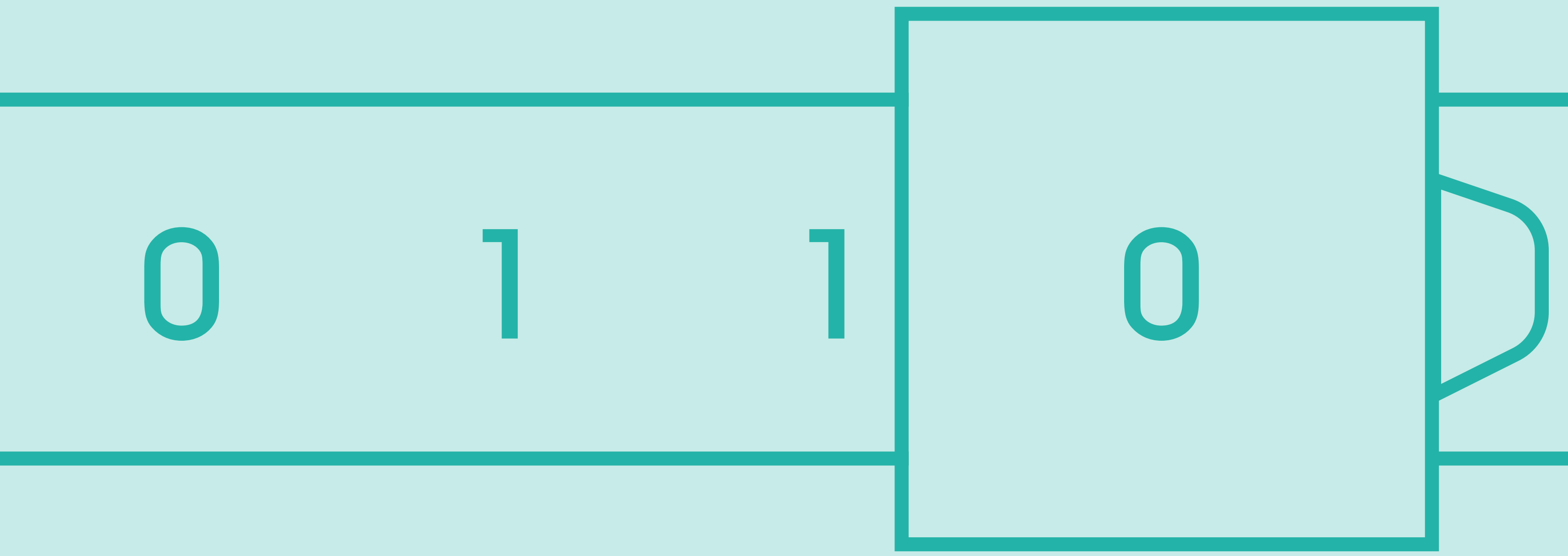
Computational Complexity and Computability



The cover depicts a Turing machine. Turing machine is a simple but powerful computation model used to study the computability and complexity of problems. It is considered one of the foundational models of theoretical computer science.

Contents

I Automata and Languages

Chapter 1 Regular Language

1.1 Alphabet and Strings

Σ denotes a finite alphabet of symbols. Σ^* denotes a set of all strings, including the empty string ε, consist of elements of Σ.

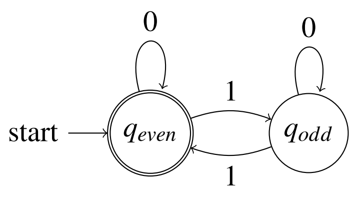
For string $x \in \Sigma^*$, $|x|$ is the length of x. A language is a subset of Σ^*.

1.2 Regular Language

See 240 notes.

Example:
$mathrmEven = \{w \in \{0,1\}^* \mid w \text{ contains an even number of 1's}\}$. It can be expressed using the regular expression $0^* + (0^*10^*10^*)^*$.

It also has a 2-state deterministic finite automaton that decides the language Even.



More formally,

Definition 1.2.1 — Deterministic Finite State Automata. A deterministic finite state automaton (DFA) is a quintuple (5-tuple)

$$M = (Q, \Sigma, \delta, s, F)$$

where

- Q is a finite set of states
- Σ is a finite set called the alphabet
- $\delta : Q \times \Sigma \to Q$, the transition function
- $s \in Q$, the starting state

- $F \subseteq Q$, the set of accepting states

Chapter 2 Context-Free Language

Chapter 3 Turing Machine

3.1 Turing Machine

While finite state automata and pushdown automata are all valid models of computation, they are too restricted as models of general purpose computers. Our goal is to have a model so that we can define computation as abstractly and general as possible.

How do we define algorithms rigorously? What does it mean for an algorithm to run in polynomial time. How do we argue that efficient algorithms do not exist.

Turing machine is a model of computation first proposed by Alan Turing in 1936. It is much more powerful than previous models that we have looked at. It is sufficiently general and can model what a human can do. A Turing machine can be think of a finite automata with unlimited and unrestricted memory.

In a Turing machine, we have an **one-way infinite tape** divided into "cells", each holding one symbol, including the blank symbol $\sqcup$. It also has a **read-write** head positioned in one square at a time that can move to the left or right. The control of the Turing is in of a fixed number of states. (The blank symbol $\sqcup$ is sometimes denoted by $\not{b}$ or $\square$)

Initially, the input tape contains a finite number of symbols starting at the left-most cell with the remaining cells blank. The head starts at the left-most input symbol. Current state and symbol determine the next state, symbol written, and the movement of the head (either one square left or one square right).

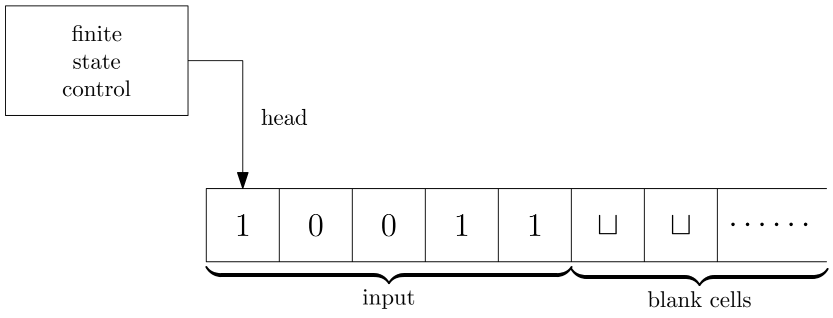


Figure 3.1: A Turing machine

Definition 3.1.1 — Turing Machine. A Turing machine is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ where Q, Σ, Γ are all finite sets, and

- Q is the set of states
- Σ is the input alphabet not containing the blank symbol $\sqcup$
- Γ is the tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subset \Gamma$
- $\delta : \ (Q - \{q_{accept}, q_{reject}\}) \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the transition function
- $q_0 \in Q$ is the start state
- $q_{accept} \in Q$ is the accept state
- $q_{reject} \in Q$ is the reject state, where $q_{accept} \neq q_{reject}$

$\delta(q,a) = (q', a', x)$ where $x \in \{L, R\}$ means if a machine is in state q and head positioned on a square containing a, then the machine replaces a with a', moves to state q, and moves the head left (L) or right (R) depending on the direction given by x.

A Turing machine M works as follows on input string $x \in \Sigma^*$.

1. Initially $x = x_1 x_2 \ldots x_n \in \Sigma^*$ appears on leftmost n squares of the input tape. Rest of the tape is blank;
2. Head of M starts on the leftmost square of tape;
3. Initial state is q_0
4. M moves according to the transition function δ
5. Continue until M reaches q_{accept} or q_{reject} and then M halts. Otherwise, continue on forever.

Definition 3.1.2 — Language Recognized by Turing Machine. We say a Turing machine M **accepts** a string $x \in \Sigma^*$ if M upon reading the input x eventually **halts** in state q_{accept}.

Let $L(M) = \{x \in \Sigma^* \mid M \text{ accepts } x\}$. We say $L(M)$ is the **language recognized/accepted by** M. We say $x \notin L(M)$ if M upon reading x either **halts** in q_{reject} or **loops**.

■ **Example 3.1 — Palindrome.**

$$\text{Palindrome} = \{yy^R \mid y \in \{0,1\}^*\}$$

At a high level, the **Turing machine** that decides this language scans back and forth, matching and erasing leftmost and rightmost symbols. Accept if the input is completely earased or reject if otherwise.
■

3.2 Configuration of Turing Machine

The **configuration** of a Turing machine describes the state, head position, and tape contents. It is denoted xqy where $x, y \in \Gamma^*$ and $q \in Q$. In this notation, the head position is implied to be the leftmost symbol of y.

Note that xqy and $xqy\sqcup$ are equivalent configurations, but xqy and $\sqcup xqy$ are not equivalent. Whether or not the first symbol is blank is important.

Definition 3.2.1 Given two configurations C_1 and C_2, we say configuration C_1 **yields** C_2 if C_2 follows from C_1 by one step of M (one application of δ).

■ **Example 3.2** Let $x, y \in \Gamma^*$ and $a, b \in \Gamma$. If $\delta(q,a) = (q', a', R)$, then $xqay$ yields $xa'q'y$. If $\delta(q,b) = (q', b', L)$, then $xaqby$ yields $xq'ab'y$. qby yields $q'b'y$ because the tape head cannot move left anymore.
■

Definition 3.2.2 — Computation. The **computation** of M on input $x \in \Sigma^*$ is the sequence $C_0C_1C_2\ldots$ where $C_0 = q_0x$ and each configuration follows from the previous one. We say the computation is **halting** if it eventually reaches accept or reject state. Otherwise, we say the computation is **looping** (infinite).

3.3 Decidability and Recognizability

Definition 3.3.1 — Decider. A Turing machine M is a decider if it halts on all inputs $x \in \Sigma^*$.

Definition 3.3.2 — Turing Decidable and Turing Recognizable. A language $A \in \Sigma^*$ is Turing decidable if and only if there is a **decider** M such that $L(M) = A$.

A is semidecidable/Turing recognizable if and only if there is a Turing machine M such that $L(M) = A$. In this case, M may not halt if it does not accept A (reject by looping).

3.4 Some Classes of Languages

$\mathsf{D} = \{A \subseteq \Sigma^* \mid A$ is decidable

$\mathsf{SD} = \{A \subseteq \Sigma^* \mid A$ is semidecidable

$\mathsf{P} = \{A \subseteq \Sigma^* \mid A = L(M)$ for some M that halts in polynomial time$\}$

Later, we will show that $\mathsf{P} \subsetneq \mathsf{D} \subsetneq \mathsf{SD}$.

3.5 Difference Between FSA and TM

- TM can read and write symbols. Infinite tape.
- Head can move left or right, unless the head is at left-most position.
- Special "accept" and "reject" states that stop computation immediately. The machine halts only when it reaches an accept or reject state. On the other hand, an FSA can only perform a finite amount of transitions before it halts.

3.6 Multi-tape Turing Machines

$$\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, R\}^k$$

We will show a 2-tape TM can be simulated by a single tape TM. The idea is to store tape 1 on odd cells, tape 2 on even cells, and represent the tape head using new symbols with $\cdot$ on top.

Theorem 3.6.1 Every multi-tape Turing machine is equivalent to an ordinary Turing machine.

To simulate a transition:

- scan the entire tape to find the symbols at tape heads;
- transition
- scan again and update the values at each tape and move tape heads left or right (by shifting the dots)

3.7 Non-deterministic Turing Machines

$$\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R\})$$

Instead of giving a single instruction, the transition function for TMs gives a set of instructions.

Theorem 3.7.1 Every non-deterministic Turing machine is equivalent to an ordinary Turing.

We cam view the computation of a nondeterministic as a tree.

The idea is to run a BFS on the tree to find an accepting path. DFS may not work because it may get stuck in a looping state. At each state (internal node), the branching factor is at most b. We will address each node in level k the tree with a string of length k in $\{1, 2, \ldots, b\}^k$. A BFS is then searching through $\{1, 2, \ldots, b\}^k$ in standard string order (i.e. $\varepsilon, 1, 2, \ldots, b, 11, 12, \ldots$).

We will simulate this with a 3-tape TM. The first tape stores the input. The second tape is for simulation. The third tape is the address incrementor.

```
NONDETERMINISTIC-TM
1  while true
2      run $M$ on tape 2
3      whenever we need to make a choice, consult tape 3
4      if $q_{accept}$ accept
5      increment tape 3
```

3.8 Enumerator

An enumerator E is a 2-tape Turing machine with a work tape and an output tape (printer). The output tape is readonly. More formally, an enumerator is a tuple $E = (Q, \Sigma, \Gamma, \delta, q_0, q_{print}, q_{reject})$ where

$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\} \times \Sigma_\varepsilon$$

is the transition function and Σ is the output alphabet. The purpose of the print state q_{print} is to end the printing of the current string (we can think of this as a carriage return on a typewriter). If q_{reject} is entered, the computation/printing will stop.

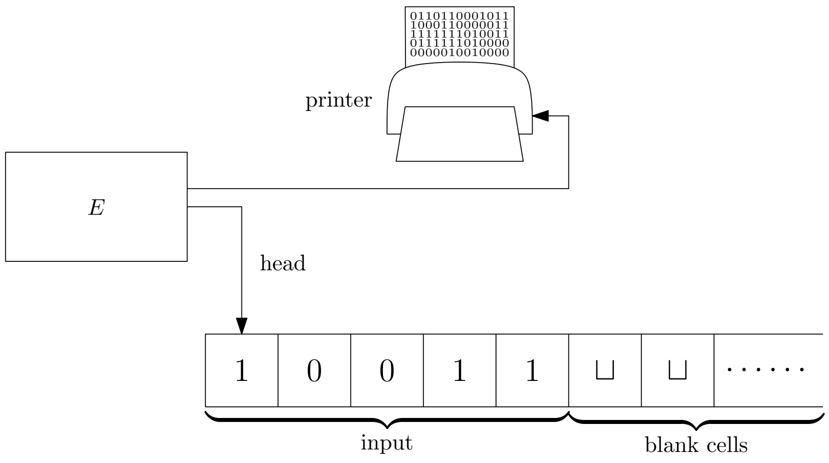


Figure 3.2: An enumerator.

Theorem 3.8.1 A language $A \subseteq \Sigma^*$ is Turing-recognizable if and only if some enumerator enumerates it.

Proof.

($\Leftarrow$): Suppose E enumerates a language A, we will show that there exists a TM M that recognizes A. We will construct a Turing machine M such that $L(M) = A$ as follows

$M =$ "On input w,
1. Run E. Every time E outputs a string, compare it with w;
2. If w ever appears in the output E, accept; Otherwise, keep running"

M accepts those strings that are printed by E. Note that M does not necessarily terminate and hence $L(M)$ is not necessarily decidable.

($\Rightarrow$): Suppose A is Turing-recognizable, then $L(M) = A$ for some Turing machine M. Let $s_1, s_2, s_3, \ldots \in \Sigma^*$ be the list of all possible strings in Σ^* in lexicographical ordering, and construct E as follows:

$E =$ "1. Repeat the following for $i = 1, 2, 3, \ldots$
2. Run M for i steps on each input $s_1, s_2, \ldots, s_i$;
3. If any computations accept, print out the corresponding s_j"

■

It might be tempting to run M on all possible strings at each iteration or to use a nondeterministic Turing machine, but both might get stuck running forever because there can be infinitely many possible strings to consider. In the case of a nondeterministic Turing machine, the computation tree can have a infinite branching factor.

Theorem 3.8.2 A language is decidable if and only if some enumerator E enumerates it in lexicographic order.

3.9 Church-Turing Thesis

II

Decidability and Computability

Chapter 4 Undecidable Language

4.1 Countability

Definition 4.1.1 — Countable Set. A set A is countable if A is empty or there is a injective function $f : A \to \mathbb{N}$, or equivalently, a surjective function $f : \mathbb{N} \to A$.

Theorem 4.1.1 $\mathbb{R}$ is not countable.

Theorem 4.1.2 $2^{\mathbb{N}} = \mathcal{P}(\mathbb{N})$ is not countable.

Corollary 4.1.3 There exists some $A \subseteq \Sigma^*$ such that A is not Turing-recognizable.

Proof. We will prove this by showing that the set of all languages in Σ^* is not countable, but the set of Turing-recognizable languages is countable.

We first show that Σ^* is countable. To show this, we simply list all strings in Σ^* in lexicographic order.

Observe that $\mathsf{TR} = \{A \subseteq \Sigma^* \mid A = L(M)\}$ is also countable because the number of Turing machines is countable, each of which can be encoded as a finite length string.

But the set of all languages $\{A \subseteq \Sigma^*\} = 2^{\Sigma^*}$. By Cantor's theorem, since Σ^* is countably infinite, the power set of Σ^* is uncountable.

Hence, there exists some language $A \subseteq \Sigma^*$ that is not Turing-recognizable. ■

Let $\langle M \rangle$ denote the encoding of a Turing machine M. For $n \in \mathbb{N}$, let $\langle n \rangle \in \{0,1\}^*$ be the binary encoding of n. For a sequence of numbers $n_1, n_2, \ldots, n_k$, $\langle n_1, n_2, \ldots, n_k \rangle \in \{0,1,\#\}^*$ be $\langle n_1 \rangle \# \langle n_2 \rangle \# \cdots \# \langle n_k \rangle$.

Then, any Turing machine can be determined by a finite sequence of numbers specifying $|Q|$, $|\Sigma|$, $|\Gamma|$, identity of $q_0, q_{accept}, q_{reject}$, and a sequence of numbers specifying δ.

$$\langle M \rangle = \text{the string that determines } M$$

Furthermore, we can denote a Turing machine T with input w as $\langle M, w \rangle$.

Theorem 4.1.4 Let
$$\mathsf{DIAG} = \{\langle M\rangle \mid M \text{ is a TM and } \langle M\rangle \notin L(M)\}$$
DIAG is not Turing recognizable.

Proof. Suppose, for contradiction, that there exists a TM M such that $T(M) = \mathsf{DIAG}$. Then, we can ask if $\langle M\rangle \in L(M)$.

Suppose that $\langle M\rangle \in L(M)$. Then

$$\langle M\rangle \in L(M) \iff \langle M\rangle \in \mathsf{DIAG} \iff \langle M\rangle \notin\in L(M)$$

which is a contradiction. ∎

This is similar to Russell's paradox.

Theorem 4.1.5 Every finite language is Turing recognizable and decidable.

Theorem 4.1.6 Every regular language is Turing recognizable and decidable.

4.2 Universal Turing Machine

Let A_{TM} be the language
$$\{\langle M, w\rangle \mid M \text{ accepts } w\}$$

Theorem 4.2.1 A_{TM} is Turing-recognizable.

Proof. We show that there exists a TM U such that $L(U) = A_{\mathsf{TM}}$. U decodes $\langle M\rangle$ into a TM M and simulates M on w. Thus, A_{TM} is Turing-recognizable. ∎

The U that we have constructed to recognize A_{TM} is called a universal Turing machine.

Theorem 4.2.2 A_{TM} is not decidable.

4.3 Decidability

Theorem 4.3.1 A language L is decidable if and only if there is an enumerator that enumerates L in standard string order.

Proof. Decidable implies standard string order enumerator.

Let M be a decider for L. We will design an standard string order.

```
E
for $w$ in standard string order
    run $M$ on $w$
    if $M$ accepts $w$
        print($w$)
```

Standard string enumerator implies decidable.

```
M($w$)
run $E$
while HAS-NEXT($E$)                       // while $E$ still has string to print
    $x$ = NEXT($E$)                                      // get next string
    if $x == w$
        accept
    if $x \geq w$
        reject
```

■

Chapter 5 Reducibility

5.1 Proving Undecidability by Reduction

Theorem 5.1.1 A_{TF} is not decidable.

Proof. By contradiction.

Suppose that A_{TM} is decidable. Then there exists some Turing machine that decides A_{TM}. We will derive a contradiction by showing that if A_{TM} is decidable, then DIAG must also be decidable.

To decide $\langle M \rangle \in \mathsf{DIAG}$, we feed $\langle M, \langle M \rangle \rangle$ into a Turing machine that decides A_{TM}. ■

Corollary 5.1.2 Let D be the class of decidable languages, and let TR is the class of Turing-recognizable.

$$\mathsf{D} \subsetneq \mathsf{TR}$$

Let $\overline{A} = \{w \in \Sigma^* \mid w \notin A\} = \Sigma^* - A$.

Let co-TR $= \{A \mid \overline{A} \in \mathsf{TR}\}$.

Theorem 5.1.3 If $A \subseteq \Sigma^*$, then $A \in \mathsf{D} \iff A \in \mathsf{TR} \land \overline{A} \in \mathsf{TR}$.

Proof. Suppose $A \in \mathsf{TR} \cap \text{co-}\mathsf{TR}$. Let M_1 be a TM that recognizes A. Let M_2 be a TM that recognizes $\overline{A}$.

Let M be a multi-tape Turing machine that simulates M_1 and M_2 in parallel. On input w, since $w \in A$ or $w \notin A$, either $M_1(w)$ accepts or $M_2(w)$ accepts. If M_1 accepts, then M accepts. If M_2 accepts, then M rejects. ■

5.2 Closure Properties

5.2.1 Union, Intersection, and Complement

Theorem 5.2.1 — Closure Under Union, Intersection, and Complement. If $A, B \in \mathsf{D}$, then $A \cup B \in \mathsf{D}$, $A \cap B \in \mathsf{D}$, and $\overline{A} \in \mathsf{D}$.

Theorem 5.2.2 The class of languages TR is closed under union, intersection, but NOT under complement.

An intuitive explanation for why TR is not closed under complement, then this would imply that all languages in TR would also be decidable.

Proof. Consider $A_{\mathsf{TM}} \in \mathsf{TR}$. If $\overline{A_{\mathsf{TM}}} \in \mathsf{TR}$, then $A_{\mathsf{TM}} \in \text{co-}\mathsf{TR}$, which implies that $A_{\mathsf{TM}} \in \mathsf{D}$. A contradiction. ■

Proof. Let $A, B \in \mathsf{TR}$. Then $A = L(M_1)$ and $B = L(M_2)$ for some TM M_1 and M_2.

$A \cap B = L(M_{A\cap B})$ where M_{AB} on input w runs M_1 and M_2 in parallel on two tapes and accepts if and only if M_1 and M_2 both halts and accepts.

$A \cup B = L(M_{A\cup B})$ where M_{AB} on input w runs M_1 and M_2 in parallel on two tapes and accepts if either M_1 or M_2 halts and accepts. ■

5.3 Computable Functions

So far, we have been using TM to accept and reject inputs. However, we can also use a TM to compute a function. If a function can be computed (evaluated) using a Turing machine, we say the function is computable.

Definition 5.3.1 — Computable Function. A function $f : \Sigma^* \to \Sigma^*$ is computable if there is some Turing machine M on every input $w \in \Sigma^*$, M halts with $f(w)$ on the tape.

■ **Example 5.1 — Duplicate.** Consider the function $f(w) = w \cdot w$ that duplicates a string w. ■

Note that f can take as input $\langle\langle\rangle M\rangle$ and output an encoding for some other Turing machine $\langle M'\rangle$.

■ **Example 5.2 — Example of Uncomputable Function - Halting Problem.** The function

$$h(\langle\langle M\rangle, w\rangle) = \begin{cases} 1 & \text{if } M \text{ halts on input } w \\ 0 & \text{if } M \text{ does not halt on input } w \end{cases}$$

solves the Halting problem and hence is not computable. ■

Theorem 5.3.1 — Church-Turing Thesis. Any function that can be computed by an "algorithm", then the function is computable.

It can be shown by the diagonal argument that the set of all functions is uncountable, and hence not all functions are computable because the set of computable functions is countable.

5.4 Mapping Reducible

Definition 5.4.1 — Mapping Reducible. A language A is mapping reducible to B, denoted $A \leq_m B$ if there is a computable function $f : \Sigma^* \to \Sigma^*$ where for every w,

$$w \in A \iff f(w) \in B$$

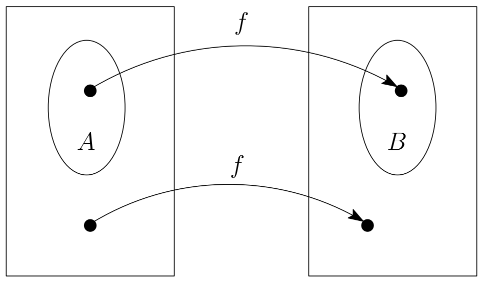


Figure 5.1: Function f reducing A to B.

Observation: If $A \leq_m B$, then $\overline{A} \leq_m \overline{B}$. If $A \leq_m B$ and $B \leq_m C$, then $A \leq_m C$ (this follows from function composition).

R Later, we will see other types of reductions. For example, $\leq_p$ for polytime reduction.

Theorem 5.4.1 If $A \leq_m B$ and B is decidable, then A is decidable. If B is Turing-recognizable, then A is also Turing-recognizable.

Proof. Let M be a decider for B. Let f be the computable mapping reduction from A to B, and let N be the Turing machine that computes this function.

Then A is decided by a Turing machine which on input w, runs N to obtain $f(w)$ and then runs M on $f(w)$. ■

Corollary 5.4.2 If $A \leq_m B$ and A is not decidable, then B is not decidable.

Application: Consider DIAG $= \{\langle M \rangle \mid \langle M \rangle \notin \mathcal{L}(M)\}$. Then, DIAG $\leq_m \overline{A_{\text{TM}}}$ via the reduction $f : \Sigma^* \to \Sigma^*$ defined as

$$f(\langle M \rangle) = \langle M, \langle M \rangle \rangle$$

and if the input w to f is not an encoding of a TM, then

$$f(w) = \langle M_{accept}, \varepsilon \rangle$$

Since DIAG is not decidable, $\overline{A_{\text{TM}}}$ is not decidable.

5.5 Applications of Reduction

5.5.1 Halting Problem

Consider the language

$$\text{HALT}_{\mathsf{TM}} = \{\langle M, w\rangle \mid M \text{ halts on } w\}$$

We have $\text{HALT}_{\mathsf{TM}} \leq_m A_{\mathsf{TM}}$ and $A_{\mathsf{TM}} \leq_m \text{HALT}_{\mathsf{TM}}$.

Proof.

($\text{HALT}_{\mathsf{TM}} \leq_m A_{\mathsf{TM}}$): If x is any string of the form $\langle M, w\rangle$, then $f(x) = \langle M', w\rangle$ where M' accepts if and only if M halts on input w. M' will simulate M running on w and accepts when M halts. If x is not of the correct form, $f(x) = \langle M_{loop}, \varepsilon\rangle$.

($A_{\mathsf{TM}} \leq_m \text{HALT}_{\mathsf{TM}}$): If x is a string of the form $\langle M, w\rangle$, then $f(x) = \langle M', w\rangle$ where M' simulates M on w, and halts if and only if M accepts. If x is not of the correct form, then $f(x) = \langle M_{loop}, \varepsilon\rangle$. ■

Corollary 5.5.1 $\text{HALT}_{\mathsf{TM}}$ is mapping equivalent (equivalent under mapping reduction) to A_{TM}.

$$\text{HALT}_{\mathsf{TM}} \equiv_m A_{\mathsf{TM}}$$

Hence,

$$\begin{aligned} \text{HALT}_{\mathsf{TM}} &\in \mathsf{TR} \\ &\notin \mathsf{co\text{-}TR} \\ &\notin \mathsf{D} \end{aligned}$$

5.5.2 EQ

Consider the language

$$\text{EQ}_{\mathsf{TM}} = \{\langle M_1, M_2\rangle \mid M_1 \text{ and } M_2 \text{ are Turing machines and } \mathcal{L}(M_1) = \mathcal{L}(M_2)\}$$

We show that

$$A_{\mathsf{TM}} \leq_m \text{EQ} \qquad \text{and} \qquad A_{\mathsf{TM}} \leq_m \overline{\text{EQ}}$$

Proof. Given $\langle M, w\rangle$, if M accepts w, we want to output M_1, M_2 such that $\mathcal{L}(M_1) = \mathcal{L}(M_2)$.

Let M_1 be the TM which accepts all inputs. $\mathcal{L}(M_1) = \Sigma^*$.

Let M_2 be the TM which on any input, runs M on w and accepts if and only if M accepts.

So, $f(\langle M, w\rangle) = \langle M_1, M_2\rangle$.

$$\mathcal{L}(M_2) = \begin{cases} \Sigma^* & \text{if } M \text{ accepts } w \\ \emptyset & \text{otherwise} \end{cases}$$

Reduction from A_{TM} to $\overline{\mathrm{EQ}}$. ■

Let A be a set of TM descriptions $\{\langle M_1\rangle, \langle M_2\rangle, \ldots\}$. Show that if $A \in \mathsf{TR}$, then there is some set of TM descriptions $B \in \mathsf{D}$ such that the set of associated languages of A and B are the same. Formally,

$$\{\mathcal{L}(M) \mid \langle M\rangle \in A\} = \{\mathcal{L}(M) \mid \langle M\rangle \in B\}$$

Let E_B be an enumerator for B.

```
1  $prev\text{-}length = 0$
2  **for** $\langle M_1\rangle, \langle M_2\rangle, \ldots$ from $E_A$
3      $M' = \textsc{Pad}(\langle M_1\rangle, prev\text{-}length + 1)$
4      $prev\text{-}length = |\langle M'\rangle|$
5      **print** $\langle M'\rangle$
```

$\mathcal{L}(E_B)$ is the decidable language that is the same as the language recognized by Turing machines in A.

Let $A \in \mathsf{TR}$ be a recognizable set of decider descriptions. Show that there is some decidable set B such that B is not the language of any TM in A.

Let $A = \{\langle D_1\rangle, \langle D_2\rangle, \ldots\}$. We want to show that there exists some B such that $\forall i. B \notin \mathcal{L}(D_1)$. Let $\langle D_1\rangle, \langle D_2\rangle, \ldots$ be an enumeration of A. Also, let $w_1, w_2, \ldots$ be an enumeration of Σ^* in standard string order.

Let $B = \{w_i \mid w_i \notin \mathcal{L}(D_i)\}$. We first show that B is decidable. We can construct a decider for B as follows.

```
$D_B(x)$
1  determine $i$ such that $w_i = x$
2  run the enumerator for $A$ to get $\langle D_i\rangle$
3  simulate $D_i$ on $w_i = x$
4  **if** $D_i$ accepts
5      **reject**
6  **else**
7      **accept**
```

We then show that $B \notin \mathcal{L}(D_i)$ for some $\langle D_i\rangle \in A$. If $w_i \in B$, then $w_i \notin \mathcal{L}(D_i)$, and $B \neq \mathcal{L}(D_i)$. If $w_i \notin B$, then $w_i \in \mathcal{L}(D_i)$ and $B \neq \mathcal{L}(D_i)$.

A language A is recognizable if and only if there is a decider V such that for all $x \in \Sigma^*$,

$$x \in A \iff \exists y. V \text{ accepts } (x,y)$$

Think V as a verifier and y as a "certificate" that $x \in A$.

L is recognizable by TM M.

```
V(x,y)
1  simulate $M$ on $x$ for $|y|$ steps
2  **if** $M$ accepts
3      **accept**
4  **elseif** $M$ rejects or has not completed
5      **reject**
```

If $x \in A$, then M accepts within a finite number of steps k. Then, $V(x, y = 1^k)$.

Now suppose that there exists a decider V such that for all $x \in A$, there exists $y \in \Sigma^*$ such that V accepts (x,y).

We need to find a recognizer for A.

```
M(x)
1  **for** $y \in \Sigma^*$
2      run $V(x,y)$
3      **if** $V$ accepts
4          **accept**
5      **else**
6          **continue**
```

If $x \in A$, there exists y such that $V(x,y)$ accepts, then y eventually show up in the standard string order of Σ^* and when y is run on V, V accepts. Otherwise, M loops.

5.6 Rice's Theorem

Using reduction, we can prove a really strong theorem about the decidability of certain languages, which can cover many of the languages that we have discussed so far.

Theorem 5.6.1 — Rice's Theorem. Suppose P is a language of Turing machine descriptions such that

1. P is nontrivial: it contains some but not all Turing machine descriptions;
2. P is a ***property of the Turing machine's languages*** (instead of a property of the Turing

machine itself): whenever $\mathcal{L}(M_1) = \mathcal{L}(M_2)$, then $\langle M_1 \rangle \in P \iff \langle M_2 \rangle \in P$.

Then, P is decidable.

Here are some example languages that are covered by Rice's theorem.

■ Example 5.3

- $\text{EMPTY}_{\text{TM}} = \{\langle M \rangle \mid \mathcal{L}(M) = \emptyset\}$
- $\text{FINITE}_{\text{TM}} = \{\langle M \rangle \mid \mathcal{L}(M) \text{ is finite}\}$
- $\text{DECIDABLE}_{\text{TM}} = \{\langle M \rangle \mid \mathcal{L}(M) \text{ is decidable}\}$

Rice's theorem claims that all of these languages are undecidable.

Note that Rice's theorem does not apply to $\{\langle M \rangle \mid M \text{ is a decider}\}$ because the property being satisfied by the TM descriptions in this language is a property of the Turing machine itself, not its language. Criterion 2 of Rice's theorem is not satisfied. ■

Now, let's prove Rice's theorem using reduction.

Proof. Let T_0 be the TM which always rejects. Without loss of generality, suppose that $\langle T_0 \rangle \notin P$. Otherwise, we can consider $\overline{P}$ which also satisfies the criteria for Rice's theorem.

Since P is nontrivial, there exists TM T_1 such that $\langle T_1 \rangle \in P$. We will show that $\text{A}_{\text{TM}} \leq_m P$. To this end, we map $\langle M, w \rangle$ to M_w where

```
$M_w$ = "on input $x$
    1. Simulate $M$ on $w$
    2. If $M$ rejects $w$, reject
    3. If $M$ accepts $w$, then run $T_1$ on $x$
       $M_w$ accepts/rejects/loops according to what $T_1$ does on input $x$"
```

$\mathcal{L}(M_w)$ is either $\emptyset = \mathcal{L}(T_0)$ or the langauge of T_1, $\mathcal{L}(T_1)$, depending on whether w is accepted by M. We use M_w's ability to decide between T_0 and T_1 to decide A_{TM}.

Claim. $\langle M, w \rangle \in \text{A}_{\text{TM}} \iff \langle M_w \rangle \in P$.

Let R be the decider for P.

```
$S$ = "on input $\langle M, w \rangle$
    1. Run $R$ on $\langle M, w \rangle$ to check if $\langle M_w \rangle \in P$.
    2. If yes, accept; otherwise, reject "
```

If M accepts w, then $\mathcal{L}(M_w) = \mathcal{L}(T_1)$, so $\langle M_w \rangle \in P$.
If M rejects w, then $\mathcal{L}(M_w) = \emptyset = \mathcal{L}(T_0)$, so $\langle M_w \rangle \notin P$.
If M loops on w, M_w loops on all inputs so $\mathcal{L}(M_w) = \emptyset = \mathcal{L}(T_0)$. ■

Chapter 6 Computation History Method

6.1 Configuration and Computation History

Recall that the ***configuration*** of a Turing machine can be expressed as a triple (q, p, t) where q is the current state, p is the head position, and t is the tape content. Equivalently, this can be written as an encoding t_1qt_2 where $t = t_1t_2$ and the head position is on the first symbol of t_2.

■ **Example 6.1 — Configuration of a Turing Machine.** For example, the configuration

$$(q_3, 6, aaaaaabbbbb)$$

means the Turing machine is in state q_3, with the head pointing at the 6th symbol, and the current tape content being $aaaaaabbbbb$. This can be expressed in a more compact form as

$$aaaaaq_3abbbbb$$

■

A computation history of a Turing machine is a sequence of configurations of the Turing machine until it reaches an accepting state.

Definition 6.1.1 — Computation History. An ***accepting computation history***, or ***computation history***, for a Turing machine M on input w is a sequence of configurations $C_1, C_2, \dots, C_{accept}$ that M enters until it accepts. Each configuration in this sequence is yielded from the configuration immediately before it.

We encode a computation history as a sequence of configurations separated by the pound sign #.

$$C_1\#C_2\#\dots\#C_{accept}$$

■ **Example 6.2** A computation history for M on $w = w_1w_2\dots w_n$ given $\delta(q_0, w_1) = \delta(q_7, a, R)$ and $\delta(q_7, w_2) = (q_8, c, R)$ is as follows

$$\underbrace{q_0w_1w_1\dots w_n}_{C_1}\#\underbrace{aq_7w_2\dots w_n}_{C_2}\#\underbrace{acq_8w_3\dots w_n}_{C_3}\#\cdots\#\underbrace{\dots q_{accept}\dots}_{C_{accept}}$$

■

6.2 Decidability of Problems Concerning Linearly Bounded Automata

To understand how to prove undecidability using the computation history method, we first examine a type of machine known as linearly bounded automata (LBA).

Definition 6.2.1 — Linearly Bounded Automaton. A ***linearly bounded automaton (LBA)*** is a single-tape Turing machine that cannot move its head off the input portion of the tape. In other words, the size of the tape is restricted to the size of the input.

6.2.1 A_{LBA} is Decidable

Let

$$A_{\mathsf{LBA}} = \{\langle B, w\rangle \mid \text{LBA } B \text{ accepts } w\}$$

Although it may come as surprising at a first glance, A_{LBA} is, in fact, decidable. This is because of the fac that, for an LBA of input size n, the number of configurations of the LBA is finite, namely equal to $|Q| \times n \times |\Gamma|^n$.

Claim. For inputs of length n, an LBA can only have $|Q| \times n \times |\Gamma|^n$ configurations.

Theorem 6.2.1 A_{LBA} is decidable.

Proof. If B on w runs for too long (more than $|Q| \times n \times |\Gamma|^n$), then by the pigeonhole principle, B must be looping and will never halts or accept. More formally, let us construct a decider for A_{LBA}.

```
$D_{A_{LBA}}$ = “on input $\langle B, w\rangle$
        1. Let $n = |w|$
        2. Run $B$ on $w$ for $|Q| \times n \times |\Gamma|^n$ steps
        3. If $B$ has accepted, accept
        4. If $B$ has rejected or it is still running, reject″
```

■

6.2.2 E_{LBA} is Undecidable

Now, let us consider the language

$$E_{\mathsf{LBA}} = \{\langle B\rangle \mid B \text{ is an LBA and } \mathcal{L}(B) = \emptyset\}$$

The next natural question is whether or not E_{LBA} is also decidable. We will show using reduction that A_{TM} can be reduced to E_{LBA} and thus E_{LBA} is undecidable.

Theorem 6.2.2 E_{LBA} is undecidable.

Proof. Assume that E_{LBA} is decidable. Then, there exists a Turing machine R that decides E_{LBA}. We construct a Turing machine S deciding A_{TM}.

$S =$ "on input $\langle M, w\rangle$

1. Construct LBA $B_{\langle M,w\rangle}$ that tests whether its input x is an accepting computation history for M on w, and accepts only if x is an accepting computation history
2. Use R to determine whether $\mathcal{L}(B_{\langle M,w\rangle}) = \emptyset$
3. Accept if no. Reject if yes."

More specifically, we define $B_{\langle M,w\rangle}$ as follows.

$B_{\langle M,w\rangle} =$ "on input x

1. Check if x begins $C_1\#$ where C_1 is the start configuration of M on w
2. Check if each C_{i+1} yields from C_i
3. Check if the final configuration is accepting
4. Accept if all checks pass. Otherwise, reject.

Clearly, S accepts if and only if M on w accepts, and it halts on all inputs. Hence, S decides A_{TM}, but since A_{TM} is undecidable, this is a contradiction. ■

6.3 Post Correspondence Problem

A domino has the form $\left[\frac{t}{b}\right]$ where $t, b \in \Sigma^*$.

Given a collection of dominos $P = \left\{\left[\frac{t_1}{b_1}\right], \ldots, \left[\frac{t_k}{b_k}\right]\right\}$, we say there is a ***match*** if we can make a list using the dominos from P (possibly with repetitions) such that the whole string on the top row is equal to the whole string on the bottom row. The ***Post Correspondence Problem (PCP)*** is to determine whether a collection of dominos P has a match. That is, we want to determine if there is a sequence $i_1, i_2, \ldots, i_m \in \{1, 2, \ldots, k\}$ such that $t_{i_1} \cdot t_{i_2} \cdot \ldots \cdot t_{i_m} = b_{i_1} \cdot b_{i_2} \cdot \ldots \cdot b_{i_m}$.

Figure 6.1 shows a set of dominos and a match.

$$P = \left\{\left[\frac{ab}{aba}\right], \left[\frac{aa}{aba}\right], \left[\frac{ba}{aa}\right], \left[\frac{abab}{b}\right]\right\}$$

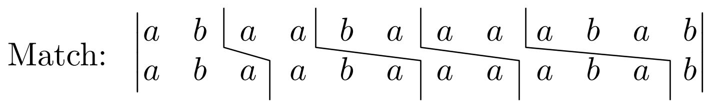


Figure 6.1: A set of dominos P and the corresponding match.

Theorem 6.3.1 PCP is undecidable.

The main idea of the proof: reduction $A_{\mathsf{TM}} \leq_m$ PCP using computation histories. We will in fact show two reductions:

$$A_{\mathsf{TM}} \leq_m \text{MPCP} \leq_m \text{PCP}$$

To make the proof simpler, we first restrict ourselves to the Modified Post Correspondence Problem (MPCP). MPCP adds the additional requirement that the match starts with the first domino $\left[\frac{t_1}{b_1}\right]$.

Proof *(*$A_{\mathsf{TM}} \leq_m$ MPCP*)*. Let $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{acc}, q_{rej})$ such that $w \in \Sigma^*$.

We are given $\langle M, w \rangle$ as input and must construct a set of dominos P such that P has a match if and only if M accepts on input w. We assume, without loss of generality, M never attempts to move head left when its head is in the leftmost position.

Given $\langle M, w \rangle$, we construct P. To do that, we introduce the following types of dominos. We put each type of dominos in the order as they are introduced below.

Step 1: $\left[\frac{\#}{\#q_0 w_1 w_2 \ldots w_n \#}\right]$ or $\left[\frac{\#}{\#q_0 \sqcup}\right]$ if $w = \varepsilon$.

This domino is used as the first domino, which simulates the initial configuration of M.

Step 2: For all $a, a' \in \Gamma$ and $q, q' \in Q$ where $q \neq q_{rej}$, if $\delta(q, a) = (q', a', R)$, we put the domino

$$\left[\frac{qa}{a'q'}\right]$$

This domino simulates the transitions where the head moves to the right.

3: For all $a, a', b \in \Gamma$ and $q, q' \in Q$ where $q \neq q_{rej}$, if $\delta(q, a) = (q', a', L)$, we put the domino

$$\left[\frac{bqa}{q'ba'}\right]$$

This domino simulates the left movement of the head.

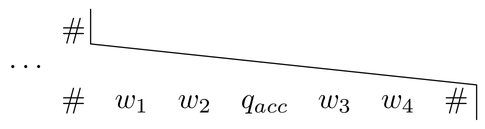
Step 4: For every $a \in \Gamma$, we put $\left[\frac{a}{a}\right]$.

This domino fills in the remaining contents of the tape after a transition.

Step 5: $\left[\frac{\#}{\#}\right]$ and $\left[\frac{\#}{\sqcup \#}\right]$

This domino places the delimiter marking the end of a configuration.

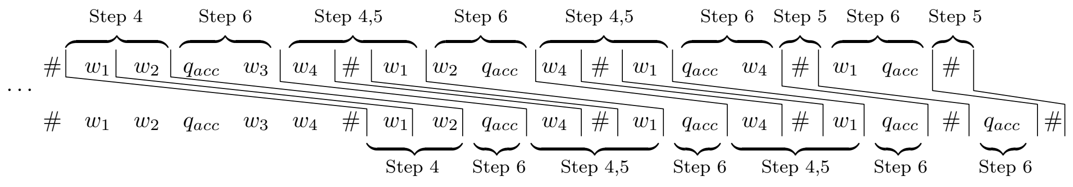
We repeat Step 2-5 to simulate the configurations of M until it reaches the accepting configuration. After all these are finished, we should have a sequence of dominos of the form



Step 6: For every $a \in \Gamma$, put

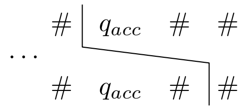
$$\left[\frac{aq_{acc}}{q_{acc}}\right] \quad \text{and} \quad \left[\frac{q_{acc}a}{q_{acc}}\right]$$

This step eats away the symbols adjacent to q_{acc} at the bottom, one at a time, as illustrated below.



Finally,

Step 7: Put the domino $\left[\frac{q_{acc}\#\#}{\#}\right]$. This step finishes off the alignment of the top and bottom row.



This concludes the construction of P.

Claim. P is an accepting instance of MPCP $\iff$ M accepts w.

Next, we provide the construction that shows MPCP $\leq_m$ PCP. It trivially holds that any accepting instance of MPCP is also a PCP. To see why, consider the match $\left[\frac{\#}{\#}\right]$.

Now, we show how to convert an instance of MPCP to PCP such that the converted instance still simulates M on w. Let $P = \left\{ \left[\frac{t_1}{b_1}\right], \dots, \left[\frac{t_k}{b_k}\right] \right\}$ be an input to MPCP. Further, let $\{*, \diamond\}$ be new symbols that are not in any t_i or b_i.

For any string $u = u_1u_2\dots u_n$, let

$$\circledast u = *u_1 * u_2 \dots * u_n$$
$$u\circledast = u_1 * u_2 * \dots u_n *$$
$$\circledast u \circledast = *u_1 * u_2 * \dots * u_n *$$

Consider the domino set P'

$$P' = \left\{ \left[\frac{\circledast t_1}{\circledast b_1 \circledast}\right] \right\} \cup \left\{ \left[\frac{\circledast t_i}{b_i \circledast}\right] \mid 1 \leq i \leq k \right\} \cup \left\{ \left[\frac{*\diamond}{\diamond}\right] \right\}$$

Then, there is a match in P starting with $\left[\frac{t_1}{b_1}\right]$ if and only if there is a match in P'. The only domino in P' that could start a match is $\left[\frac{\circledast t_1}{\circledast b_1 \circledast}\right]$ because it is the only one where both the top string and bottom string starts with $*$. ■

Chapter 7 Information

7.1 Quantifying Information

Can we quantify how much information is in a string.

Consider
$$x = 0101010101010101$$
and
$$y = 110001010110010$$
Intuitively, y appears to have more information than x, which is simply repeated 01.

We would like to have a formal definition that captures the intuitive definition of "information".

Idea: The more we can compress a string, the less information it contains.

Thesis: The amount of information in a string is the shortest way to describe the string.

Consider $\langle M, w\rangle$, where $M(w)$ halts with only x on its tape. We need to specify an encoding of $\langle M, w\rangle$ (what is the alphabet, and when does $\langle M\rangle$ end and w begin).

There are different encodings. For the purpose of our discussion, we restrict outselves to binary strings. A specific encoding of $\langle M, w\rangle$, let M be a TM with input alphabet $\{0,1\}$ and $w = w_1w_2\dots w_n \in \{0,1\}^*$.

If $\langle M\rangle = z_1z_2\dots z_k \in \{0,1\}^*$ is a binary encoding of M, let

$$\langle M, w\rangle = 0z_10z_2\dots 0z_k1w_1w_2\dots w_n$$

Then, $|\langle M, w\rangle| = 2|\langle M\rangle| + |w| + 1$. We use 0 to separate each character in the Turing machine description, and we use 1 to separate $\langle M\rangle$ and w.

Alternatively, we can use 11 to represent 1 in the description of $\langle M\rangle$, 00 to represent 0; additionally, use 01 as the delimiter separating $\langle M\rangle$ and w. This is the encoding presented in Sipser.

Definition 7.1.1 — Shortest Description. If $x \in \{0,1\}^*$, then the ***shortest description*** x, denoted $d(x)$ is the lexicographically minimal string $\langle M, w\rangle$ such that $M(w)$ halts with only x on its tape.

Definition 7.1.2 — Kolmogorov Complexity. The ***Kolmogorov complexity*** (or descriptive complexity, Kolmogorov-Chaitin complexity) of x, denoted $K(x)$ is $|d(x)|$.

Theorem 7.1.1 There is a constant c such that for all $x \in \{0,1\}^*$,

$$K(x) \leq |x| + c$$

The amount of information in x is not much more than $|x|$. The Kolmogorov complexity of a string is at most a fixed constant more than its length.

Proof. Define M

M on input w, halts. On any string x, $M(x)$ halts with x on its tape.

$$K(x) \leq |\langle M, x\rangle| \leq 2|\langle M\rangle| + |x| + 1 \leq |x| + c$$

So we can let c be the length of the trivial Turing machine that computes the identity function. ■

Theorem 7.1.2 There exists a constant c such that for all $x \in \{0,1\}^*$,

$$K(xx) \leq K(x) + c$$

This says if a string is repetitive, such string has no more information than x.

Proof. Consider the Turing machine M defined as follows

```
M = “on input ⟨N, w⟩, where N is a TM and w is a string
    1. Run N on w until it halts and produces an output string s
    2. Output the string ss ”
```

Let $\langle M, w\rangle$ be the shortest description of x, then $\langle N, \langle M, w\rangle\rangle$ is a description of xx.

$$K(xx) \leq |\langle N, \langle M, w\rangle\rangle| \leq 2|\langle N\rangle| + K(x) + 1 \leq K(x) + c$$

So letting $c = 2|\langle N\rangle| + 1$, the theorem holds. ■

Corollary 7.1.3 There is a constant s such that for all $n \geq 2$ and $x \in \{0,1\}^*$,

$$K(x^n) \leq K(x) + c \log n$$

In particular, $K((01)^n) \in O(\log n)$.

Proof. Let

```
N = “on input ⟨n, ⟨M, w⟩⟩
    run M(w) and print x for n times”
```

If $\langle M, w\rangle$ is a shortest description of x, then,

$$\begin{aligned} K(x^n) &\leq K(\langle N, \langle n, \langle M, w\rangle\rangle\rangle) \\ &\leq 2|\langle N\rangle| + 2\lceil \log n\rceil + |\langle M, w\rangle| + 2 \\ &= K(x) + O(\log n) \end{aligned}$$

■

7.2 The Invariance Theorem

The Kolmogorov complexity of a string is independent of the model (as long as we restrict ourselves to classical computational models). The model does not matter. Turing machiens can be viewed as programming languages. If we use another programming lanuage, we will not get significantly shorter description. Intuitively, we can always write a compiler/interpreter to translate from one language to another, and the size of the compiler is constant.

Definition 7.2.1 — Interpreter. An ***interpreter*** is a semi-computable function $p : \{0,1\}^* \to \{0,1\}^*$ which takes a program as inputs and prints their outputs.

Note that an interpreter is semi-computable, meaning it may not halt on all inputs.

Definition 7.2.2 The shortest description of x under p, denoted $d_p(x)$, is the lexicographically shortest string s for which $p(s) = x$. We define the Kolmogorov complexity under p, as $K_p(x) = |d_p(x)|$.

■ **Example 7.1 — Complexity under the Python interpreter.** For example, $d_{\text{Python}}(x)$ is the shortest binary string encoding of a Python program that outputs x. ■

We have the following theorem.

Theorem 7.2.1 — Invariance Theorem. For every interpreter p, there is a constant c such that for all $x \in \{0,1\}^*$,

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

This theorem implies that we only change the Kolmogorov complexity of x by a constant c by using a different programming language.

Proof. Define the Turing machine M such that

$$\begin{aligned} M = \text{“on input } w \\ \text{output } p(x)\text{”} \end{aligned}$$

Then, $\langle M, d_p(x)\rangle$ is a description of x. So,

$$\begin{aligned} K(x) &\leq |\langle M, d_p(x)\rangle| \\ &\leq 2|\langle M\rangle| + K_p(x) + 1 \\ &\leq K_p(x) + c \end{aligned}$$

■

7.3 Incompressible Strings

Definition 7.3.1 — Incompressible Strings. Let x be a string. We say that x is c-compressible if

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

If x is not c-compressible, we say x is incompressibe by c. If x is incompressible by 1, we say that x is ***incompressible*** or ***Kolmogorov random***.

Theorem 7.3.1 For all n, there is an $x \in \{0,1\}^*$ such that $K(x) \geq n$. In other words, incompressible strings of every length exist.

Proof. The number of binary strings of length n is equal to 2^n. However, the number of descriptions $\langle M, w\rangle$ of length $|\langle M, w\rangle| < n$ is equal to

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

Therefore, there exists at least one n-bit string that does not have a description of length less than n. ■

Theorem 7.3.2 For all n and c,

$$\operatorname*{Prob}_{x \in \{0,1\}^n}\left[K(x) \geq n - c\right] \geq 1 - \frac{1}{2^c}$$

In other words, most strings are very incompressibe.

Proof. We assume that all strings of length n are uniformly distributed. The number of descriptions of length less than $n-c$ is

$$1+2+4+\cdots+2^{n-c-1} < 2^{n-c}.$$

Recall that there are 2^n strings of length n. So, the probably of a string being c-compressible is $2^{n-c}/2^n = 2^{-c} = 1/2^c$. It follows that the probably of a string being c-incompressibe is $1 - 1/2^c$. ■

A natural follow-up question to ask is: given a string x, how hard is it to find a short algorithm for generating the string? Let's look at the following numbers:

- 10101010011110101000110010
- 1235813213455891442333776
- 12624120720504040320362880

As we can see, it seems quite hard in general to find a succinct algorithm that generates an arbitrary string. In fact, the problem of determining whether a string is compressible is undecidable, and not even recognizable.

Definition 7.3.2 — Language of Incompressible Strings. Let

$$\text{INCOMP} = \{x \in \{0,1\}^* \mid K(x) \geq |x|\}$$

be the set of incompressibe strings.

Theorem 7.3.3 INCOMP is undecidable.

Intuition: If INCOMP were decidable, then we could design an algorithm that prints the first incompressibe string of length n. But then, such a string can be succinct described by giving the algorithm and n in binary.

R Berry's paradox: "the smallest integer that cannot be defined in less than thirteen words". If such integer exists, it is "the smallest integer that cannot be defined in less than thirteen words", but then it can be defined in twelve words.

Proof. For contradiction, assume that M is a decider for INCOMP. And let M' be

$$\begin{aligned} M' = \text{``on input } \langle n \rangle,\\ &\text{run } M \text{ on all } \langle n \rangle \text{ bit strings and print the}\\ &\text{lexicographically first string that } M \text{ accepts''} \end{aligned}$$

Let $s_n \in \{0,1\}^n$ be the output of M' on $\langle n \rangle$. Then, $\langle M \rangle$ accepts s_n, so $s_n \in$ INCOMP and $K(s_n) \geq n$.

On the other hand, $\langle M', \langle n \rangle \rangle$ is a description of s_n. Therefore,

$$\begin{aligned} K(s_n) \leq |\langle M', \langle n \rangle \rangle| &\leq 2\langle M' \rangle + \lceil \log n \rceil + 1\\ &\leq \log n + c \end{aligned}$$

for some constant c. Now, choose n large enough so that $\log n + c < n$. Then, $K(s_n) = \log n + c < n$. But then, since s_n is in INCOMP, $K(s_n) \geq n$. Contradiction. ■

Theorem 7.3.4 INCOMP is co-Turing recognizable, but not Turing recognizable.

The idea of the proof is similar to the proof that INCOMP is not decidable, except that we use dovetailing to prevent looping.

Proof. We prove that INCOMP is not Turing recognizable by proving a stronger version of the theorem: every infinite subset of INCOMP is not Turing recognizable.

Let $I \subseteq$ INCOMP be an infinite subset of INCOMP. Let x_i denote the ith string in I. For contradiction, assume that I is Turing recognizable. Then, there exists some Turing machine M that recognizes I.

We construct a Turing machine M' as follows:

```
$M'$ = “on input $n$,
    **for** $i = 1, 2, 3, \ldots$
        run $M$ for $i$ steps on $x_1, \ldots, x_i$
        **if** $M$ accepts $|x_j|$ for some $j \leq i$ and $|x_j| \geq n$
            print $x_i$ ”
```

The number of binary strings of a given length n is finite (namely, there are 2^n strings of length n). This implies that since I is infinite, there must be string of infinitely many lengths in I. Hence, for all n, there will always be some string x such that $|x| \geq n$. Consider such x printed by M'.

Since $x \in I \subseteq \text{INCOMP}$, x is incompressible and $K(x) \geq n$. On the other hand, $\langle M', \langle n \rangle \rangle$ is also a description of x. Hence,

$$K(x) \leq |\langle M', \langle n \rangle \rangle \leq 2|\langle M' \rangle| + \lceil \log n \rceil + 1 \leq \log n + c$$

for some constant c. We choose a sufficiently large n such that $\log n + c < n$. Then, $K(x) = \log n + c < n$, but then since $x \in I$, $K(x) \geq n$. This is a contradiction. Therefore, I is not recognizable. ■

III Complexity Theory

Chapter 8 Time Complexity

8.1 Complexity and Asymptotic Notation

We express the running time of an algorithm as a function of the length of the string repreenting the input and do not consider other parameters. In worst-case analysis, we consider the longest running time of all inputs of a particular length. In average-case analysis, we considerthe average of all the running times of inputs of a particular length. Formally, we define the worst-case time complexity as follows

Definition 8.1.1 — Time Complexity and Worst-Case Time Complexity. Let M be a Turing machine over input alphabet Σ. For each $x \in \Sigma^*$, let $t_M(x)$ be the number of steps required by M to halt. If M never halts on x, we define $t_M(x) = \infty$. Then, the worst case time complexity of M is the function $T_M : \mathbb{N} \to \mathbb{R} \cup \{\infty\}$ defined by

$$T_M(n) = \max\{t_M(x) \mid x \in \Sigma^* \wedge |x| = n\}$$

The asymptotic upper bounds are defined the same way as we have discussed in previous courses.

We say that a Turing machine M is a $O(t(n))$ time Turing machine if $T_M(n) \in O(t(n))$. With that, we can define the time complexity classes.

Definition 8.1.2 — TIME. Let $t : \mathbb{N} \to \mathbb{R}$ be a function. We define the time complexity class $\mathsf{TIME}(t(n))$ to be the function of all languages that are decidable by an $O(t(n))$ time Turing machine.

8.2 Models of Computation

In computability theory, many problems are invariant under all reasonable (and classical) models of computation. A language that is not decidable by a Turing machine, is not decidable by a nondeterministic Turing machine or multi-tape Turing machine either. However, in the study of complexity theory, the model of computation often does make a difference.

Theorem 8.2.1 Let $t(n)$ be a function, where $t(n) \geq m$. Then, every $t(n)$ time multi-tape Turing machine has an equivalent $O(t^2(n))$ single-tape Turing machine.

Proof. ■

8.3 The Class P

Definition 8.3.1 — P.

$$\mathsf{P} = \bigcup_k \mathsf{TIME}(n^k)$$

or equivalently, without using the class $\mathsf{TIME}(\cdot)$, we can define the class P as

$$\mathsf{P} = \{L \mid L = \mathcal{L}(M) \text{ for some polynomial time Turing machine } M\}$$

8.3.1 Some Languages in P

PATH

Definition 8.3.2

$$\text{PATH} = \{\langle G,s,t\rangle \mid G \text{ is a directed graph with a path from } s \text{ to } t\}$$

Theorem 8.3.1

$$\text{PATH} \in \mathsf{P}$$

Proof. We construct a decider M for PATH.

```
M = "on input ⟨G,s,t⟩
    1. Mark s
    2. Repeat until nothing new is marked
       For each marked node x,
          Scan G to mark all y where (x,y) ∈ E
    3. Accept if t is marked. Reject if not."
```

This is essential the breadth-first search algorithm. Clearly, Step 2 takes $O(n^4)$ steps because the outer loop takes at most n iterations, and the inner loop also takes at most n^2 steps. The step within the inner loop marks at most n^2 times. Overall, the algorithm should run in $O(n^4)$ time, so PATH $\in \mathsf{P}$. ■

To show polynomial time, we need to show that each stage of the algorithm should be clearly polynomial and the total number of steps should also be polynomial.

8.4 The Class NP

8.4.1 Verifier

Definition 8.4.1 A ***verifier*** for a language A is an algorithm V, where

$$A = \{w \mid V \text{ accepts } \langle w, c\rangle \text{ for some string } c\ \}$$

A ***polynomial time verifier*** runs in polynomial time in the length of w. A language A is ***polynomially verifiable*** if it has a polynomial time verifier. The c in the definition is called a ***certificate*** or ***proof*** of membership in A.

Definition 8.4.2 — Verifier-based Definition of NP. NP is the class of languages that have polynomial time verifiers.

8.4.2 NP

Equivalently, we can define the class NP in a way similar to how we have defined P. To do that, we first define $\mathsf{NTIME}(\cdot)$.

Definition 8.4.3 — NTIME.

$$\mathsf{NTIME}(t(n)) = \{L \mid L \text{ is the language decided by an } O(t(n)) \text{ time nondeterministic TM }\}$$

Then,

Definition 8.4.4 — NP.

$$\mathsf{NP} = \bigcup_k \mathsf{NTIME}(n^k)$$

Theorem 8.4.1 A language has a polynomial time verifiers if and only if it is decided by some nondeterministic polynomial time Turing machine.

Proof. The idea of the proof is to construct a polynomial time NTM using a verifier by nondeterministically guessing the certificate for V, and to construct a verifier using the sequence of nondeterministic choices made by the NTM as the certificate c. ■

We show that HAMPATH $\leq_P$ UHAMPATH. More specifically, given $\langle G, s, t\rangle$, where G is a directed graph where G' is an undirected graph such that

$$\langle G, s, t\rangle \in \text{HAMPATH} \iff \langle G', s', t'\rangle \in \text{UHAMPATH}$$

Chapter 9 The Cook-Levin Theorem

9.1 NP-Completeness

Definition 9.1.1 — NP-complete. A language A is ***NP-complete*** if it is in NP and every language $B \in \mathsf{NP}$ is polynomial time reducible to A.

Let A be any NP-complete problem, then $\mathsf{P} = \mathsf{NP}$ if and only if $A \in \mathsf{P}$.

If A is NP-complete, then assuming $\mathsf{P} \neq \mathsf{NP}$ (we don't know if this is true), A is not solvable in time $O(n^k)$ for any k.

9.2 The Cook-Levin Theorem

Theorem 9.2.1 — Cook and Levin 1971. SAT and 3SAT are NP-complete.

SAT is the ***Boolean satisfiability problem*** of propositional formulas.

Boolean variables: $x, y, z, \ldots$ taking values of TRUE and FALSE (represented by 1 and 0, respectively).

Boolean operations: AND ($\vee$), OR ($\wedge$), NOT (top bar or $\neg$)

Boolean formula: expressions involving Boolean variables and operations.

A Boolean formula is satisfiable if some assignment of 0s and 1s to its variable makes the formula evaluate to 1.

The satisfiability problem is to test whether a Boolean formula is satisfiable. More formally, consider the language

$$\text{SAT} = \{\langle \phi \rangle \mid \phi \text{ is a satisfiable Boolean formula}\}$$

Literal: a Boolean variable and its negation.

CNF formula: conjunctive normal form, and of ors. Each group of literals or'ed together is called a clause.

3CNF formula: each clause has 3 literals.

9.3 Proof of the Cook-Levin Theorem

Proof idea.

First, we show that SAT $\in$ NP: there is a polynomial time verifier that checks that a certificate is a satisfying assignment. Alternatively, we can construct an NTM that guesses the satisfying assignments.

Next, we need to show that SAT is NP-hard, that is, every language in NP is polynomial time reducible to SAT.

For all $A \in$ NP, show that $A \leq_p$ SAT. Such polynomial time reduction takes a string w and maps it to a formula such ϕ such that $w \in A \iff \phi \in$ SAT.

Since $A \in$ NP, there is a constant k and NTM N with running time n^k such that $A = \mathcal{L}(N)$. ϕ will simulate the machine N on w where $A = \mathcal{L}(N)$.

A tableau for N on w is an $n^k \times (n^k + 3)$ tbale, whose rows are

$$\#C_0\#, \#C_1\#, \#C_2\#, \dots, \#C_{n^k}\#$$

where each C_i is a configuration with n^k tape symbols and C_i yields C_{i+1} under N's transition function for all $i \in \{1, \dots, n^{k-1}\}$.

N accepts $w \iff$ there is an accepting tableau for N on w.

A tableau is accepting if and only if C_i is an accepting configuration. We want to construct the formula ϕ that will describe all logical constraints tha any accepting tableau for N on w must satisfy.

So,

$$\phi \text{ is satisfiable} \iff \text{there is an accepting tableau for } N \text{ on } w \iff N \text{ accepts } w$$

Let $C = Q \cup \Gamma \cup \{\#\}$ be the alphabet of the tableau (i.e. all characters tha can appear in the tableau).

- each entry of the tableau is a cell
- cell$[i, j]$ denotes the value of the cell at row i and column j
- for every i, j such that $1 \leq i \leq n^k$ and $1 \leq j \leq n^k + 3$ and every $s \in C$, there is a variable $x_{i,j,s}$. $x_{i,j,s}$ is a variable of ϕ

Hence, the total number of variables is $n^k(n^k + 3)|C| \in O(n^{2k})$.

$x_{i,j,s} = 1$ corresponds to cell$[i, j] = s$.

We will now design ϕ such that any satisfying assignment for ϕ corresponds to an accepting tableau for N on w.

Formula ϕ will be the AND of four CNF formulas

$$\phi = \phi_{cell} \wedge \phi_{start} \wedge \phi_{accept} \wedge \phi_{move}$$

where

ϕ_{cell}: for all i, j, exactly one $s \in C$ has $x_{i,j,s} = 1$.

$$\phi_{cell} = \bigvee_{\substack{1 \le i \le n^k \\ 1 \le j \le n^k+3}} \left[\left(\bigwedge_{s \in C} x_{i,j,s} \right) \wedge \left(\bigvee_{\substack{s,t \in C \\ s \neq t}} (\overline{x_{i,j,s}} \vee \overline{x_{i,j,t}}) \right) \right]$$

$\bigwedge_{s \in C} x_{i,j,s}$ enforces that at least one variable is set to 1; $\bigvee_{\substack{s,t \in C \\ s \neq t}} (\overline{x_{i,j,s}} \vee \overline{x_{i,j,t}})$ enforces that at most one variable is set to 1.

ϕ_{start}: the first row of the tableau is the start configuration of N on w

$$\phi_{start} = x_{1,1,\#} \wedge x_{1,2,q_0} \wedge x_{1,3,w_1} \wedge \ldots \wedge x_{1,n^k+2,\sqcup} \wedge x_{1,n^k+3,\#}$$

ϕ_{accept}: an accepting configuration is last row of the tableau

$$\phi_{accept} = \bigvee_{\substack{1 \le i \le n^k \\ 1 \le j \le n^k+3}} x_{i,j,q_{accept}}$$

ϕ_{move}: the tableau is legal, i.e. every row is a configuration that legally follows from the previous configuration

This formula will check that every 2×3 "windows" of cells is legal, i.e. it obeys transition function.

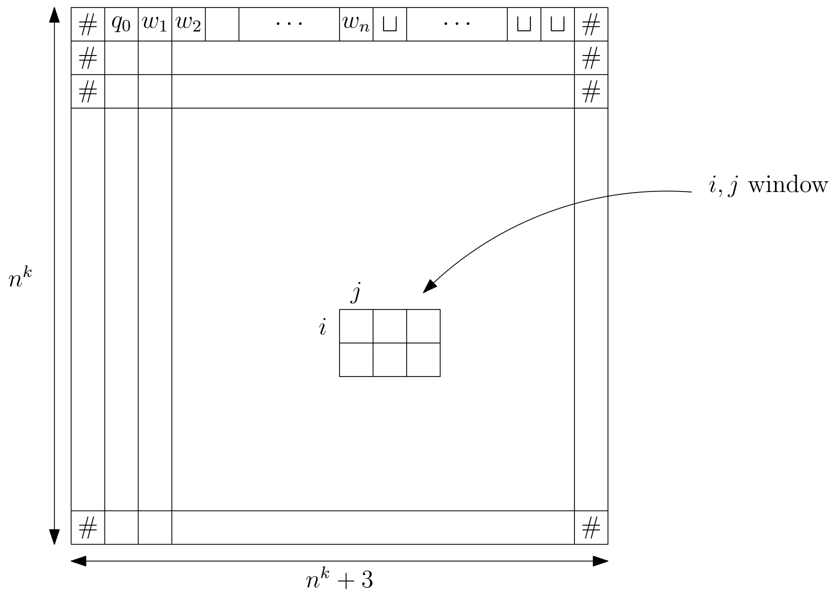


Figure 9.1: A tableau.

Informally,

$$\phi_{move} = \bigwedge_{\substack{1\le i\le n^k-1\\ 1\le j\le n^k+1}} (\text{the } (i,j) \text{ window is legal})$$

We express "(i,j) window is legal" using the formula

$$\bigvee_{\substack{(a_1,a_2,\ldots,a_6)\\ \text{is a legal window}}} (x_{i,j,a_1} \wedge x_{i,j+1,a_2} \wedge x_{i,j+2,a_3} \wedge x_{i+1,j,a_4} \wedge x_{i+1,j+1,a_5} \wedge x_{i+1,j+2,a_6})$$

which is equivalent to

$$\bigwedge_{\substack{(a_1,\ldots,a_6)\\ \text{isn't a legal window}}} (\overline{x_{i,j,a_1}} \vee \overline{x_{i,j+1,a_2}} \vee \overline{x_{i,j+2,a_3}} \vee \overline{x_{i+1,j,a_4}} \vee \overline{x_{i+1,j+1,a_5}} \vee \overline{x_{i+1,j+2,a_6}})$$

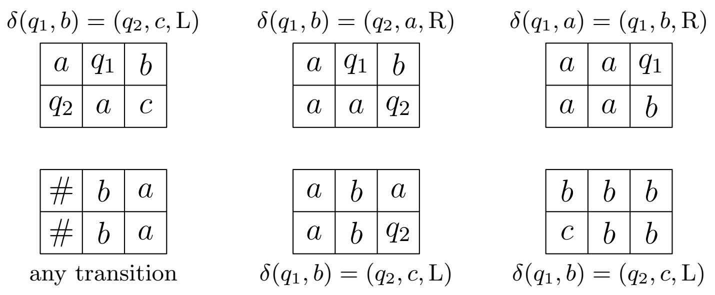

$\delta(q_1,b)=(q_2,c,\mathrm{L})$
a q_1 b
q_2 a c
$\delta(q_1,b)=(q_2,a,\mathrm{R})$
a q_1 b
a a q_2
$\delta(q_1,a)=(q_1,b,\mathrm{R})$
a a q_1
a a b
$\#$ b a
$\#$ b a
any transition
a b a
a b q_2
$\delta(q_1,b)=(q_2,c,\mathrm{L})$
b b b
c b b
$\delta(q_1,b)=(q_2,c,\mathrm{L})$

Figure 9.2: Some legal windows under the transition function $\delta(q_1,b)=(q_2,c,\mathrm{L})$, $\delta(q_1,b)=(q_2,a,\mathrm{R})$, and $\delta(q_1,a)=(q_1,b,\mathrm{R})$.

Lemma 9.3.1 If the top row of the tableau is the start configuration and every window in the tableau is legal, each row of the tableau is a configuration that legally follows the preceding one. In other words, if some configuration does not follow from the previous one, we are guaranteed to detect the violation in some window.

Proof. See Sipser Claim 7.41. ■

Thus $A \le_p$ SAT so SAT is NP-complete.

9.4 3SAT

Everything is already in AND of ORs (CNF), we need to make this ORs small (3CNF). Let's start with a short example of a 4CNF with one clause.

$$(a_1 \vee a_2 \vee a_3 \vee a_4) \iff (a_1 \vee a_2 \vee z_1) \wedge (\neg z_1 \vee a_3 \vee a_4)$$

Every assignment that satisfies the formula on the LHS satisfies the formula on the RHS, and vice versa.

In general, for each clause in the original formula,

$$(a_1 \vee a_2 \vee a_3 \vee \cdots \vee a_t)$$

we can replace it with

$$(a_1 \vee a_2 \vee z_1) \wedge (\neg z_1 \vee a_3 \vee z_2) \wedge (\neg z_2 \vee a_4 \vee z_3) \wedge (\neg z_3 \vee \cdots) \wedge \cdots \wedge (\neg z_{t-3} \vee a_{t-1} \vee a_t)$$

with $t-2$ clauses.

Chapter 10 NP-Complete Problems

10.1 Clique

A clique in an undirected graph $G = (V, E)$ is sa set $S \subseteq V$ such that for all pairs of distinct $u, v \in S$, $\{u, v\} \in E$.

$$\text{CLIQUE} = \{\langle G, k\rangle \mid G \text{ is a graph, } k \in \mathbb{N} \text{ and } G \text{ has a clique of size } k\ \}$$

Theorem 10.1.1 CLIQUE is NP-compelte.

Proof. Clearly, CLIQUQE $\in$ NP. To show that CLIQUE is NP-hard, we will show the reduction 3SAT $\leq_p$ CLIQUE.

Given a 3CNF formula ϕ, construct in polynomial time $\langle G_\phi, k_\phi\rangle$ such that

$$\phi \text{ is satisfiable } \iff G_\phi \text{ has a clique of size } k$$

Suppose $\phi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \vee \cdots \vee (a_k \vee b_k \vee c_k)$ with k clauses and 3 literals per clause.

We construct G such that it has k groups of 3 vertices (giving us $3k$ vertices) where each vertex is labeled according to a literal in a clause. We connect all pairs of vertices except:

- vertices in the same triple (same clause)
- vertices with contradictory labels (x_i and $\overline{x_i}$)

Now assume that ϕ is satisfiable, then there exists some assignment of variables that satisfies ϕ. Take such satisfying assignment. In this assignment, there must be at least one true literal in each clause. In each triple, pick any one vertex corresponding to the true literal. These vertices form a k-clique containing k vertices, distinct triples (because there is no edge among vertices within the same triple), and no contradicting labels (becuase there is no edge between contradictory labels).

Suppose a graph G has a k-clique. Then, no two vertices in the clique are in the same triple. Each triple has exactly one vertex in the clique. Assign values to variables so that all clique nodes are true literals (this is possible since there is no contradictory labels in clique). This assignment satisfies ϕ. Therefore, ϕ is satisfiable. ■

Now consider the language k-CLIQUE. In this language, k is fixed.

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

Theorem 10.1.2 k-CLIQUE $\in$ P for all k.

Proof. Check all $\binom{n}{k}$ subset of vertices. Since k is fixed, this number of subset needs be checked is polynomial in n. ■

10.2 Independent Sets

$G = (V,E)$, $S \subseteq V$ is an independent set in G if $\langle u,v\rangle \notin E$ for all distinct $u,v \in S$.

$$\text{IND-SET} = \{\langle G,k\rangle \mid G \text{ is a graph which contains an independent set of size } k\}$$

Theorem 10.2.1 IND-SET is NP-complete.

Proof. We show that CLIQUE $\leq_p$ IND-SET. ■

10.3 Vertex Cover

Let $G = (V,E)$ be an undirected graph. $S \subseteq V$ is a vertex cover for G if every edge in G has at least one endpoint in S.

$$\text{VERTEX-COVER} = \{\langle G,k\rangle \mid G \text{ has vertex color of size } k\}$$

Theorem 10.3.1 VERTEX-COVER is NP-complete.

Proof. Showing VERTEX-COVER $\in$ NP is easy. We show VERTEX-COVER is NP-hard by showing

$$\text{IND-SET} \leq_p \text{VERTEX-COVER}$$

For the reduction, we map $\langle G,k\rangle$ to $\langle G,|V|-k\rangle$. We claim that $G = (V,E)$ where $S \subseteq V$ is an independent set in G if and only if $V - S$ is a vertex cover. ■

10.4 Coloring

An undirected graph G is k-colorable if there is a map $f : V \to \{1, 2 \ldots, k\}$ such that for every edge $(u, v) \in E$, $f(u) \neq f(v)$.

Theorem 10.4.1 2COL $\in$ P.

Proof. Run BFS. Greedily assign colors by alternating coloring for each level. ■

Theorem 10.4.2 3COL is NP-complete.

Proof. We show that 3SAT $\leq_p$ 3COL.

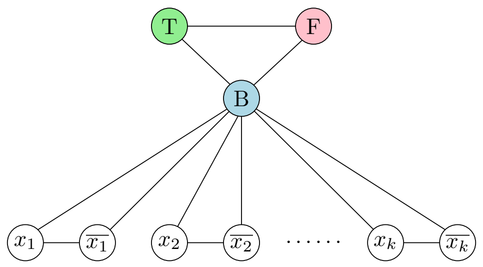


Figure 10.1: Construction that ensures that a coloring corresponds to a valid truth assignment.

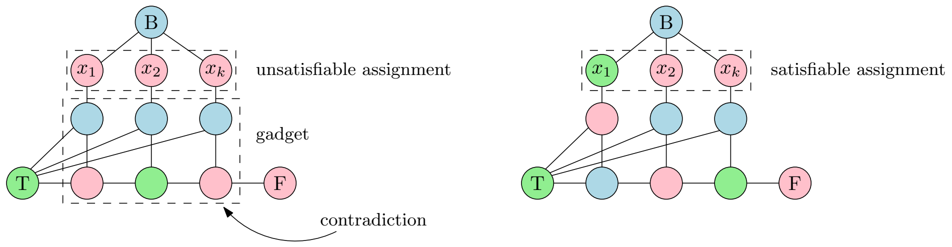


Figure 10.2: Construction using gadgets that ensures only satisfiable truth assignment can be colored.

An alternative construction uses the OR-gadget. The OR-gadget forces the three vertices within the gadget to have different colors. This, in turn, forces a contradiction at the output gadget when we have all literals being false.

■

10.5 Subset Sum

Given a collection of numbers $x_1, \ldots, x_k$ and a target t, is there a subcollection adding up to t.

$$\text{SUBSET-SUM} = \{\langle s, t\rangle \mid s = \{x_1, \ldots, x_k\} \text{ and for some } \{y_1, \ldots, y_i\} \subseteq \{x_1, \ldots, x_k\}, \Sigma y_i = t\}$$

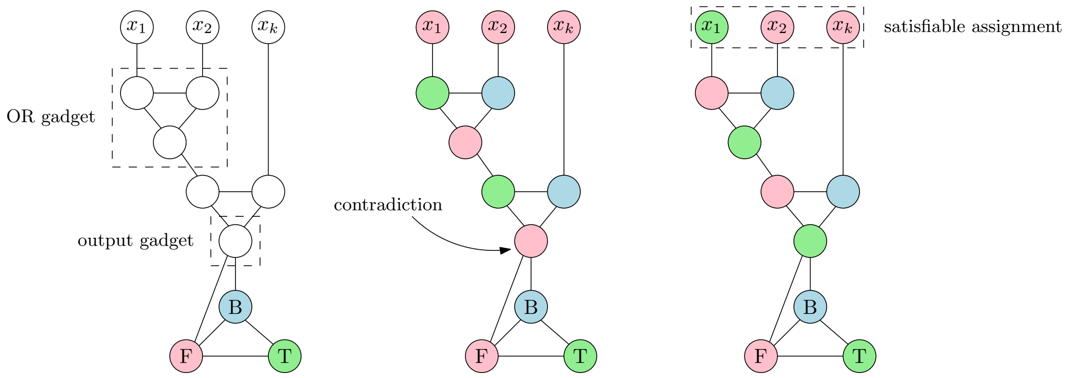


Figure 10.3: Construction using gadgets that ensures only satisfiable truth assignment can be colored.

Proof. ■

Chapter 11 Space Complexity

11.1 Space Complexity

We define the space complexity as the largest tape index reached during the computation.

Definition 11.1.1 — Space Complexity. The space complexity of M is the function $f : \mathbb{N} \to \mathbb{N}$ where $f(n)$ is the largesttape index reached by M on any input of length n.

Definition 11.1.2 — Space Complexity Class.

$$\mathsf{SPACE}(f(n)) = \{L \mid L \text{ is decided by a TM with } O(f(n)) \text{ space complexity}\}$$

Theorem 11.1.1 3SAT $\in \mathsf{SPACE}(n)$

Proof. Given 3SAT formula ϕ of length n, try all possible assignments to at most n variables. Evaluate ϕ on each assignment. Accept if and only if there is a satisfying assignment. This can be carried out in $O(n)$ space. ■

Theorem 11.1.2 $\mathsf{NTIME}(t(n)) \subseteq \mathsf{SPACE}(t(n))$

Definition 11.1.3 — PSPACE.

$$\mathsf{PSPACE} = \bigcup_{k \in \mathbb{N}} \mathsf{SPACE}(n^k)$$

PSPACE formalizes problems solvable by computers with bounded memory. PSPACE is more general because we can reuse space.

Open question: $\mathsf{P} \stackrel{?}{=} \mathsf{PSPACE}$

If $\mathsf{P} = \mathsf{PSPACE}$, then $\mathsf{P} = \mathsf{NP}$.

11.2 Time Complexity of SPACE

Let M be a Turing machine that halts with space complexity $f(n)$. How many steps could M take on inputs of length n?

Number of steps is at most number of possible configurations. Recall that a configuration encodes the head position, state, $f(n)$ cells of tape content, so the length of a configuration is in $O(f(n))$. It follows that the total number of configurations is in $O(2^{f(n)})$.

Hence,

$$\mathsf{PSPACE} \subseteq \mathsf{EXPTIME}$$

So far, we have

$$\mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSPACE} \subseteq \mathsf{EXPTIME}$$

Theorem 11.2.1 $\mathsf{P} \neq \mathsf{EXPTIME}$

Proof. By the time hierarchy theorem. ■

So, at least one of $\mathsf{P} \neq \mathsf{NP}$ and $\mathsf{NP} \neq \mathsf{PSPACE}$ and $\mathsf{PSPACE} \neq \mathsf{EXPTIME}$ must be true.

11.3 Nondeterministic Space

Definition 11.3.1 — NSPACE.

$$\mathsf{NSPACE}(f(n)) = \{L \mid L \text{ is decided by a nondeterministic TM with } O(f(n)) \text{ space complexity}\}$$

Definition 11.3.2 — NPSPACE.

$$\mathsf{NPSPACE} = \bigcup_{k\in\mathbb{N}} \mathsf{NSPACE}(n^k)$$

11.4 Savitch's Theorem

Theorem 11.4.1 — Savitch's Theorem. For any function $f : \mathbb{N} \to \mathbb{R}^+$ where $f(n) \geq n$,

$$\mathsf{NSPACE}(f(n)) \subseteq \mathsf{SPACE}(f^2(n))$$

Proof. Let $N = (Q, \Sigma, \Gamma, \delta, q_0, q_{acc}, q_{rej})$ be NTM with space complexity $f(n)$. We will construct a DTM M with space complexity $O(f^2(n))$ such that $\mathcal{L}(M) = \mathcal{L}(N)$.

Let $w \in \Sigma^n$ be input of length n. We define a directed graph $G = (V, E)$ called $f(n)$ space configuration graph of N where

$$V = \{\text{configurations of } N \text{ with at most } f(n)\ \}$$

Observe that $w \in \mathcal{L}(N)$ if and only if there exists a path in G of length at most $2^{df(n)}$ from $q_0 w$ to an accepting configuration of the form $xq_{acc}y$.

```
CAN-YIELD($c_1, c_2, t$)
1   if $t == 1$
2       if $c_1$ yields $c_2$
3           accept
4       else reject
5   if $t \geq 2$
6       for $c_3 \in V$ in order
7           CAN-YIELD($c_1, c_3, \lceil t/2 \rceil$)
8           CAN-YIELD($c_3, c_2, \lfloor t/2 \rfloor$)
9           if both accept for some $c_3$
10              accept
11          else reject
```

```
$M$ = "on input $w$ of length $n$
  1.  for each $c \in V$ in orders
  2.      if $c$ is an accepting configuration
  3.          run CAN-YIELD($q_0 w, c, 2^{df(n)}$)
  4.          if it accepts then accept
  5.          else continue
  6.      if CAN-YIELD($q_0 w, c, 2^{df(n)}$) rejects for every accepting configuration $c \in V$, reject"
```

■

Thus, by Savitch's theorem

$$\text{PSPACE} = \text{NPSPACE}$$

We can simulate any NPSPACE Turing machine using a deterministic TM with only a quadratic overhead.

11.5 PSPACE-Complete

Definition 11.5.1 — PSPACE-Complete. Language B is PSPACE-complete if

1. B is in PSPACE
2. Every A in PSPACE is **poly-time** reducible to B (i.e. B is PSPACE-hard)

Theorem 11.5.1 If B is PSPACE-complete and B is in P, then P = PSPACE.

Proof idea. Let $A \in$ PSPACE. The poly-time TM for A first reduces input x to an instance y of B. Then it runs the polytime TM for B on y and outputs the answer. ■

Theorem 11.5.2 If B is NPSPACE-complete and B is in NP, then NP $=$ PSPACE.

11.5.1 TQBF

Definition 11.5.2 — Fully Quantified Boolean Formula. A ***fully quantified Boolean formula*** is a Boolean formula where every variable in the formula is quantified ($\exists$ or $\forall$). If all quantifiers appear at the beginning of the statement and that each quantifier's scope is everything following it, we say the statement is in ***prenex normal form*** (PNF).

$$\text{TQBF} = \{\langle \phi \rangle \mid \phi \text{ is a true fully quantified Boolean formula}\}$$

Theorem 11.5.3 TQBF is PSPACE-complete.

Note that SAT is a special case where all quantifiers are $\exists$. And TAUT is a special case where all quantifiers are $\forall$.

TAUT is a special case where all quantifiers are $\forall$. So clearly, SAT $\leq_p$ TQBF and TAUT $\leq_p$ TQBF.

Theorem 11.5.4 — Meyer-Stockmeyer. TQBF is PSPACE-complete.

Proof. We need to show two things: (1) TQBF is in PSPACE; and (2) TQBF is PSPACE-hard.

We show that TQBF $\in$ PSPACE. Consider the Turing machine T that decides TQBF.

$T =$"on input $\langle \phi \rangle$

1. if ϕ has no quantifiers, evaluate ϕ. Accept if it evaluates to 1.
2. If $\phi = \exists x.\, \psi$, recursively call T on ψ, first with $x = 0$ and then $x = 1$. If either result is accept, then accept; otherwise reject.
3. If $\phi = \forall x.\, \psi$, recursively call T on ψ first with $x = 0$ and then $x = 1$. If both results are accept, then accept; otherwise reject."

Note that we can reuse space. The depth of recursion is number of variables. At each level, we just need to store value of one variable. Hence, the space used is in $O(m)$ where m is the number of variables.

To show that TQBF is PSPACE-hard, we show that for all $A \in$ PSPACE, $A \leq_p$ TQBF.

For every $A \in$ PSPACE, there exists some constant k and Turing machine M that decides A in space of at most n^k. However, M may run exponentially many steps, so simply encoding the computation history tableau directly won't work because we cannot construct an exponentially-sized formula in polynomial time.

Fix M and w, we will construct a quantified Boolean formula ϕ such that M accepts $w \iff \phi$ is true. We will actually show how to construct $\phi_{c_1,c_2,t}$ to be a formula that is true if and only if M can go from configuration c_1 and c_2 in at most t steps. Note that in the construction that follows, c_1, c_2, m_1 are not necessarily one single variable, but collections of variables that represent some configurations. We can encode a configuration using $l \in O(n^k)$ variables from the proof of Cook-Levin's theorem.

When $t = 1$, it is easy to construct $\phi_{c_1,c_2,1}$. This translates to "c_2 follows from c_1". This is essentially the same construction used in the proof of Cook-Levin theorem.

When $t > 1$, we construct $\phi_{c_1,c_2,t}$ recursively.

$$\phi_{c_1,c_2,t} = \exists m_1.[\phi_{c_1,m_1,t/2} \wedge \phi_{m_1,c_2,t/2}]$$

where m_1 represents variables describing a configuration of M. Notice, however, at each stage, the size of the formula doubles. This will lead to an exponential explosion of the size of the formula. In the end, we would have a formula of size in the order of $t = 2^{dn^k}$.

Instead, we use a clever construction:

$$\phi_{c_1,c_2,t} = \exists m_1.\forall c_3,c_4.[(c_3,c_4) = (c_1,m_1) \vee (c_3,c_4) = (m_1,c_2) \implies \phi_{c_3,c_4,t/2}]$$

This construction essentially says: "for all variables c_3, c_4 encoding a configuration, if c_3, c_4 represents c_1, m_1, then $\phi_{c_3,c_4,t/2}$ must be true; and the same is true for the case when c_3, c_4 represents c_1, m_1".

With this construction, the size of the formula increases at most by an additive factor of size $O(n^k)$. At the end, following the recursive construction described as above, we have

$$\phi_{c_{start},c_{end},2^{dn^k}}$$

The level of recursion is $\log(2^{dn^k})$ and each level of recursion adds a portion of the formula with size linear in the size of the variables encoding a configuration, which is in $O(n^k)$. Therefore, in the end, the size of the formula is $O(n^{2k})$, polynomial in the input size. ■

Chapter 12 The Class L and NL

12.1 L and NL

Sublinear space bounds where $f(n)$ is much smaller than n. This won't make sense given our existing one-tape Turing machine model since we won't even have enough space to store the input.

We modify our model and have a 2-tape Turing machine where we have one tape as a read-only input tape, and another tape as the read/write work tape. We don't have to store all data on the main memory, so we would like to consider *only the computation spaced used on the work tape*.

$$\mathsf{L} = \mathsf{SPACE}(\log n)$$
$$\mathsf{NL} = \mathsf{NSPACE}(\log n)$$

Savitch's theorem still holds, so $\mathsf{NL} \subseteq \mathsf{SPACE}(\log^2 n)$.

Theorem 12.1.1 $\mathsf{L} \subseteq \mathsf{P}$

Proof. Let A be a language in L. There exsits some logspace-bounded 2-tape TM M that decides A. We define the configuration for a logspace-bounded 2-tape Turing machine M on w as (q, p_1, p_2, t) where q is a state, p_1 and p_2 are the tape positions (for the read-only input tape and work tape, respectively), and t is the tape content on the work tape. Then, the number of possible configurations is $|Q| \times n \times O(\log n) \times |\Gamma|^{O(\log n)} = O(n^k)$ for some k. Therefore, M runs in polynomial time. ■

12.2 PATH is in NL

$$\text{PATH} = \{\langle G, s, t\rangle \mid G \text{ has directed path from } s \text{ to } t\ \}$$

Theorem 12.2.1 — PATH in NL.

$$\text{PATH} \in \mathsf{NL}$$

Proof. At each vertex, we use nondeterminism to guess the next vertex. Suppose we have the path $p = ss_1s_2\ldots t$. At each vertex s_i, we look at the next vertex s_{i+1} and checks if s_{i+1} is connected to s_i. If any of the nondeterministic paths lead to t. At any time during the computation of one nondeterministic path, we have at most the name of two vertices on the work tape. In addition, we keep track of the vertices visited, using $\log n$ space since any simple path contains at most n vertices. It takes at most $\log n$ space to write down the name of a vertex, so PATH $\in$ NL. ■

We do not know if PATH $\overset{?}{\in}$ L. We also don't know if L $\overset{?}{=}$ NL.

12.3 NL-Complete

A language A is in NL-complete if $A \in$ NL and for all $B \in$ NL, $B \leq_L A$. The symbol $\leq_L$ stands for log-space reducible, which we will define here.

As we will show later, all problems in NL are also in P. It does not make a lot of sense to define NL-complete in terms of polytime reduction. Note that for all $A, B \in$ NL except $\emptyset$ and Σ^*, $A \leq_p B$ and $B \leq_p A$. We need a type of reduction that is more restrictive than polytime reduction.

12.3.1 Log-Space Reduction

A log-space transducer is a Turing machine with

- read-only input tape
- write-only output tape (head can only move to the right)
- read/write work tape (only $O(\log n)$ symbols)

that computes a function $f : \Sigma^* \to \Sigma^*$ where $f(w)$ is written on output tape when M halts when w is on the input tape.

We say f is a log-space computable function.

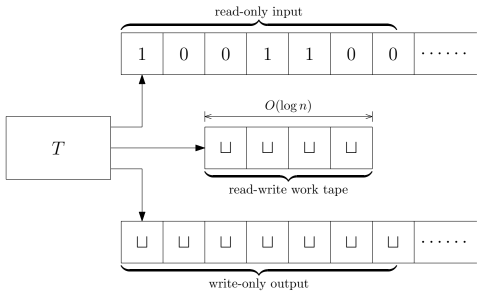


Figure 12.1: A log-space transducer

Theorem 12.3.1 if $A \leq_L B$ and $B \in$ L, then $A \in$ L.

Proof. Suppose there exists a log-space decider M_B for B. We construct a log-space decider for A. A seemingly obvious construction would be:

```
M = "on input w
  1. run w on the log-space transducer to compute f(w)
  2. run M_B on f(w)"
```

The issue with this construction is that we don't have enough space to actually write down the entirety of $f(w)$, so this construction won't quite work.

We use this algernative construction. Whenever B needs a bit of $f(w)$, we recompute it on the fly. More formally, M_A computes individual symbols of $f(w)$ by calling the logspace transducer as requested by M_B. In the simulation, we run M_B without actual input on the tape. When it requests the ith bit of the input, we run M_A with a counter to get the ith bit of $f(w)$.

To see why this construction of M_A runs in logspace, observe first that M_A itself needs to keep track of the position requested by M_B when calling the transducer, which takes $O(\log n)$ space. In addition, M_B uses $O(\log|f(w)|)+1$ space (the one additional bit is used to store the bit requested by M_B at each step of its computation) when we simulate M_B. Since $f(w)$ has length of as most the number of configurations that the transducer can have, which is $|w|2^{O(\log|w|)}$. It follows that the space used by M_B is also in $O(\log|w|) = O(\log n)$. Therefore, M_A runs in logspace. ■

12.3.2 PATH is NL-Complete

Theorem 12.3.2 PATH is NL-complete.

Proof. We have shown that PATH is in NL. It reamins to be shown that for every language $A \in \mathsf{NL}$, $A \leq_L$ PATH.

Let $A \in \mathsf{NL}$. There exists a nondeterministic log-space Turing machine M that decides A. Given a string w, we want to reduce it to a graph representing the computation of M on w where:

- nodes: various configurations
- edges: $(c_1, c_2) \in E$ if c_1 yields y_2

Why is this construction log-space computable? We need to give a log-space transducer that outputs $\langle G, s, t\rangle$ on input w. Note that when constructing a log-space transducer, we don't actually care about the time complexity of the transducer, although the construction that we are about to show is both log-space and runs in polytime. The transducer needs to do two things:

1. List the nodes: This is relatively easy. Each node is a configuration of M on w which can be represented in $c\log n$ space for some constant c. The transducer can simply go through every strings of length $c\log n$ using brute force, checks if it encodes a legal configuration of M on w, and outputs those that do. This runs in log-space because at any time, we only write at most one string on the work tape.
2. List the edges: We list the edges of the graph by testing every pair of possible configurations (c_1, c_2) and checks if c_1 yields c_2. This can be done using log-space as well because we only need to write c_1, c_2 on the work tape while looking at the transition table to determine if c_1 yields c_2.

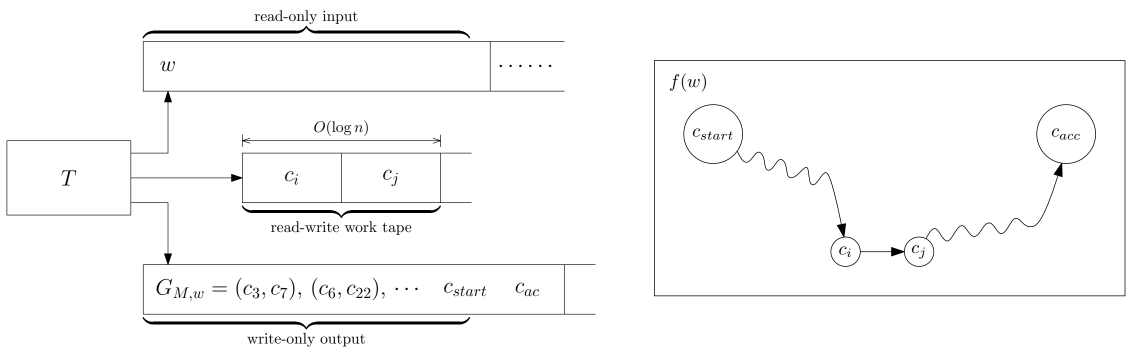


Figure 12.2: A log-space transducer for reducing A to PATH and the computation graph constructed by the logspace transducer.

■

Corollary 12.3.3 NL $\subseteq$ P

Proof. 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. Therefore, any language in NL is polynomial time reducible to PATH, which is in P. ■

12.4 NL and coNL

Theorem 12.4.1 — Immerman-Szelepcsényi (1987). NL $=$ co-NL

Appendix

Commonly Used Axioms & Theorems

Rules of Inference

Axiom 1 — Modus Ponens. $(P \wedge (P \implies Q)) \implies Q$

Axiom 2 — Modus Tollens. $(\neg Q \wedge (P \implies Q)) \implies \neg P$

Axiom 3 — Hypothetical Syllogism (transitivity).

$((P \implies Q) \wedge (Q \implies R)) \implies (P \implies R)$

Axiom 4 — Disjunctive Syllogism. $((P \vee Q) \wedge \neg P) \implies Q$

Axiom 5 — Addition. $P \implies (P \vee Q)$

Axiom 6 — Simplification. $(P \wedge Q) \implies P$

Axiom 7 — Conjunction. $((P) \wedge (Q)) \implies (P \wedge Q)$

Axiom 8 — Resolution. $((P \lor Q) \land (\neg P \lor R)) \implies (Q \lor R)$

Laws of Logic

Axiom 9 — Implication Law. $(P \implies Q) \equiv (\neg P \lor Q)$

Axiom 10 — Distributive Law.

$$(P \land (Q \lor R)) \equiv ((P \land Q) \lor (P \land R))$$
$$(P \lor (Q \land R)) \equiv ((P \lor Q) \land (P \lor R))$$

Axiom 11 — De Morgan's Law.

$$\neg(P \land Q) \equiv (\neg P \lor \neg Q)$$
$$\neg(P \lor Q) \equiv (\neg P \land \neg Q)$$

Axiom 12 — Absorption Law.

$$(P \lor (P \land Q)) \equiv P$$
$$(P \land (P \lor Q)) \equiv P$$

Axiom 13 — Commutativity of AND. $A \land B \equiv B \land A$

Axiom 14 — Associativity of AND. $(A \land B) \land C \equiv A \land (B \land C)$

Axiom 15 — Identity of AND. $\mathbf{T} \land A \equiv A$

Axiom 16 — Zero of AND. $\mathbf{F} \land A \equiv \mathbf{F}$

Axiom 17 — Idempotence for AND. $A \land A \equiv A$

Axiom 18 — Contradiction for AND. $A \land \neg A \equiv \mathbf{F}$

Axiom 19 — Double Negation. $\neg(\neg A) \equiv A$

Axiom 20 — Validity for OR. $A \vee \neg A \equiv \mathbf{T}$

Induction

Axiom 21 — Well Ordering Principle. Every nonempty set of nonnegative integers has a smallest element. i.e., For any $A \subset \mathbb{N}$ such that $A \neq \emptyset$, there is some $a \in A$ such that $\forall a' \in A . a \leq a'$.

Recurrences

Theorem 22 — The Master Theorem. Suppose that for $n \in \mathbb{Z}^+$.

$$T(n) = \begin{cases} c & \text{if } n < B \\ a_1 T(\lceil n/b \rceil) + a_2 T(\lfloor n/b \rfloor) + dn^i & \text{if } n \geq B \end{cases}$$

where $a_1, a_2, B, b \in \mathbb{N}$.

Let $a = a_1 + a_2 \geq 1$, $b > 1$, and $c, d, i \in \mathbb{R} \cup \{0\}$. Then,

$$T(n) \in \begin{cases} O(n^i \log n) & \text{if } a = b^i \\ O(n^i) & \text{if } a < b^i \\ O(n^{\log_b a}) & \text{if } a > b^i \end{cases}$$

Linear Recurrences:

A linear recurrence is an equation

$$f(n) = \underbrace{a_1 f(n-1) + a_2 f(n-2) + \cdots + a_d f(n-d)}_{\text{homogeneous part}} + \underbrace{g(n)}_{\text{inhomogeneous part}}$$

along with some boundary conditions.

The procedure for solving linear recurrences are as follows:

1. Find the roots of the characteristic equation. Linear recurrences usually have exponential solutions (such as x^n). Such solution is called the **homogeneous solution**.

$$x^n = a_1 x^{n-1} + a_2 x^{n-2} + \cdots + a_{k-1} x + a_k$$

2. Write down the homogeneous solution. Each root generates one term and the homogeneous solution is their sum. A non-repeated root r generates the term cr^n, where c is a constant to be determined later. A root with r with multiplicity k generates the terms

$$d_1 r^n \quad d_2 n r^n \quad d_3 n^2 r^n \quad \cdots \quad d_k n^{k-1} r^n$$

where $d_1, \cdots, d_k$ are constants to be determined later.

3. Find a **particular solution** for the full recurrence including the inhomogeneous part, but without considering the boundary conditions.
   If $g(n)$ is a constant or a polynomial, try a polynomial of the same degree, then of one higher degree, then two higher. If $g(n)$ is exponential in the form $g(n) = k^n$, then try $f(n) = ck^n$, then $f(n) = (bn + c)k^n$, then $f(n) = (an^2 + bn + c)k^n$, etc.
4. Write the **general solution**, which is the sum of homogeneous solution and particular solution.
5. Substitute the boundary condition into the general solution. Each boundary condition gives a linear equation. Solve such system of linear equations for the values of the constants to make the solution consistent with the boundary conditions.

Basic Prerequisite Mathematics

Basic Prerequisite Mathematics

SET THEORY

Common Sets

- $\mathbb{N} = \{0, 1, 2, \dots\}$: the natural numbers, or non-negative integers. The convention in computer science is to include 0 in the natural numbers.
- $\mathbb{Z} = \{\dots, -2, -1, 0, 1, 2, \dots\}$: the integers
- $\mathbb{Z}^+ = \{1, 2, 3, \dots\}$: the positive integers
- $\mathbb{Z}^- = \{-1, -2, -3, \dots\}$: the negative integers
- $\mathbb{Q}$ the rational numbers, $\mathbb{Q}^+$ the positive rationals, and $\mathbb{Q}^-$ the negative rationals.
- $\mathbb{R}$ the real numbers, $\mathbb{R}^+$ the positive reals, and $\mathbb{R}^-$ the negative reals.

Notation

For any sets A and B, we will use the following standard notation.

- $x \in A$: "x is an element of A" or "A contains x"
- $A \subseteq B$: "A is a subset of B" or "A is included in B"
- $A = B$: "A equals B" (Note that $A = B$ if and only if $A \subseteq B$ and $B \subseteq A$.)
- $A \subsetneq B$: "A is a proper subset of B"
  (Note that $A \subsetneq B$ if and only if $A \subseteq B$ and $A \neq B$.)
- $A \cup B$: "A union B"
- $A \cap B$: "A intersection B"
- $A - B$: "A minus B" (*set* difference)
- $|A|$: "cardinality of A" (the number of elements of A)
- $\varnothing$ or $\{\}$: "the empty set"
- $\mathcal{P}(A)$ or 2^A: "powerset of A" (the set of all subsets of A)
  If $A = \{a, 34, \triangle\}$, then $\mathcal{P}(A) = \big\{\{\}, \{a\}, \{34\}, \{\triangle\}, \{a, 34\}, \{a, \triangle\}, \{34, \triangle\}, \{a, 34, \triangle\}\big\}$.
  $S \in \mathcal{P}(A)$ means the same as $S \subseteq A$.
- $\{x \in A \mid P(x)\}$: "the set of elements x in A for which $P(x)$ is true"
  For example, $\{x \in \mathbb{Z} \mid \cos(\pi x) > 0\}$ represents the set of integers x for which $\cos(\pi x)$ is greater than zero, *i.e.*, it is equal to $\{\dots, -4, -2, 0, 2, 4, \dots\} = \{x \in \mathbb{Z} \mid x \text{ is even}\}$.

- $A \times B$: "the cross product or Cartesian product of A and B"
  $A \times B = \{(a,b) \mid a \in A \text{ and } b \in B\}$.
  If $A = \{1,2,3\}$ and $B = \{5,6\}$, then $A \times B = \{(1,5),(1,6),(2,5),(2,6),(3,5),(3,6)\}$.)

- A^n: "the cross product of n copies of A"
  This is set of all sequences of $n \geq 1$ elements, each of which is in A.

- B^A or $A \to B$: "the set of all functions from A to B."

- $f : A \to B$ or $f \in B^A$: "f is a function from A to B"
  f associates one element $f(x) \in B$ to every element $x \in A$.

NUMBER THEORY

For any two natural numbers a and b, we say that a *divides* b if there exists a natural number c such that $b = ac$. In such a case, we say that a is a *divisor* of b (*e.g.*, 3 is a divisor of 12 but 3 is not a divisor of 16). Note that any natural number is a divisor of 0 and 1 is a divisor of any natural number. A number a is *even* if 2 divides a and is *odd* if 2 does not divide a.

A natural number p is *prime* if it has exactly two positive divisors (*e.g.*, 2 is prime since its positive divisors are 1 and 2 but 1 is **not** prime since it only has one positive divisor: 1). There are an infinite number of prime numbers and any integer greater than one can be expressed in a unique way as a finite product of prime numbers (*e.g.*, $8 = 2^3$, $77 = 7 \times 11$, $3 = 3$).

Inequalities

For any integers m and n, $m < n$ if and only if $m + 1 \leq n$ and $m > n$ if and only if $m \geq n + 1$. For any real numbers w, x, y, and z, the following properties always hold (they also hold when $<$ and $\leq$ are exchanged throughout with $>$ and $\geq$, respectively).

- if $x < y$ and $w \leq z$, then $x + w < y + z$

- if $x < y$, then $\begin{cases} xz < yz & \text{if } z > 0 \\ xz = yz & \text{if } z = 0 \\ xz > yz & \text{if } z < 0 \end{cases}$

- if $x \leq y$ and $y < z$ (or if $x < y$ and $y \leq z$), then $x < z$

Functions

Here are some common number-theoretic functions together with their definitions and properties of them. (Unless noted otherwise, in this section, x and y represent arbitrary real numbers and k, m, and n represent arbitrary positive integers.)

- $\min\{x, y\}$: "minimum of x and y" (the smallest of x or y)

  Properties: $\min\{x, y\} \leq x$
  $\min\{x, y\} \leq y$

- $\max\{x, y\}$: "maximum of x and y" (the largest of x or y)

  Properties: $x \le \max\{x, y\}$
  $y \le \max\{x, y\}$

- $\lfloor x \rfloor$: "floor of x" (the greatest integer less than or equal to x, *e.g.*, $\lfloor 5.67 \rfloor = 5$, $\lfloor -2.01 \rfloor = -3$)

  Properties: $x - 1 < \lfloor x \rfloor \le x$
  $\lfloor -x \rfloor = -\lceil x \rceil$
  $\lfloor x + k \rfloor = \lfloor x \rfloor + k$
  $\lfloor \lfloor k/m \rfloor / n \rfloor = \lfloor k/mn \rfloor$
  $(k - m + 1)/m \le \lfloor k/m \rfloor$

- $\lceil x \rceil$: "ceiling of x" (the least integer greater than or equal to x, *e.g.*, $\lceil 5.67 \rceil = 6$, $\lceil -2.01 \rceil = -2$)

  Properties: $x \le \lceil x \rceil < x + 1$
  $\lceil -x \rceil = -\lfloor x \rfloor$
  $\lceil x + k \rceil = \lceil x \rceil + k$
  $\lceil \lceil k/m \rceil / n \rceil = \lceil k/mn \rceil$
  $\lceil k/m \rceil \le (k + m - 1)/m$

  Additional property of $\lfloor\ \rfloor$ and $\lceil\ \rceil$: $\lfloor k/2 \rfloor + \lceil k/2 \rceil = k$.

- $|x|$: "absolute value of x" ($|x| = x$ if $x \ge 0$; $-x$ if $x < 0$, *e.g.*, $|5.67| = 5.67$, $|-2.01| = 2.01$)

  BEWARE! The same notation is used to represent the cardinality $|A|$ of a set A and the absolute value $|x|$ of a number x so be sure you are aware of the context in which it is used.

- $m \operatorname{div} n$: "the quotient of m divided by n" (integer division of m by n, *e.g.*, $5 \operatorname{div} 6 = 0$, $27 \operatorname{div} 4 = 6$, $-27 \operatorname{div} 4 = -6$)

  Properties: If $m, n > 0$, then $m \operatorname{div} n = \lfloor m/n \rfloor$
  $(-m) \operatorname{div} n = -(m \operatorname{div} n) = m \operatorname{div} (-n)$

- $m \operatorname{rem} n$: "the remainder of m divided by n" (*e.g.*, $5 \operatorname{rem} 6 = 5$, $27 \operatorname{rem} 4 = 3$, $-27 \operatorname{rem} 4 = -3$)

  Properties: $m = (m \operatorname{div} n) \cdot n + m \operatorname{rem} n$
  $(-m) \operatorname{rem} n = -(m \operatorname{rem} n) = m \operatorname{rem} (-n)$

- $m \bmod n$: "m modulo n" (*e.g.*, $5 \bmod 6 = 5$, $27 \bmod 4 = 3$ $-27 \bmod 4 = 1$)

  Properties: $0 \le m \bmod n < n$
  n divides $m - (m \bmod n)$.

- $\gcd(m, n)$: "greatest common divisor of m and n" (the largest positive integer that divides both m and n)

  For example, $\gcd(3, 4) = 1$, $\gcd(12, 20) = 4$, $\gcd(3, 6) = 3$

- $\operatorname{lcm}(m, n)$: "least common multiple of m and n" (the smallest positive integer that m and n both divide)

  For example, $\operatorname{lcm}(3, 4) = 12$, $\operatorname{lcm}(12, 20) = 60$, $\operatorname{lcm}(3, 6) = 6$

  Properties: $\gcd(m, n) \cdot \operatorname{lcm}(m, n) = m \cdot n$.

CALCULUS

Limits and Sums

An infinite sequence of real numbers $\{a_n\} = a_1, a_2, \ldots, a_n, \ldots$ *converges* to a limit $L \in \mathbb{R}$ if, for every $\varepsilon > 0$, there exists $n_0 \geq 0$ such that $|a_n - L| < \varepsilon$ for every $n \geq n_0$. In this case, we write $\lim\limits_{n\to\infty} a_n = L$ Otherwise, we say that the sequence *diverges*.

If $\{a_n\}$ and $\{b_n\}$ are two sequences of real numbers such that $\lim\limits_{n\to\infty} a_n = L_1$ and $\lim\limits_{n\to\infty} b_n = L_2$, then

$$\lim_{n\to\infty} (a_n + b_n) = L_1 + L_2 \quad \text{and} \quad \lim_{n\to\infty} (a_n \cdot b_n) = L_1 \cdot L_2.$$

In particular, if c is any real number, then

$$\lim_{n\to\infty} (c \cdot a_n) = c \cdot L_1.$$

The sum $a_1 + a_2 + \cdots + a_n$ and product $a_1 \cdot a_2 \cdots a_n$ of the finite sequence $a_1, a_2, \ldots, a_n$ are denoted by

$$\sum_{i=1}^{n} a_i \text{ and } \prod_{i=1}^{n} a_i.$$

If the elements of the sequence are all different and $S = \{a_1, a_2, \ldots, a_n\}$ is the set of elements in the sequence, these can also be denoted by

$$\sum_{a\in S} a \text{ and } \prod_{a\in S} a.$$

Examples:

- For any $a \in \mathbb{R}$ such that $-1 < a < 1$, $\lim\limits_{n\to\infty} a^n = 0$.
- For any $a \in \mathbb{R}^+$, $\lim\limits_{n\to\infty} a^{1/n} = 1$.
- For any $a \in \mathbb{R}^+$, $\lim\limits_{n\to\infty} (1/n)^a = 0$.
- $\lim\limits_{n\to\infty} (1 + 1/n)^n = e = 2.718281828459045 23536\ldots$

- For any $a, b \in \mathbb{R}$, the *arithmetic* sum is given by:

$$\sum_{i=0}^{n} (a + ib) = (a) + (a + b) + (a + 2b) + \cdots + (a + nb) = \frac{1}{2}(n + 1)(2a + nb).$$

- For any $a, b \in \mathbb{R}^+$, the *geometric* sum is given by:

$$\sum_{i=0}^{n} (ab^i) = a + ab + ab^2 + \cdots + ab^n = \frac{a(1 - b^{n+1})}{1 - b}.$$

EXPONENTS AND LOGARITHMS

Definition: For any $a, b, c \in \mathbb{R}^+$, $a = \log_b c$ if and only if $b^a = c$.

Notation: For any $x \in \mathbb{R}^+$, $\ln x = \log_e x$ and $\lg x = \log_2 x$.

For any $a, b, c \in \mathbb{R}^+$ and any $n \in \mathbb{Z}^+$, the following properties always hold.

- $\sqrt[n]{b} = b^{1/n}$
- $b^a b^c = b^{a+c}$
- $(b^a)^c = b^{ac}$
- $b^a / b^c = b^{a-c}$
- $b^0 = 1$
- $a^b c^b = (ac)^b$
- $b^{\log_b a} = a = \log_b b^a$
- $a^{\log_b c} = c^{\log_b a}$
- $\log_b(ac) = \log_b a + \log_b c$
- $\log_b(a^c) = c \cdot \log_b a$
- $\log_b(a/c) = \log_b a - \log_b c$
- $\log_b 1 = 0$
- $\log_b a = \log_c a / \log_c b$

BINARY NOTATION

A *binary number* is a sequence of bits $a_k \cdots a_1 a_0$ where each bit a_i is equal to 0 or 1. Every binary number represents a natural number in the following way:

$$(a_k \cdots a_1 a_0)_2 = \sum_{i=0}^{k} a_i 2^i = a_k 2^k + \cdots + a_1 2 + a_0.$$

For example, $(1001)_2 = 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 8 + 1 = 9$, $(01110)_2 = 8 + 4 + 2 = 14$.

Properties:

- If $a = (a_k \cdots a_1 a_0)_2$, then $2a = (a_k \cdots a_1 a_0 0)_2$, *e.g.*, $9 = (1001)_2$ so $18 = (10010)_2$.
- If $a = (a_k \cdots a_1 a_0)_2$, then $\lfloor a/2 \rfloor = (a_k \cdots a_1)_2$, *e.g.*, $9 = (1001)_2$ so $4 = (100)_2$.
- The smallest number of bits required to represent the positive integer n in binary is called the *length* of n and is equal to $\lceil \log_2(n+1) \rceil$.

Make sure you know how to add and multiply two binary numbers. For example, $(1111)_2 + (101)_2 = (10100)_2$ and $(1111)_2 \times 101)_2 = (1001011)_2$.

Proof Templates

Proof Outlines

LINE NUMBERS: Only lines that are referred to have labels (for example, L1) in this document. For a formal proof, all lines are numbered. Line numbers appear at the beginning of a line. You can indent line numbers together with the lines they are numbering or all line numbers can be unindented, provided you are consistent.

INDENTATION: Indent when you make an assumption or define a variable. Unindent when this assumption or variable is no longer being used.

1. **Implication**: Direct proof of A IMPLIES B.

   > L1. Assume A.
   > $\vdots$
   > L2. B

   A IMPLIES B; direct proof: L1, L2

2. **Implication**: Indirect proof of A IMPLIES B.

   > L1. Assume $NOT(B)$.
   > $\vdots$
   > L2. $NOT(A)$

   A IMPLIES B; indirect proof: L1, L2

3. **Equivalence**: Proof of A IFF B.

   > L1. Assume A.
   > $\vdots$
   > L2. B

   L3. A IMPLIES B; direct proof: L1, L2

   > L4. Assume B.
   > $\vdots$
   > L5. A

   L6. B IMPLIES A; direct proof: L4, L5
   A IFF B; equivalence: L3, L6

4. **Proof by contradiction** of A.

   > L1. To obtain a contradiction, assume $NOT(A)$.
   > $\vdots$
   >> L2. B
   >
   > $\vdots$
   >> L3. $NOT(B)$
   >
   > L4. This is a contradiction: L2, L3

   Therefore A; proof by contradiction: L1, L4

5. **Modus Ponens**.

   $\vdots$
   L1. A
   $\vdots$
   L2. A IMPLIES B
   B; modus ponens: L1, L2

6. **Conjunction:** Proof of A AND B:

   $\vdots$
   L1. A
   $\vdots$
   L2. B
   A AND B; proof of conjunction; L1, 2

7. **Use of Conjunction**:

   $\vdots$
   L1. A AND B
   A; use of conjunction: L1
   B; use of conjunction: L1

8. **Implication with Conjunction**: Proof of (A_1 AND A_2) IMPLIES B.

   > L1. Assume A_1 AND A_2.
   > A_1; use of conjunction, L1
   > A_2; use of conjunction, L1
   > $\vdots$
   > L2. B

   (A_1 AND A_2) IMPLIES B; direct proof, L1, L2

9. **Implication with Conjunction**: Proof of A IMPLIES (B_1 AND B_2).

   > L1. Assume A.
   > $\vdots$
   > L2. B_1
   > $\vdots$
   > L3. B_2
   > L4. B_1 AND B_2; proof of conjunction: L2, L3

   A IMPLIES (B_1 AND B_2); direct proof: L1, L4

10. **Disjunction**: Proof of A OR B and B OR A.

    $\vdots$
    L1. A
    A OR B; proof of disjunction: L1
    B OR A; proof of disjunction: L1

11. **Proof by cases**.

    L1. C OR $NOT(C)$ tautology
    L2. Case 1: Assume C.

    - $\vdots$
    - L3. A

    L4. C IMPLIES A; direct proof: L2, L3
    L5. Case 2: Assume $NOT(C)$.

    - $\vdots$
    - L6. A

    L7. $NOT(C)$ IMPLIES A; direct proof: L5, L6
    A proof by cases: L1, L4, L7

12. **Proof by cases** of A OR B.

    L1. C OR $NOT(C)$ tautology
    L2. Case 1: Assume C.

    - $\vdots$
    - L3. A
    - L4. A OR B; proof of disjunction, L3

    L5. C IMPLIES (A OR B); direct proof, L2, L4
    L6. Case 2: Assume $NOT(C)$.

    - $\vdots$
    - L7. B
    - L8. A OR B; proof of disjunction, L7

    L9. $NOT(C)$ IMPLIES (A OR B); direct proof: L6, L8
    A OR B; proof by cases: L1, L5, L9

13. **Implication with Disjunction**: Proof by cases of (A_1 OR A_2) IMPLIES B.

    L1. Case 1: Assume A_1.

    - $\vdots$
    - L2. B

    L3. A_1 IMPLIES B; direct proof: L1,L2
    L4. Case 2: Assume A_2.

    - $\vdots$
    - L5. B

    L6. A_2 IMPLIES B; direct proof: L4, L5
    (A_1 OR A_2) IMPLIES B; proof by cases: L3, L6

14. **Implication with Disjunction**: Proof by cases of A IMPLIES (B_1 OR B_2).

    L1. Assume A.
    L2. C OR NOT(C) tautology
    L3. Case 1: Assume C.
        ⋮
        L4. B_1
        L5. B_1 OR B_2; disjunction: L4
    L6. C IMPLIES (B_1 OR B_2); direct proof: L3, L5
    L7. Case 2: AssumeNOT(C).
        ⋮
        L8. B_2
        L9. B_1 OR B_2; disjunction: L8
    L10. NOT(C) IMPLIES (B_1 OR B_2); direct proof: L7, L9
    L11. B_1 OR B_2; proof by cases: L2, L6, L10

    A IMPLIES (B_1 OR B_2): direct proof. L1, L11

15. **Substitution of a Variable in a Tautology**:
    Suppose P is a propositional variable, Q is a formula, and R' is obtained from R by replacing *every* occurrence of P by (Q).

    L1. R tautology
    R'; substitution of all P by Q: L1

16. **Substitution of a Formula by a Logically Equivalent Formula**:
    Suppose S is a subformula of R and R' is obtained from R by replacing *some* occurrence of S by S'.

    L1. R
    L2. S IFF S'
    L3. R'; substitution of an occurrence of S by S': L1, L2

17. **Specialization**:

    L1. $c \in D$
    L2. $\forall x \in D.P(x)$
    $P(c)$; specialization: L1, L2

18. **Generalization**: Proof of $\forall x \in D.P(x)$.

    L1. Let x be an arbitrary element of D.
    ⋮
    L2. $P(x)$

    Since x is an arbitrary element of D,
    $\forall x \in D.P(x)$; generalization: L1, L2

19. **Universal Quantification with Implication**: Proof of $\forall x \in D.(P(x)$ IMPLIES $Q(x))$.

```
    L1. Let $x$ be an arbitrary element of $D$.
        L2. Assume $P(x)$
        ⋮
        L3. $Q(x)$
    L4. $P(x)$ IMPLIES $Q(x)$; direct proof: L2, L3
Since $x$ is an arbitrary element of $D$,
$\forall x \in D.(P(x)$ IMPLIES $Q(x))$; generalization: L1, L4
```

20. **Implication with Universal Quantification**: Proof of $(\forall x \in D.P(x))$ IMPLIES A.

```
    L1. Assume $\forall x \in D.P(x)$.
    ⋮
    L2. $a \in D$
    $P(a)$; specialization: L1, L2
    ⋮
    L3. $A$
Therefore $(\forall x \in D.P(x))$ IMPLIES $A$; direct proof: L1, L3
```

21. **Implication with Universal Quantification**: Proof of A IMPLIES $(\forall x \in D.P(x))$.

```
    L1. Assume $A$.
        L2. Let $x$ be an arbitrary element of $D$.
        ⋮
        L3. $P(x)$
    Since $x$ is an arbitrary element of $D$,
    L4. $\forall x \in D.P(x)$; generalization, L2, L3
$A$ IMPLIES $(\forall x \in D.P(x))$; direct proof: L1, L4
```

22. **Instantiation**:

```
L1. $\exists x \in D.P(x)$
    Let $c \in D$ be such that $P(c)$; instantiation: L1
    ⋮
```

23. **Construction**: Proof of $\exists x \in D.P(x)$.

```
    L1. Let $a = \cdots$
    ⋮
    L2. $a \in D$
    ⋮
    L3. $P(a)$
$\exists x \in D.P(x)$; construction: L1, L2, L3
```

24. **Existential Quantification with Implication**: Proof of $\exists x \in D.(P(x)$ IMPLIES $Q(x))$.

```
    L1. Let $a = \cdots$
    ⋮
    L2. $a \in D$
        L3. Suppose $P(a)$.
        ⋮
        L4. $Q(a)$
    L5. $P(a)$ IMPLIES $Q(a)$; direct proof: L3, L4
$\exists x \in D.(P(x)$ IMPLIES $Q(x))$; construction: L1, L2, L5
```

25. **Implication with Existential Quantification**: Proof of $(\exists x \in D.P(x))$ IMPLIES A.

```
    L1. Assume $\exists x \in D.P(x)$.
        Let $a \in D$ be such that $P(a)$; instantiation: L1
        ⋮
        L2. $A$
$(\exists x \in D.P(x))$ IMPLIES $A$; direct proof: L1, L2
```

26. **Implication with Existential Quantification**: Proof of A IMPLIES $(\exists x \in D.P(x))$.

```
    L1. Assume $A$.
        L2. Let $a = \cdots$
        ⋮
        L3. $a \in D$
        ⋮
        L4. $P(a)$
    L5. $\exists x \in D.P(x)$; construction: L2, L3, L4
$A$ IMPLIES $(\exists x \in D.P(x))$; direct proof: L1, L5
```

27. **Subset**: Proof of $A \subseteq B$.

```
    L1. Let $x \in A$ be arbitrary.
    ⋮
    L2. $x \in B$
    The following line is optional:
    L3. $x \in A$ IMPLIES $x \in B$; direct proof: L1, L2
$A \subseteq B$; definition of subset: L3 (or L1, L2, if the optional line is missing)
```

28. **Weak Induction**: Proof of $\forall n \in N.P(n)$

    Base Case:

    $\vdots$

    L1. $P(0)$

        L2. Let $n \in N$ be arbitrary.

            L3. Assume $P(n)$.

            $\vdots$

            L4. $P(n+1)$

        The following two lines are optional:

        L5. $P(n)$ IMPLIES $(P(n+1)$; direct proof of implication: L3, L4

    L6. $\forall n \in N.(P(n)$ IMPLIES $P(n+1))$; generalization L2, L5

    $\forall n \in N.P(n)$ induction; L1, L6 (or L1, L2, L3, L4, if the optional lines are missing)

29. **Strong Induction**: Proof of $\forall n \in N.P(n)$

        L1. Let $n \in N$ be arbitrary.

            L2. Assume $\forall j \in N.(j < n$ IMPLIES $P(j))$

            $\vdots$

            L3. $P(n)$

        The following two lines are optional:

        L4. $\forall j \in N.(j < n$ IMPLIES $P(j))$ IMPLIES $P(n)$; direct proof of implication: L2, L3

    L5. $\forall n \in N.[\forall j \in N.(j < n$ IMPLIES $P(j))$ IMPLIES $P(n)]$; generalization: L1, L4

    $\forall n \in N.P(n)$; strong induction: L5 (or L1, L2, L3, if the optional lines are missing)

30. **Structural Induction**: Proof of $\forall e \in S.P(e)$, where S is a recursively defined set

    Base case(s):

        L1. For each base case e in the definition of S

        L2. $P(e)$.

    Constructor case(s):

        L3. For each constructor case e of the definition of S,

            L4. assume $P(e')$ for all components e' of e.

            $\vdots$

            L5. $P(e)$

    $\forall e \in S.P(e)$; structural induction: L1, L2, L3, L4, L5

31. **Well Ordering Principle**: Proof of $\forall e \in S.P(e)$, where S is a well ordered set, i.e. every nonempty subset of S has a smallest element.

```
L1. To obtain a contradiction, suppose that $\forall e \in S.P(e)$ is false.
L2. Let $C = \{e \in S \mid P(e) \text{ is false}\}$ be the set of counterexamples to $P$.
L3. $C \neq \phi$; definition: L1, L2
    L4. Let $e$ be the smallest element of $C$; well ordering principle: L2, L3
        Let $e' = \cdots$
        $\vdots$
        L5. $e' \in C$
        $\vdots$
        L6. $e' < e$.
L7. This is a contradiction: L4, L5, L6
$\forall e \in S.P(e)$; proof by contradiction: L1, L7
```

Index

Index

Bibliography

Courses

[1] Allan Borodin. *CSC364*. University of Toronto. 2004.

[2] Allan Borodin. *CSC373 Spring 2011*. University of Toronto. 2011.

[3] Stephen A. Cook. *CSC463 Computational Complexity and Computability Winter 2018*. University of Toronto. 2018.

[5] Faith Ellen. *CSC240 Enriched Introduction to Theory of Computation Winter 2021*. University of Toronto. 2021.

[6] Faith Ellen. *CSC265 Enriched Data Structures and Analysis Fall 2021*. University of Toronto. 2021.

[8] Shubhangi Saraf. *CSC463 Computational Complexity and Computability Winter 2022*. University of Toronto. 2022.

Books

[4] Thomas H. Cormen et al. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 0262033844.

[7] Jon Kleinberg and Éva Tardos. *Algorithm Design*. 1st. Pearson, 2005. ISBN: 978-0321295354.

[9] Michael Sipser. *Introduction to the Theory of Computation*. Third. Boston, MA: Course Technology, 2013. ISBN: 113318779X.

Journal Articles