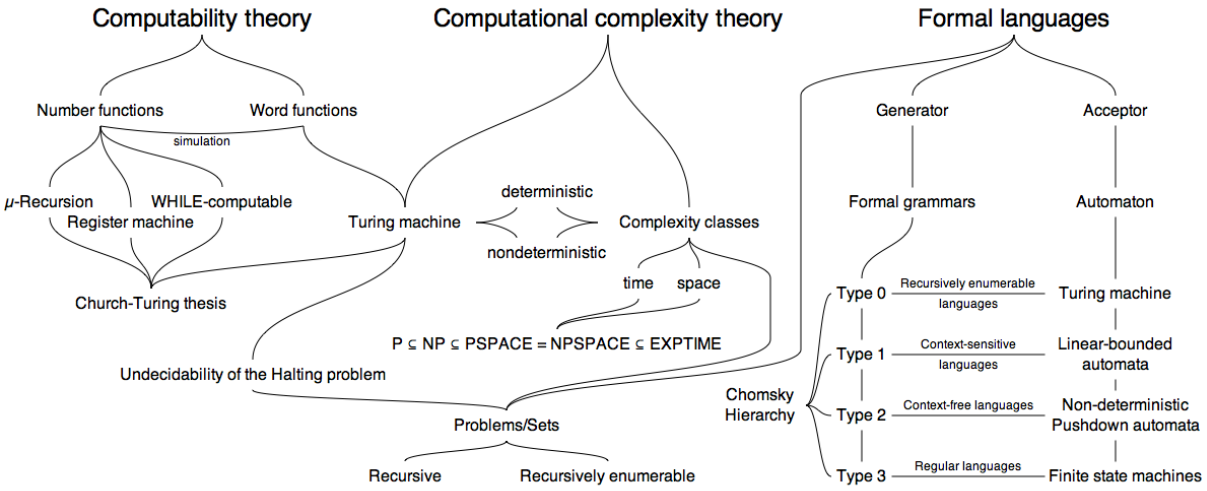
CS 373
a.k.a. Theory of Computation

Spring 2012, University of Illinos
as taught by Steve LaValle

notes by Michele R. Esposito

Please, let report any error or type-o at micheleresposito@gmail.com

Contents



1 Basic Definitions

1.1 String

A **string over an alphabet** is a finite sequence of symbols form that alphabets.

1.2 Language

A **language** is a set of strings.

1.3 Alphabet

A finite set of object called symbols

1.4 Empty string

The string of length zero

2 Regular Languages

2.1 DFAs

A **finite automaton** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where,

1. Q is a finite set called the **states**
2. Σ is a finite set called the **alphabet**.
3. $\delta : Q \times \Sigma \to Q$ is the **transition function**.
4. $q_0 \in Q$ is the **start state**
5. $F \subseteq Q$ is the **set of accept states**.

We say that a machine M accepts w if a sequence of states $r0, r1, \ldots, r_n \in Q$ exists with three conditions:

1. $r_0 = q_0$
2. $\delta(r_i, w_{i+1}) = r_{i+1}$, for $i = 0, \ldots, n-1$.
3. $r_n \in F$.

We say that M **recognizes language** A if $A = \{w | M \text{ accepts } w\}$.

2.2 Regular language

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

2.3 The regular operations

Let A and B be languages. We define the regular operations **union,concatenation,** and **star** as follows:

- **Union** $A \cup B = \{x | x \in A \text{ or } x \in B\}$.
- **Concatenation** $AB = \{xy | x \in A \text{ and } y \in B\}$.
- **Star**: $A^* = \{x_1 x_2 \ldots x_k | k \geq 0 \text{ and each } x_i \in A\}$. Otherwise, can be defined recursively as: $V_0 = \epsilon$, then $V_{i+1} = \{wv | w \in V_i \text{ and } v \in V\}$ where $i \geq 0$. Then, $V^* = \bigcup_{i \in \mathbb{N}} V_i$. REMEMBER! $\varnothing^* = \{\epsilon\}$

The set of regular languages is closed under the following operations: (taken from Wikipedia)

1. Union: $K \cup L$.
2. Intersection: $K \cup L$.
3. Complement: K^c.
4. Difference: $K - L$.
5. Kleene star: K^*.
6. String homomorphism, inverse homomorphism. See section 2.8 for details.
7. Reverse: K^R.
8. Quotient: $L_1 \backslash L_2$. See section 2.8 for details.

2.4 NFAs

A **NFA** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set called the **states**
2. Σ is a finite set called the **alphabet**.
3. $\delta : Q \times \Sigma_\epsilon \to 2^Q$ is the **transition function**.

4. $q_0 \in Q$ is the **start state**
5. $F \subseteq Q$ is the **set of accept states**.

2^Q is the power set.

2.5 Regular expression

Say that R is a **regular expression** if R is

1. a for some a in the alphabet Σ.
2. ϵ.
3. $\varnothing$
4. $R_1 \cup R_2$ where R_1 and R_2 are regular expressions
5. $R_1 R_2$ where R_1 and R_2 are regular expressions
6. R_1^* where R_1 is a regular expression.

2.6 GNFAs

A **generalized nondeterministic finite automaton** is 5-tuple $(Q, \Sigma, \delta, q_{start}, q_{accept})$, where

1. Q is a finite set called the **states**
2. Σ is a finite set called the **alphabet**.
3. $\delta : Q \times \Sigma_\epsilon \to R$ is the **transition function**.
4. $q_0 \in Q$ is the **start state**
5. $F \subseteq Q$ is the **set of accept states**.

2.7 Pumping lemma

if A is a regular language, then there is a number p (the pumping length) where, if s is any string in A of length at least p, then s may be divided into three pieces, $s = xyz$, satisfying the conditions:

1. for each $i \geq 0$, $xy^iz \in A$,
2. $|y| > 0$, and
3. $|xy| \leq p$.

Imagine there is an "adversary". He gets to pick: p, breaks s into xyz. You get to pick s and i. Use to prove irregularity by contradiction.

2.8 Closure Proprieties of Regular Sets

Substitution
The class of regular sets has the interesting propriety that it is closed under substitution. $\forall a \in \Sigma$ of a regular subset R, let R_a be a particular regular set. Suppose that we replace each word $a_1 a_2 \dots a_n \in R$ by the set of words of the form $w_1 w_2 \dots w_n$, where w_i is an *substitution* mapping of an alphabet Σ onto subset of Δ^*. The mapping f is extended to the strings as follows:

1. $f(\epsilon) = \epsilon$;
2. $f(xa) = f(x)f(a)$.

The mapping f is extended to the languages by defining $f(L) = \bigcup_{x \in L} f(x)$.
Homomorphism
A homomorphism h is a substitution s.t. $h(a)$ contains a single string for each a. We generally take $h(a)$ to be the string itself, rather than the set containing that string.
Inverse homomorphism
An *inverse homomorphism* of a language L be:

$$h^{-1}(L) = \{x | h(x) \in L\}$$

Or, using a string:

$$h^{-1}(w) = \{x | h(x) = w\}$$

Quotients of languages
Define the *quotient* of languages L_1, L_2, written L_1, L_2 to be:

$$\{x | \exists y \in L_2 \text{ such that } xy \in L_1\}.$$

2.9 Myhill-Nerode Theorem

Right invariant
An equivalence relation R s.t. xRy implies $xzRyz$ is said to be *right invariant*. We see that every finite automaton induces a right invariant equivalence relation, on its set of input strings.
Myhill-Nerode Theorem
The following three statements are equivalent:

1. The set $L \subseteq \Sigma^*$ is accepted by some finite automaton.
2. L is the union of some of the equivalence classes of a right invariant equivalence relation of finite index.
3. Let equivalence relation R_l be defined by: xR_Ly if and only if $\forall z \in \Sigma^*$, xz is in L exactly when yz is in L. Then R_L is of finite index.

We can use the Myhill-Nerode theorem to prove nonregularity. For a language A over Σ, the relation A has a finite number of equivalence classes iff A is regular. This, we can prove nonregularity by:

1. Choose some infinite set $S \subseteq \Sigma^*$;

2. Show that $x, y \in S, x \neq y$ belongs to different equivalence classes. Thus, $\exists z$ s.t. $xz \in A$, but $yz \notin A$.

3. Since S is infinite and every member has a equivalence class, A, does not have a finite index. By the Myhill-Nerode theorem, A is nonregular. (condition 3).

3 Context-Sensitive Languages

3.1 Context-free grammars

A **context-free grammar** is a 4-tuple (V, Σ, R, S), where

1. V is a finite set called the *variables*, or *nonterminals*,
2. Σ is a finite set, disjoint from V, called the **terminals**,
3. R is a finite set of **rules**, with each rule being a variable and string of variables and terminals, and
4. $S \in V$ is the start variable.

If u, v and w are strings of variables and terminals, and $A \to w$ is a rule of the grammar, we say that uAv **yields** uwv, written $uAv \implies uwv$. Say that u **derives** v, written $u \implies v$, if $u = v$ or if a sequence $u_1, u_2, \ldots, u_k$ exists for $k \geq 0$ and

$$u \implies u_1 \implies u_2 \implies \ldots \implies u_k \implies v$$

The **language of the grammar** is $\{w \in \Sigma^* | S \implies {}^* w\}$.

Ambiguity

A string w is derived **ambiguously** in context-free grammar G is it has two or more different leftmost derivations. Grammar G is **ambiguous** if it generates some string ambiguously.

Chomsky Normal Form

Chomsky normal form is useful in giving algorithms for working with context-free grammars. Also, is good for parsing, because it produces a binary tree.

A context-free grammar is in **Chomsky normal form** if every rule is of the form

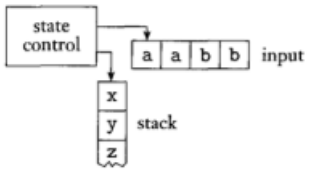
$$A \to BC$$
$$A \to a$$

where a is any terminal, and $A, B,$ and C are any variables- except that B and C may not be the start variable. In addition we permit the rule $S \to \epsilon$, where S is the start variable.

Any context-free language is generated by a context-free grammar in Chomsky normal form.

3.2 Pushdown Automata

A pushdown automata is a type of NFA, but they have an extra component called stack, which allows them to have infinite memory. They have the same power as context-free grammar, as we will see later. Here is a schematic representation of a PDA.



Now, here is a formal definition of a PDA. A **pushdown automaton** is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$ where Q, Σ, Γ and F are all finite sets, and

1. Q is the set of states,
2. Σ is the input alphabet,
3. Γ is the stack alphabet,
4. $\delta : Q \times \Sigma_\epsilon \times \Gamma_\epsilon \to 2^{Q \times \Gamma_\epsilon}$ is the transition function,
5. $q_o \in Q$ is the start state, and
6. $F \subseteq Q$ is the set of accept states.

A pushdown automaton $M = (Q, \Sigma, \Gamma, \delta, q_o, F)$ computes as follows, It accepts input w if w can be written as $w = w_1 w_2 \dots w_m$, where each $w_i \in \Sigma_\epsilon$ and sequences of states $r_0, r_1, \dots, r_m \in Q$ and string $s_0, s_1, \dots, s_m \in \Gamma^*$ exist that satisfy the following three conditions. The string s_i represents the sequence of stack contents that M has on the accepting branch of the computation.

1. $r_0 = q_0$ and $s_0 = \epsilon$. This condition signifies that M starts out properly, in the start state and with an empty stack.
2. For $i = 0, \dots, m-1$, have $(r_{i+1}, b) \in \delta(r_i, w_{i+1}, a)$, where $s_i = at$ and $s_{i+1} = bt$ for some $a, b \in \Gamma_\epsilon$ and $t \in \Gamma^*$. This condition states that M moves properly according to the state, stack, and next input symbol.
3. $r_m \in F$. This condition states that an accept state occurs at the input end.

3.3 Pumping Lemma for CFL

If A is a context-free language, then there is a number p where, if s is any string in A of the length at least p, then s may be divided into five pieces $s = uvxyz$ satisfying the conditions:

1. for each $i \geq 0, uv^i xy^i z \in A$,
2. $|vy| > 0$, and
3. $|vxy| \leq p$.

3.4 Closure Properties of CFL's

Context free languages are closed under the following operations:

1. Union
2. Concatenation
3. Kleene star
4. Substitution
5. Homomorphism, and inverse homomorphism
6. Reverse.

3.5 Context-Sensitive grammars

A **context-sensitive grammar** is a formal grammar in which the left-hand sides and right-hand sides of any production rules may be surrounded by a context of terminal and nonterminal symbols.

1. V is a finite set of variables
2. Σ is finite set of terminals, and $\Sigma \cap V = \varnothing$.
3. R is a finite set of rules of the form $\alpha A \beta \to \alpha \gamma \beta$ in which A is a variable and α, β, γ are strings of terminals and variables.
4. $S \in V$ is the start variable.

3.6 CYK Algorithm

In order to test for membership of a certain string to a language, we can use the CYK algorithm. In order to do so, we first need to write the grammar in CNF. Then, given x of length $n \geq 1$ and a grammar G, determine for each i and j and for each variable A, whether $A \implies {}^{*}x_{ij}$, where xij is the substring of x of length j beginning at position i. To state the CYK.

4 Turing Machines

A Turing machine is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ with Q, Σ, Γ finite, nonempty sets.

1. Q is a set of states
2. Σ is the input alphabet, which does not contains the blank symbol.
3. Γ is the tape alphabet, in which $blak \in \Gamma$, and $\Sigma \subset \Gamma$.
4. $\delta : Q \times \Gamma \to Q \times \Gamma \{L, R\}$ is the transition function.
5. $q_0 \in Q$ is the start state.
6. $q_{accept} \in Q$ is the accept state

7. $q_{reject} \in Q$ is the reject state, and $q_{accept} \neq q_{reject}$.

A Turing machine M **accepts** input w if a sequence of configurations $C_1, C_2, \ldots_k$ exists where

1. C_1 is the start configuration of M of input w,
2. each C_i yields C_{i+1}, and
3. C_k is an accepting configuration.

The collection of strings that M accepts if the **language of M**, denoted $L(M)$. Call a language **Turing-recognizable** if some Turing machine recognizes it. That is, the machine ends on accept states, or keeps looping. Therefore, a TM recognizes A if

- Accepts it $x \in A$
- Loops it $x \notin A$

Call a language **Turing-decidable** if some Turing machine decides it. That is, the machine always ends in an accept state. Therefore, a TM decides A if

- Accepts if $x \in A$
- Rejects if $x \notin A$

4.1 Variants of Turing machines

There are some alternatives version of Turing machines. We call those **variants** of Turing machine, and we analyze the power. For more details. In more detail, we look at two variants:

4.1.1 Multitape Turing machine

A **multitape TM** is an ordinary TM with several tapes. WE define the transition function to be

$$\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, R, S\}^k \tag{1}$$

Where k is the number of tapes. the expression $\delta(q_1, a_1, \ldots, a_k) = (q_j, b_1, \ldots, b_k, L, R, \ldots, L)$ means that if the machine is in state q_i and heads 1 through k are reading symbols a_1 through a_k, then machine goes to state q_j, writes symbols b_1 through b_k, and directs each head with proper movement.
Theorem: Every multitape Turing machine has an equivalent single-tape Turing machine. Look at lecture notes for proof detail.

4.1.2 Nondeterministic Turing Machines

A NTM is a TM which at any point in computation can move in several possible ways. The transition function has the form

$$\delta : Q \times \Gamma \to 2^{Q \times \Gamma \times \{L,R\}} \tag{2}$$

The computation of a NTM is a tree whose branches correspond to $\neq$ possibilities for the machine. If some branch of the computation leads to the accept state, the machine accepts its input.
Theorem: Every Nondeterministic Turing machine has an equivalent deterministic TM. Look at lecture notes for proof detail.

4.1.3 Enumerators

An enumerator is a TM with an "attached printer", which is used to output strings. An enumerator E starts with a blank input tape. If the enumerator doesn't halt, it may print infinite strings. The language enumerated by E is a collection of all the strings that it eventually prints out. Moreover, E may generate the strings of the language in a ny order, possibly with repetitions.
Theorem: A language is Turing-recognizable iff some enumerator enumerates it.

4.2 Definition of algorithms

The **Church-Turing** thesis provides a definition of an algorithm. That is, the intuitive notion of algorithms equals Turing machine algorithms.

5 Decidability

We now begin to investigate the power of algorithms. In fact, in this section we will look at which problems are solvable by using the power of algorithms.

5.1 Decidable languages

The following languages, and machines, can be simulated into a universal Turing machine. Here are the machines, with a descriptions of the algorithms.

(a) $\mathbf{A_{DFA}} = \{< B, w > | B$ if a DFA that accepts input string $w\}$. Then, A_{DFA} is decidable with the following algorithm:
$M =$"on input$< B, W >$, where B is a DFA and w is a string:

1. Simulate B on input w.
2. if the simulation ends in an accept state, *accept*, If it ends in a non accepting state, *reject*.

(b) An $\mathbf{A_{NFA}} = \{< B, w > | B$ is an NFA that accepts input string $w\}$ is a decidable language. Here is a turing machine M that accepts it:
$N =$"On input $< B, w >$ where B is an NFA, and w is a string:

1. Convert NFA B to an equivalent $DFAC$.
2. Simulate B on input w.
3. if the simulation ends in an accept state, *accept*, If it ends in a non accepting state, *reject*.

(c) $\mathbf{A_{REX}} = \{< R, w > |$ R is a regular expression that generates string $w\}$.
$P =$" On input $< R, w >$ where R is a regular expression and w is a string:

1. Convert regular expression R to an equivalent DFA A by using the procedure for for this conversion.
2. Simulate B on input w.
3. if the simulation ends in an accept state, *accept*, If it ends in a non accepting state, *reject*.

(d) $\mathbf{E_{DFA}} = \{< A > | A \text{ is a DFA and } L(A) = \varnothing\}$ is a decidable language
$T =$"On input $< A >$ where A is a DFA:

1. Mark the start state of A.
2. Repeat until no new states get marked:
3. Mark any state that has a transition coming into it from any state that is already marked
4. If no accept state is marked, *accept*; otherwise *reject*."

(e) $\mathbf{A_{CFG}} = \{< G, w > | G \text{ is a CFG that generates string} w\}$. Define the TM S for A_{CFG} as follow:
$S =$"On input $< G, w >$, where G is a CFG and w is a string:

1. Convert G to an equivalent grammar in Chomsky normal form
2. List all derivations with $2n - 1$ steps, where $n = |w|$. If $n = 0$, then list all derivations with 1 step.
3. If any of these derivations generate w, *accept*; if not *reject*."

(f) $\mathbf{E_{CFG}} = \{< A > | A \text{ is a CFG and } L(A) = \varnothing\}$ is a decidable language.

(g) Every context-free language is decidable.

5.2 The Halting problem

Let $A_{TM} = \{< M, w > | M \text{ is a TM and M accepts } w\}$. Then, A_{TM} is undecidable. This proof, which is in the lecture notes, is based on construction a decider H and D that, when they run on the description of themselves, they will lead to a contradiction. This is a rough sketch of the machine, and of the proof:
Let H be a decider such that

$$H(< M, w >) = \begin{cases} accept & \text{if M accepts} w \\ reject & \text{if M does not} w \end{cases} \tag{3}$$

Now, construct D, having H as a subroutine:
$D =$ "On input ¡M¿, where M is a TM:

1. Run H on input $< M, < M >>$.
2. Output the opposite of what H outputs.

Therefore, we have the following steps:

- H accepts $< M, w >$ exactly when M accepts w.
- D rejects $< M >$ exactly when M accepts $< M >$.
- D rejects $< D >$ exactly when D accepts $< D >$.

Note that the final step obviously leads to a contradiction, therefore the machines H and D cannot exists.

5.3 A Turing unrecognizable language

Recall that the complement of a language is the language consisting of all strings that re not in the language. We say that a language is **co-Turing-recognizable** if it the complement of a Turing-recognizable language.
Theorem: A language is decidable iff it is Turing-recognizable and co-Turing-recognizable.
That is, a language is decidable exactly when both it and its complement are Turing-recognizable. Corollary: $\bar{A_{TM}}$ is not Turing-recognizable.
Proof: We know that A_{TM} is Turing-recognizable. If $\bar{A_{TM}}$ also were Turing-recognizable, A_{TM} would be decidable. Theorem 4.11 tells us that A_{TM} is not decidable, so $\bar{A_{TM}}$ must not be Turing-recognizable

6 Reducibility

A **reduction** is a ways of converting one problem into another problem in such a way that a solution to the second problem can be used to solve the first problem. Reducibility plays an important role in classifying problems by decidability and later in complexity theory as well.

6.1 Undecidable problems from language theory

Let's try to establish the undecidability of few problems. Let $HALT_{TM}$ be:

$$HALT_{TM} = \{< M, w > \mid \text{M is a TM and M halt on input } w\}. \qquad (4)$$

Then, we can establish that $HALT_{TM}$ is undecidable, with the following proof: Let's assume for the purposes of obtaining a contradiction that TM R decides $HALT_{TM}$. We construct TMS to decide A_{TM}, with S operating as follows.
$S =$ "On input $< M, w >$, and encoding of a TMM and a string w:

1. Run TM R on input $< M, w >$.

2. If R rejects, reject.

3. If R accepts, simulate M on w until it halts.

4. If M has accepted, accept; if M has rejected, reject."

Clearly, if R decides $HALT_{TM}$, then S decides A_{TM}. Then, $HALT_{TM}$ must be undecidable.

The following languages are all undecidable:

- $\mathbf{E_{TM}} = \{< M > |\text{M is a TM and L(M)} = \varnothing\}$. Here is the proof.
  First, construct a machine M_1 so that:
  $M_1 =$"On input x:

  1. if $x \neq w$, reject
  2. if $x = w$, run M on input w and accept if M does".

  Now, assume that TM R decides E_{TM} and construct TM S that decides A_{TM} as follows:
  $S =$"On input $< M, w >$, and encoding of a TM M and a string w:

1. Use the description of M and w and construct the TM M_1 just described
2. Run R on input $< M_1 >$
3. If R accepts, reject; if R reject, accept.

- **REGULAR$_{\mathbf{TM}}$** $= \{< M > | M$ is a TM and L(M) is regular language$\}$ is undecidable.

  Proof idea: We assume that $REGULAR_{TM}$ is decidable by a TM R and use this assumption to construct a TM S that decides A_{TM}. The idea is for S to take its input $< M, w >$ and modify M so that the resulting TM recognizes a regular language iff M accepts w. We call the modified machine M_2. We design M_2 to recognize the nonregular language $\{0^n 1^n | n \geq 0\}$ if M does not accept w.
  Proof: We let R be a TM that decides $REGULAR_{TM}$ and construct TM S to decide A_{TM}. Then S works in the following manner:
  $S =$ "On input $< M, w >$, where M is a TM and w is a string:

  1. Construct the following TM M_2.
  $M_2 =$'On input x:

  (a) If x has the form $0^n 1^n$, *accept*.
  (b) If x does not have this form, run M on input w and accept if M accepts w.'

  2. Run R on input $< M_2 >$.

  3. If R accepts, *accept*; if R rejects, *reject*."

- **EQ$_{\mathbf{TM}}$** $= \{< M_1, M_2 > | M_1$ and M_2 are TMs and$L(M_1) = L(M_2)\}$ is undecidable
  Proof Idea: Show that if EQ_{TM} were decidable, E_{TM} also would be decidable, by giving a reduction from E_{TM} to EQ_{TM}. If one of these languages happens to be $\varnothing$, we end up with the problem of determining whether the language of the other machine is empty- that is, the E_{TM} problem. So in a sense, the E_{TM} problem is a special case of the EQ_{TM}
  Proof: We let TM R decide EQ_{TM} and construct TM S to decide E_{TM} as follows.
  $S =$"On input $< M >$, where M is a TM:

  1. Run R on input $< M, M_1 >$, where M_1 is a TM that rejects all inputs.

  2. If R accepts, *accept*; if R rejects, *reject*."

  If R decides EQ_{TM}, S decides E_{TM}, which is a contradiction.

6.2 Mapping reducibility

The intent of this section is that of formalizing reducibility. Doing so allows us to use reducibility in more refined ways, such as for proving that certain languages are not Turing-recognizable and for applications is complexity theory.
Being able to reduce problem A to problem B by using a mapping reducibility means that a computable function exists that converts instances of problem A to instances of problem B. If we have such a conversion function, called a *reduction*, we can solve by first using the reduction to convert it to an instance of B and then applying to the solver for B.

6.2.1 Computable functions

A Turing machine computes a function by starting with the input to the function on the tape and halting with the output of the function on the tape.
Definition: a function $f : \Sigma^* \to \Sigma^*$ is a **computable function** if some Turing machine M, on every input w, halts with just $f(w)$ on its tape.

6.2.2 Formal definition of mapping reducibility

Language A is **mapping reducible** to a language B, written $A \leq_m B$, if there is a computable function $f : \Sigma^* \to \Sigma^*$, where for every w,

$$w \in A \leftrightarrow f(w) \in B \tag{5}$$

The function f is called the **reduction** of A to B.
The test whether $w \in A$, we use the reduction f to map w to $f(w)$ and test whether $f(w) \in B$. The term *mapping reduction* comes from the function or mapping that provides the means of doing the reduction.
Theorem:
If $A \leq_m B$ and B is decidable, then A is decidable.
Also, its contrapositive works:
If $A \leq_m B$ and A is undecidable, then B is undecidable. We can computer the reduction $A_{TM} \leq_M HALT_{TM}$ with the following TM F that computers a reduction f:
$F =$"On input $< M, w >$:

1. Construct the following machine M'.
$M' =$"On input x:

(a) Run M on x.

(b) If M accepts, *accept*.

(c) If M rejects, enter a loop.'

2. Output $< M', w >$."

So, if a language is in A, then it will also be in $HALT$, because we accept. However, if the language is not in A, then we reject, and we keep halting in $HALT$.

6.3 Rice's theorem

Rice's Theorem states that, for any non-trivial property of partial functions, there is no general and effective method to decide whether an algorithm computes a partial function with that property. Formally states: Let P be any property about TMs that so

1. For every TMs M_1 and M_2, such that $L(M_1) = L(M_2)$, $< M_1 > \in P$ iff $< M_2 > \in P$
2. There exists TMs M_1 and M_2 for which $< M_1 > \in P$ and $< M_2 > \notin P$.

Trivial means that every TM has that property. For example, the language of a Turing machine has characters is trivial.
That is, a Turing machine cannot decide whether another Turing machine has that property. We have $L(M_1) = L(M_2)$ because we care about every Turing machine that has that language, regardless of its implementation.

7 Complexity Theory

Even if a problem is decidable, we want to be able to analyze the time and space that are required to solve that problem. Our objective in this chapter is to present the basic time requirements to solve such a problem.

7.1 Measuring complexity

We analyze the algorithms to determine how much time it uses. The number of steps that an algorithm uses on a particular input may depend on several parameters. For instance, if the input is a graph, the number of steps may depend on the number of steps may depend on the number of nodes. We compute the running time of an algorithm purely as a function of length of the string representing the input and don't consider any other parameters. In **worst-case analysis**, we consider the longest possible running time. In **average-case analysis**, we consider the average of all the running times of inputs of a particular length.

7.1.1 Running time

Let M be a deterministic TM that halts on all inputs. The **running time** or **time complexity** of M is the function $f : \mathbb{N} \to \mathbb{N}$, where $f(n)$ is the maximum number of steps that M uses on any input of length n. If $f(n)$ is the running time of M, we say that M runs in time $f(n)$ and that M is an $f(n)$ time TM. Customarily we use n to represent the length of the input.

7.2 Big-O and Small-O Notation

Because the exact running time of an algorithm often is a complex expression, we analyze the **asymptotic analysis**. We do so by considering only the highest order term of the expression for the running time of the algorithm, disregarding both the coefficient of that term and any lower ordure terms.

7.2.1 Big O:

Let f and g be functions, $f, g : \mathbb{N} \to \mathbb{R}^+$. Say that $f(n) = O(g(n))$ if positive integers c and n_0 exist such that $\forall n \geq n_0$

$$f(n) \leq cg(n) \tag{6}$$

Then, $g(n)$ is an **upper bound** for $f(n)$.

7.2.2 Small-o notation

Frequently we derive bounds of the form n^c for $c \geq 0$. Such bounds are called **polynomial bounds**. Bounds of the form 2^{n^θ} are called **exponential bounds** when $\theta \in \mathbb{R}, \theta > 0$. To say that one function is asymptotically less than another, we use small -o notation:
Let f and g be function $f, g : \mathbb{N} \to \mathbb{R}^+$. Say that $f(n) = o(g(n))$ if

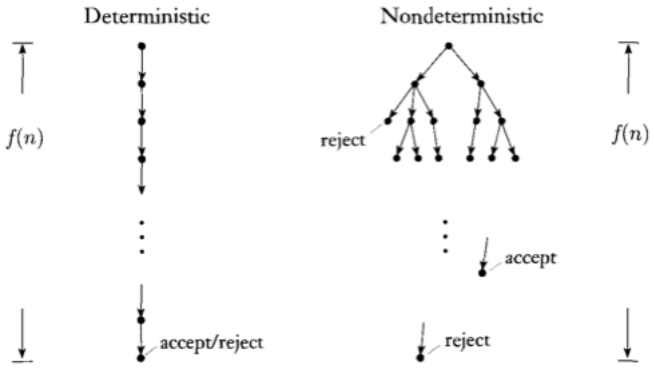
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \tag{7}$$

7.3 Analyzing algorithms

To analyze any algorithm, we consider each step in four separate stages. Then, we look at how they are linked and the collective result. we now set up some notation for classifying languages according to their time requirements:
Let $f : \mathbb{N} \to \mathbb{R}^+$ be a function. Define the **time complexity class, TIME(t(n))**, to be the collection of all languages that are decidable by an $O(t(n))$ time TM.

7.4 Complexity relationship among models

Theorem: Let $t(n)$ be a function, where $t(n) \geq n$. Then every $t(n)$ time multitape Turing machine has an equivalent $O(t^2(n))$ time single-tape Turing machine.
Definition: Let N be a nondeterministic Turing machine that is a decider. Then **running time** of N is the function $f : \mathbb{N} \to \mathbb{N}$, where $f(n)$ is the maximum number of steps that N uses on any branch of its computation on any input of length n.



Theorem: Let $t(n)$ be a function, where $t(n) \geq n$. Then every $t(n)$ time nondeterministic single-tape TM has an equivalent $2^{O(t(n))}$ time deterministic single-tape TM.

7.5 The class P

We consider polynomial differences in running time to be small, whereas exponential differences are considered to be large. The reason is that exponential time algorithms grow too fast to be useful for any practical purpose. They typically arise when we solve problems by using brute force search. For example, one way to factor a number into its constituent primes is to search through all potential divisors.
Definition: **P** is the class of languages that are decidable in polynomial time on a deterministic single-tape Turing machine. In other words,

$$P = \bigcup_k TIME(n^k) \tag{8}$$

The class P is central in our theory because

1. P is invariant $\forall$ models of computation that are polynomially equivalent to the deterministic single-tape Turing machine, and
2. P roughly corresponds to the class of problems that are realistically solvable on a computer

7.5.1 Examples of problems in P:

- PATH=$\{< G, s, t > | G$ is a directed graph that has a directed path from s to $t\}$
- RELPRIME=$\{< x, y > | x$ and y are relatively prime$\}$
- Every context-free language is a member of P.

7.6 The class NP

The problem of finding a **Hamiltonian path** is said to be undecidable. That is, HAMPATH: $\{< G, s, t > | G$ is a directed graph with a Hamiltonian path from s to $t\}$. This problem also has a feature called **polynomial verifiability** that is important for understanding of complexity. Even though we don't know of a fast way to determine whether a graph contains a Hamiltonian path, if such a path were discovered somehow, we could easily convince someone else of its existence, by presenting it.

Definition 1. *A* **verifier** *for a language A is an algorithm V, where*

$$A = \{w | V \text{ accepts} < w, c > \text{ for some string } c\} \tag{9}$$

We measure the time of a verifier only in terms of the length of w, so a **polynomial time verifier** *runs in polynomial time in the length of w. A language A is a* **polynomially verifiable** *if it has a polynomial time verifier.*

Definition 2. NP *is the class of languages that have polynomial time verifiers*

Theorem: A language is NP iff it is decided by some nondeterministic polynomial time Turing machine.

Definition 3.

$$\textbf{NTIME}(\textbf{t}(\textbf{n})) = \{L | L \textit{ is a language decided by a O(t(n)) time nondeterministic TM}\} \tag{10}$$

Which implies

$$NP = \bigcup_k NTIME(n^k) \tag{11}$$