binary strings are not codes for Turing machines
• For instance, 00, 011110, 111000111 and 1110101010111 are not legal codes for TMs according to the chosen encoding
• We associate with illegal binary strings a trivial machine, $M_{\text{triv}}$, rejecting all inputs:

\[
M_b = \begin{cases} 
  M, & \text{if } b = \langle M \rangle \text{ is the code for Turingin machine } M \\
  M_{\text{triv}}, & \text{otherwise}
\end{cases}
\]
• Hence, all Turing machines over \( \{ 0, 1 \} \) can be enumerated:

\[
M_\varepsilon, M_0, M_1, M_{00}, M_{01}, M_{10}, M_{11}, M_{000}, \ldots
\]

• At the same time we obtain an enumeration of the Turing-
recognizable languages over \( \{ 0, 1 \} \):

\[
L(M_\varepsilon), L(M_0), L(M_1), L(M_{00}), L(M_{01}), L(M_{10}), \ldots
\]

• A language can appear more than once in this enumeration

• By diagonalization we can prove that the language \( D \)
corresponding to the decision problem

\[
\text{Does the Turing machine } M \text{ reject its own description } \langle M \rangle?\]

is not Turing-recognizable

• Hence, the decision problem corresponding to \( D \) is unsolvable.
Lemma D  

Language $D = \{ b \in \{0, 1\}^* \mid b \notin L(M_b) \}$ is not Turing-recognizable.

Proof. Let us assume that $D = L(M)$ for some standard Turing machine $M$.

Let $d = \langle M \rangle$ be the binary code of $M$; i.e., $D = L(M_d)$. However,

$$d \in D \iff d \notin L(M_d) = D.$$  

We have a contradiction and the assumption cannot hold. Hence, there cannot exist a standard Turing machine $M$ s.t. $D = L(M)$.

Therefore, $D$ cannot be Turing-recognizable. \qed
4 Decidability

- The **acceptance problem** for DFAs: Does a particular DFA $B$ accept a given string $w$?
- Expressed as a formal language
  \[ A_{DFA} = \{ \langle B, w \rangle \mid B \text{ is a DFA that accepts string } w \} \]

- We simply need to represent a TM that decides $A_{DFA}$
- The Turing machine can easily simulate the operation of the DFA $B$ on input string $w$
  - If the simulation ends up in an accept state, *accept*
  - If it ends up in a nonaccepting state, *reject*
- DFAs can be represented (encoded) in the same vein as Turing machines
Theorem 4.1 \( A_{\text{DFA}} \) is a decidable language.

- NFAs can be converted to equivalent DFAs, and hence a corresponding theorem holds for them
- Regular expressions, on the other hand, can be converted to NFAs, and therefore a corresponding result also holds for them
- A different kind of a problem is emptiness testing: recognizing whether an automaton \( A \) accepts any strings at all

\[ E_{\text{DFA}} = \{ \langle A \rangle \mid A \text{ is a DFA and } L(A) = \emptyset \} \]

Theorem 4.4 \( E_{\text{DFA}} \) is a decidable language.
• The TM can work as follows on input $\langle A \rangle$:
  1. Mark the start state of $A$
  2. Repeat until no new states get marked:
     Mark any state that has a transition coming in from any state that is already marked
  3. If no accept state is marked, accept; otherwise, reject

• Also determining whether two DFAs recognize the same language is decidable

• The corresponding language is
  \[ EQ_{DFA} = \{ \langle A, B \rangle \mid A \text{ and } B \text{ are DFAs and } L(A) = L(B) \} \]

• The decidability of $EQ_{DFA}$ follows from Theorem 4.4 by turning to consider the symmetric difference of $L(A)$ and $L(B)$
\[ L(C) \quad = \quad (L(A) \setminus L(B)) \cup (L(B) \setminus L(A)) \]
\[ \quad = \quad (L(A) \cap \overline{L(B)}) \cup (\overline{L(A)} \cap L(B)) \]

- \( L(C) = \emptyset \) if and only if \( L(A) = L(B) \)
- Because the class of regular languages is closed under complementation, union, and intersection, the TM can construct automaton \( C \) given \( A \) and \( B \)
- We can use Theorem 4.4 to test whether \( L(C) \) is empty

**Theorem 4.5** \( EQ_{DFA} \) is a decidable language.

- Turning to context-free grammars we cannot go through all derivations because there may be an infinite number of them
In the acceptance problem we can consider the grammar converted into Chomsky normal form in which case any derivation of a string \( w \), \(|w| = n\), has length \( 2n - 1 \).

The emptiness testing of a context-free grammar \( G \) can be decided in a different manner:

- Mark all terminal symbols in \( G \).
- Repeat until no new variables get marked:
  - Mark any variable \( A \) where \( G \) has a rule \( A \rightarrow U_1...U_k \) and each symbol \( U_1...U_k \) has already been marked.
  - If the start variable has not been marked, accept; otherwise reject.

The equivalence problem for context-free grammars, on the other hand, is not decidable.
• Because of the decidability of the acceptance problem, all context-free languages are decidable
• Hence, regular languages are a proper subset of context-free languages, which are decidable
• Furthermore, decidable languages are a proper subset of Turing-recognizable languages
4.2 The Halting Problem

- The technique of *diagonalization* was discovered in 1873 by Georg Cantor who was concerned with the problem of measuring the sizes of infinite sets.

- For finite sets we can simply count the elements.

- Do infinite sets \( \mathbb{N} = \{ 0, 1, 2, \ldots \} \) and \( \mathbb{Z}^+ = \{ 1, 2, 3, \ldots \} \) have the same size?

  - \( \mathbb{N} \) is larger because it contains the extra element 0 and all other elements of \( \mathbb{Z}^+ \).
  - The sets have the same size because each element of \( \mathbb{Z}^+ \) can be mapped to an unique element of \( \mathbb{N} \) by \( f(z) = z - 1 \).
We have already used this comparison of sizes:

- $|A| \leq |B|$ iff there exists a **one-to-one** function $f: A \rightarrow B$
- One-to-one (injection): $a_1 \neq a_2 \Rightarrow f(a_1) \neq f(a_2)$

We have also examined the equal size of two sets $|A| = |B|$ through a bijective $f: A \rightarrow B$ mapping

- Correspondence (bijection) = one-to-one + **onto**
- Onto (surjection): $f(A) = B$ or $\forall b \in B: \exists a \in A: b = f(a)$
- A bijection uniquely pairs the elements of the sets $A$ and $B$

- $|\mathbb{N}| = |\mathbb{Z}^+|$
- An infinite set that has the same size as $\mathbb{N}$ is **countable**
\[ |\mathbb{Z}| = |\mathbb{N}| \]

\[ \begin{array}{ccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & \ldots \\
\downarrow & \downarrow & \downarrow & \uparrow & \uparrow & \uparrow & \uparrow & \\
0 & -1 & 1 & -2 & 2 & -3 & 3 & \ldots \\
\end{array} \]

\[ |\mathbb{N}^2| = |\mathbb{N}| \]

\[ 0,0 \quad 1,0 \quad 2,0 \quad 3,0 \quad 4,0 \quad \ldots \]
\[ 0,1 \quad 1,1 \quad 2,1 \quad 3,1 \quad 4,1 \quad \ldots \]
\[ 0,2 \quad 1,2 \quad 2,2 \quad 3,2 \quad 4,2 \quad \ldots \]
\[ 0,3 \quad 1,3 \quad 2,3 \quad 3,3 \quad 4,3 \quad \ldots \]
\[ 0,4 \quad 1,4 \quad 2,4 \quad 3,4 \quad 4,4 \quad \ldots \]

\[ |\mathbb{Z}^2| = |\mathbb{N}| \]

- \[ |\mathbb{N}| = |\mathbb{N}^2| \]
- \[ |\mathbb{N}^2| = |\mathbb{Z}^2| \text{ (relatively easy)} \]
- Follows by transitivity
The set of rational numbers

\[ \mathbb{Q} = \{ m/n \mid m, n \in \mathbb{Z} \land n \neq 0 \} \]

In between any two integers there is an infinite number of rational numbers

Nevertheless, \(|\mathbb{Q}| = |\mathbb{N}|\)

Mapping \(f: \mathbb{Q} \to \mathbb{Z}^2, f(m/n) = (m, n)\)

- Integers \(m\) and \(n\) have no common factors!
- Mapping \(f\) is a one-to-one. Hence, \(|\mathbb{Q}| \leq |\mathbb{Z}^2| = |\mathbb{N}|\)

On the other hand, \(\mathbb{N} \subseteq \mathbb{Q} \Rightarrow |\mathbb{N}| \leq |\mathbb{Q}|\)

\[ \therefore |\mathbb{Q}| = |\mathbb{N}| \]
• $|\mathbb{N}| < |\mathbb{R}|$

• Let us assume that the interval $[0, 1]$ is countable and apply Cantor's diagonalization to the numbers $x_1, x_2, x_3, \ldots$ in $[0, 1]$.

• Let the decimal representations of the numbers within the interval be (excluding infinite sequences of 9s)

\[ x_i = \sum_{j \in \mathbb{Z}^+} d_{ij} \cdot 10^{-j} \]

• Let us construct a new real number

\[ x = \sum_{j \in \mathbb{Z}^+} d_j \cdot 10^{-j} \]

such that

\[ d_j = \begin{cases} 0, & \text{if } d_{jj} > 0 \\ 1, & \text{if } d_{jj} = 0 \end{cases} \]
• If, for example

\[
x_1 = 0,23246...
\]
\[
x_2 = 0,30589...
\]
\[
x_3 = 0,21754...
\]
\[
x_4 = 0,05424...
\]
\[
x_5 = 0,99548...
\]
\[\vdots\]

then \(x = 0,01000...\)

• Hence, \(x \neq x_i\) for all \(i\)

• The assumption about the countability of the numbers within the interval \([0, 1]\) is false

• \(|\mathbb{R}| = |[0, 1]| \neq |\mathbb{N}|\)
Recall that …

- Finite languages \( \not\subset \) Regular languages \( \not\subset \) Context-free languages \( \not\subset \) Turing-decidable languages \( \not\subset \) Turing-recognizable
- There are automata and grammar descriptions for all these language classes
- Different variations of Turing machines and unrestricted grammars (e.g.) are universal models of computation
- A Turing-\textit{decidable} language has a \textit{total} TM that halts on each input
- A Turing-\textit{recognizable} language has a TM that halts on all positive instances, but not necessarily on the negative ones
- Turing-recognizable languages that are not decidable are semidecidable
- There are languages that are not even Turing-recognizable
Universal Turing Machines

• The universal language $U$ over the alphabet $\{0, 1\}$ is

$$U = \{ \langle M, w \rangle \mid w \in L(M) \}.$$ 

• The language $U$ contains information on all Turing-recognizable languages over $\{0, 1\}$:
  
  • Let $A \subseteq \{0, 1\}^*$ be some Turing-recognizable language and $M$ a standard TM recognizing $A$. Then
  
  $$A = \{ w \in \{0, 1\}^* \mid \langle M, w \rangle \in U \}.$$ 

• Also $U$ is Turing-recognizable.

• Turing machines recognizing $U$ are called universal Turing machines.
**Theorem E** Language $U$ is Turing-recognizable.

**Proof.** The following three-tape TM $M_U$ recognizes $U$

1. First $M_U$ checks that the input $cw$ in tape 1 contains a legal encoding $c$ of a Turing machine. If not, $M_U$ rejects the input

2. Otherwise $w = a_1a_2...a_k \in \{0, 1\}^*$ is copied to tape 2 in the form $00010^{a_1+1}10^{a_2+1}1...10^{a_k+1}10000$

3. Now $M_U$ has to find out whether the TM $M$ ($c = \langle M \rangle$) would accept $w$. Tape 1 contains the description $c$ of $M$, tape 2 simulates the tape of $M$, and tape 3 keeps track of the state of the TM $M$:

$$q_i \sim 0^{i+1}$$
4. $M_U$ works in phases, simulating one transition of $M$ at each step

1. First $M_U$ searches the position of the encoding of $M$ (tape 1) that corresponds to the simulated state (tape 3) of $M$ and the symbol in tape 2 at the position of the tape head

2. Let the chosen sequence of encoding be $0^{i+1}10^{j+1}10^{r+1}10^{s+1}10^{t+1}$, which corresponds transition function $\delta$ rule $\delta(q_i, a_j) = (q_r, a_s, \Delta_t)$.

   tape 3: $0^{i+1} \leftrightarrow 0^{r+1}$
   tape 2: $0^{j+1} \leftrightarrow 0^{s+1}$

   In addition the head of tape 2 is moved to the left so that the code of one symbol is passed, if $t = 0$, and to the right otherwise
3. When tape 1 does not contain any code for the simulated state \( q_i \), \( M \) has reached a final state. Now \( i = k + 1 \) or \( i = k + 2 \), where \( q_k \) is the last encoded state. The TM \( M_U \) transitions to final state \textit{accept} or \textit{reject}.

Clearly the TM \( M_U \) accepts the binary string \( \langle M, w \rangle \) if and only if \( w \in L(M) \).
Theorem F *Language* $U$ *is not decidable.*

**Proof.** Let us assume that $U$ has a total recognizer $M_U^T$.

Then we could construct a recognizer $M_D$ for the "diagonal language" $D$, which is not Turing-recognizable (Lemma D), based on $M_U^T$ and the following total Turing machines:

- $M_{\text{OK}}$ tests whether the input binary string is a valid encoding of a Turing machine
- $M_{\text{DUP}}$ duplicates the input string $c$ to the tape: $cc$
Combining the machines as shown in the next picture, we get the Turing machine $M_D$, which is total whenever $M_{UT}$ is. Moreover,

$$c \in L(M_D)$$

$$\iff c \notin L(M_{OK}) \lor cc \notin L(M_{UT})$$

$$\iff c \notin L(M_c)$$

$$\iff c \in D = \{ c \mid c \notin L(M_c) \}$$

By Lemma D the language $D$ is not decidable. Hence, we have a contradiction and the assumption must be false. Therefore, there cannot exist a total recognizer $M_{UT}$ for the language $U$. \qed
TM $M_D$ recognizing the diagonal language
Corollary G  $\tilde{U} = \{ \langle M, w \rangle \mid w \notin L(M) \}$ is not Turing-recognizable.

Proof. $\tilde{U} = \tilde{U} \cup \text{ERR}$, where \text{ERR} is the easy to decide language:

$$\text{ERR} = \{ x \in \{0, 1\}^* \mid x \text{ does not have a prefix that is a valid code for a Turing machine} \}.$$ 

Counter-assumption: $\tilde{U}$ is Turing-recognizable

- Then by Theorem B, $\tilde{U} \cup \text{ERR} = \tilde{U}$ is Turing-recognizable.
- $U$ is known to be Turing-recognizable (Th. E) and now also $\tilde{U}$ is Turing-recognizable. Hence, by Theorem 4.22, $U$ is decidable.

This is a contradiction with Theorem F and the counter-assumption does not hold. I.e., $\tilde{U}$ is not Turing-recognizable. $\square$
The Halting Problem is Undecidable

- Analogously to the acceptance problem of DFAs, we can pose the halting problem of Turing machines:
  
  *Does the given Turing machine $M$ halt on input $w$?*

- This is an undecidable problem. If it could be decided, we could easily decide also the universal language

**Theorem 5.1** $HALT_{TM} = \{ \langle M, w \rangle \mid M$ is a TM and halts on input $w \}$ is Turing-recognizable, but not decidable.

**Proof.** $HALT_{TM}$ is Turing-recognizable: The universal Turing machine $M_U$ of Theorem E is easy to convert into a TM that simulates the computation of $M$ on input $w$ and accepts if and only if the computation being simulated halts.
HALT$_{TM}$ is not decidable: counter-assumption: HALT$_{TM} = L(M_{HALT})$
for the total Turing machine $M_{HALT}$.

Now we can compose a decider for the language $U$ by combining
machines $M_U$ and $M_{HALT}$ as shown in the next figure.

The existence of such a TM is a contradiction with Theorem F. Hence, the counter-assumption cannot hold and HALT$_{TM}$ is not
decidable.

**Corollary H** $\hat{H} = \{ \langle M, w \rangle \mid M \text{ is a TM and does not halt on input } w \}$
is not Turing-recognizable.

**Proof.** Like in corollary G.
A total TM for the universal language $U$
Chomsky hierarchy

- A formal language $L$ can be recognized with a Turing machine if and only if it can be generated by an unrestricted grammar.

- Hence, the languages generated by unrestricted grammars are Turing-recognizable languages.

- They constitute type 0 languages of Chomsky hierarchy.

- Chomsky’s type 1 languages are the context-sensitive ones. It can be shown that they are all decidable.

- On the other hand, there exists decidable languages, which cannot be generated by context-sensitive grammars.
Finite lang.

3: Regular languages

2: Context-free languages

e.g. \{a^k b^k | k \geq 0 \}

1: Context-sensitive languages

Decidable languages

0: Turing-recognizable languages

e.g. \{ a^k b^k c^k | k \geq 0 \}

\(U, \text{HALT}_{TM}, ...\)

\(A_{DFA}, E_{DFA}, ...\)

\(D, \hat{U}, \hat{A}, ...\)
Halting Problem in Programming Language

The correspondence between Turing machines and programming languages:

All TMs ~ programming language
One TM ~ program
The code of a TM ~ representation of a program in machine code
Universal TM ~ interpreter for machine language

The interpretation of the undecidability of the halting problem in programming languages:

``There does not exist a Java method, which could decide whether any given Java method $M$ halts on input $w$".```
Let us assume that there exists a total Java method \( h \) that returns \texttt{true} if the method represented by string \( m \) halts on input \( w \) and \texttt{false} otherwise:

\[
\text{boolean } h(String \ m, \ String \ w)
\]

Now we can program the method \( hHat \)

\[
\text{boolean } hHat(\ String \ m )
\]
\[
\{ \text{ if } (h(m,m)) \}
\]
\[
\quad \text{while } (\text{true}) ;\}
\]

Let \( H \) be the string representation of \( hHat \). \( hHat \) works as follows:

\[
hHat(H) \text{ halts } \Leftrightarrow h(H,H) = \texttt{false} \Leftrightarrow hHat(H) \text{ does not halt}
\]
5. Reducibility

- The proof of unsolvability of the halting problem is an example of a reducibility:
  - a way of converting problem A to problem B in such a way that a solution to problem B can be used to solve problem A
  - If the halting problem were decidable, then the universal language would also be decidable
  - Reducibility says nothing about solving either of the problems alone; they just have this connection
    - We know from other sources that the universal language is not decidable
  - When problem A is reducible to problem B, solving A cannot be harder than solving B because a solution to B gives one to A
    - If an unsolvable problem is reducible to another problem, the latter also must be unsolvable
Non-emptiness Testing for TMs

(Observe that the book deals with $E_{TM}$.)

The following decision problem is undecidable:
``Does the given Turing machine accept any inputs?''
$NE_{TM} = \{ \langle M \rangle | M \text{ is a Turing machine and } L(M) \neq \emptyset \}$

**Theorem (5.2)** $NE_{TM}$ is Turing-recognizable, but not decidable

**Proof.** The fact that $NE_{TM}$ is Turing-recognizable will be shown in the exercises.

- Let us assume that $NE_{TM}$ has a decider $M^r_{NE}$
- Using it we can construct a total Turing machine for the language $U$
- Let $M$ be an arbitrary Turing machine, whose operation on input $w$ is under scrutiny
• Let $M^w$ be a Turing machine that replaces its actual input with the string $w = a_1 a_2 \ldots a_k$ and then works as $M$.
• Operation of $M^w$ does not depend in any way about the actual input.
  • The TM either accepts or rejects all inputs:

$$L(M^w) = \begin{cases} \{0,1\}^*, & \text{if } w \in L(M) \\ \emptyset, & \text{if } w \notin L(M) \end{cases}$$
The Turing machine $M^w$
Let $M_{\text{ENC}}$ be a TM, which
- Inputs the concatenation of the code $\langle M \rangle$ for a Turing machine $M$ and a binary string $w$, $\langle M, w \rangle$, and
- Leaves to the tape the code $\langle M^w \rangle$ of the TM $M^w$

By combining $M_{\text{ENC}}$ and the decider $M_{\text{NE}}^T$ for the language $\text{NE}_{\text{TM}}$ we are now able to construct the following Turing machine $M_{U^T}$
A decider $M_U^T$ for the universal language $U$
• $M^T_U$ is total whenever $M^T_{NE}$ is, and $L(M^T_U) = U$ because

\[
\langle M, w \rangle \in L(M^T_U)
\ \iff \ \langle M^w \rangle \in L(M^T_{NE}) = NE_{TM}
\ \iff \ L(M^w) \neq \emptyset
\ \iff \ w \in L(M)
\ \iff \ \langle M, w \rangle \in U
\]

• However, by Theorem F $U$ is not decidable, and the existence of the TM $M^T_U$ is a contradiction

• Hence, the language $NE_{TM}$ cannot have a total recognizer $M^T_{NE}$ and we have, thus, proved that the language $NE_{TM}$ is not decidable. \qed
TMs Recognizing Regular Languages

- Similarly, we can show that recognizing those Turing machines that accept a regular language is undecidable by reducing the decidability of the universal language into this problem.

The decision problem is:

"Does the given Turing machine accept a regular language?"

\[ \text{REG}_{TM} = \{ \langle M \rangle \mid M \text{ is a TM and } L(M) \text{ is a regular language} \} \]

**Theorem 5.3** \(\text{REG}_{TM}\) is undecidable.

**Proof.**
- Let us assume that \(\text{REG}_{TM}\) has a decider \(M^{T_{REG}}\).
Using $M_{\text{REG}}^T$ we could construct a decider for the universal language $U$

Let $M$ be an arbitrary Turing machine, whose operation on input $w$ we are interested in

The language corresponding to balanced pairs of parenthesis
\[ \{ 0^n 1^n \mid n \geq 0 \} \]

is not regular, but easy to decide using a TM

Let $M_{\text{parenth}}$ be a decider for the language

Now, let $M_{\text{ENC}}$ be a TM, which on input $\langle M, w \rangle$ composes an encoding for a TM $M^w$, which on input $x$

- First works as $M_{\text{parenth}}$ on input $x$.
- If $M_{\text{parenth}}$ rejects $x$, $M^w$ operates as $M$ on input $w$.
- Otherwise $M^w$ accepts $x$
Deciding a regular language: the TM $M^w$
Thus, $M^w$ either accepts the regular language $\{0, 1\}^*$ or non-regular $\{0^n1^n \mid n \geq 0\}$

Accepting/rejecting the string $w$ on $M$ reduces to the question of the regularity of the language of the TM $M^w$

$$L(M^w) = \begin{cases} 
\{0,1\}^* & \text{if } w \in L(M) \\
\{0^n1^n \mid n \geq 0\} & \text{if } w \notin L(M) 
\end{cases}$$

Let $M_{\text{ENC}}$ be a TM, which
- inputs the concatenation of the code $\langle M \rangle$ for a Turing machine $M$ and a binary string $w$, $\langle M, w \rangle$, and
- Leaves to the tape the code $\langle M^w \rangle$ of the TM $M^w$

Now by combining $M_{\text{ENC}}$ and $M^{T_{\text{REG}}}$ would yield the following Turing machine $M^{U_T}$
A decider $M^T_U$ for the universal language $U$
• $M_U^T$ is total whenever $M_{\text{REG}}^T$ is and $L(M_U^T) = U$, because

$$\langle M, w \rangle \in L(M_U^T)$$

$\iff \langle M^w \rangle \in L(M_{\text{REG}}^T) = \text{REG}_{\text{TM}}$

$\iff L(M^w)$ is a regular language

$\iff w \in L(M)$

$\iff \langle M, w \rangle \in U$

• By Theorem F, $U$ is not decidable, and the existence of the TM $M_U^T$ is a contradiction

• Hence, language $\text{REG}_{\text{TM}}$ cannot have a decider $M_{\text{REG}}^T$

• Thus, we have shown that the language $\text{REG}_{\text{TM}}$ is not decidable
Rice’s Theorem

- Any property that only depends on the language recognized by a TM, not on its syntactic details, is called a *semantic property* of the Turing machine.

  E.g.
  - "$M$ accept the empty string",
  - "$M$ accepts some string" (NE),
  - "$M$ accept infinitely many strings",
  - “The language of $M$ is regular” (REG) etc.

- If two Turing machines $M_1$ and $M_2$ have $L(M_1) = L(M_2)$, then they have exactly the same semantic properties.
• More abstractly: a semantic property $S$ is any collection of Turing-recognizable languages over the alphabet $\{0, 1\}$

• Turing machine $M$ has property $S$ if $L(M) \in S$.
• *Trivial* properties are $S = \emptyset$ and $S = TR$.
• Property $S$ is *solvable*, if language $codes(S) = \{ \langle M \rangle \mid L(M) \in S \}$ is decidable.

**Rice’s theorem** *All non-trivial semantic properties of Turing machines are unsolvable*
Computation Histories

- The computation history for a Turing machine on an input is simply the sequence of configurations that the machine goes through as it processes the input.
- An accepting computation history for $M$ on $w$ is a sequence of configurations $C_1, C_2, ..., C_l$, where
  - $C_1$ is the start configuration $q_0 w$,
  - $C_l$ an accepting configuration of $M$, and
  - each $C_i$ legally follows from $C_{i-1}$ according to the rules of $M$

- Similarly one can define a rejecting computation history
- Computation histories are finite sequences — if $M$ doesn’t halt on $w$, no accepting or rejecting computation history exists for $M$ on $w$
Linear Bounded Automata

- A linear bounded automaton (LBA) is a Turing machine that cannot use extra working space.
- It can only use the space taken up by the input.
- Because the tape alphabet can, in any case, be larger than the input alphabet, it allows the available memory to be increased up to a constant factor.
- Deciders for problems concerning context-free languages.
- If a LBA has
  - $q$ states,
  - $g$ symbols in its tape alphabet, and
  - an input of length $n$,
then the number of its possible configurations is $q \cdot n \cdot g^n$. 
Theorem 5.9

The acceptance problem for linear bounded automata

\[ A_{\text{LBA}} = \{ \langle M, w \rangle \mid M \text{ is an LBA that accepts string } w \} \]

is decidable.

Proof. As \( M \) computes on \( w \), it goes from configuration to configuration. If it ever repeats a configuration, it will go on to repeat this configuration over and over again and thus be in a loop.

Because an LBA has only \( q \cdot n \cdot g^n \) distinct configurations, if the computation of \( M \) has not halted in so many steps, it must be in a loop.

Thus, to decide \( A_{\text{LBA}} \) it is enough to simulate \( M \) on \( w \) for \( q \cdot n \cdot g^n \) steps or until it halts.
Theorem 5.10

The emptiness problem for linear bounded automata

\[ E_{\text{LBA}} = \{ \langle M \rangle \mid M \text{ is an LBA and } L(M) = \emptyset \} \]

is undecidable.

**Proof.** Reduction from the universal language (acceptance problem for general TMs).

Counter-assumption: \( E_{\text{LBA}} \) is decidable; i.e., there exists a decider \( M^T_E \) for \( E_{\text{LBA}} \).

Let \( M \) be an arbitrary Turing machine, whose operation on input \( w \) is under scrutiny. Let us compose an LBA \( B \) that recognizes all accepting computation histories for \( M \) on \( w \).
Now we can reduce the acceptance problem for general Turing machines to the emptiness testing for LBAs:

\[
\begin{align*}
L(B) \neq \emptyset & \quad \text{if } w \in L(M) \\
L(B) = \emptyset & \quad \text{if } w \notin L(M)
\end{align*}
\]

The LBA $B$ must accept input string $x$ if it is an accepting computation history for $M$ on $w$.

Let the input be presented as $x = C_1#C_2#\cdots#C_l$. 
B checks that \( x \) satisfies the conditions of an accepting computation history:

- \( C_1 = q_0 \, w \),
- \( C_l \) is an accepting configuration for \( M \); i.e. \texttt{accept} is the state in \( C_l \), and
- \( C_{i-1} \Rightarrow_M C_i \):
  - configurations \( C_{i-1} \) and \( C_i \) are identical except for the position under and adjacent to the head in \( C_{i-1} \), and
  - the changes correspond to the transition function of \( M \).

Given \( M \) and \( w \) it is possible to construct LBA \( B \) mechanically.
By combining machines $B$ and $M^T_E$ as shown in the following figure, we obtain a decider for the acceptance problem of general Turing machines (universal language).

$$\langle M, w \rangle \in L(M_U^T)$$
$$\iff \langle B \rangle \notin L(M^T_E)$$
$$\iff L(B) \neq \emptyset$$
$$\iff w \in L(M)$$
$$\iff \langle M, w \rangle \in U$$

This is a contradiction, and the language $E_{LBA}$ cannot be decidable.

$\square$
A decider $M_U^T$ for the universal language $U$
5.3 Mapping Reducibility

- Let $M = (Q, \Sigma, \Gamma, \delta, q_0, \text{accept, reject})$ be an arbitrary standard Turing machine.
- Let us define the partial function $f_M : \Sigma^* \to \Gamma^*$ computed by the TM as follows:

$$f_M(w) = \begin{cases} u, & \text{if } q_0 \xrightarrow{w}^* u q, \\ q \in \{\text{accept, reject}\} & \text{otherwise} \\ \text{undefined} & \text{otherwise} \end{cases}$$

- Thus, the TM $M$ maps a string $w \in \Sigma^*$ to the string $u$, which is the contents of the tape, if the computation halts on $w$.
- If it does not halt, the value of the function is not defined in $w$. 
Definition 5.20

- Partial function $f$ is **computable**, if it can be computed with a total Turing machine. I.e. if its value $f(w)$ is defined for every $w$

- Let us formulate the idea that problem $A$ is "at most as difficult as" problem $B$ as follows

  - Let $A \subseteq \Sigma^*$, $B \subseteq \Gamma^*$ be two formal languages
  - $A$ is **mapping reducible** to $B$, written $A \leq_m B$, if there is a computable function $f: \Sigma^* \rightarrow \Gamma^*$ s.t.
    $$w \in A \Leftrightarrow f(w) \in B \quad \forall w \in \Sigma^*$$
  - The function $f$ is called the **reduction** of $A$ to $B$
• Mapping an instance \( w \) of \( A \) computably into an instance \( f(w) \) of \( B \) and

"does \( w \) have property \( A \)?" \( \iff \)
"does \( f(w) \) have property \( B \)?"
Lemma J  For all languages $A$, $B$, $C$ the following hold

i. $A \leq_m A$, (reflexive)

ii. if $A \leq_m B$ and $B \leq_m C$, then $A \leq_m C$, (transitive)

iii. if $A \leq_m B$ and $B$ is Turing-recognizable, then so is $A$, and

iv. if $A \leq_m B$ and $B$ is decidable, then so is $A$

Proof.

i. Let us choose $f(x) = x$ as the reduction.

ii. Let $f$ be reduction of $A$ to $B$ and $g$ a reduction of $B$ to $C$. 
   In other words, $f: A \leq_m B$, $g: B \leq_m C$.

   We show that the composite function $h$, $h(x) = g(f(x))$ is a reduction $h: A \leq_m C$. 
1. $h$ is computable: Let $M_f$ and $M_g$ be the total Turing machines computing $f$ and $g$. $M_{REW}$ replaces all symbols to the right of the tape head with $\square$ and moves the tape head to the beginning of the tape. The total machine depicted in the following figure computes function $h$.

2. $h$ is a reduction:

$$x \in A \iff f(x) \in B \iff g(f(x)) = h(x) \in C,$$

hence, $h: A \leq_m C$.

iii. (and iv.) Let $f: A \leq_m B$, $M_B$ the recognizer of $B$ and $M_f$ the TM computing $f$. The TM depicted below recognizes language $A$ and it is total whenever $M_B$ is. $\square$
The TM computing the composite mapping

\[ M_f \xrightarrow{f(x)} M_{\text{REW}} \xrightarrow{f(x)} M_g \xrightarrow{g(f(x)) \downarrow} \]

\[ x \rightarrow f(x) \rightarrow f(x) \rightarrow g(f(x)) \rightarrow h(x) \]
The TM recognizing $A$
• We have already used the following consequence of Lemma J to prove undecidability.

**Corollary 5.23** If $A \preceq_m B$ and $A$ is undecidable, then $B$ is undecidable.

Let us call language $B \subseteq \{0, 1\}^*$ **TR-complete**, if
1. $B$ is Turing-recognizable (TR), and
2. $A \preceq_m B$ for all Turing-recognizable languages $A$

**Theorem K** The universal language $U$ is TR-complete.

**Proof.** We know that $U$ is Turing-recognizable. Let $B$ be any Turing-recognizable language. Furthermore, let $B = L(M_B)$.

Now, $B$ can be reduced to $U$ with the function $f(x) = \langle M_B, x \rangle$, which is clearly computable, and for which it holds

$$x \in B = L(M_B) \iff f(x) = \langle M_B, x \rangle \in U.$$
**Theorem L** Let $A$ be a TR-complete language, $B$ TR, and $A \leq_m B$. Then also $B$ is a TR-complete language.

- All "natural" languages belonging to the difference of TR and decidable languages are TR-complete, but it contains also other languages.
- The class of TR languages is not closed under complementation, thus it has the dual class
  \[ \text{co-TR} = \{ \overline{A} \mid A \in \text{TR} \} \]
- $\text{TR} \cap \text{co-TR} = \text{decidable languages}$ (by Theorem 4.22)
- $B \subseteq \{0, 1\}^*$ is co-TR-complete, if $B \in \text{co-TR}$ and $A \leq_m B$ for all $A \in \text{co-TR}$
- A language $A$ is co-TR-complete, if and only if the language $\overline{A}$ is TR-complete.
- Language $\text{TOT} = \{ \langle M \rangle \mid M \text{ halts on all inputs} \}$ does not belong to either TR or co-TR.
6. Advanced Topics in Computability

- The Church-Turing thesis gives a universally acceptable definition of *algorithm*
- Another fundamental concept in computer science is *information*
- No equally comprehensive definition of information is known
- Several definitions of information are used – depending upon the application
- In the following we present one way of defining information, using computability theory
6.4 A Definition of Information

- Consider the following two binary sequences
  \[ A = 0101010101010101010101010101010101010101 \]
  \[ B = 1110010110100011101000011101010011010111 \]
- Intuitively, \( A \) contains little information: it is merely a repetition of the pattern \( 01 \) twenty times
- In contrast, \( B \) appears to contain more information
- We define the quantity of information contained in an object to be the size of that object’s smallest representation or description
  - a precise and unambiguous characterization of the object so that we may recreate it from the description alone
- Sequence \( A \) contains little information: it has a small description
- Sequence \( B \) apparently contains more information because it seems to have no concise description
• We may always describe an object (e.g., string) by placing a copy of the object directly into the description.
• This type of a description is never shorter than the object itself.
• Neither does it tell us anything about its information quantity.
• A description that is significantly shorter than the object implies that the information contained within it can be compressed into a small volume.
• Therefore, the amount of information cannot be very large.

• *The size of the shortest description determines the amount of information.*
Minimal Length Descriptions

• We describe a binary string $x$ with a Turing machine $M$ and a binary input $w$ to $M$

• The length of the description is the combined length of representing $M$ and $w$: $\langle M, w \rangle = \langle M \rangle w$

Definition 6.23  Let $x$ be a binary string. The minimal description of $x$, denoted $d(x)$, is the shortest string $\langle M, w \rangle$ where TM $M$ on input $w$ halts with $x$ on its tape. If several such strings exist, select the lexicographically first among them.

The Kolmogorov complexity (also Kolmogorov-Chaitin c. and descriptive c.) of $x$ is

$$K(x) = |d(x)|$$
• In other words, \( K(x) \) is the length of the minimal description of \( x \)
• It is intended to capture our intuition for the amount of information in the string \( x \)
• The Kolmogorov complexity of a string is at most a fixed constant more than its length
• The constant is universal, it is not dependent on the string

**Theorem 6.24** \( \exists c \forall x: K(x) \leq |x| + c \)

**Proof**  Consider the following description of the string \( x \).

Let \( M \) be a TM that halts as soon as it is started. This machine computes the identity function – its output is the same as input. A description of \( x \) is simply \( \langle M \rangle x \). Letting \( c \) be the length of \( \langle M \rangle \) completes the proof. \( \square \)
• Theorem 6.24 conforms to our intuition: the information contained by a string cannot be (substantially) more than its length
• Similarly, the information contained by the string $xx$ is not significantly more than the information contained by $x$

**Theorem 6.25** $\exists c \forall x: K(xx) \leq K(x) + c$

**Proof** Consider the following Turing machine, which expects an input of the form $\langle N, w \rangle$, where $N$ is a TM and $w$ is an input for it.

$M = \text{"On input } \langle N, w \rangle \text{, where } N \text{ is a TM and } w \text{ is a string:}
1. \text{Run } N \text{ on } w \text{ until it halts and produces an output string } s.
2. \text{Output the string } ss."

A description of $xx$ is $\langle M \rangle d(x)$. Recall that $d(x)$ is the minimal description of $x$. The length of this description is $|\langle M \rangle| + |d(x)|$, which is $c + K(x)$, where $c$ is the length of $\langle M \rangle$. \qed
By the above results we might expect the Kolmogorov complexity of the concatenation $xy$ of two strings $x$ and $y$ to be at most the sum of their individual complexities (plus a fixed constant).

That bound, however, cannot be reached.

The cost of combining two descriptions leads to a greater bound:

**Theorem 6.26** $\exists c \forall x, y: K(xy) \leq 2K(x) + K(y) + c$

**Proof** We construct a TM $M$ that breaks its input into two separate descriptions. The bits of the first description $d(x)$ are doubled and terminated with string $01$ before the second description $d(y)$ appears.

Once both descriptions have been obtained, they are run to obtain the strings $x$ and $y$ and the output $xy$ is produced.

The length of this description of $xy$ is clearly twice the complexity of $x$ plus the complexity of $y$ plus a fixed constant describing $M$. $\square$
• A more efficient method of indicating the separation between the two descriptions improves this theorem somewhat
• Instead of doubling the bits of \( d(x) \), we may prepend the length of \( d(x) \) as a binary integer that has been doubled to differentiate it from \( d(x) \)
• The description still contains enough information to decode it into the two descriptions of \( x \) and \( y \)
• The description now has length at most
  \[
  2\log_2 K(x) + K(x) + K(y) + c
  \]
Optimality of the Definition

- Kolmogorov complexity $K(x)$ has an optimality property among all possible ways of defining descriptive complexity with algorithms.
- Consider a general description language to be any computable function $p: \Sigma^* \rightarrow \Sigma^*$.
- E.g., $p$ could be the programming language LISP (encoded in binary).
- Define the minimal description of $x$ w.r.t. $p$ – denoted $d_p(x)$ – to be the lexicographically shortest string $s$ where $p(s) = x$.
- Let $K_p(x) = |d_p(x)|$.
- If, for example, $p = \text{LISP}$, then
  - $d_{\text{LISP}}(x)$ is the minimal LISP program that outputs $x$.
  - $K_{\text{LISP}}(x)$ is the length of the minimal program.
Theorem 6.27  For any description language $p$, a fixed constant $c$ exists that depends only on $p$, where

$$
\forall x: K(x) \leq K_p(x) + c
$$

Proof: Take any description language $p$ and consider the following Turing machine:

$M = \text{"On input } w: \text{ Output } p(w)."$

Then $\langle M \rangle d_p(x)$ is a description of $x$ whose length is at most the fixed constant $|\langle M \rangle|$ greater than $d_p(x)$. \hfill \square

Let $p =$ LISP. Suppose that $x$ has a short description $w$ in LISP. Let $M$ be a TM that can interpret LISP and use the LISP program $w$ for $x$ as input to $M$.

Then $\langle M, w \rangle$ is a description of $x$ that is only a fixed amount $|M|$ larger than the LISP description of $x$. 
Incompressible Strings and Randomness

- For some strings the minimal description may be much shorter than the string itself (if the information in the string appears sparsely or redundantly)
- Do some strings lack short descriptions?
  - I.e., is the minimal description of some strings actually as long as the string itself?
- Such strings exist: They cannot be described more concisely than simply writing them out explicitly

**Definition 6.28** Say that string $x$ is $c$-compressible if

$$K(x) \leq |x| - c.$$  

If $x$ is not $c$-compressible, we say that $x$ is incompressible by $c$.

If $x$ is incompressible by 1, we say that $x$ is incompressible.
Theorem 6.29 Incompressible strings of every length exist.

**Proof:** The number of distinct binary strings of length $n$ is $2^n$. Each description is a binary string, so the number of descriptions of length $< n$ is at most the sum of the number of strings of each length up to $n-1$:

$$\sum_{i=0}^{n-1} 2^i = 1 + 2 + 4 + \cdots + 2^{n-1} = 2^n - 1$$

The number of short descriptions is less than the number of strings of length $n$. Therefore at least one string of length $n$ is incompressible. □

**Corollary 6.30** At least $2^n - 2^{n-c+1} + 1$ strings of length $n$ are incompressible by $c$. 
Incompressible strings have many properties we would expect to find in randomly chosen strings:

- For example, any incompressible string of length $n$ has roughly an equal number of 0s and 1s.
- The length of its longest run of 0s is approx. $\log_2 n$.
- Any computable property that holds for “almost all” strings also holds for all sufficiently long incompressible strings:
  - A property of strings is simply a function that maps strings to the truth values.
  - A property holds for almost all strings if the fraction of strings of length $n$ on which it is false approaches 0 as $n$ grows large.
- Also a randomly chosen long string is likely to satisfy a computable property that holds for almost all strings.
Theorem 6.31 Let $f$ be a computable property that holds for almost all strings. Then for any $b > 0$, the property $f$ is False on only finitely many strings that are incompressible by $b$.

Proof: Let us consider the following algorithm:

$$M = "\text{On input } i, \text{ a binary integer:}
1. \text{Find the } i^{\text{th}} \text{ string } s \text{ where } f(s) = \text{False, considering the strings ordered lexicographically.}
2. \text{Output string } s."$$

• We can use $M$ to obtain short descriptions of strings that fail to have property $f$ as follows.
• For any such string $x$, let $i_x$ be the index of $x$ on a list of all strings that fail to have property $f$, ordered lexicographically.
• Then $\langle M, i_x \rangle$ is a description of $x$ with length $|i_x| + c$.
• Because only few strings fail to have property $f$, the index of $x$ is small and its description is correspondingly short.
• Fix any number $b > 0$. Select $n$ such that at most a $1/2^{b+c+1}$ fraction of strings of length $n$ or less fail to have property $f$.

• All sufficiently large $n$ satisfy this condition because $f$ holds for almost all strings.

• Let $x$ be a string of length $n$ that fails to have property $f$.

• We have $2^{n+1} - 1$ strings of length $n$ or less, so

$$i_x \leq \frac{2^{n+1} - 1}{2^{b+c+1}} \leq 2^{n-b-c}.$$ 

• Therefore, $|i_x| \leq n-b-c$, so the length of $\langle M, i_x \rangle$ is at most $(n-b-c) + c = n-b$, which implies that $K(x) \leq n-b$.

• Thus, every sufficiently long $x$ that fails to have property $f$ is compressible by $b$.

• Hence, only finitely many strings that fail to have property $f$ are incompressible by $b$. □
Examples of incompressible strings?

- Unfortunately, Kolmogorov complexity is not computable (exercises next week)
- No algorithm can decide in general whether strings are incompressible (exercises next week)
- Indeed, no infinite subset of them is Turing-recognizable
- So we have no way of obtaining long incompressible strings
- In any case, we would have no way to determine whether a string is incompressible even if we could somehow obtain such

- The following theorem describes certain strings that are nearly incompressible, but doesn't provide a way to exhibit them explicitly
**Theorem 6.32** For some constant $b$, for every string $x$, the minimal description $d(x)$ of $x$ is incompressible by $b$.

**Proof**: Consider the following TM:

$M = "On$ input $\langle R, y \rangle$, where $R$ is a TM and $y$ is a string:
1. Run $R$ on $y$ and reject if its output isn’t of the form $\langle S, z \rangle$
2. Run $S$ on $z$ and halt with its output on the tape."

Let $b$ be $|\langle M \rangle| + 1$. We show that $b$ satisfies the claim. Suppose to the contrary that $d(x)$ is $b$-compressible for some string $x$. Then

$$K(d(x)) = |d(d(x))| \leq |d(x)| - b.$$  

But then $\langle M \rangle d(d(x))$ is a description of $x$ whose length is at most

$$|\langle M \rangle| + |d(d(x))| \leq (b - 1) + (|d(x)| - b) = |d(x)| - 1.$$  

This description of $x$ is shorter than $d(x)$, contradicting the latter’s minimality.  

$\square$
7. Time Complexity

- Before we have not paid any attention to the time required to solve a problem
- Now we turn to consider solvable problems and their time complexity
- A problem that in principle is solvable can in practice require so much time that it effectively is unsolvable

- In TSP (traveling salesperson problem) the problem is to find the shortest *Hamilton cycle* from a weighted graph
- Trivial algorithm: in a graph of $n$ nodes, go through all possible $n!$ routes and choose the shortest one
In a computer in which examining one route takes 0.001 s, the trivial algorithm would require more time than the current age of the universe for a graph of \( n = 22 \) nodes.

If we had a billion times more efficient computer then we still couldn’t solve the TSP for a graph of \( n = 31 \) nodes within the current age of the universe.

Hence, it is not reasonable to call the exponential \((n! \approx O(n^n))\) algorithm that goes through all the potential routes as a solution for the TSP.

One does not know an algorithm whose time consumption would be bounded by a polynomial of \( n \); on the other hand, one is not able to show that such an algorithm does not exist.
7.1 Measuring Complexity

- The length of the computation of a standard Turing machine
  \( M = (Q, \Sigma, \Gamma, \delta, q_0, \text{accept, reject}) \)
  
  \[ q_0 \, w \Rightarrow_M \, q \, a \, v, \quad q \in \{ \text{accept, reject} \} \]

  is the number of steps included

- Time complexity on input \( w \):

  \[
  \text{time}_M = \begin{cases} 
  \text{length of } q_0 \, w \xrightarrow{*} M \cdots, & \text{if the computation halts} \\
  \infty, & \text{if computation doesn’t halt on } w 
  \end{cases}
  \]

- The running time or time complexity of \( M \) is the function

  \[
  \text{time}_M : \Sigma^* \to \mathbb{N} \cup \{ \infty \}
  \]
• One usually examines time complexity as a function of the length $n$ of the input:
  
  • *In the average case* when inputs of length $n$ are drawn from the probability distribution $P_n(w)$

  $$time_M^{\text{avg}}(n) = \sum_{|w|=n} P_n(w) \cdot time_M(w)$$

  • Or more commonly *in the worst case*

  $$time_M^{\text{max}}(n) = \max_{|w|=n} time_M(w)$$

• Usually the notation is also simplified

  $$time_M : \mathbb{N} \to \mathbb{N} \cup \{\infty\},$$

  $$time_M(n) = time_M^{\text{max}}(n)$$

• It would be interesting to analyze the average case, but it is typically so difficult that one has to be content with dealing with the worst-case analysis
Asymptotic Analysis

- Even the worst-case running times are still too complex for exact examination.
- Therefore, it is usual to just estimate their growth rates by asymptotic notation.

- Let $f, g : \mathbb{N} \rightarrow \mathbb{R}^+$ be arbitrary functions.
  - $f = O(g)$: $g$ is an (asymptotic) upper bound for $f$, if $\exists c, n_0 \in \mathbb{Z}^+$:
    \[ f(n) \leq c \cdot g(n), \quad \forall n \geq n_0 \]
  - $f = \Theta(g)$: $g$ is an asymptotically tight bound for $f$, if $f = O(g)$ and $g = O(f)$.
• $f = o(g)$: $g$ is an asymptotically tight upper bound for $f$,
  if $\forall c > 0: \exists n_c \in \mathbb{N}$:
  \[ f(n) < c \cdot g(n), \ \forall n \geq n_c \]

• $f = \Omega(g)$: $g$ is an asymptotic lower bound for $f$,
  if $\exists c > 0$: for infinitely many $n \in \mathbb{N}$:
  \[ f(n) \geq c \cdot g(n) \]

• $f = o(g) \iff f = O(g) \land g \neq O(f)$
  $f \neq \Omega(g) \iff f = o(g)$

• One usually talks more vaguely and says, e.g.,
  • "function $n!$ has growth rate $O(n^n)$" or
  • "$2n^2$ has growth rate $n^2$"
Lemma M

1. \( \log_a n = \Theta(\log_b n) \quad \forall a, b > 0, \)
2. \( n^a = o(n^b), \) if \( a < b, \)
3. \( 2^{an} = o(2^{bn}), \) if \( a < b, \)
4. \( \log_a n = o(n^b) \quad \forall a, b > 0, \)
5. \( n^a = o(2^{bn}) \quad \forall a, b > 0, \)
6. \( c \cdot f(n) = \Theta(f(n)) \quad \forall c > 0 \) and functions \( f, \)
7. \( f(n) + g(n) = \Theta(\max \{ f(n), g(n) \}) \) for all functions \( f \) and \( g, \) and
8. if \( p(n) \) is a polynomial of degree \( r, \) then \( p(n) = \Theta(n^r) \)
Lemma N  Let $f, g : \mathbb{N} \to \mathbb{R}^+$ be arbitrary functions. If the limit

$$L = \lim_{n \to \infty} \frac{f(n)}{g(n)}$$

exists and

1. $0 < L < \infty \Rightarrow f = \Theta(g)$
2. $L = 0 \Rightarrow f = o(g)$
3. $L > \infty \Rightarrow g = o(f)$
4. $L < \infty \Rightarrow f = O(g)$
5. $L > 0 \Rightarrow f = \Omega(g)$  \hfill \Box
Let us examine the recognition of the familiar, non-regular language \( A = \{ 0^k1^k \mid k \geq 0 \} \).

How much time does a single-tape Turing machine require in deciding \( A \)?

On input string \( w \):

1. Scan across the tape and reject if a 0 is found to the right of a 1.
2. Repeat if both 0s and 1s remain on the tape:
   3. Scan across the tape, crossing off a single 0 and a single 1.
4. If 0s still remain after all the 1s have been crossed off, or if 1s still remain after all the 0s have been crossed off, reject. Otherwise, if neither 0s nor 1s remain on the tape, accept.
In stage 1 of the recognizer for $A$ uses $n$ steps on input $w$, $|w| = n$

- Repositioning the head at the left-hand end of the tape uses another $n$ steps
- In total stage 1 uses $2n = O(n)$ steps
- In stages 2 and 3 crossing off a pair of 0s and 1s requires $O(n)$ steps and at most $n/2$ such scans can occur
- Altogether, stages 2 and 3 take $O(n^2)$ steps
- Stage 4 again takes a linear number of steps

- All in all the Turing machine requires time $O(n) + O(n^2) + O(n) = O(n^2)$
Recognizing $A$, however, is not an $\Omega(n^2)$ task; the following Turing machine only takes $O(n\log n)$ time.

On input string $w$:

1. Scan across the tape and reject if a 0 is found to the right of a 1.
2. Repeat as long as some 0s and 1s remain on the tape:
   a) If the total number of 0s and 1s remaining is odd, reject.
   b) Cross off every other 0 starting with the first 0. Do the same with the 1s.
3. If no 0s and no 1s remain on the tape, accept. Otherwise, reject.
On every scan performed in stage 2, the total number of 0s remaining is cut in half and any remainder is discarded.

Every stage takes $O(n)$ time and there are at most $1 + \log_2 n$ iterations of stage 2.

The total time is $O(n \cdot \log n)$.

Examples of the correctness of the TM: if there initially were, e.g., seven 0s and six 1s, then the parity test immediately rejects.

If seven 0s and five 1s, then after the first halving of 0s and 1s there remains three 0s and two 1s and the parity test rejects.

If seven 0s and three 1s, then it takes two rounds of halving before the parity test gets to reject.
A single-tape Turing machine cannot decide the language $A$ in less asymptotic time than $O(n \cdot \log n)$.

Any language that can be decided in $o(n \cdot \log n)$ time on a single-tape TM is regular.

If, on the other hand, we have a second tape, we can decide the language in linear time.

On input string $w$:
1. If a 0 is found to the right of a 1, then reject.
2. Copy the 0s onto tape 2.
3. Scan across the 1s, for each cross off a 0 on tape 2. If all 0s are crossed off before all the 1s are read, reject.
4. If all 0s have now been crossed off, accept. If any 0s remain, reject.
• Universal models of computation are thus not equally efficient
• In complexity theory it does matter which is the model being used, while in computability theory they are all equivalent

• Time requirements for deterministic models do not differ greatly
• Let \( t: \mathbb{N} \rightarrow \mathbb{R}^+ \) be an arbitrary function
• A language \( A \) can be decided in time \( t \), if there exists a deterministic Turing machine \( M \) s.t.
  • \( L(M) = A \) and
  • \( \text{time}_M(n) \leq t(n) \) for all \( n \)
• The time complexity class for formal languages
  \[ \text{DTIME}(t) = \{ A \mid A \text{ can be decided in time } t \} \]
• Every multitape Turing machine has an equivalent single-tape Turing machine (Theorem 3.13)

• Simulating each step of the \( k \)-tape machine uses at most \( O(t(n)) \) steps on the single-tape machine, where \( t(n) \geq n \) is the time complexity of the multitape machine

• There are in total \( O(t(n)) \) steps, and hence

\[ \text{Theorem 7.8} \quad \text{Every} \; t(n) \; \text{time multitape Turing machine has an equivalent} \; O(t^2(n)) \; \text{time single-tape Turing machine.} \]

• The running time of a nondeterministic Turing machine \( N \):

\[ \text{time}_N(w) \text{ is the length of the longest computation} \; (q_0 \; w \; \Rightarrow_N \; ...) \]
• The computations of a nondeterministic decider on input $w$ may be thought as a computation tree.

• In the leaves of the tree the computation halts. The TM accepts the input $w$, if some leaf corresponds to the accept state.

• The worst-case time complexity:

  $$\text{time}_N(n) = \max_{|w|=n} \text{time}_N(w)$$

• Language $A$ can be decided nondeterministically in time $t$, if there exists a nondeterministic decider $N$ s.t.
  
  • $L(N) = A$ and
  
  • $\text{time}_N(n) \leq t(n)$ $\forall n$

• Nondeterministic time complexity classes (Definition 7.21): $\text{NTIME}(t) = \{ A \mid A$ can be decided nondeterministically in time $t \}$
Theorem 7.11 Every $t(n)$ time nondeterministic single-tape Turing machine has an equivalent $2^{O(t(n))}$ time deterministic single-tape Turing machine.

Proof. The deterministic Turing machine $M$ of the proof of Theorem 3.16 systematically searches the computation tree of the nondeterministic machine $N$.

Let the running time of $N$ be $t(n)$. Then, on input of length $n$ every branch of $N$’s nondeterministic computation tree has length at most $t(n)$.

The branching factor of the tree is determined by the transition possibilities in the transition function of $N$. Let $b \geq 2$ its upper bound.

The number of leaves in the tree is at most $b^{t(n)}$. 
In the worst case we have to examine all the nodes in the tree. The total number of nodes in the tree is at most less than twice that of the leaves; i.e., of the order $O(b^n)$. Hence, the time required by the three-tape TM $M$ is asymptotically $2^{O(t(n))}$.

By Theorem 7.8 converting to a single-tape TM at most squares the running time. Thus, the running time of the single-tape simulator is $(2^{O(t(n))})^2 = 2^{O(2t(n))} = 2^{O(t(n))}$.

- The efficiency difference between a single-tape and a multitape TM is at most the square of $t$; i.e., polynomial in $t$
- On the other hand, the efficiency difference between a deterministic and a nondeterministic TM may be exponential in $t$
Thus, for any nondeterministic Turing machine $M$ that runs in some polynomial time $p(n)$, we can devise an algorithm that takes an input $w$ of length $n$ and produces $E_{p,w}$. The running time is $O(p(n))$ on a multitape deterministic Turing machine and...

WTF, man. I just wanted to learn how to program video games.
7.2 The Class P

- For our purposes, polynomial differences in running time are considered to be small, whereas exponential differences are considered to be large.
- In time complexity polynomial time requirement is thought as useful, whereas exponential time requirement most often is useless.
- Exponential time algorithms typically arise in exhaustive searching (brute-force search).

- All reasonable models of computation are polynomially equivalent.
- Any one of them can simulate another with only a polynomial increase in running time.
Definition 7.12 \( P = \bigcup \{ \text{DTIME}(t) \mid t \text{ is a polynomial} \} \)
\[ = \bigcup_{k \geq 0} \text{DTIME}(n^k) \]

- The class \( P \) plays a central role and is important because
  - It is invariant for all models of computation that are polynomially equivalent to the deterministic single-tape Turing machine, and
  - It roughly corresponds to the class of problems that are realistically solvable on a computer
- Algorithms with a high-degree polynomial time complexity of course are not that practical, but they tend to be quite rare
- The role of \( P \) in complexity theory is similar to the role of decidable languages in computability theory
Examples of Problems in P

- Let $G$ be a directed graph containing nodes $s$ and $t$
- Problem PATH: Is there a directed path in $G$ from $s$ to $t$?

- Brute-force algorithm would examine all potential paths in $G$ to determine whether any is the required path
- In the worst case the number of potential paths is exponential in the number of nodes of the graph
- Thus we need a more sophisticated algorithm:
  1. Place a mark on node $s$
  2. As long as the set of marked nodes keeps growing:
     - Scan all the edges of $G$. If an edge $(a, b)$ goes from a marked node $a$ to an unmarked node $b$, mark node $b$
  3. If $t$ is marked, accept. Otherwise, reject.
Let $m$ be the number of nodes in graph $G$

- Stages 1 and 3 are executed only once
- Stage 2 runs at most $m$ times, because each time except the last it marks an additional node in $G$
- Thus, the total number of stages used is $m + 2$
- Each stage is easy to implement in polynomial time

**Theorem 7.14** PATH $\in$ P.
Say that two numbers are \textit{relatively prime} if 1 is the largest integer that evenly divides them both. For example, 10 and 21 are relatively prime, even though neither of them is a prime number by itself. On the other hand, 9 and 21 are both divisible by 3. Let \textit{RP} be the problem of testing whether two numbers are relatively prime:

\[ \text{RP} = \{ \langle x, y \rangle \mid x \text{ and } y \text{ are relatively prime} \} \]

Again, brute-force algorithm is inefficient. Searching through all possible divisors of both numbers requires exponential time, because the number of possible divisors is exponential in the length of the binary representation of the given numbers \( x \) and \( y \).
RP is in P

- An ancient numerical procedure, called the Euclidean algorithm, for computing the greatest common divisor (gcd) helps us to solve RP efficiently.

- For example, $\gcd(18, 24) = 6$

- Obviously, $x$ and $y$ are relatively prime iff $\gcd(x, y) = 1$

- Let $x$ and $y$ be natural numbers in binary representation
  1. Repeat until $y = 0$
     a) $x \leftarrow x \mod y$
     b) exchange $x$ and $y$
  2. Output $x$
E.g., $x = 18, y = 24$

- $x \leftarrow 18 \mod 24 = 18$
- $x \leftarrow 24, y \leftarrow 18$
- $x \leftarrow 24 \mod 18 = 6$
- $x \leftarrow 18, y \leftarrow 6$
- $x \leftarrow 18 \mod 6 = 0$
- $x \leftarrow 6, y \leftarrow 0$

Output $x = 6$
• After the stage 1 has been executed for the first time, it definitely holds that $x > y$
• Thereafter, every execution of stage 1(a) cuts the value of $x$ by at least half:
  • $x \geq 2y \Rightarrow x \mod y < y \leq x/2$
  • $x < 2y \Rightarrow x \mod y = x - y < x/2$
• Every other loop through of stage 1 reduces each of the original values $x$ and $y$ by at least half, so in the worst case the number of loops is at most
  $$\min\{ 2\log_2 x, 2\log_2 y \}$$
• The binary representations of the numbers have logarithmic length, and hence the Euclidean algorithm requires a linear time
7.3 The Class NP

- The class $\text{NP}$ [Theorem 7.20, Corollary 7.22]:
  \[
  \text{NP} = \bigcup \{ \text{NTIME}(t) \mid t \text{ is a polynomial} \} = \bigcup_{k \geq 0} \text{NTIME}(n^k)
  \]
- In other words, the set of languages decidable in polynomial time by a nondeterministic Turing machine.

- Obviously, $\text{P} \subseteq \text{NP}$
- When we talk about problems in $\text{NP}$, we usually mean those problems that do not belong to $\text{P}$: $\text{NP} \setminus \text{P}$
  - For example, one does not know how to find a Hamiltonian path in a directed graph in polynomial time, instead one has to resort to the brute-force algorithm.
The following nondeterministic Turing machine decides the directed Hamiltonian path problem in polynomial time

On input $\langle G, s, t \rangle$, where $G$ is a directed graph with nodes $s$ and $t$, the number of nodes in $G$ is $m$:

1. Write a list of $m$ numbers, $p_1, \ldots, p_m$, where each $p_i$ is nondeterministically selected to be between 1 and $m$
2. If there are repetitions in the list $p_1, \ldots, p_m$, reject
3. If $p_1 \neq s$ or $p_m \neq t$, reject
4. For each $i \in \{1, \ldots, m\}$ check whether $(p_i, p_{i+1})$ is an edge of $G$. If any are not, reject. Otherwise, we have verified the nondeterministic selection, so accept
**CLIQUE is in NP**

- A **clique** in an undirected graph is a subgraph, wherein every two nodes are connected by an edge.
- A $k$-clique is a clique that contains $k$ nodes.

$$\text{CLIQUE} = \{ \langle G, k \rangle \mid G \text{ is an undirected graph with a } k\text{-clique} \}$$

- We can solve clique nondeterministically by first selecting (guessing) $k$ nodes of the graph $G$.
- Thereafter we deterministically verify that all selected nodes really are connected by an edge in $G$.
- If any are not, *reject*. Otherwise, *accept*.
SUBSET-SUM is in NP

• Given a multiset of numbers \( S = \{ x_1, \ldots, x_k \} \) and a target number \( t \), does \( S \) contain a subcollection \( \{ y_1, \ldots, y_l \} \) s.t. \( \sum_i y_i = t \)

• E.g., the pair \( \langle \{ 4, 11, 16, 21, 27 \}, 25 \rangle \) belongs to the language of this problem, because \( 4 + 21 = 25 \)

• Exhaustive search: examine all \( 2^k \) subcollections and determine whether the sum of the elements is \( t \)

• Nondeterministically select (guess) a subcollection of numbers from the given multiset \( S \)

• Deterministically verify that the the chosen numbers sum to \( t \)
P =?= NP

- P is the class of languages for which membership can be decided quickly
- NP is the class of languages for which membership can be verified quickly

- It would seem clear that NP is significantly larger class than P, but it has not been proved for a single language in NP \ P that it would not have a polynomial-time decider algorithm

- Hence, in principle it is possible that P = NP

- For problems in NP \ P one is only aware of deterministic algorithms requiring an exponential time

\[ \text{NP} \subseteq \text{EXPTIME} = \bigcup_k \text{DTIME}(2^{nk}) \]
7.4 NP-Completeness

- A function \( f: \Sigma^* \rightarrow \Gamma^* \) is a polynomial time computable if there exists a Turing machine \( M \) and a polynomial \( p \) for which
  - \( f = f_M \) and
  - \( \text{time}_M(n) \leq p(n) \) for all \( n \)

- Let \( A \subseteq \Sigma^*, B \subseteq \Gamma^* \) be two formal languages

**Definition 7.29** Language \( A \) is polynomial time reducible to language \( B \), written

\[
A \leq_{m^p} B,
\]

if a polynomial time computable function \( f: \Sigma^* \rightarrow \Gamma^* \) exists, where for every \( x \in \Sigma^* \),

\[
x \in A \Leftrightarrow f(x) \in B
\]
Theorem 7.31 (Extended)

For all languages $A$, $B$, $C$ it holds

i. $A \leq_{m}^{p} A$, (reflexive)

ii. if $A \leq_{m}^{p} B$ and $B \leq_{m}^{p} C$, then $A \leq_{m}^{p} C$ (transitive),

iii. if $A \leq_{m}^{p} B$ and $B \in \text{NP}$, then $A \in \text{NP}$, and

iv. if $A \leq_{m}^{p} B$ and $B \in \text{P}$, then $A \in \text{P}$.

Note: for the part of mapping reducibility this theorem is exactly the same as Lemma J (Theorem 5.22). The difference is the polynomial time computability of the reduction.
Proof.

i. We choose \( f(x) = x \) to be the reduction.

ii. The composite function \( h(x) = g(f(x)) \) is a reduction from \( A \) to \( C \),
\[ h: A \leq_m C \] (see Lemma J, Theorem 5.22).

\( h \) can be computed in polynomial time:

Let \( M_f \) (\( M_g \)) be the Turing machine computing function \( f \) (\( g \)) in time bounded by polynomial \( p \) (\( q \)).

We can assume that \( p \) and \( q \) are everywhere non-descending.

Let \( M_g \), \( M_{REW} \), and \( M_f \) work as in the proof of Lemma J.
The TM computing the composite mapping

\[
M_f \xrightarrow{f(x)} M_{\text{REW}} \xrightarrow{f(x)} M_g \xrightarrow{g(f(x))} h(x)
\]
By combining the TMs as previously, we get a TM $M_h$ that computes the function $h$, and uses the following time on input $x$:

$$
time_{M_f}(x) + time_{M_{REW}}(f(x)v) + time_{M_g}(f(x))
\leq p(|x|) + 2p(|x|) + q(|f(x)|)
\leq 3p(|x|) + q(p(|x|))
= O(q(p(|x|))),
$$

which is polynomial in the length of $x$.

iii. (and iv.) By combining

- the TM $M_f$, which computes the reduction $f: A \leq_m^p B$ in time bounded by the polynomial $p$,
- the TM $M_B$, which decides the language $B$ in time bounded by $q$, and
- $M_{REW}$

similarly as in the proof of Lemma J, we get the TM $M_A$, which decides the language $A$ in time $O(q(p(|x|)))$. It is deterministic whenever $M_B$ is. $\square$
Satisfiability of Boolean formulas, SAT

Given a Boolean formula $\varphi$, which consists of
- Boolean variables $x_1, \ldots, x_n$,
- Constant values 0 (false) and 1 (true), and
- Boolean operations $\lor$, $\land$, and $\neg$.

Is $\varphi$ satisfiable? Is there an assignment of values 0 and 1 to the variables $t: \{ x_1, \ldots, x_n \} \rightarrow \{ 0, 1 \}$, such that
$$\varphi(t(x_1), \ldots, t(x_n)) = 1$$

Let us guess the assignment $t$ of values for the variables and verify that $\varphi(t) = 1$.

If $\varphi$ contains $n$ Boolean variables, then $t$ can be represented as a binary string of $n$ bits and it can be verified in polynomial time
• Stephen Cook and Leonid Levin discovered in the early 1970s that there exists the class of $NP$-complete problems.

• The individual complexity of an $NP$-complete problem is related to that of the entire class of $NP$.

• If a polynomial-time algorithm exists for any of the $NP$-complete problems, all problems in $NP$ would be polynomial-time solvable.

• $NP$-complete problems help to study the question $P =?= NP$ and to recognize difficult practical problems.

**Theorem 7.27 (Cook-Levin theorem)**

$$SAT \in P \iff P = NP$$
• The satisfiability problem for many special forms of Boolean formulas is also NP-complete.
• A formula \( \varphi \) is in conjunctive normal form (cnf), if it comprises several conjuncts

\[
\varphi = C_1 \land C_2 \land \ldots \land C_m,
\]

where each clause \( C_i \) is a disjunction

\[
C_i = \alpha_{i1} \lor \alpha_{i2} \lor \ldots \lor \alpha_{ir}
\]

• Terms \( \alpha_{ij} \) are literals: Boolean variables or their negations
• CSAT is the satisfiability problem for cnf-formulas:

\[
\{ \varphi \mid \varphi \text{ is a satisfiable cnf-formula} \}
\]
• Obviously, \( \text{CSAT} \in \text{NP} \). An arbitrary Boolean formula can be converted to a cnf-formula in polynomial time.
By restricting the number of terms in a clause of a cnf-formula to be exactly $k$ literals, we get the $k$-conjunctive normal form ($k$-cnf).

A family of languages:

$$k\text{SAT} = \{ \varphi \mid \varphi \text{ is a satisfiable } k\text{-cnf-formula} \}$$

Language 2SAT belongs to $P$.

**Theorem** $CSAT \leq_{m}^{P} 3\text{SAT}$

**Proof.**
- The given cnf-formula $\varphi$ can be converted in polynomial time into an equivalent 3-cnf-formula $\varphi'$
- Let $\varphi = C_1 \land C_2 \land \ldots \land C_m$
• Each clause $C_k = \alpha_1 \lor \alpha_2 \lor \ldots \lor \alpha_r$, $r \geq 3$, is replaced by a 3-cnf-formula

$C_k' = (\alpha_1 \lor \alpha_2 \lor t_1) \land (\neg t_1 \lor \alpha_3 \lor t_2) \land \ldots \land (\neg t_{r-3} \lor \alpha_{r-1} \lor \alpha_r)$,

where $t_1, ..., t_{r-3}$ are new variables. The formula $C_k'$ can clearly be obtained from clause $C_k$ in polynomial time.

We still need to check that the transformation satisfies reducibility $\varphi \in \text{CSAT} \iff \varphi' \in \text{3SAT}$:

1. $\varphi$ satisfiable $\Rightarrow$ $\varphi'$ satisfiable:
   For all clauses $C_k$ the assignment satisfying $\varphi$ must set $\alpha_i = 1$ for some $\alpha_i \in C_k$.
   $C_k'$ gets satisfied when we set the values of literals as in the assignment satisfying $C_k$ and the new variables get values as
follows

\[ t_j = \begin{cases} 
1, & \text{if } j \leq i - 2 \\
0, & \text{if } j > i - 2 
\end{cases} \]

2. \( \varphi \) is satisfiable \( \Leftrightarrow \varphi' \) is satisfiable:
Also the subformulas \( C_k' \) corresponding to the clauses \( C_k \) of \( \varphi \) must be satisfied. Then either
a) Some literal \( \alpha_i = 1, \alpha_i \in C_k \), and \( C_k \) gets satisfied by it, or
b) For some \( i < r-3 \):
\[
t_i = 1 \land t_{i+1} = 0,
\]
and it must be that \( \alpha_{i+2} = 1 \), and again \( C_k \) gets satisfied.
If \( r \leq 3 \), then

\[
C_k = \alpha_1 \lor \alpha_2 \lor \alpha_3 \Rightarrow \\
C_k' = C_k
\]

\[
C_k = \alpha_1 \lor \alpha_2 \Rightarrow \\
C_k' = (\alpha_1 \lor \alpha_2 \lor \top) \land (\alpha_1 \lor \alpha_2 \lor \bot)
\]

\[
C_k = \alpha \Rightarrow \\
C_k' = (\alpha \lor t_1 \lor t_2) \land (\alpha \lor t_1 \lor \neg t_2) \land (\alpha \lor \neg t_1 \lor t_2) \land (\alpha \lor \neg t_1 \lor \neg t_2)
\]

The equivalence of satisfiability of the formulas is maintained. \( \square \)
Vertex Cover, VC

Given an undirected graph $G$ and a natural number $k$.

Does $G$ contain a subset of $k$ nodes that cover every edge of $G$?

- A node covers an edge if the edge touches the node.

To represent VC as a formal language we need to encode graphs as strings. Similar encoding techniques as those used with Turing machines apply.

We can guess the given number $k$ of nodes from the given graph $G$, and then verify in time polynomial in the size of the graph that the chosen $k$ nodes cover all edges of $G$.
Theorem 7.44  \(3\text{SAT} \leq_{m^p} \text{VC}\)

**Proof.** Let \(\varphi = C_1 \land C_2 \land \ldots \land C_m\) be a 3-cnf-formula with variables \(x_1, \ldots, x_n\).

The corresponding instance of vertex cover \(\langle G, k \rangle\) is as follows:

- \(G\) has a node corresponding to each literal
- \(G\) has 3 nodes \(C_j^1, C_j^2, C_j^3\) corresponding to each clause \(C_j\) of \(\varphi\)
- \(G\) has edges:
  - \((x_i, \neg x_i)\),
  - \((C_j^1, C_j^2), (C_j^2, C_j^3), (C_j^3, C_j^1)\), and
  - If \(C_j = \alpha_1 \lor \alpha_2 \lor \alpha_3\), then \((C_j^1, \alpha_1), (C_j^2, \alpha_2), (C_j^3, \alpha_3)\)
- \(k = n + 2m\)

Clearly \(G\) can be composed in polynomial time from formula \(\varphi\).
The graph for formula $\varphi = (x_1 \lor \neg x_3 \lor \neg x_4) \land (\neg x_1 \lor x_2 \lor \neg x_4)$
1. \( \varphi \) is satisfiable \( \Rightarrow \)

\( G \) has a vertex cover of at most \( k = n + 2m \) nodes:

- Let us include into the vertex cover corresponding to the value assignment, the node representing the literal which obtains value 1 (\( n \) nodes)

- For each clause \( C_j \), one edge \((C_j^*, \alpha_r)\) of the corresponding triangle is now covered

- We take to the vertex cover the two remaining corners of the triangle (altogether \( 2m \) nodes)
2. \( G \) has a vertex cover of at most \( k \) nodes \( \Rightarrow \) \( \varphi \) is satisfiable

- Let \( V' \), \( |V'| \leq k \), be a vertex cover of \( G \)
- For \( V' \) to be able to cover all edges of \( G \), it must contain one node per each variable and at least two nodes from each \( C_j \)-triangle
- Hence, \( |V'| = k \)
- Let us set

\[
t(x_i) = \begin{cases} 
1, & \text{if } x_i \in V' \\
0, & \text{if } \neg x_i \in V' 
\end{cases}
\]

- One of the edges starting from the corners of each \( C_j \)-triangle is covered by a literal node \( \alpha \in V' \)
- Then \( t(\alpha) = t(C_j) = 1 \) \( \square \)
• Hence, \( \text{SAT} \leq_{m}^{p} \text{CSAT} \leq_{m}^{p} \text{3SAT} \leq_{m}^{p} \text{VC} \)

**Definition 7.34** A language \( B \) is **NP-complete** if it satisfies:

1. \( B \in \text{NP} \), and
2. \( A \leq_{m}^{p} B \) for every \( A \in \text{NP} \)

• A **NP-complete** language can be decided deterministically in polynomial time if and only if all other languages in \( \text{NP} \) can also be decided deterministically in polynomial time.

**Theorem 7.35** If \( B \) is **NP-complete** and \( B \in \text{P} \), then \( \text{P} = \text{NP} \).
Theorem 7.36  If $B$ is \textit{NP}-complete and $B \leq_{m}^{p} C$ for $C \in \textit{NP}$, then $C$ is \textit{NP}-complete.

\textbf{Proof.} Because $B$ is \textit{NP}-complete, by definition $A \leq_{m}^{p} B$ for every language $A \in \textit{NP}$. On the other hand, $B \leq_{m}^{p} C$, and by the transitivity of polynomial time reductions (Theorem 7.31) it must hold that $A \leq_{m}^{p} C$ for all $A \in \textit{NP}$. By assumption $C \in \textit{NP}$, and the claim holds.

- Hence, to show that language $C$ is \textit{NP}-complete, it suffices to reduce in polynomial time some language $B$ known to be \textit{NP}-complete to $C$ and in addition verify that $C \in \textit{NP}$
- However, we should find the first \textit{NP}-complete language
Theorem 7.37 (Cook-Levin) Language

\[ \text{SAT} = \{ \varphi \mid \varphi \text{ is a satisfiable Boolean formula} \} \]

is \text{NP}-complete.

- We need to show that \( A \leq^p_m \text{SAT} \) for any \( A \in \text{NP} \)
- All that we know about \( A \) is that it has a polynomial time nondeterministic decider \( N \)
- The reduction for \( A \) takes a string \( w \) and produces a Boolean formula \( \varphi_w \) that simulates \( N \) on input \( w \)
- \( \varphi_w \) is satisfiable iff \( w \in L(N) = A \)
- For each possible computation of \( N \) we have one truth value assignment of the variables in \( \varphi_w \)
- The formula \( \varphi_w \) is composed to give those conditions by which the given assignment corresponds to an accepting computation of \( N \)
Corollary 7.42 \textit{CSAT, 3SAT, and VC are NP-complete.}

**Independent Set, IS:** Given an undirected graph $G$ and a natural number $k$. Does $G$ have at least $k$ nodes which have no edges with each other?

By the following lemma it is easy to compose reductions $\text{VC} \leq^p_m \text{IS}$ and $\text{IS} \leq^p_m \text{CLIQUE}$

**Lemma** Let $G = (V, E)$ be an undirected graph and $V' \subseteq V$. Then the following conditions are equivalent:

1. $V'$ is a vertex cover in $G$,
2. $V \setminus V'$ is an independent set, and
3. $V \setminus V'$ is a clique in the complement graph of $G$: $\tilde{G} = (V, (V \times V) \setminus E)$
VC \leq_m^p IS:
Let us choose the mapping \( f \):
\[
f(\langle G, k \rangle) = \langle G, |V| - k \rangle.
\]
Clearly this transformation can be computed in polynomial time. Now, by the preceding lemma
\[
\langle G, k \rangle \in VC \iff \langle G, |V| - k \rangle \in IS.
\]
Hence, \( f : VC \leq_m^p IS \).

IS \leq_m^p CLIQUE:
Let us now choose the mapping \( f \):
\[
f(\langle G, k \rangle) = \langle \tilde{G}, k \rangle.
\]
This transformation can be computed in polynomial time and by the preceding lemma
\[
\langle G, k \rangle \in IS \iff \langle \tilde{G}, k \rangle \in CLIQUE.
\]
Thus, \( f : IS \leq_m^p CLIQUE \).
SAT / CSAT

3SAT

- Subset-sum
- VC
  - IS
  - CLIQUE
- Hamiltonian path
  - TSP
- CLIQUE
8. Space Complexity

• The space complexity of a standard Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, \text{accept, reject})$ on input $w$ is

$$\text{space}_M(w) = \max \{ |uav| : q_0 w \Rightarrow_M u q a v, q \in Q, u, a, v \in \Gamma^* \}$$

• The space complexity of a nondeterministic Turing machine $N = (Q, \Sigma, \Gamma, \delta, q_0, \text{accept, reject})$ on input $w$ is

• $\text{space}_N(w) = \text{the maximum number of tape cells scanned by the computation } q_0 w \Rightarrow_N ... \text{ requiring the most space}$
• Let \( s: \mathbb{N} \to \mathbb{R}^+ \) be an arbitrary function

• The deterministic space complexity class is:
  \[
  \text{DSPACE}(s(n)) = \{ L \mid L \text{ is a language decided by an } O(s(n)) \text{ space deterministic TM} \}
  \]

• Respectively, the nondeterministic space complexity class is:
  \[
  \text{NSPACE}(s(n)) = \{ L \mid L \text{ is a language decided by an } O(s(n)) \text{ space nondeterministic TM} \}
  \]

• Space can be reused, while time cannot

• E.g., deciding an instance of SAT only requires linear space, while it most probably is not polynomial-time decidable, since it is NP-complete
• The composite space complexity classes are:

\[ \text{PSPACE} = \bigcup \{ \text{DSPACE}(s) \mid s \text{ is a polynomial} \} = \bigcup_{k \geq 0} \text{DSPACE}(n^k) \]

\[ \text{EXPSPACE} = \bigcup_{k \geq 0} \text{DSPACE}(2^{n^k}) \]

\[ \text{NPSPACE} = \bigcup_{k \geq 0} \text{NSPACE}(n^k) \]

\[ \text{NEXPSPACE} = \bigcup_{k \geq 0} \text{NSPACE}(2^{n^k}) \]
Lemma For all $t(n), s(n) \geq n$:

1. $DTIME(t(n)) \subseteq DSPACE(t(n))$, and
2. $DSPACE(s(n)) \subseteq \bigcup_{k \geq 0} DTIME(k^{s(n)})$

Proof.

1. In $t(n)$ steps a Turing machine can write at most $t(n)$ symbols to the tape.

2. A TM requiring space $s(n)$ has at most $k^{s(n)}$ ($k$ constant) distinct configurations on an input of length $n$. By letting the computation continue for at most $k^{s(n)}$ steps we can make the TM always use time at most $k^{s(n)}$. 

Consequence: $P \subseteq \text{PSPACE} \subseteq \text{EXPTIME} \subseteq \text{EXPSPACE}$
8.1 Savitch’s Theorem

Theorem 8.5 Let $f: \mathbb{N} \rightarrow \mathbb{R}^+$ be any function, where $f(n) \geq n$. Then

$$\text{NSPACE}(f(n)) \subseteq \text{DSPACE}(f^2(n)).$$

Proof.

- Simulate a nondeterministic TM $N$ with a deterministic one $M$.
- Naïve approach: try all the branches of the computation tree of $N$ one by one.
- A branch using $f(n)$ space may run for $2^{O(f(n))}$ steps and each step may be a nondeterministic choice.
- Reach the next branch $\Rightarrow$ record all the choices of the branch.
- Thus, in the worst case the naïve approach may use $2^{O(f(n))}$ space.
- Hence, the naïve approach is not the one we are looking for.
Instead, let us examine the *yieldability problem*: can a TM $N$ get from configuration $c_1$ to configuration $c_2$ within $t$ steps.

By deterministically solving the yieldability problem, where

- $c_1$ is the start configuration of $N$ on some $w$ and
- $c_2$ is the accept configuration,

without using too much space implies the claim.

We search an intermediate configuration $c_m$ of $N$ such that

- $c_1$ can get to $c_m$ within $t/2$ steps, and
- $c_m$ can get to $c_2$ within $t/2$ steps.

We can reuse the space of the recursive calls.
CANYIELD\( (c_1, c_2, t) \):
\% \( t \) is a power of 2

1. If \( t = 1 \), then test directly whether \( c_1 = c_2 \) or whether \( c_1 \) yields \( c_2 \) in one step according to the rules of \( N \). Accept if either test succeeds; reject if both fail.
2. If \( t > 1 \), then for each configuration \( c_m \) of \( N \) on \( w \) using space \( f(n) \):
3. Run CANYIELD\( (c_1, c_m, t/2) \).
4. Run CANYIELD\( (c_m, c_2, t/2) \).
5. If steps 3 and 4 both accept, then accept.
6. If haven’t yet accepted, reject.
• \( M = \) “On input \( w \):
  1. Output the result of \( \text{CANYIELD}(c_{\text{start}}, c_{\text{accept}}, 2^{d_f(n)}) \):
• The constant \( d \) is selected so that \( N \) has no more than \( 2^{d_f(n)} \) configurations using \( f(n) \) tape, where \( n = |w| \)

• The algorithm needs space for storing the recursion stack.
• Each level of the recursion uses \( O(f(n)) \) space to store a configuration.
  • Just store the current step number of the algorithm and values \( c_1, c_2, \) and \( t \) on a stack

• Because each level of recursion divides the size of \( t \) in half, the depth of the recursion is logarithmic in the maximum time that \( N \) may use on any branch; i.e., \( \log(2^{O(f(n))}) = O(f(n)) \).

• Hence, the deterministic simulation uses \( O(f^2(n)) \) space.
Corollary

1. \( \text{NPSPACE} = \text{PSPACE} \)
2. \( \text{NEXPSpace} = \text{EXPSPACE} \)

Now our knowledge of the relationship among time and space complexity classes extends to the linear series of containments:

\[
P \subseteq \text{NP} \subseteq \left\{ \begin{array}{l}
\text{PSPACE} = \text{NPSPACE} \\
\text{EXPTIME} \end{array} \right\} \subseteq \text{EXPTIME} \subseteq \\
\text{NEXPTIME} \subseteq \left\{ \begin{array}{l}
\text{EXPSPACE} = \text{NEXPSpace} \\
\end{array} \right\}
\]
Theorem

1. $P \neq \text{EXPTIME}$
2. $\text{PSPACE} \neq \text{EXPSPACE}$

- Complexity classes that have "exponential distance" from each other are known to be distinct
  
  $P \neq \text{EXPTIME}, \ NP \neq \text{NEXPTIME}$ and $\text{PSPACE} \neq \text{EXPSPACE}$

- However, one does not know whether the following hold
  
  $P \neq \text{NP}, \ NP \neq \text{PSPACE}, \ PSPACE \neq \text{EXPTIME}$, ...

- Most researchers believe that all these inequalities hold, but cannot prove the results

- There does not exist the “most difficult” decidable language
Theorem
1. If $A$ is decidable, then there exists a computable function $t(n)$, for which $A \in \text{DTIME}(t(n))$.
2. For every computable function $t(n)$ there exists a decidable language $A$ not belonging to $\text{DTIME}(t(n))$.

Function $f(n) = f(n, n)$ is computable
$$f(0, n) = 2^{2^{2^{2^{\cdots}}}} \text{ } n \text{ times}$$
$$f(m+1, n) = 2^{2^{2^{2^{\cdots}}}} f(m, n) \text{ times}$$
$$f(0) = 1, f(1) = 4, f(2) = 2^{2^{2^{2^{2^{\cdots}}}}} \text{ 65 536 times}$$

By the above theorem: $A \in \text{DEC} \setminus \text{DTIME}(f(n)) \neq \emptyset$

In other words there are “decidable” problems, whose time complexity on input of length $n$ is not bounded by function $f(n)$. 
8.3 PSPACE-Completeness

A language $B$ is **PSPACE-complete**, if

1. $B \in \text{PSPACE}$ and
2. $A \leq_{m^p} B$ for all $A \in \text{PSPACE}$

If $B$ merely satisfies condition 2, we say that it is **PSPACE-hard**

In a *fully quantified Boolean formula* each variable appears within the scope of some quantifier; e.g.

$$\varphi = \forall x \exists y [(x \lor y) \land (\neg x \lor \neg y)]$$

TQBF is the language of true fully quantified Boolean formulas

**Theorem 8.9** TQBF is PSPACE-complete.
• We need to reduce any \( A \in \text{PSPACE} \) polynomially to TQBF starting from the polynomial-space bounded TM \( M \) for \( A \).

• Imitating the proof of Cook-Levin theorem is out of the question because \( M \) can in any case run for exponential time.

• Instead, one has to resort to a proof technique resembling the one used in the proof of Savitch’s theorem.

• The reduction maps string \( w \) to a quantified Boolean formula \( \varphi \) that is true if and only if \( M \) accepts \( w \).

• Collections of variables denoted \( c_1 \) and \( c_2 \) represent two configurations.
• We construct a formula $\varphi(c_1, c_2, t)$, which is true if and only if $M$ can go from $c_1$ to $c_2$ in at most $t > 0$ steps.
• This yields a reduction when we construct the formula $\varphi(c_{\text{start}}, c_{\text{accept}}, h)$, where $h = 2^{d f(n)}$.
• The constant $d$ is chosen so that $M$ has no more than $2^{d f(n)}$ possible configurations on an input of length $n$.
• $f(n) = n^k$.

• The formula encodes the contents of tape cells.
• Each cell has several variables associated with it,
  • one for each tape symbol and one for each state of $M$. 

Each configuration has \( n^k \) cells and so is encoded by \( O(n^k) \) variables.

If \( t = 1 \), then
1. either \( c_1 = c_2 \), or
2. \( c_2 \) follows from \( c_1 \) in a single step of \( M \).

Now constructing formula \( \varphi(c_1, c_2, t) \) is easy
1. Each of the variables representing \( c_1 \) has the same Boolean value as the corresponding variable representing \( c_2 \).
2. (As in the proof of Cook-Levin theorem) writing expressions stating that the contents of each triple of \( c_1 \)'s cells correctly yields the contents of the corresponding triple of \( c_2 \)'s cells.
• If $t > 1$, we construct $\varphi(c_1, c_2, t)$ recursively.

• A straightforward approach would be to define

$$\varphi(c_1, c_2, t) = \exists m_1 \left[ \varphi(c_1, m_1, t/2) \land \varphi(m_1, c_2, t/2) \right],$$

where $m_1$ represents a configuration of $M$.

• Writing $\exists m_1$ is a shorthand for $\exists x_1, \ldots, x_l$, where $l = O(n^k)$ and $x_1, \ldots, x_l$ are the variables that encode $m_1$.

• $\varphi(c_1, c_2, t)$ has the correct value, its value is true whenever $M$ can go from $c_1$ to $c_2$ within $t$ steps.

• However, the formula becomes too big.

• Every level of recursion cuts $t$ in half, but roughly doubles the size of the formula $\Rightarrow$ we end up with a formula of size $\approx t$.

• Initially $t = 2^{df(n)} \Rightarrow$ exponentially large formula.
• To shorten formula $\varphi(c_1, c_2, t)$ we change it to form
  
  $\exists m_1 \forall(c_3, c_4) \in \{(c_1, m_1), (m_1, c_2)\} : [\varphi(c_3, c_4, t/2)]$
  
  • This formula preserves the original meaning, but folds the two recursive subformulas into a single one
  
  • To obtain a syntactically correct formula
    
    $\forall x \in \{y, z\} : [...]$ may be replaced with the equivalent
    
    $\forall x \ [ (x = y \land x = z) \Rightarrow [...] ]$

  • Each level of recursion adds a portion of the formula that is linear in the size the configurations, i.e., $O(f(n))$
  
  • The number of levels of recursion is $\log(2^{df(n)}) = O(f(n))$
  
  • Hence the size of the resulting formula is $O(f^2(n))$
The course book presents the PSPACE-completeness of two "artificial" games. For them no polynomial time algorithm exists unless \( P = PSPACE \).

Standard chess on a \( 8 \times 8 \) board does not directly fall under the same category.

There is only a finite number of different game positions.

In principle, all these positions may be placed in a table, along with the best move for each position.

Thus, the control of a Turing machine (or finite automaton) can store the same information and it can be used to play optimally in linear time using table lookup.

Unfortunately the table would be too large to fit inside our galaxy.
• Current complexity analysis methods are asymptotic and apply only to the rate of growth of the complexity as the problem size increases
• The complexity of problems of any fixed size cannot be handled using these techniques

• Generalizing games to an $n \times n$ board gives some evidence for the difficulty of computing optimal play
• For example, generalizations of chess and GO have been shown to be at least PSPACE-hard
8.4 The Classes L and NL

- In time complexity there is really no point in considering *sublinear* complexity classes, because they are not sufficient for reading the entire input.
- On the other hand, one does not necessarily need to store the entire input and, therefore, it makes sense to consider sublinear space complexity classes.

- To make things meaningful, we must modify our model of computation.
- Let a Turing machine have in addition to the read/write work tape a read only input tape.
- Only the cells scanned on the work tape contribute to the space complexity of the TM.
• \( L \) is the class of languages that are decidable in logarithmic space on a deterministic Turing machine:
  \[
  L = \text{DSPACE}(\log n)
  \]

• \( \text{NL} \) is the respective class for nondeterministic TMs

• Earlier we examined deciding language \( A = \{ 0^k 1^k \mid k \geq 0 \} \) with a Turing machine requiring linear space

• We can decide whether a given string belongs to the language using only a logarithmic space

• It suffices to count the numbers of zeros and ones, separately (and make the necessary checks). The binary representations of the counters uses only logarithmic space.

• Therefore, \( A \in L \)
Earlier we showed that the language \textsc{PATH}: \("G\) has a directed path from \(s\) to \(t.\)" is in \(P\)

The algorithm given uses linear space

We do not know whether \textsc{PATH} can be solved deterministically in logarithmic space

A nondeterministic logarithmic space algorithm, on the other hand, is easy to find

Starting at node \(s\) repeat at most \(m\) steps

- If current node is \(t,\) then \textit{accept}
- Record the position of the current node on the work tape (logarithmic space)
- Nondeterministically select one of the followers of the current node
• For very small space bounds the earlier claim that any $f(n)$ space bounded TM also runs in time $2^{O(f(n))}$ is no longer true
• For example, a Turing machine using constant $O(1)$ space may run for a linear $O(n)$ number of steps

• For small space bounds the time complexity has the asymptotic upper bound $n2^{O(f(n))}$
• If $f(n) \geq \log n$, then $n2^{O(f(n))} = 2^{O(f(n))}$
• Also Savitch’s theorem holds as such when $f(n) \geq \log n$
• As mentioned, PATH is known to be in $NL$ but probably not in $L$
• In fact, we don’t know of any problem in $NL$ that can be proved to be outside $L$
8.5 NL-Completeness

- $L =?= NL$ is an analogous question to $P =?= NP$
- We can define $NL$-complete languages as the most difficult languages in $NL$

- Polynomial time reducibility, however, cannot be used in the definition, because all problems in $NL$ are solvable in polynomial time
- Therefore, every two problems in $NL$ except $\emptyset$ and $\Sigma^*$ are polynomial time reducible to one another

- Instead we use logarithmic space reducibility $\leq_L$
- A function is log space computable if the Turing machine computing it uses only $O(\log n)$ space from its work tape
**Theorem 8.23**  If $A \leq_L B$ and $B \in L$, then $A \in L$. □

**Corollary 8.24**  If any NL-complete language is in $L$, then $L = NL$. □

**Theorem 8.25**  PATH is NL-complete. □

**Corollary 8.26**  $NL \subseteq P$.

**Proof.** Theorem 8.25 shows that for any language $A$ in NL it holds that $A \leq_L$ PATH. A Turing machine that uses space $f(n)$ runs in time $n2^{O(f(n))}$, so a reducer that runs in log space also runs in polynomial time.

Because $A \leq_m^p$ PATH and PATH $\in P$, by Theorem 7.14, also $A \in P$. □
8.6 NL = coNL

- For Example, NP and coNP are generally believed to be different
- At first glance, the same would appear to hold for NL and coNL
- There are, however, gaps in our understanding of computation:

Theorem 8.27  \( \text{NL} = \text{coNL} \)

Proof. The complement of \( \text{PATH} \) – “The input graph \( G \) does not contain a path from \( s \) to \( t \).” – is in \( \text{NL} \). Because \( \text{PATH} \) is \( \text{NL} \)-complete, every problem in \( \text{coNL} \) is also in \( \text{NL} \).

- Our knowledge of the relationship among complexity classes \( L \subseteq \text{NL} = \text{coNL} \subseteq \text{P} \subseteq \text{PSPACE} \)
- Whether any of these containments are proper is unknown, but all are believed to be
10 Advanced Topics in Complexity Theory

- What to do with a problem that is *intractable* and does not accept a deterministic exact solution in polynomial time
- Relax the problem:
  1. Instead of solving it exactly, approximate the solution
  2. Instead of using a deterministic algorithm, use a probabilistic (a.k.a. randomized) algorithm
- Approximation algorithm finds a solution that is guaranteed to be close to the optimal exact solution
- Probabilistic algorithm comes up with the exact solution with a high probability
- Sometimes it may fail to give the correct answer (Monte Carlo) or may have a high time requirement (Las Vegas)
10.1 Approximation Algorithms

- Let us examine a problem, where we are given
  - A ground set $U$ with $m$ elements
  - A collection of subsets of the ground set $S = \{ S_1, \ldots, S_n \}$ s.t. it is a cover of $U$: $US = U$

- The aim is to find a subcover $S' \subseteq S$, $US' = U$, containing as few subsets as possible
- This problem is known as the Minimum Set Cover (minSC)

- One of the oldest and most studied combinatorial optimization problems
The corresponding decision problem
- Given: a ground set \( U \), cover \( S \) and a natural number \( k \)
- Question: Does \( U \) have a subcover \( S' \subseteq S \) s.t. \(|S'| \leq k\)?

**Theorem** The decision version of minimum set cover problem is NP-complete.

**Proof.** Obviously \( \text{minSC} \in \text{NP} \): Let us guess from the given cover \( S \) a subcover \( S' \) containing \( k \) subsets and verify deterministically in polynomial time that we really have a subcover.
Polynomial time reduction $\text{VC} \leq_m^p \text{minSC}$ is easy to give. Let $\langle G, k \rangle$ be an instance of the vertex cover in which $G = (V, E)$. We choose the mapping $f$:

$$f(\langle (V, E), k \rangle) = \langle E, V_E, k \rangle,$$

where $V_E$ is the collection of edges connected to the nodes of $G$. In other words, for each $v \in V$ has a corresponding set

$$\{ e \in E \mid e = (v, w) \}.$$

Clearly $f$ is computable in polynomial time and is a reduction.  \qed
• Hence, minSC is an intractable problem – we do not know of a polynomial time algorithm for solving it
• Therefore, we attempt to find a polynomial time algorithm
  • that does not necessarily give the best possible (optimal) solution, but
  • can be shown always to be at most a function of the input length worse than the optimal solution
• Such an algorithm is called an approximation algorithm

• Let us denote by
  • Opt the cost of the solution given by an optimal algorithm and
  • App that of the solution given by an approximation algorithm
Since minSC is a minimization problem, $\frac{\text{App}}{\text{Opt}} \geq 1$.

The closer to 1 this ratio is, the better the solution produced approximates the optimal solution.

From an approximation algorithm one requires that the fraction is bounded by a function of the length $n$ of the input

$$\frac{\text{App}}{\text{Opt}} \leq \rho(n)$$

- $\rho(n)$ is the approximation ratio of the algorithm.
- The algorithm is called an $\rho(n)$-approximation algorithm.
- At the best the approximation ratio does not depend at all on the length $n$ of the input, but is constant.
Let us examine the following algorithm for vertex cover
We will show that it is an 2-approximation algorithm for the problem

**Input:** An undirected graph $G = (V, E)$

**Output:** Vertex cover $C$

1. $C \leftarrow \emptyset$
2. $E' \leftarrow E$
3. while $E' \neq \emptyset$ do
   a. Let $(u, v)$ be any edge of the set $E'$;
   b. $C \leftarrow C \cup \{u, v\}$
   c. Remove from $E'$ all edges connected to nodes $u$ and $v$
4. od
5. return $C$
Selection of the first random edge: $(b, c)$
We remove other edges connected with nodes $b$ and $c$
The next random choice: \((e, f)\) and
Removal of other edges connected with its nodes
The only remaining choice \((d, g)\)

We end up with a cover of 6 nodes, while the optimal one has 3 nodes (e.g., \(b, d, e\))
Theorem 10.1 The above given algorithm is polynomial time 2-approximation algorithm for vertex cover.

Proof. The time complexity of the algorithm, using adjacency list representation for the graph, is \( O(V + E) \), and thus uses a polynomial time.

The set of nodes \( C \) returned by the algorithm obviously is a vertex cover for the edges of \( G \), because nodes are inserted into \( C \) in the loop of row 3 until all edges have been covered.

Let \( A \) be the set of edges chosen by algorithm in row 3a. In order to cover the edges of \( A \) any vertex cover — in particular also the optimal vertex cover — has to contain at least one of the ends of each edge in \( A \).
Because the end points of the edges in $A$ are distinct by the design of the algorithm, $|A|$ is a lower bound for the size of any vertex cover.

In particular,

$$\text{Opt} \geq |A|.$$  

The above algorithm always selects in row 3a an edge whose neither end point is yet in the set $C$. Hence,

$$\text{App} = |C| = 2|A|.$$  

Combining the above equations yields

$$\text{App} = 2|A| \leq 2 \text{Opt},$$  

and therefore

$$\text{App}/\text{Opt} \leq 2.$$  

$\Box$
Also set cover has a simple greedy approximation algorithm
Neither this nor any other polynomial time deterministic algorithm can attain a constant approximation ratio

**Input:** Ground set $U$ and its cover $S$

**Output:** Set cover $C$

1. $X \leftarrow U; C \leftarrow \emptyset$
2. **while** $X \neq \emptyset$ **do**
   a. select $S' \in S$ s.t. $|S' \cap X|$ is maximized;
   b. $X \leftarrow X \setminus S'$;
   c. $C \leftarrow C \cup \{S'\}$
3. **od**;
4. return $C$;
Greedy: 4 subsets
Optimal: 3 subsets

$S_3 \quad S_4 \quad S_5$
• The greedy algorithm can quite easy to implement to run in polynomial time in the length of the input $|U|$ and $|S|$
  • The loop in row 2 is executed at most $\min(|U|, |S|)$ times and the body of the loop itself can be implemented to require time $O(|U| \cdot |S|)$
  • Altogether the time requirement thus is $O(|U| \cdot |S| \cdot \min(|U|, |S|))$
  • It is also possible to give a linear time implementation for the greedy approximation algorithm for set cover

• The collection $C$ returned by the algorithm is obviously a set cover, because the loop of row 2 is executed until there are no more elements to cover
In order to relate the cost of the set cover returned by the greedy algorithm, we set cost 1 to each of the chosen sets.

Let $S_i$ be the set selected by the greedy algorithm at round $i$

We distribute the cost of $S_i$ evenly among all those elements in it that now become covered for the first time.

Let $c_u$ denote the cost assigned on element $u \in U$

Each element gets assigned a cost only once, the first time it is covered by some set.

If $u$ is first covered by the set $S_i$, the cost assigned to it is:

$$c_u = \frac{1}{|S_i \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|}$$
• Each set selected by the greedy algorithm is assigned cost 1 so that

\[ \text{App} = |C| = \sum_{u \in U} c_u \]

• On the other hand, the cost of the optimal cover \( C^* \) is

\[ \sum_{S' \in C^*} \sum_{u \in S'} c_u \]

• Because each \( u \in U \) belongs to at least one \( S' \in C^* \), we have

\[ \sum_{S' \in C^*} \sum_{u \in S'} c_u \geq \sum_{u \in U} c_u \]

• Combining the above given yields

\[ \text{App} \leq \sum_{S' \in C^*} \sum_{u \in S'} c_u \]
• Let $H(k)$ denote the $k$-th harmonic number

$$H(k) = \sum_{j=1}^{k} \frac{1}{j} = 1 + \frac{1}{2} + \cdots + \frac{1}{k}$$

• We define $H(0) = 0$

• Next we show that for any $S' \in S$ it holds

$$\sum_{u \in S'} c_u \leq H(|S'|)$$

• Then, by the previous inequality,

$$\text{App} \leq \sum_{S' \in C^*} H(|S'|)$$

$$\leq |C^*| \cdot H(\max\{|S'|: S' \in S\})$$

$$= \text{Opt} \cdot H(\max\{|S'|: S' \in S\})$$
Lemma For each \( S' \in S \) it holds

\[
\sum_{u \in S'} c_u \leq H(|S'|)
\]

Proof. Let \( S' \in S \) be arbitrary and \( i = 1, 2, \ldots, |C| \). Furthermore, let

\[
n_i = |S' \setminus (S_1 \cup S_2 \cup \ldots \cup S_i)|
\]

be the number of those elements of \( S' \) that have not yet been covered when the greedy algorithm has chosen sets \( S_1, S_2, \ldots, S_i \) to the set cover.

Let \( n_0 = |S'| \).

Let \( k \) be the smallest index s.t. \( n_k = 0 \); i.e., every element of \( S' \) belongs to at least one of the sets \( S_1, S_2, \ldots, S_k \).

Then \( n_{i-1} \geq n_i \) and \( S_i, i = 1, 2, \ldots, k \), covers \( n_{i-1} - n_i \) elements for the first time.
Now

\[ \sum_{u \in S'} c_u = \sum_{i=1}^{k} (n_{i-1} - n_i) \frac{1}{|S_i \setminus (S_1 \cup \cdots \cup S_{i-1})|}. \]

Since \( S_i \) is chosen greedily, it must cover at least as many elements as the set \( S' \) (or otherwise \( S' \) should have been selected). Hence,

\[ |S_i \setminus (S_1 \cup \cdots \cup S_{i-1})| \geq |S' \setminus (S_1 \cup \cdots \cup S_{i-1})| = n_{i-1} \]

Which further yields

\[ \sum_{u \in S'} c_u \leq \sum_{i=1}^{k} (n_{i-1} - n_i) \frac{1}{n_{i-1}}. \]
\[
\sum_{u \in S'} c_u \leq \sum_{i=1}^{k} (n_{i-1} - n_i) \frac{1}{n_{i-1}}
\]

\[
= \sum_{i=1}^{k} \sum_{j=n_i+1}^{n_{i-1}} \frac{1}{n_{i-1}}
\]

\[
\leq \sum_{i=1}^{k} \sum_{j=n_i+1}^{n_{i-1}} \frac{1}{j},
\]

because \(j \leq n_{i-1}\). Moreover,

\[
= \sum_{i=1}^{k} \left( \sum_{j=1}^{n_i} \frac{1}{j} - \sum_{j=1}^{n_{i-1}} \frac{1}{j} \right)
\]

\[
= \sum_{i=1}^{k} \left( H(n_{i-1}) - H(n_i) \right)
\]

\[
= H(n_0) - H(n_k),
\]

since the other terms in the sum cancel each other out.
We have chosen $n_k = 0$ and defined $H(0) = 0$. Therefore, further

$$= H(n_0) - H(0)$$

$$= H(n_0)$$

$$= H(|S'|)$$

and we have proved the lemma.

- For the harmonic number $H(k)$ it holds $\ln k < H(k) \leq \ln k + 1$
- From the above results it follows:

**Theorem** *For the greedy algorithm of the set cover problem it holds that*

$$\frac{\text{App}}{\text{Opt}} \leq H(\max \{|S'|: S' \in S\}) \leq \ln |U| + 1$$
In some applications \( \max \{ |S'| : S' \in S \} \) is a small constant.

Then the solution returned by the greedy algorithm is only a small constant away from the optimal one.

In particular, if subsets \( S' \) have an upper bound \( d \) for their size,
\[
\text{App/Opt} \leq H(d)
\]

E.g., when the nodes of the graph of vertex cover have maximum degree 3, then

- the solution returned by the greedy set cover algorithm is at most \( H(3) = \frac{11}{6} < 2 \) times as large as the optimal cover.
• Feige, 1996: no polynomial-time algorithm can approximate minSC within \((1-\epsilon) \ln m\), for any \(\epsilon > 0\), unless \(\text{NP} \subseteq \text{DTIME}(n^{\log \log n})\).

• Hence, it is not possible to find an approximation algorithm for minSC that would be significantly better than the greedy one.

• Slavík, 1996: A more exact upper bound for the approximation ratio of the greedy algorithm is

\[ \ln m - \ln \ln m + \Theta(1) \]

• In fact this is also a lower bound for the approximation ratio of the greedy algorithm.

• \(\ln m - \ln \ln m + \Theta(1)\) is thus the asymptotically exact approximation ratio of the greedy algorithm.
10.2 Probabilistic Algorithms

- A.k.a. randomized algorithms
- Another way of dealing with too time consuming computation
- Certain types of problems seem to be more easily solvable by “flipping a coin” than by deterministic algorithms
- Calculating the best choice may require excessive time
- Estimating it may introduce a bias that invalidates the result
- For example, statisticians use random sampling
  - Instead of querying all the individuals for their political preferences might take too long
  - Randomly selected subset of voters gives reliable results at a small cost
The Class BPP

- A probabilistic Turing machine $N$ is a nondeterministic TM in which each nondeterministic step is called a coin-flip step.
- Such a step has two legal next moves.
- We assign a probability to each branch $b$ of $N$’s computation on input $w$ as follows:
  \[ \Pr[b] = 2^{-k}, \]
  where $k$ is the number of coin-flip steps that occur on branch $b$.
- The probability that $N$ accepts $w$ is
  \[ \Pr[N \text{ accepts } w] = \sum_{b \in A} \Pr[b] \]
  where $A$ is the set of accepting branches of computation.
\begin{itemize}
\item \( \Pr[\text{N rejects } w] = 1 - \Pr[\text{N accepts } w] \)
\item As usual, when a probabilistic TM recognizes a language, it must accept all strings in the language and reject all those out of the language.
\item Except that now we allow the machine a small probability of error.
\item For all \( 0 \leq \varepsilon < \frac{1}{2} \) we say that \( \text{N} \) recognizes language \( A \) with probability of error \( \varepsilon \) if
  \begin{itemize}
  \item \( w \in A \Rightarrow \Pr[\text{N accepts } w] \geq 1 - \varepsilon \)
  \item \( w \notin A \Rightarrow \Pr[\text{N rejects } w] \geq 1 - \varepsilon \)
  \end{itemize}
\item I.e., the probability of obtaining the wrong answer by simulating \( \text{N} \) is at most \( \varepsilon \).
\item Sometimes error probability bounds depend on the input length \( n \); e.g., exponentially small: \( \varepsilon = 2^{-n} \).
\end{itemize}
**Definition 10.4**  \( \text{BPP} \) is the class of languages that are recognized by probabilistic polynomial time Turing machines with an error probability \( \frac{1}{3} \).

- Any constant error instead of \( \frac{1}{3} \) would yield an equivalent definition as long as it is in the interval \( ]0, \frac{1}{2} [ \)
- By the virtue of the following amplification lemma we can always make the error probability exponentially small
- A probabilistic algorithm with an error probability of \( 2^{-100} \) is far more likely to give an erroneous result because of a hardware failure than because of an unlucky toss of its coin
Amplification Lemma

Lemma 10.5 Let $\varepsilon$ be a fixed constant strictly in between 0 and $\frac{1}{2}$. Then for any polynomial $p(n)$ a probabilistic polynomial time TM $N$ that operates with error probability $\varepsilon$ has an equivalent probabilistic polynomial time TM $N_2$ that operates with an error probability of $2^{-p(n)}$.

Proof (Idea) $N_2$ simulates $N$ by running it a polynomial number of times and taking the majority vote of the outcomes. The probability of error decreases exponentially with the number of runs of $N$ made.
Primality

Let \( \mathbb{Z}_p^+ = \{ 0, \ldots, p - 1 \} \)

Every integer is equivalent modulo \( p \) to some member of the set \( \mathbb{Z}_p^+ \)

Theorem 10.6 (Fermat's little theorem)

If \( p \) is prime and \( a \in \mathbb{Z}_p^+ \), then

\[
a^{p-1} \equiv 1 \pmod{p}.
\]

For example, \( 2^{7-1} = 2^6 = 64 \) and \( 64 \mod 7 = 1 \)

while \( 2^{6-1} = 2^5 = 32 \) and \( 32 \mod 6 = 2 \)

hence \( 6 \) is not prime

We show that \( 6 \) is a composite number without factoring it!
• Fermat’s little theorem, thus, (almost) gives a test for primality
• We say that $p$ passes the Fermat test at $a$, if
  \[ a^{p-1} \equiv 1 \pmod{p} \]

• Call a number $p$ pseudoprime if it passes Fermat tests for all smaller $a$ relatively prime to it
• Only infrequent *Carmichael numbers* are pseudoprime without being prime
• If a number is not pseudoprime, it fails at least half of all Fermat tests
• Hence, we easily get a pseudoprimality algorithm with an exponentially small error probability
Pseudoprime($p$)
1. Select random $a_1, \ldots, a_k \in \mathbb{Z}_p^+$
2. Compute $a_i^{p-1} \mod p$ for each $i$
3. If all computed values are 1 accept, otherwise reject

- If $p$ isn’t pseudoprime, it passes each randomly selected test with probability at most $\frac{1}{2}$
- The probability that it passes all $k$ tests is thus at most $2^{-k}$
- The algorithm operates in polynomial time

- To convert this algorithm to a primality algorithm, we should still avoid the problem with the Carmichael numbers
The number 1 has exactly two square roots, 1 and -1, modulo any prime $p$.

For many composite numbers, including all the Carmichael numbers, 1 has four or more square roots.

For example, $\pm 1$ and $\pm 8$ are the four square roots of 1 modulo 21.

We can obtain square roots of 1 if $p$ passes the Fermat test at $a$ because

- $a^{p-1} \mod p \equiv 1$ and so
- $a^{(p-1)/2} \mod p$ is a square root of 1.

We may repeatedly divide the exponent by two, so long as the resulting exponent remains an integer.
Prime($p$)
% accept = input $p$ is prime

1. If $p$ is even, **accept** if $p = 2$, otherwise **reject**
2. Select random $a_1, ..., a_k \in \mathbb{Z}_p^+$
3. For each $i \in \{1, ..., k\}$
   a) Compute $a_i^{p-1} \mod p$ and **reject** if different from 1
   b) Let $p - 1 = st$ where $s$ is odd and $t = 2^h$ is a power of 2
   c) Compute the sequence $a_i^{s \cdot 2^0}, a_i^{s \cdot 2^1}, ..., a_i^{s \cdot 2^h}$ modulo $p$
   d) If some element of this sequence is not 1, find the last element that is not 1 and **reject** if that element is not $-1$
4. All test have been passed, so **accept**
Lemma 10.7 *If* $p$ *is an odd prime,*

$$\Pr[\text{Prime accepts } p] = 1.$$  

**Proof** If $p$ is prime, no branch of the algorithm rejects: Rejection in step 3a means that $(a^{p-1} \mod p) \neq 1$ and Fermat’s little theorem implies that $p$ is composite.

If rejection happens in step 3d, there exists some $b \in \mathbb{Z}_p^+$ s.t.

$b \not\equiv \pm 1 \pmod{p}$ and $b^2 \equiv 1 \pmod{p}$. Therefore $b^2 - 1 \equiv 0 \pmod{p}$. Factoring yields

$$(b - 1)(b + 1) \equiv 0 \pmod{p},$$

which implies that $(b - 1)(b + 1) = cp$ for some positive integer $c$.

Because $b \not\equiv \pm 1 \pmod{p}$, both $b - 1$ and $b + 1$ are in the interval $]0, p[$. Therefore $p$ is composite because a multiple of a prime number cannot be expressed as a product of numbers that are smaller than it is.  

$\square$
• The next lemma shows that the algorithm identifies composite numbers with high probability

• An important elementary tool from number theory, *Chinese remainder theorem*, says that a one-to-one correspondence exists between \( \mathbb{Z}_{pq} \) and \( \mathbb{Z}_p \times \mathbb{Z}_q \) if \( p \) and \( q \) are relatively prime:
  • Each number \( r \in \mathbb{Z}_{pq} \) corresponds to a pair \((a, b)\), where \( a \in \mathbb{Z}_p \) and \( b \in \mathbb{Z}_q \) s.t.
    • \( r \equiv a \pmod{p} \) and
    • \( r \equiv b \pmod{q} \)
Lemma 10.8 If $p$ is an odd composite number,

$$\Pr[\text{Prime accepts } p] \leq 2^{-k}.$$  

Proof Omitted, takes advantage of the Chinese remainder thm. \qed

• Let PRIMES = \{ $n$ | $n$ is a prime number in binary \}

• The preceding algorithm and its analysis establishes:

Theorem 10.9 PRIMES $\in$ BPP

Note that the probabilistic primality algorithm has one-sided error. When it rejects, we know that the input must be composite. An error may only occur in accepting the input.
Thus an incorrect answer can only occur when the input is a composite number. For all primes we get the correct answer.

The one-sided error feature is common to many probabilistic algorithms, so the special complexity class $\text{RP}$ is designated for it:

**Definition 10.10** $\text{RP}$ is the class of languages that are recognized by probabilistic polynomial time Turing machines where inputs in the language are accepted with a probability of at least $\frac{1}{2}$ and inputs not in the language are rejected with a probability of 1.

Our earlier algorithm shows that $\text{COMPOSITES} \in \text{RP}$
**PRIMES ∈ P**

- A generalization of Fermat’s little theorem:

**Theorem A.** Let $a$ and $p$ be relatively prime and $p > 1$. $p$ is a prime number if and only if 

$$
(X - a)^p \equiv X^p - a \pmod{p}
$$

- $X$ is not important here, only the coefficients of the polynomial $(X - a)^p - (X^p + a)$ are significant.

- For $0 < i < p$, the coefficient of $X^i$ is $\binom{p}{i}a^{p-i}$. Supposing that $p$ is prime, $\binom{p}{i} = 0 \pmod{p}$ and hence all the coefficients are zero.

- Therefore, we are left with the first term $X^p$ and the last one $-a^p$, which is $-a$ modulo $p$.

- Unfortunately, deciding the primality of $p$ based on this requires an exponential time.
• Agrawal (1999): it suffices to examine the polynomial \((X - a)^p\) modulo \(X^r - 1\)

• If \(r\) is large enough, the only composite numbers that pass the test are powers of odd primes

• On the other hand, \(r\) should be quite small so that the complexity of the approach does not grow too much

• Kayal & Saxena (2000): Based on an unproven conjecture, \(r\) doesn’t have to be larger than \(4(\log^2 p)\), in which case the complexity of the test procedure is only of the order \(O(\log^3 n)\); that is, belongs to \(P\)

• The only difficulty is that the result is based on an unproven claim
• A pair of odd numbers is called *Sophie Germain primes* if both \( q \) and \( 2q + 1 \) are primes (related to Fermat’s last theorem).

• Agrawal, Kayal & Saxena (2002): If one can find a pair of SG primes \( q \) and \( 2q + 1 \) s.t.

\[
q > 4\left(\sqrt{2q + 1}\right) \cdot \log p
\]

then \( r \) does not need to larger than

\[
2\left(\sqrt{2q + 1}\right) \cdot \log p
\]

• Unfortunately this test is recursive and has time requirement of \( O(\log^{12} n) \) instead of the \( O(\log^3 n) \) mentioned above.
Deterministic-Prime($p$)

1. if $p = a^b$ for some $b > 1$ then reject;
2. $r ← 2$;
3. while $r < p$ do
   a) if $\gcd(p, r) \neq 1$ then reject;
   b) if Deterministic-Prime($r$) then
      i. Let $q$ be the largest factor of $r−1$;
      ii. if $q > 4\sqrt{r} \log p$ and $p^{(r−1)/q} \neq 1 \pmod{r}$ then break;
   c) $r ← r + 1$;
4. for $a ← 1$ to $2\sqrt{r} \log p$ do
   if $(x−a)^p \neq x^p−a \pmod{x^r−1, p}$ then reject;
5. accept the input;
The test of row 1 removes the powers of odd primes as required by the test of Agrawal (1999)
The loop of row 3 searches a pair of Sophie Germain primes \( q \) and \( r \)
Row 3a) tests for Theorem A that \( p \) and \( r \) are relatively prime
The loop of row 4 examines primality using a variation of Theorem A (Agrawal, 1999) up to value \( 2\sqrt{r} \log p \) (AKS, 2002)

Because Theorem A holds if and only if \( p \) is prime, the decision of the algorithm is correct
The other variations only affect the complexity of the algorithm, not its correctness
Course Recap

✓ All computational problems cannot be solved algorithmically.
✓ A deterministic finite automaton (DFA) has a unique minimal automaton. The minimal automaton can be constructed in a straightforward manner.
✓ Nondeterministic finite automata (NFA) do not recognize more languages than DFAs.

✓ A language is regular
  ⇔ it can be recognized with a finite automaton
  ⇔ it can be described with a regular expression
  ⇔ it can be generated with a right-linear grammar.
Strings of a regular language can be "pumped"

There are sensible formal languages that are not regular

Context-free languages are a proper superset of regular languages

A language is context-free if and only if it can be recognized with a pushdown automaton (PDA)

By the Church-Turing thesis any problem solvable on a computer can also be solved using a Turing machine (TM)

Variants of Turing machines – including nondeterministic Turing machines – have equal recognition power to the standard single-tape machine
The "efficiency" of different machines varies
Languages generated by unrestricted grammars are equivalent to those recognized by Turing machines
A total Turing machine (decider) halts on every input
A formal language is Turing-recognizable (TR), if it can be recognized with a TM and Turing-decidable, if it has a decider

- $A$ and $B$ decidable $\Rightarrow \bar{A}$, $A \cup B$ and $A \cap B$ are decidable
- $A, B \in \text{TR} \Rightarrow (A \cup B), (A \cap B) \in \text{TR}$
- $A$ decidable $\iff A, \bar{A} \in \text{TR}$
- $A \in \text{TR}$, not decidable $\Rightarrow \bar{A} \notin \text{TR}$
\( D = \{ c \in \{0, 1\}^* \mid c \notin L(M_c) \} \notin \text{TR} \)

- E.g., \( A_{\text{DFA}}, E_{\text{DFA}}, \) and \( E_{\text{QDFA}} \) are decidable languages

\( U = \{ \langle M, w \rangle \mid w \in L(M) \} \in \text{TR}, \) not decidable

\( \tilde{U} = \{ \langle M, w \rangle \mid w \notin L(M) \} \notin \text{TR} \)

\( H = \{ \langle M, w \rangle \mid M(w) \downarrow \} \in \text{TR}, \) not decidable

\( \tilde{H} = \{ \langle M, w \rangle \mid M(w) \uparrow \} \notin \text{TR} \)

- Chomsky hierarchy:
  finite \( \not\in \) regular \( \not\in \) context-free \( \not\in \) context-sensitive \( \not\in \) languages generated by unrestricted grammars (\( = \text{TR} \))
NE = \{ \langle M \rangle \mid M \text{ is a TM and } L(M) \neq \emptyset \} \in \text{TR}, \text{ not decidable}

REG = \{ \langle M \rangle \mid M \text{ is a TM and } L(M) \text{ is regular} \} \text{ is not decidable}

Rice’s theorem: All nontrivial semantic properties of Turing machines are undecidable

Linear bounded automaton (LBA) cannot use more work space than that already required by input

A_{LBA} \text{ is decidable, while } E_{LBA} \text{ is undecidable}

A \subseteq \Sigma^* \text{ is reducible to } B, B \subseteq \Gamma^*, \text{ denoted } A \leq_m B, \text{ if there exists a computable function } f: \Sigma^* \rightarrow \Gamma^* \text{ s.t.}

x \in A \iff f(x) \in B \quad \forall x \in \Sigma^*$
✓ $A \subseteq \{0, 1\}^*$ is TR-complete, if
   1. $A \in \text{TR}$ and
   2. $B \leq_m A$ for all $B \in \text{TR}$

✓ Language $U$ is TR-complete

✓ In time complexity analysis of Turing machines one examines the worst case with inputs on certain length

✓ Relating growth rates of functions: $O, \Theta, o, \Omega$

✓ The number of tapes does not have a significant impact on the efficiency of a Turing machine

✓ The efficiency difference of a deterministic and nondeterministic TM, on the other hand, is exponential
\[ P = \bigcup_{k \geq 0} \text{DTIME}(n^k + k) \]

\[ \text{EXPTIME} = \bigcup_{k \geq 0} \text{DTIME}(2^{n^k}) \]

- \( P \) includes languages that can be decided in time that is polynomial in the length of the input.
- E.g., finding a directed path in a graph, PATH, and deciding whether two numbers are relatively prime are examples of problems in \( P \).
- The corresponding nondeterministic composite classes are \( \text{NP} \) and \( \text{NEXPTIME} \).
- For instance, CLIQUE and SUBSET-SUM are problems in \( \text{NP} \).
- Problems belonging to \( P \) are solvable in practice.
✓ $P \subseteq \text{NP}$. In addition, $\text{NP}$ contains problems for which no polynomial time algorithm is known.

✓ $A \subseteq \Sigma^*$ is *polynomial time reducible* to $B$, $B \subseteq \Gamma^*$, denoted $A \leq_m^p B$, if there exists a polynomial time computable function $f: \Sigma^* \rightarrow \Gamma^*$ s.t.

$$x \in A \iff f(x) \in B \quad \forall x \in \Sigma^*$$

✓ $A \subseteq \{0, 1\}^*$ is *NP-complete*, if

1. $A \in \text{NP}$ and
2. $B \leq_m^p A$ for all $B \in \text{NP}$

✓ All problems in $\text{NP}$ are polynomial time reducible to a $\text{NP}$-complete problem.

✓ If any $\text{NP}$-complete problem is in $P$, then $P = \text{NP}$
Showing that $A \in \text{NP}$ is \text{NP}-complete:

1. Select a similar problem $B$ that is known to be \text{NP}-complete
2. Give a polynomial time reduction $f: B \leq_m^p A$; by Theorem 7.36 also $A$ is \text{NP}-complete

NP-complete problems: \text{SAT, CSAT, 3SAT, VC, IS, CLIQUE, Hamiltonian path, TSP, Subset-sum, and minSC}

\[
\begin{align*}
\text{PSPACE} & = \bigcup_{k \geq 0} \text{DSPACE}(n^k) \\
\text{NPSPACE} & = \bigcup_{k \geq 0} \text{NSPACE}(n^k) \\
\text{P} & \subseteq \text{NP} \subseteq \left\{ \begin{array}{l} \text{PSPACE} = \text{NPSPACE} \end{array} \right\} \subseteq \text{EXPTIME} \subseteq \\
\text{NEXPTIME} & \subseteq \left\{ \begin{array}{l} \text{EXPSPACE} = \text{NEXPSPACE} \end{array} \right\}
\end{align*}
\]
✓ TQBF is PSPACE-complete
✓ So are the "asymptotic" versions of chess and GO

\[ L = \text{DSPACE}(\log n) \]
\[ NL = \text{NSPACE}(\log n) \]

✓ PATH ∈ NL is NL-complete
✓ L =?= NL
✓ NL ⊆ P
✓ One can try to approximate an intractable problem can efficiently
✓ Vertex cover has an efficient 2-approximation algorithm
✓ Minimum set cover has an efficient \((\ln m + 1)\)-approximation algorithm
Keep in mind that programming video games also requires understanding computationally demanding problems …
• Decidable languages:

\[ A_{\text{DFA}} = \{ \langle B, w \rangle \mid B \text{ is a DFA that accepts string } w \} \]
\[ E_{\text{DFA}} = \{ \langle A \rangle \mid A \text{ is a DFA and } L(A) = \emptyset \} \]
\[ EQ_{\text{DFA}} = \{ \langle A, B \rangle \mid A \text{ and } B \text{ are DFAs and } L(A) = L(B) \} \]

• Undecidable languages: \( EQ_{\text{CFG}}, U, HALT_{\text{TM}} \)

• Turing-recognizable:

\[ U = \{ \langle M, w \rangle \mid w \in L(M) \} \]
\[ HALT_{\text{TM}} = \{ \langle M, w \rangle \mid M \text{ is a TM and halts on input } w \} \]

• Not Turing-recognizable:

\[ D = \{ b \in \{ 0, 1 \}^* \mid b \notin L(M_b) \} \]
\[ \tilde{U} = \{ \langle M, w \rangle \mid w \notin L(M) \} \]
\[ \hat{H} = \{ \langle M, w \rangle \mid M \text{ is a TM and does not halt on input } w \} \]