CIS511 Theory of Computation Notes

Chao Qu

May 13, 2017

1 Introduction

1.1 Mathematical Notions and Terminology

1.1.1 Sequences and Tuples

A **sequence** of objects is a list of these objects in some order.
A sequence with k elements is a **k-tuple**.

Functions and Relations

The set of possible inputs to a function is called its **domain**.
The outputs of a function come from a set called its **range**.

$$f: D \to R$$

Functions with k arguments is called a **k-ary function**.
A binary relation R is an equivalence relation if R satisfies three conditions:

1. R is **reflexive** if for every x, xRx;
2. R is **symmetric** if for every x and y, xRy implies yRx;
3. R is **transitive** if for every x, y, and z, xRy and yRz implies xRz.

1.1.2 Graphs

The number of edges at a particular node is the **degree** of that node.
No more than one edge is allowed between any two nodes.
For a graph G, (i, j) represents an edge between node i and node j. Order of i and j doesn't matter in an undirected graph. We can also describe undirected edges using set notation $\{i, j\}$.
A **simple path** is a path that doesn't repeat any node.
A graph is **connected** if every two nodes have a path between them.
A path is a **cycle** if it starts and ends in the same node.
A **simple cycle** is one that contains at least three nodes and repeats only the first and last nodes.
A graph is a **tree** if it is connected and has no simple cycles.
A path in which all the arrows point in the same direction as its steps is called a **directed path**.
A directed graph is **strongly connected** if a directed path connects every two nodes.

1.1.3 Strings and Languages

An **alphabet** (Σ) is any nonempty ***finite*** set.
The members of the alphabet are the **symbols** of the alphabet.
A **string over an alphabet** (s or w) is a ***finite*** sequence of symbols from that alphabet.
The string of length zero is called the **empty string** (ϵ).
A **language** (L) is a set of strings.
Σ^* is the set of all strings over Σ. Different from power set, Σ^* is usually infinite.
$\emptyset$ is a language, $\{\epsilon\}$ is a different language.
Strings in a language can be infinite, but alphabets have to be finite.
Σ^k is the set of strings of length k, each of whose symbol is in Σ. $\Sigma^0 = \{\epsilon\}$.
$\Sigma = \{0, 1\}$ is the alphabet, $\Sigma^1 = \{0, 1\}$ is a set of strings.
A **problem** is the question of deciding whether a given string is a member of some particular language.

1.1.4 Boolean Logic

the **implication** operation $\rightarrow$ and is 0 if its first operand is 1 and its second operand is 0.

$$A \rightarrow B \Leftrightarrow \neg A \vee B$$

Some examples

$$\begin{aligned} P \vee Q &\Leftrightarrow \neg(\neg P \wedge \neg Q) \\ P \rightarrow Q &\Leftrightarrow \neg P \vee Q \\ P \leftrightarrow Q &\Leftrightarrow (P \rightarrow Q) \wedge (Q \rightarrow P) \\ P \oplus Q &\Leftrightarrow \neg(P \leftrightarrow Q) \end{aligned}$$

2 Regular Languages

2.1 Finite Automata

Finite state machine (FSM) is a computation model with limited and constant memory. constant memory means its size is independent of input length
Markov chains are probabilistic counterparts of finite automata.

2.1.1 Formal Definition of a Finite Automaton

ITOC - Definition 1.5 - Finite Automaton

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. $\delta\colon Q \times \Sigma \rightarrow Q$ is the **transition function**,

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

Some properties:

- 0 accept states is possible with $F = \emptyset$.
- exactly 1 transition arrow exits every state for each possible input symbol.

If A is the set of all strings that machine M accepts, then A is the language of machine M and write $L(M) = A$.
We say that a machine **recognizes** a language and **accepts** a string.
A machine may accept several strings, but it always recognizes only one language.
If a machine accepts no strings, then it recognizes the empty language $\emptyset$.

2.1.2 Formal Definition of Computation

ITOC - Computation for a DFA

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a finite automaton and let $w = w_1 w_2 \cdots w_n$ be a string where $w_i \in \Sigma$, for $i = 1, 2, \ldots n$.
M **accepts** w, if a sequence of states $r_0, r_1, \ldots, r_n$ in Q exists with 3 conditions:

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

M **recognizes** language A if $A = \{w \mid M \text{ accepts } w\}$.

ITOC - Definition 1.16 - Regular Language

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

What languages are not regular? Anything that requires memory.

- cannot store the string,
- cannot count.

2.1.3 The Regular Operations

ITOC - Definition 1.23 - Regular Operations

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

- Union: $A \cup B = \{x \mid x \in A \vee x \in B\}$.
- Concatenation: $A \circ B = \{xy \mid x \in A \wedge y \in B\}$.
- Star: $A^* = \{x_1 x_2 \dots x_k \mid k \geq 0, \forall x_i \in A\}$.

The empty string ϵ is always a member of A^*, no matter what A is.
The collection of regular languages is closed under all three of the regular operations.

ITOC - Theorem 1.25

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

Proof. Proof by construction.
Assume $A_1 = L(M_1), A_2 = L(M_2)$, then build M to recognize $L_1 \cup L_2$. Simulate M_1 and M_2 simultaneously, each state in M corresponds to two states, one in M_1 and one in M_2.
Let M_1 recognize A_1, where $M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$.
Let M_2 recognize A_2, where $M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$.
Construct M to recognize $A_1 \cup A_2$, where $M = (Q, \Sigma, \delta, q_0, F)$.

1. $Q = Q_1 \times Q_2 = \{(r_1, r_2) \mid r_1 \in Q_1 \wedge r_2 \in Q_2\}$,
2. Σ is the same as Σ in M_1 and M_2,
3. $\delta((r_1, r_2), a) = (\delta_1(r_1, a), \delta_2(r_2, a))$,
4. $q_0 = (q_1, q_2)$,
5. $F = (F_1 \times Q_2) \cup (Q_1 \times F_2) = \{(r_1, r_2) \mid r_1 \in F_1 \vee r_2 \in F_2\}$.

□

2.2 Nondeterminism

In a **nondeterministic** machine, several choices may exist for the next state at any point.
Nondeterminism is a **generalization** of determinism. Every DFA is an NFA.

Difference between DFA and NFA:

1. Every state of a DFA always has exactly one exiting transition arrow for each symbol in the alphabet. In an NFA, a state may have zero, one, or many exiting arrows for each alphabet symbol.

2. In a DFA, labels on the transition arrows are symbols from the alphabet. This NFA has an arrow with the label ϵ.

If any one of these copies of the machine is in an accept state at the end of the input, the NFA accepts the input string.

2.2.1 Formal Definition of a Nondeterministic Finite Automaton

NFA and DFA differ in one essential way: in the type of transition function. In an NFA, the transition function takes a state and an input symbol or the empty string and produces *the set of possible next states.*

ITOC - Definition 1.37 - Nondeterministic Finite Automaton

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

1. Q is a finite set of states,
2. Σ is a finite alphabet,
3. $\delta\colon Q \times \Sigma_\epsilon \to \mathcal{P}(Q)$ is the transition function,
4. $q_0 \in Q$ is the start state,
5. $F \subseteq Q$ is the set of accept states.

ITOC - Computation for an NFA

Let $N = (Q, \Sigma, \delta, q_0, F)$ be an NFA and let $w = y_1 y_2 \cdots y_n$ be a string where $y_i \in \Sigma_\epsilon$, for $i = 1, 2, \ldots n$.
N **accepts** w, if a sequence of states $r_0, r_1, \ldots, r_n$ in Q exists with 3 conditions:

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

2.2.2 Equivalence of NFAs and DFAs

NFAs and DFAs recognize the same class of languages, which is the class of regular languages.
Two machines are **equivalent** if they recognize the same language.

ITOC - Theorem 1.39

Every NFA has an equivalent DFA.

If k is the number of states of the NFA, then the DFA simulating the NFA will have 2^k states. Define $E(R)$ to be the collection of states that can be reached from members of R by going only along ϵ arrows, including members of R.

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

Proof. Proof by construction.
Let $N = (Q, \Sigma, \delta, q_0, F)$ be some NFA that recognizes A.
Construct $M = (Q', \Sigma, \delta', q_0', F')$ that also recognizes A.

1. $Q' = \mathcal{P}(Q)$,
2. $\delta'(R, a) = \{q \in Q \mid q \in \delta(r, a) \text{ for some } r \in R\} = \bigcup_{r \in R} \delta(r, a)$
   $\delta'(R, a) = \{q \in Q \mid q \in E(\delta(r, a)) \text{ for some } r \in R\} = \bigcup_{r \in R} E(\delta(r, a))$
3. $q_0' = \{q_0\}$
   $q_0' = E(\{q_0\})$
4. $F' = \{R \in Q' \mid R \text{ contains an accept state of } N\} = \{R \in Q' \mid R \cap \bigcup_{f \in F} E(f) \neq \emptyset\}$.

□

We follow the ϵ arrows after each input symbol is read. An alternative procedure based on following the ϵ arrows before reading each input symbol works equally well.

ITOC - Corollary 1.40

A language is regular if and only if some NFA recognizes it.

Proof. NFA recognizes a language ⇒ there is a DFA that also recognizes the language ⇒ the language is regular.
A language is a regular ⇒ there is a DFA that recognizes it ⇒ DFA is a special case of NFA ⇒ the language is recognized by an NFA. □

When constructing a DFA from an NFA, if no arrows point at some states, they can be removed without affecting the performance of the machine.

2.2.3 Closure under the Regular Operations

ITOC - Theorem 1.45

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

Proof. NFA N_1 recognizes A_1, NFA N_2 recognizes A_2, build N to recognize $A_1 \cup A_2$.
$N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1), N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2) \rightarrow N = (Q, \Sigma, \delta, q_0, F)$.

1. $Q = Q_1 \cup Q_2 \cup \{q_0\}$, add an additional new start state q_0.
2. q_0 is the start state of N.
3. $F = F_1 \cup F_2$.
4. $$\delta(q,a) = \begin{cases} \delta_1(q,a) & q \in Q_1 \\ \delta_1(q,a) & q \in Q_2 \\ \{q_1, q_2\} & q = q_0, a = \epsilon \\ \emptyset \& q = q_0, a \neq \epsilon \end{cases}$$

□

ITOC - Theorem 1.47

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

Proof. Proof by construction.
$N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1), N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2) \rightarrow N = (Q, \Sigma, \delta, q_1, F_2)$.

1. $Q = Q_1 \cup Q_2$.
2. q_1 is the start state of N_1.
3. Accept states F are the same as N_2, $F = F_2$.
4. $$\delta(q,a) = \begin{cases} \delta_1(q,a) & q \in Q_1, q \notin F_1 \\ \delta_1(q,a) & q \in F_1, a \neq \epsilon \\ \delta_1(q,a) \cup \{q_2\} & q \in F_1, a = \epsilon \\ \delta_2(q,a) & q \in Q_2 \end{cases}$$

□

Theorem 1.49

The class of regular languages is closed under the star operation.

Proof. Proof by construction.
$N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1) \to N = (Q, \Sigma, \delta, q_0, F)$.

1. $Q = Q_1 \cup \{q_0\}$.
2. q_0 is the new start state.
3. $F = F_1 \cup \{q_0\}$.
4. $$\delta(q,a) = \begin{cases} \delta_1(q,a) & q \in Q_1, q \notin F_1 \\ \delta_1(q,a) & q \in F_1, a \neq \epsilon \\ \delta_1(q,a) \cup \{q_1\} & q \in F_1, a = \epsilon \\ \{q_1\} & q = q_0, a = \epsilon \\ \emptyset & q = q_0, a \neq \epsilon \end{cases}$$

□

Important closures:

1. Set theoretic: Union $A \cup B$, Intersection $A \cap B$, Complement $\overline{A}$, Relative complement $B - A$.
2. Regular operations: Union $A \cup B$, Concatenation $A \circ B$, Kleene Star A^*
3. Reverse: A^R
   (a) Reverse all the arcs in the transition diagram for A.
   (b) Make the start state of A the only accepting state for the new automaton.
   (c) Make the accepting states of A regular states.
   (d) Create a new start state p_0 with ϵ-transitions to all the formal accepting states of A.

2.3 Regular Expression

2.3.1 Formal Definition of a Regular Expression

The value of a regular expression is a language.
If Σ is an alphabet, the regular expression Σ describes the language consisting of all strings of length 1 over this alphabet. Σ^* describes the language consisting of all strings over that alphabet.

ITOC - Definition 1.52

R is a **regular expression** if R is

1. a fro some $a \in \Sigma$, (language $\{a\}$)
2. ϵ, (language $\{\epsilon\}$)
3. $\emptyset$, (language $\emptyset$)

4. $(R_1 \cup R_2)$, where R_1 and R_2 are regex,
5. $(R_1 \circ R_2)$, where R_1 and R_2 are regex,
6. (R_1^*), where R_1 is a regex.

If Σ is an alphabet, the regular expression Σ describes the language consisting of all strings of length 1 over this alphabet, and Σ^* describes the language consisting of all strings over that alphabet. The expression ϵ represents the language containing a single string, whereas $\emptyset$ represents the language that doesn't contain nay strings.
$R^+ = RR^*$, $R^+ \cup \epsilon = R^*$.
$L(R)$ is the language of R.

Some examples:

- $1^*\emptyset = \emptyset$.

  Concatenating the empty set to any set yields the empty set.

- $\emptyset^* = \{\epsilon\}$.

  If the language is empty, the star operation can put together 0 strings, giving only the empty string.

Let R be any regular expression

1. $R \cup \emptyset = R$.

   Adding the empty language to any other language will not change it.

2. $R \circ \epsilon = R$

   Joining the empty string to any string will not change it.

3. $R \cup \epsilon$ may not equal R.
4. $R \circ \emptyset$ may not equal $R/$, $R \circ \emptyset = \emptyset$.

2.3.2 Equivalence with Finite Automata

Regular expressions and finite automata are equivalent in their descriptive power. A regular language is one that is recognized by some finite automata

ITOC - Theorem 1.54

A language is regular if and only if some regular expression describes it.

ITOC - Lemma 1.55

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

Proof. Say that a regular expression R describing some language A. Show how to convert R into an NFA recognizing A. Then if an NFA recognizes A then A is regular.
For the first 3 cases,

1. $R = a$ for some $a \in \Sigma$
   $N = (\{q_1, q_2\}, \Sigma, \delta, q_1, \{q_2\})$
   $\delta(q_1, a) = \{q_2\}$ and $\delta(r, b) = \emptyset$ for $r \neq q_1$ or $b \neq a$.
2. $R = \epsilon$, $L(R) = \{\epsilon\}$
   $N = (\{q_1\}, \Sigma, \delta, q_1, \{q_1\})$
   $\delta(r, b) = \emptyset$ for any r and b
3. $R = \emptyset$, $L(R) = \emptyset$
   $N = (\{q_1\}, \Sigma, \delta, q_1, \emptyset)$
   $\delta(r, b) = \emptyset$ for any r and b

For the last three cases, use the construction given in the proofs that the class of regular languages is closed under the regular operations. □

2.4 Non-regular Languages

2.4.1 The Pumping Lemma for Regular Languages

All strings in the language can be "pumped" if they are at least as long as a certain special value, called the **pumping length**.
Such string contains a section that can be repeated any number of times with the resulting string remaining in the language.

Theorem 1.70

If A is a regular language, then there is a number p where if $s \in A$, $|s| \geq p$, then $s = xyz$, satisfying the following conditions:

1. $\forall i \geq 0, xy^i z \in A$.
2. $|y| > 0$.
3. $|xy| \leq p$.

Either x or z may be ϵ, but $y \neq \epsilon$.
Intuitive explanation: We can always a nonempty string y not too far from the beginning of w that can be pumped; that is, repeating y any number of times, or deleting it (the case $i = 0$), keeps the resulting string in the language L.
By the pigeonhole principle, the first p+1 states in the sequence must contain a repetition. Therefore $|xy| \leq p$.

Proof. Construct a DFA $M = (Q, \Sigma, \delta, q_1, F)$ that recognizes A. Let number of states of M to be p.
Let string $s = s_1 \dots s_n \in A$, with $|s| = n \geq p$.
The sequence of states that M goes through when computing s has length $n+1$. $r_{i+1} = \delta(r_i, s_i)$, for $1 \leq i \leq n$.
This sequence $r_1 \cdots r_{n+1}$ has length $n + 1 \geq p + 1$. But M only has p states. Thus, the first $p + 1$ elements in the sequence must have a repeated state (pigeonhole principle).
When we have $p + 1$ states, we have p symbols, so $|xy| \leq p$.
Let the first repeat state r_j and the second r_l, and r_l occurs among the first $p + 1$ of the state sequence $(r_1 \cdots r_n)$, we have $l \leq p + 1$.
Let $x = s_1 \cdots s_{j-1}$, $y = s_j \cdots s_{l-1}$ and $z = s_l \cdots s_n$.
M must accept xy^iz (condition 1). Since $j \neq l$, $|y| > 0$ (condition 2). $l \leq p + 1$, so $|xy| \leq p$ (condition 3). □

How to use pumping lemma to prove that a language is not regular.

1. Assume that B is regular
2. Use pumping lemma to guarantee the existence of pumping length p, such that all strings with length at least p can be pumped.
3. Find a string s in B that has length p or greater but cannot be pumped.
4. Demonstrate that s cannot be pumped by considering all ways of dividing s into x, y, and z.
5. For each such division, find a value i where $xy^iz \notin B$.
6. Contradiction.

2.5 Exercises

1.11 Prove that every NFA can be converted to an equivalent one that has a single accept state.

Proof. Let $N = (Q, \Sigma, \delta, q_0, F)$ be an NFA. Construct an NFA N' with a single accept state that recognizes the same language as N. Informally, N' is exactly like N except it has ϵ-transitions from the states corresponding to the accept states of N, to a new accept state, q_{accept}. State q_{accept} has no emerging transitions. Formally $N' = (Q \cup \{q_{\text{accept}}\}, \Sigma, \delta', q_0, \{q_{\text{accept}}\})$, where for each $q \in Q$ and $a \in \Sigma_\epsilon$

$$\delta'(q, a) = \begin{cases} \delta(q, a) & \text{if } a \neq \epsilon \text{ or } q \notin F \\ \delta(q, a) \cup \{q_{\text{accept}}\} & \text{if } a = \epsilon \text{ and } q \in F \end{cases}$$

and $\delta'(q_{\text{accept}}, a) = \emptyset$ for each $a \in \Sigma_\epsilon$. □

1.29 Using the pumping lemma to show that the following languages are not regular.
a. $A_1 = \{0^n1^n2^n \mid n \geq 0\}$

Proof. Assume that $A_1 = \{0^n 1^n 2^n \mid n \geq 0\}$ is regular. Let p be the pumping length given by the pumping lemma. Choose s to be the string $0^p 1^p 2^p$. Because $s \in A_1$ and $|s| \geq p$, the pumping lemma guarantees that s can be split into three pieces, $s = xyz$, where for any $i \geq 0$ the string $xy^i z \in A_1$. To satisfy pumping lemma, we must have $|y| > 0$ and $|xy| \leq p$. Therefore, y consists only of 0s. Hence $xyyz$ will have more 0s than 1s and 2s, thus not a member of A_1, a contradiction. □

c. $A_3 = \{a^{2^n} \mid n \geq 0\}$

Proof. Assume that $A_3 = \{a^{2^n} \mid n \geq 0\}$ is regular. Let p be the pumping length given by the pumping lemma. Choose s to be the string a^{2^p}. Because s is a member of A_3 and s is longer than p, the pumping lemma guarantees that s can be split into three pieces, $s = xyz$, satisfying the three conditions of the pumping lemma.
The Third condition tells us that $|xy| \leq p$. Furthermore, $p < 2^p$ and so $|y| < 2^p$. Therefore, $|xyyz| = |xyz| + |y| < 2^p + 2^p = 2^{p+1}$. The second condition requires $|y| > 0$ so $2^p < |xyyz| < 2^{p+1}$. Therefore the length of $xyyz$ cannot be a power of 2. Hence $xyyz$ is not a member of A_3, a contradiction. □

2.6 Problems

1.31 For any string $w = w_1 w_2 \cdots w_n$, the **reverse** of w, written w^R, is the string w in reverse order, $w_n \cdots, w_2, w_1$. For any language A, let $A^R = \{w^R \mid w \in A\}$. Show that if A is regular, so is A^R.

Proof. A is regular then there exists a DFA D that recognizes A. We then construct a NFA N from A, that recognizes A^R. The construction works as follows.

1. Reverse all the arrows of D.
2. Make the start state of D be the only accept state of N.
3. Add a new start state q_0' to N, and add ϵ-transitions from q_0' to all the old accepting state of D.
4. Make all the old accepting states of D normal states in N.

Formal definition of N.

1. $Q' = Q \cup \{q_0'\}$.
   The states of N are the states of M, with the addition of a new start state q_0'.
2. The state q_0' is the start state of N.
3. The set of accept states $F' = \{q_0\}$.

4. Define δ so that for any $q \in Q'$ and $a \in \Sigma_\epsilon$,

$$\delta'(q,a) = \begin{cases} F, & \text{if } q = q_0',\ a = \epsilon \\ \emptyset, & \text{if } q = q_0',\ a \neq \epsilon \\ \{p \mid \delta(p,a) = q\}, & \text{if } q \in Q \end{cases}$$

It is easy to verify that for any $w \in \Sigma^*$, there is a path following w from the start state to an accept state in D iff there is a path following w^R from q_0' to q_{accept}' in N. It then follows that $w \in A$ iff $w^R \in A^R$. □

1.38 An **all**-NFA M is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ that accepts $x \in \Sigma^*$ if every possible state that M could be in after reading input x is a state from F. Prove that all-NFAs recognize the class of regular languages.

Proof. There are two directions in this proof.

We first show that if a language is regular, then it can be recognized by an all-NFA.
If a language is regular, then it can be recognized by a DFA. But a DFA is also an all-NFA. Since a DFA has only one possible final state, and that state is an accept state.

We then show that given an all-NFA, the language it recognizes is regular.
Let $M = (Q, \Sigma, \delta, q_0, F)$ be an all-NFA and $A = L(M)$ be the language that M recognizes. We construct a standard NFA $N = (Q, \Sigma, \delta, q_0, Q - F)$.
Let $s \in A$, then M accepts s. By definition of all-NFAs, all possible final states of s is a state from F. If we then pass s to N, no possible states of s will be in $Q - F$, because they will all be in F. This means x is not accepted by N. Thus, $A = \overline{L(N)}$. Since regular language is closed under complement, and $L(N)$ is regular, $A = \overline{L(N)} = L(M)$ must also be regular. □

1.53 Let $\Sigma = \{0, 1, +, =\}$ and

$$ADD = \{x = y + z \mid x, y, z \text{ are binary integers, and } x \text{ is the sum of } y \text{ and } z\}$$

Show that ADD is not regular.

Proof. Use pumping lemma to show that ADD is not regular. Assume that ADD is regular. Let p be the pumping length given by the pumping lemma. Choose $s = (1^p = 1^{p-1}0 + 1)$. Since $s \in ADD$ and $|s| \geq p$, then the pumping lemma guarantees that s can be split into three pieces, $s = xyz$. To satisfy pumping lemma, we must have $|y| > 0$ and $|xy| \leq p$. Therefore, y consists only of 1s, say $|y| = 1^k$, where $0 < k \leq p$. Then $xyyz = (1^{p+k} = 1^{p-1}0 + 1) \notin ADD$. This is a contradiction and thus ADD is not regular. □

3 The Church-Turing Thesis

3.1 Turing Machines

Turning Machine is similar to a finite automaton but with an unlimited and unrestricted memory.

A Turing machine can do everything that a real computer can do.
Initially the tape contains only the input string and is black everywhere else. The outputs accept and reject are obtained by entering designated accepting and rejecting states. If it doesn't enter an accepting or a rejecting state, it will go on forever, never halting.
Differences between finite automata and Turing machines:

1. A TM can both write on the tape and read from it.
2. The read-write head can move both to the left and to the right.
3. The tape is infinite.
4. The special state for rejecting and accepting take effect immediately

3.2 Formal Definition of a Turing Machine

Definition 3.3

A **Turing machine** is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$

1. Q is the set of states,
2. Σ is the input alphabet, $\sqcup \notin \Sigma$,
3. Γ is the tape alphabet, $\sqcup \in \Gamma$, $\Sigma \subset \Gamma$,
4. $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\text{L, R}\}$,
5. q_0 is the start state,
6. q_{accept} is the accept state,
7. q_{reject} is the reject state, $q_{\text{accept}} \neq q_{\text{reject}}$.

Computation of a Turing machine

1. M receives input $w = w_1 \cdots w_n \in \Sigma^*$ on the leftmost n squares of the tape and the rest of the tape is blank.
2. If M ever tries to move to the left off the left-hand end of the tape, the head stays in the same place for that move.
3. Computation halts when it enters either accept or reject state. If neither occurs, M loops forever.

A **configuration** of the TM consists of the current state, the current tape content, and the current head location.
Configuration C_1 **yields** configuration C_2 if the TM can legally go from C_1 to C_2 in a single step.
The **start configuration** of M on input w is the configuration $q_0 w$.

Accepting and rejecting configurations are **halting configurations** and do not yield further configurations.

ITOC - Computation of Turing Machine

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

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

The collection of strings that M accepts is **the language of** M, or **the language recognized by** M, denoted $L(M)$.

ITOC - Definition 3.5

Call a language **Turing-recognizable** if some Turing machine recognizes it.

When we start a Turing machine on an input, 3 outcomes are possible. The machine may *accept*, *reject* or *loop*.
A Turing machine M can fail to accept an input by entering the q_{reject} state and rejecting, or by looping.
Turning machines that halt on all inputs are called **deciders** because they always make a decision to accept or reject. A deiceder that recognizes some language also is said to **decide** that language.

ITOC - Definition 3.6

Call a language **Turing-decidable** or simply **decidable** if some Turing machine decides it.

Every decidable language is Turing-recognizable.

3.2.1 Variants of Turing Machines

The original model and its reasonable variants all have the same power - they recognize the same class of languages.

Multitape Turing Machines
A **multitape Turing machine** is like an ordinary Turing machine with several tapes.
Initially the input appears on tape 1, and the others start out blank.
Transition function: $\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{\text{L, R, S}\}^k$

IOTC - Theorem 3.13

Every multitape Turing machine has an equivalent single-tape Turing machine.

3.2.2 The Definition of Algorithm

The input to a Turing machine is always a string. If we want to provide an object other than a string as input, we must first represent that object as a string.
We indicate the block structure of the algorithm with further indentation. The first line of the algorithm describes the input to the machine.
A graph is **connected** if every node can be reached from every other node by traveling along the edges of the graph.

3.3 Exercises

3.4 Problems

3.10 Say that a **write-once Turing machine** is a single-tape TM that can alter each tape square at most once (including the input portion of the tape). Show that this variant Turing machine model is equivalent to the ordinary Turing machine model. (Hint: As a first step, consider the case whereby the Turing machine may alter each tape square at most twice. Use lots of tape.)

Proof. First simulate an ordinary TM by a write-twice TM. The write-twice TM simulates a single step of the original TM by copying the entire tape over to a fresh portion of the tape to the right-hand side of the currently used portion. The copying procedure operates character by character, marking a character as it is copied.
This procedure alters each tape square twice: once to write the character for the first time, and again to mark that it has been copied. The position of the original TM's tape head is marked on the tape. When copying the cells at or adjacent to the marked position, the tape content is updated according to the rules of the original TM.

To carry out the simulation with a write-once TM, operate as before, except that each cell of the original TM's tape is now represented by two cells. The first contains the original TM's tape symbol and the second is for the mark used in the copying procedure. The input is not presented to the TM in the format with two cells per symbol, so the very first time the tape is copied , the copying marks are put directly over the input symbols. □

3.11 A **Turing machine with doubly infinite tape** is similar to an ordinary Turing machine, but its tape is infinite to the left as well as to the right. The tape is initially filled with blanks except for the portion that contains the input. Computation is defined as usual except that the head never encounters an end to the tape as it moves leftward. Show that this type of Turing machine recognizes the class of Turing-recognizable languages.

Proof. A TM with doubly infinite tape can easily simulate an ordinary TM. It needs to mark the left-hand end of the input so that it can prevent the head from moving off that end.

To simulate the double infinite tape TM by an ordinary TM, we show how to simulate it with a 2-tape TM, which in turn can be simulated by an ordinary TM.
The first tape of the 2-tape TM is written with the input string and the second tape is blank. We cut the tape of the doubly infinite tape TM into two parts, at the starting cell of the input string. The portion with the input string and all the blank spaces to its right appears on the

first tape of the 2-tape TM. The portion to the left of the input string appears on the second tape, in reverse order. □

3.12 A **Turing machine with left reset** is similar to an ordinary TM, but the transition function has the form

$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{\mathrm{R}, \mathrm{RESET}\}$$

If $\delta(q, a) = (r, b, \mathrm{RESET})$, when the machine is in state q reading an a, the machines head jumps to the left-hand end of the tape after it writes b on the tape and enters state r. Note that these machines do not have the usual ability to move the head one symbol left. Show that Turing machines with left reset recognize the class of Turing-recognizable languages.

Proof. **Textbook solution**
We simulate an ordinary TM with the left-reset TM. When the ordinary TM moves its head right, the reset TM does the same. When the ordinary TM moves its head left, the reset TM cannot, so it gets the same effect by marking the current head location on the tape, then resetting and copying the entire tape one cell to the right, except for the mark, which is kept on the same tape cell. Then it resets again, and scans right until it find the mark.
Homework solution
This proof has two directions.

We first show that regular Turing machines can simulate Turing machine with left reset.
We will replace the state transition function in left-reset Turning machines which is of the form $\delta(q, a) = (r, b, \mathrm{RESET})$ to a series of state transition functions in regular Turning machines which simulate the same move.
To achieve this, we use the technique in example 3.7 from the book, which "marks the leftmost symbol in some way when the machine starts with its head on that symbol. Then the machine may scan left until it finds the mark when it wants to reset its head to the left-hand end".

We then show that Turing machine with left reset can simulate regular Turing machine.
For a move to the right on the regular TM, the left-reset TM will do the same.
For a move to the left on the regular TM, the left-reset TM simulates that with the following algorithm:
Assume the head is currently at some cell on the tape,

1. Mark the current cell and left reset.
2. Check the current cell (which is the leftmost cell after the previous left reset)
3. If it is marked, then we are already at the left end of the tape. Unmark the cell and do a left reset and we are done.
4. If it is not marked, then the machine starts to copy each cell to its right (we will write a $\sqcup$ in the first cell). When it reaches the marked cell containing symbol $\dot{x}$, it writes down a marked version of the previous cell $\dot{y}$ and at the next step, it writes down the unmarked version x.
5. It then keeps copying until the entire input tape is shifted to the right by one cell.

6. Do a left reset to go back to the left end side of the tape, then keep going right to find the marked cell. Now the marked cell is one cell left to the previous head position.

□

3.13 A **Turing machine with stay put instead of left** is similar to an ordinary TM, but the transition function has the form

$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{R, S\}$$

At each point, the machine can move its head right or let it stay in the same position. Show that this Turing machine variant is not equivalent to the usual version. What class of languages do these machines recognize?

Proof. It is easy to see that we can simulate any DFA with a stay-put TM. The only modification is to add transitions from state in F to q_{accept} and from states not in F to q_{reject} upon reading a blank.
TODO □

3.15 Show that the collection of decidable languages is closed under the operation of
a. union

Proof. Let M_1 decides L_1 and M_2 decides L_2. We construct a TM M' that decides the union of L_1 and L_2:
M' = On input w:

1. Run M_1 on w. If it accepts, *accept*.
2. Run M_2 on w. If it accepts, *accept*. Otherwise, *reject*.

M' accepts w if either M_1 or M_2 accept it. If both reject, M' rejects. □

b. concatenation

Proof. Let M_1 decides L_1 and M_2 decides L_2. We construct a TM M' that decides the concatenation of L_1 and L_2:
M' = On input w:

1. Nondeterministically split w into strings x and y.
2. Run M_1 on x and M_2 on y.
3. If both accept, *accept*; otherwise, *reject*.

If there is an accepting computation path, then we have found a successful split and the string is in L_1L_2. If all computation paths reject, then the string is not in L_1L_2. In either case, M' halts. Thus M' decides L_1L_2. □

c. star

Proof. Let M decides L. Construct a TM M' that decides L^*.
M = On input w:

1. Split w into $w_1 w_2 \cdots w_n$ for every possible way
2. Run M on w_i for $i = 1, \ldots, n$. (Since M is a decider, it will always halt).
3. If M accepts each of the strings w_i, *accept*
4. All possible splits tried, *reject*.

□

e. complementation

Proof. Let M decides L. Construct a TM M' that decides $\overline{L}$.
M' = On input w:

1. Run M on w.
2. Output the opposite of whatever M outputs.

□

f. intersection

Proof. Let M_1 decide L_1 and M_2 decides L_2. Construct a TM M' that decides $L_1 \cap L_2$.
M' = On input w:

1. Run M_1 on w.
2. If M_1 rejects, *reject*.
3. Run M_2 on w.
4. If M_2 rejects, *reject*; Otherwise, *accept*

□

3.16 Show that the collection of Turing-recognizable languages is closed under the operation of
a. union

Proof. For any two Turing-recognizable languages L_1 and L_2, let M_1 and M_2 be the TMs that recognize them. We construct a TM M' that recognizes the union of L_1 and L_2:
M' = On input w:

1. Run M_1 and M_2 alternately on w step by step.
2. If either accepts, *accept*. If both halt and reject, *reject*.

If either M_1 or M_2 accepts w, M' accepts w because the accepting TM arrives to its accepting

state after a finite number of steps. Note that if both TMs reject and either of them does so by looping, then M' will loop. □

b. concatenation

Proof. Let M_1 recognize L_1 and M_2 recognize L_2, construct an NDTM M' that recognize L_1L_2.
M' = On input w:

1. Non-deterministically split w into x and y.
2. Run M_1 on x. If it halts and rejects, *reject*.
3. Run M_2 on y. If it accepts, *accept*. If it halts and rejects, *reject*.

M' will accept strings in L_1L_2 because it will guess the correct split and both M_1 and M_2 will accept. □

c. star

Proof. Let M recognize L. Construct an NDTM that recognizes L^*
M' = On input w:

1. Nondeterministically split w into $w_1w_2 \cdots w_n$.
2. Run M on w_i for all i. If M accepts all of them, *accept*. If M halts and rejects for any i, *reject*.

□

d. intersection

Proof. Let M_1 and M_2 recognize L_1 and L_2. Construct a TM M' that recognizes $L_1 \cap L_2$.
M' = On input w:

1. Run M_1 on w. If it halts and rejects, *reject*. If it accepts, go to step 2.
2. Run M_2 on w. If it halts and rejects, *reject*. If it accepts, *accept*.

□

3.17 Let $B = \{\langle M_1\rangle, \langle M_2\rangle, \ldots\}$ be a Turing-recognizable language consisting of TM descriptions. Show that there is a decidable language C consisting of TM descriptions such that every machine described in B has an equivalent machine in C and vice versa.

Proof. B is Turing-recognizable, then there is a enumerator E that enumerates strings in B. We build an enumerator E' that enumerates the equivalent of M_i in C in string order.

E = Ignore input

1. Run E.
2. When E gives the ith TM $\langle M_i\rangle$, add extra useless state to M_i such that the resulting

TM description $\langle M_i' \rangle$ is longer than $\langle M_i \rangle$.

3. Output $\langle M_i' \rangle$.

Since this enumerator E' enumerates strings in string order, it is decidable. □

3.18 Show that a language is decidable iff some enumerator enumerates the language in the standard string order.

Proof. If A is decidable, the enumerator operates by generating the strings in string order and testing each in turn for membership in A using the decider. Strings which are found to be in A are printed.

If A is enumerable in string order, consider two cases. If A is finite, it is decidable because all finite languages are decidable. If A is infinite, a decider for A operates as follows. On receiving input w, the decider enumerates all strings in A in order until some string after w appears. That must occur eventually because A is infinite. If w has appeared in the enumeration already then *accept*, but if it hasn't appeared yet, it never will, so *reject*. □

3.19 Show that every infinite Turing-recognizable language has an infinite decidable subset.

Proof. Let L be an infinite Turing-recognizable language, from theorem 3.21 we know that if L is Turing-recognizable, then some enumerator E enumerates L. We then construct another enumerator E' that enumerates a subset of L (called L') in lexicographical order. We proved above that if an enumerator that enumerates a language in lexicographical order, then that language is decidable. Therefore, the construction of E' shows that L has in infinite decidable subset L'.

E' = Ignore input

1. Run E once. Print and record the first string.
2. Keep running E. For every string w that E prints, compare it with the previously recorded string.
   (a) If w is longer than the previous recorded string, print and record w.
   (b) Otherwise don't print w.

Again to see why this machine is correct, note that E' internally uses E, so it will print a subset of the language of E which is L. And because L is infinite, E will eventually print some string that is longer than the previously recorded string in E'. Therefore, L' is an infinite decidable subset of L. □

3.20 Show that single-tape TMs that cannot write on the portion of the tape containing the input string recognize only regular languages.

Proof. □

4 Decidability

4.1 Decidable Languages

4.1.1 Decidable Problems Concerning Regular Languages

ITOC - Theorem 4.1

A_{DFA} is a decidable language.

$$A_{\text{DFA}} = \{\langle B, w\rangle \mid B \text{ is a DFA that accepts input string } w\}$$

The problem of testing whether a DFA B accepts an input w is the same as the problem of testing whether $\langle B, w\rangle$ is a member of the language A_{DFA}. Showing that the language is decidable is the same as showing the computational problem is decidable.

Proof. Construct a TM M that decides A_{DFA}.

$M =$ "On input $\langle B, w\rangle$, where B is a DFA and w is a string:

1. Simulate B on input w.
2. If the simulations ends in an accept state, *accept*. If it ends in a nonaccepting state, *reject*.

□

When M receives its input, M first determines whether it properly represents a DFA B and a string w. If not, M rejects.

ITOC - Theorem 4.2

A_{NFA} is a decidable language.

$$A_{\text{NFA}} = \{\langle B, w\rangle \mid B \text{ is a NFA that accepts input string } w\}$$

Proof. Construct a TM N that decides A_{NFA}.

$N =$ "On input $\langle B, w\rangle$, where B is an NFA and w is a string:

1. Convert NFA B to an equivalent DFA C, using the procedure for this conversion given in Theorem 1.39.
2. Run TM M from Theorem 4.1 on input $\langle C, w\rangle$.
3. IF M accepts, *accept*; otherwise, *reject*.

□

ITOC - Theorem 4.3

A_{REX} is a decidable language.

$$A_{\text{REX}} = \{\langle R, w\rangle \mid R \text{ is a regular expression that generates string } w\}$$

Proof. Construct a TM that decides A_{REX}.

N = On input $\langle R, w\rangle$, where R is a regular expression and w is a string:

1. Convert regular expression R to an equivalent NFA A by using the procedure given in Theorem 1.54.
2. Run TM N on input $\langle A, w\rangle$.
3. IF N accepts, *accept*; if N rejects, *reject*.

□

ITOC - Theorem 4.4

E_{DFA} is a decidable language.

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

Proof. Construct a TM T that decides E_{DFA} using a marking algorithm.

T = On input $\langle A\rangle$, where A 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*.

□

ITOC - Theorem 4.5

EQ_{DFA} is a decidable language.

$$EQ_{\text{DFA}} = \{\langle A, B\rangle \mid A \text{ and } B \text{ are DFAs and } L(A) = L(B)\}$$

Proof. Construct a new DFA C from A and B, where C accepts only those strings that are accepted by either A or B but not both. Thus, if A and B recognize the same language, C will accept nothing. The language of C is

$$L(C) = \left(L(A) \cap \overline{L(B)}\right) \cup \left(\overline{L(A)} \cap L(B)\right)$$

Then $L(C) = \emptyset$ iff $L(A) = L(B)$. Construct C from A and B with the constructions for proving the class of regular languages closed under complementation, union, and intersection.

$F =$ "On input $\langle A, B\rangle$, where A and B are DFAs:

1. Construct DFA C as described.
2. Run E_{DFA} decider T on $\langle C\rangle$ to test whether $L(C) = \emptyset$.
3. If T accepts, *accept*. If T rejects, *reject*.

□

4.2 Undecidability

ITOC - Theorem 4.11

A_{TM} is undecidable.

$$A_{\text{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$$

A_{TM} is Turing-recognizable. This theorem shows that recognizers are more powerful than deciders. Requiring a TM to halt on all inputs restricts the kinds of languages that it can recognize.
The following Turing machine U recognizes A_{TM}.

$U =$ "On input $\langle M, w\rangle$, where M is a TM and w is a string:

1. Simulate M on input w.
2. If M ever enters its accept state, *accept*; if M ever enters its reject state, *reject*.

This machine loops on input $\langle M, w\rangle$ if M loops on w, which is why this machine does not decide A_{TM}.
U is a ***universal Turing machine***.

4.2.1 The Diagonalization Method

ITOC - Definition 4.14

A set A is **countable** if either it is finite or it has the same size as $\mathbb{N}$.

ITOC - Theorem 4.17

$\mathbb{R}$ is uncountable.

In order to show that $\mathbb{R}$ is uncountable, we show that no correspondence exists between $\mathbb{N}$ and $\mathbb{R}$. Proof by contradiction. Suppose that a correspondence f existed between $\mathbb{N}$ and $\mathbb{R}$, we show that f fails to work as it should by finding an x in $\mathbb{R}$ that is not paired with anything in $\mathbb{N}$.

ITOC - Corollary 4.18

Some languages are not Turing-recognizable.

Proof. The set of all strings Σ^* is countable for any alphabet Σ, which can be list in string order.
The set of all Turing machines is countable because each Turing machine M has an encoding into a string $\langle M \rangle$. Omit those strings that are not legal encodings of TMs, we can obtain a list of all TMs.
The set of all **infinite binary sequences** is uncountable. Can be proved via diagonalization.
Let $\mathcal{B}$ be the set of all infinite binary sequences.
Let $\mathcal{L}$ be the set of all languages over alphabet Σ. We show that $\mathcal{L}$ is uncountable by giving a correspondence with $\mathcal{B}$. Each language $A \in \mathcal{L}$ has a unique sequence in $\mathcal{B}$, which is called the **characteristic sequence**.
The function $f : \mathcal{L} \to \mathcal{B}$, where $f(A)$ equals the characteristic sequence of A, is bijective. Therefore $\mathcal{L}$ is uncountable as $\mathcal{B}$. □

4.2.2 An Undecidable Language

ITOC - Theorem 4.11

A_{TM} is undecidable.

$$A_{\text{TM}} = \{\langle M, w \rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$$

Proof. Suppose that H is a decider for A_{TM}.

$$H(\langle M, w \rangle) = \begin{cases} accept & \text{if } M \text{ accepts } w \\ reject & \text{if } M \text{ does not accept } w \end{cases}$$

Construct a new TM D with H as a subroutine. D calls H to determine what M does when the input to M is $\langle M \rangle$ and it does the opposite.

D = On input $\langle M \rangle$, where M is a TM:

1. Run H on input $\langle M, \langle M \rangle \rangle$.
2. Output the opposite of what H outputs. If H accepts, *reject*; if H rejects, *accept*.

$$D(\langle M \rangle) = \begin{cases} accept & \text{if } M \text{ does not accept } \langle M \rangle \\ reject & \text{if } M \text{ accepts } \langle M \rangle \end{cases}$$

Then we run D with its own description $\langle D \rangle$.

$$D(\langle D \rangle) = \begin{cases} accept & \text{if } D \text{ does not accept } \langle D \rangle \\ reject & \text{if } D \text{ accepts } \langle D \rangle. \end{cases}$$

□

Assume that a TM H decides A_{TM}. Use H to build a TM D that takes an input $\langle M \rangle$, where D accepts its input $\langle M \rangle$ exactly when M does not accept its input $\langle M \rangle$. Finally, run D on itself.

- H accepts $\langle M, w \rangle$ exactly when M accepts w.
- D rejects $\langle M \rangle$ exactly when M accepts $\langle M \rangle$.
- D rejects $\langle D \rangle$ exactly when D accepts $\langle D \rangle$.

This is analogous to a formal diagonalization argument: Construct the matrix $T(i, j)$, where the value at (i, j) is the result of $H(\langle M_i, \langle M_j \rangle \rangle)$. A machine D can be constructed where a contradiction arises on the element along the diagonal where D is run on input $\langle D \rangle$.

4.2.3 A Turing-unrecognizable language

A language is **co-Turing-recognizable** if it is the complement of a Turing-recognizable language.

ITOC - Theorem 4.22

A language is decidable iff it is Turing-recognizable and co-Turing-recognizable.

A language is decidable exactly when both it and its complement are Turing-recognizable.

Proof. First, if A is decidable, then $\overline{A}$ is decidable. Since any decidable language is Turing-recognizable, then both A and its complement $\overline{A}$ are Turing-recognizable.
For the other direction, if both A and $\overline{A}$ are Turing-recognizable, let M_1 and M_2 be the recognizers for A and $\overline{A}$. Construct a TM M that decides A.

M = On input W:

1. Run both M_1 and M_2 on w in parallel.

2. If M_1 accepts, *accept*; if M_2 accepts, *reject*.

Every string w is either in A or $\overline{A}$. Therefore, either M_1 or M_2 must accept w. Because M halts whenever M_1 or M_2 accepts, M always halts and so is a decider. IT accepts all strings in A and rejects all strings not in A. So M is a decider for A and A is decidable. □

ITOC - Corollary 4.23

$\overline{A_{\text{TM}}}$ is not Turing-recognizable.

Proof. A_{TM} is Turing-recognizable. If $\overline{A_{\text{TM}}}$ were Turing-recognizable, A_{TM} would be decidable, a contradiction. □

4.3 Exercises

4.2 Consider the problem of determining whether a DFA and a regular expression are equivalent. Express this problem as a language and show that it is decidable.

Proof. Let

$$EQ_{\text{DFA,REX}} = \{\langle A, R\rangle \mid A \text{ is a DFA, } R \text{ is a regular expression and } L(A) = L(R)\}$$

The following TM E decides $EQ_{\text{DFA,REX}}$.

E = On input $\langle A, R\rangle$

1. Convert REX R to an equivalent DFA D using the procedure given in the book.
2. Use the TM M for deciding EQ_{DFA} given in the book on input $\langle A, D\rangle$.
3. If M accepts, *accept*. If M rejects, *reject*.

□

4.3 Let

$$ALL_{\text{DFA}} = \{\langle A\rangle \mid A \text{ is an DFA and } L(A) = \Sigma^*\}$$

Show that ALL_{DFA} is decidable.

Proof. Construct a TM L that decides ALL_{DFA}.

L = On input $\langle A\rangle$ where A is a DFA:

1. Construct DFA B that recognizes $\overline{L(A)}$.
2. Run E_{DFA} decider T on input $\langle B\rangle$.
3. If T accepts, *accept*. If T rejects, *reject*.

□

4.5 Let

$$E_{\text{TM}} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) = \emptyset\}$$

Show that $\overline{E_{\text{TM}}}$, the complement of E_{TM}, is Turing-recognizable.

Proof.

$$\overline{E_{\text{TM}}} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) \neq \emptyset\}$$

Let $s_1, s_2, \ldots$ be a list of all strings in Σ^* in string order. The following TM M' recognizes $\overline{E_{\text{TM}}}$.

$M' =$ On input $\langle M \rangle$, where M is a TM:

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 M has accepted any of these, *accept*.

□

4.4 Problems

4.10 Let

$$INFINITE_{\text{DFA}} = \{\langle A \rangle \mid A \text{ is a DFA and } L(A) \text{ is an infinite language}\}$$

Show that $INFINITE_{\text{DFA}}$ is decidable.

Proof. Construct TM I that decides $INFINITE_{\text{DFA}}$.

$I =$ On input $\langle A \rangle$, where A is DFA:

1. Let k be the number of states of A.
2. Construct DFA D that accepts all strings of length k or more.
3. Construct DFA M such that $L(M) = L(A) \cap L(D)$.
4. Use E_{DFA} decider to test whether $L(M)$ is $\emptyset$.
5. Output the opposite of whatever T outputs.

If A accepts infinitely many strings, it must accept strings longer than k, thus $L(M) \neq \emptyset$ and T will output *reject*, therefore I will accept.
If $L(A)$ is finite, then it cannot have strings longer than k, thus $L(M) = \emptyset$ and T will accept, therefore I will reject. □

4.12 Let

$$A = \{\langle M \rangle \mid M \text{ is an DFA that doesn't accept strings containing an odd number of 1s}\}$$

Show that A is decidable.

Proof. Construct a TM D that decides A.

D = On input $\langle M \rangle$, where M is a DFA:

1. Construct a DFA O that accepts every string containing an odd number of 1s.
2. Construct a DFA B such that $L(B) = L(M) \cap L(O)$.
3. Use E_{DFA} decider T to test whether $L(B) = \emptyset$.
4. Output whatever T outputs.

If M accepts strings containing an odd number of 1s, then $L(B)$ will not be empty, and T will reject, thus D will also reject.
If M doesn't accept strings containing an odd number of 1s, then $L(B)$ will be empty, and T will accept, thus D will also accept. □

4.13 Let

$$A = \{\langle R, S \rangle \mid R \text{ and } S \text{ are REXs and } L(R) \subseteq L(S)\}$$

Show that A is decidable.

Proof. $L(R) \subseteq L(S)$ iff $L(R) \cup \overline{L(S)} = \emptyset$. Construct a TM M that decides A.

M = On input $\langle R, S \rangle$, where R and S are REXs:

1. Construct a DFA E such that $L(E) = L(R) \cup \overline{L(S)}$.
2. Run E_{DFA} decider on $\langle E \rangle$ to test whether $L(E) = \emptyset$.
3. Output whatever T outputs.

□

4.16 Let

$$A = \{\langle R \rangle \mid R \text{ is a REX describing a language containing at least one string } w \text{ that has 111 as a substring}\}$$

Show that A is decidable.

Proof. Construct a TM X that decides A.

X = On input $\langle R \rangle$ where R is a regular expression:

1. Construct DFA E that accepts $\Sigma^* 111 \Sigma^*$.
2. Construct DFA B such that $L(B) = L(R) \cap L(E)$.
3. Use E_{DFA} decider on $\langle B \rangle$ to test whether $L(B) = \emptyset$.
4. Output the opposite of whatever T outputs.

□

4.17 Prove that EQ_{DFA} is decidable by testing the two DFAs on all strings up to a certain size. Calculate a size that works.

Proof. For any DFAs A and B, $L(A) = L(B)$ iff A and B accept the same strings up to length mn, where m and n are the numbers of states of A and B, respectively.
If $L(A) = L(B)$, A and B will accept the same strings.
If $L(A) \neq L(B)$, wee need to show that the languages differ on some string s of length at most mn. Let t be a shortest string on which the language differ, where $l = |t|$. If $l \leq mn$, then we are done. If not, consider the sequence of states $q_0, q_1, \ldots, q_l$ that A enters on input t, and the sequence of states $r_0, r_1, \ldots, r_l$ that B enters on input t. Because A has m states and B has n states, only mn distinct pairs (q, r) exist where q is a state of A and r is a state of B. By the pigeon hole principle, two pairs of states (q_i, r_i) and (q_j, r_j) must be identical. If we remove the portion from i to $j - 1$ we obtain a shorter string on which A and B behave as they would on t, Hence we have found a shorter string on which the two languages differ, even though t was supposed to be shortest. Hence t must be no longer than mn. □

4.18 Let C be a language. Prove that C is Turing-recognizable iff a decidable language D exists such that $C = \{x \mid \exists y(\langle x, y\rangle \in D)\}$

Proof. Assume that a decidable language D exists.
A TM recognizing C operates on input x by going through each possible string y and testing whether $\langle x, y\rangle \in D$. If such a y is ever found, *accept*; if not, just keep searching.

Assume that C is recognized by TM M.
Define a language D to be $\{\langle x, y\rangle \mid M \text{ accepts } x \text{ withing } |y| \text{ steps}\}$. Language D is decidable, and if $x \in C$ then M accepts x within some number of steps, so $\langle x, y\rangle \in D$ for any sufficiently long y, but if $x \notin C$ then $\langle x, y\rangle \not\in D$ for any y. □

4.20 Let A and B be two disjoint languages. Say that languages C separates A and B if $A \subseteq C$ and $B \subseteq \overline{C}$. Show that any two disjoint co-Turing-recognizable languages are separable by some decidable languages.

Proof. Let A and B be two languages such that $A \cap B = \emptyset$, and $\overline{A}$ and $\overline{B}$ are recognizable. Let J recognize $\overline{A}$ and K recognize $\overline{B}$. Construct TM T such that $C = L(T)$ separates A and B.

T = On input w:

1. Simulate J and K on w by alternating the steps of the two TMs
2. If J accepts first, *reject*. If K accepts first, *accept*.

T will halt because $\overline{A} \cup \overline{B} = \Sigma^*$. So either J or K will accept w eventually. $A \subseteq C$ because if $w \in A$, w will not be recognized by J and will be accept by K first. $B \subseteq \overline{C}$ because ,if $w \in B$, w will not be recognized by K and will be accepted by J first. Therefore, C separates A and B. □

4.21 Let

$$WWR_{\text{DFA}} = S = \{\langle M \rangle \mid M \text{ is a DFA that accepts } w^R \text{ whenever it accepts } w\}$$

Show that S is decidable.

Proof. For any language A, let $A^R = \{w^R \mid w \in A\}$. If $\langle M \rangle \in S$, then $L(M) = L(M)^R$. The following TM T decides S.

T = On input $\langle M \rangle$, where M is a DFA

1. Construct DFA N that recognizes $L(M)^R$
2. Run EQ_{DFA} decider F on $\langle M, N \rangle$.
3. Output whatever F outputs.

□

4.23 Say that an NFA is **ambiguous** if it accepts some string along two different computation branches. Let $AMBIG_{\text{NFA}} = \{\langle N \rangle \mid N \text{ is an ambiguous NFA}\}$. Show that $AMBIG_{\text{NFA}}$ is decidable. (Suggestion: One elegant way to solve this problem is to construct a suitable DFA and then run E_{DFA} on it.)

Proof. □

4.30 Let A be a Turing-recognizable language consisting of descriptions of Turing machines, $\{\langle M_1 \rangle, \langle M_2 \rangle, \ldots\}$ where every M_i is a decider. Prove that some decidable language D is not decided by any decider M_i whose description appears in A.

Proof. Since A is Turing-recognizable, then there exists an enumerator E that enumerates it. Consider $\langle M_i \rangle$ as the ith output of E. Assume $s_1, s_2, \ldots, s_i$ are all possible strings of $\{0,1\}^*$. Build a TM D as follow:

D = On input m

1. Check if $m = s_i$.
2. Use enumerator E to enumerate $\langle M_1 \rangle, \langle M_2 \rangle, \ldots, \langle M_i \rangle$.
3. Run M_i on input m.
4. If M_i accept, *reject*; if M_i reject, *accept*.

□

Since D is a decider and it is different from every M_i, it is not in A.

5 Reducibility

A **reduction** is a way of converting one problem to another problem in such a way that a solution to the second problem can be used to solve the first problem.

Reducibility always involves two problems, A and B. If A reduces to B, we can use a solution to B to solve A.
When A is reducible to B, solving A cannot be harder than solving B because a solution to B gives a solution to A.
If A is reducible to B and B is decidable, A is also decidable. If A is undecidable and reducible to B, B is undecidable.

5.1 Undecidable Problems from Language Theory

Use the undecidability of A_{TM} to prove the undecidability of the halting problem by reducing A_{TM} to $HALT_{\text{TM}}$.

ITOC - Theorem 5.1

$HALT_{\text{TM}}$ is undecidable

$$HALT_{\text{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ halts on input } w\}$$

Proof. Proof by contradiction. Assume that $HALT_{\text{TM}}$ is decidable and show that A_{TM} is decidable. The key idea is to show that A_{TM} is reducible to $HALT_{\text{TM}}$.
Assume that a TM R decides $HALT_{\text{TM}}$. Construct TM S to decide A_{TM}.

S = On input $\langle M, w\rangle$, an encoding of a TM M and a string w:

1. Run TM R on input $\langle M, w\rangle$.
2. If R rejects, *reject*.
3. If R accepts, simulate M on w until it halts.
4. If M accepts, *accept*; if M rejects, *reject*.

If R decides $HALT_{\text{TM}}$, then S decides A_{TM}. But A_{TM} is undecidable, $HALT_{\text{TM}}$ is also undecidable. □

We will prove a language undecidable by reducing A to that language.

ITOC - Theorem 5.2

E_{TM} is undecidable.

$$E_{\text{TM}} = \{\langle M\rangle \mid M \text{ is a TM and } L(M) = \emptyset\}$$

Proof. Assume that E_{TM} is decidable and then show that A_{TM} is decidable, a contradiction. We modify $\langle M\rangle$ to guarantee that M rejects all strings except w, but on input w it works as usual. Then use R to determine whether the modified machine recognizes the empty language.

The only string the machine can now accept is w, so its language will be nonempty iff it accepts w.
The modified machine M_1 is

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.

Then assume TM R decides E_{TM} and construct TM S that decides A_{TM} as follows.

S = On input $\langle M, w \rangle$

1. Use the description of M and w to construct the TM M_1.
2. Run R on input $\langle M_1 \rangle$.
3. If R accepts, *reject*; if R rejects, *accept*.

□

ITOC - Theorem 5.3

$REGULAR_{\text{TM}}$ is undecidable.

$$REGULAR_{\text{TM}} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) \text{ is a regular language}\}$$

Proof. Assume that $REGULAR_{\text{TM}}$ is decidable by a TM R and use this to construct a TM S that decides A_{TM}.
The idea is for S to take its input $\langle M, w \rangle$ and modify M so that the resulting TM recognizes a regular language iff M accepts w. Call this machine M_2.
M_2 is **not** constructed for the purposes of actually running it on some input.
We construct M_2 only for the purpose of feeding its description into the decider for $REUGLAR_{\text{TM}}$.
Once this decider returns its answer, we can use it to obtain the answer to whether M accepts w.
Let R be a TM that decides $REGULAR_{\text{TM}}$ and construct TM S to decide A_{TM}.

S = On input $\langle M, w \rangle$

1. Construct TM M_2.

   M_2 = On input x:

   (a) If x has the form 0^n1^n, *accept*.

   (b) If x does not have this from, run M on input w and *accept* if M accepts w.

2. Run R on input $\langle M_2 \rangle$

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

□

Rice's theorem, states that determining **any property** of the languages recognized by Turing machines is undecidable.

ITOC - Theorem 5.4

EQ_{TM} is undecidable.

$$EQ_{\text{TM}} = \{\langle M_1, M_2 \rangle \mid M_1 \text{ and } M_2 \text{ are TMs and } L(M_1) = L(M_2)\}$$

Proof. 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 $\emptyset$, we end up with the problem of determining whether the language of the other machine is empty.
Let TM R decide EQ_{TM} and construct TM S to decide E_{TM}.

S = On input $\langle M \rangle$:

1. Run R on input $\langle M, M_1 \rangle$, where M_1 is a TM that rejects all inputs.
2. If R accepts, *accept*; if R rejects, *reject*.

□

5.2 Mapping Reducibility

Being able to reduce A to B by mapping reducibility means that a computable function exists that converts instances of A to B. If we have such a conversion function (reduction), we can solve A with a solver for B.

5.2.1 Computable Functions

ITOC - Definition 5.17

A function $f : \Sigma^* \to \Sigma^*$ is a **computable function** if some Turing machine M, on every input w, hatls with just $f(w)$ on its tape.

5.2.2 Formal Definition of Mapping Reducibility

ITOC - Definition 5.20

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

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

The function F is called the **reduction** from A to B.

To test whether $w \in A$, we use the reduction f to map w to $f(w)$ and test whether $f(w) \in B$.

ITOC - Theorem 5.22

If $A \leq_m B$ and B is decidable, then A is decidable.

Proof. Let M be the decider for B and f be the reduction from A to B. We construct a decider N for A.

N = On input w:

1. Compute $f(w)$
2. Run M on input $f(w)$ and output whatever M outputs.

□

ITOC - Corollary 5.23

If $A \leq_m B$ and A is undecidable, then B is undecidable.

We used a reduction from A_{TM} to prove that $HALT_{\text{TM}}$ is undecidable. This reduction showed how a decider for $HALT_{\text{TM}}$ could be used to give a decider for A_{TM}. We can also use a mapping reduction from A_{TM} to $HALT_{\text{TM}}$. We need to present a computable function f that takes input of th form $\langle M, w \rangle$ and returns output of the form $\langle M', w' \rangle$, where

$$\langle M, w \rangle \in A_{\text{TM}} \text{ iff } \langle M', w' \rangle \in HALT_{\text{TM}}$$

The following machine F computes a reduction f.

F = On input $\langle M, w \rangle$:

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.

In general, when we describe a TM that computes a reduction from A to B, improperly formed inputs are assumed to map to strings outside of B.
No such reduction exists from A_{TM} to E_{TM}.
We can also use mapping reducibility to show that problems are not Turing-recognizable.

ITOC - Theorem 5.28

If $A \leq_m B$ and B is Turing-recognizable, then A is Turing-recognizable.

ITOC - Corollary 5.29

If $A \leq_m B$ and A is not Turing-recognizable, then B is not Turing-recognizable.

In a typical application of this corollary, let A be $\overline{A_{\text{TM}}}$ and it is not Turing-recognizable. Mapping reducibility implies that $A \leq_m B$ means the same as $\overline{A} \leq_m \overline{B}$. To prove that B isn't recognizable, we show taht $A_{\text{TM}} \leq_m \overline{B}$.

ITOC - Theorem 5.30

EQ_{TM} is neither Turing-recognizable nor co-Turing-recognizable.

Proof. First show that EQ_{TM} is not Turing-recognizable by showing that A_{TM} is reducible to $\overline{EQ_{\text{TM}}}$. The reducing function f works as follows.

F = On input $\langle M, w \rangle$

1. Construct the following two machines, M_1 and M_2.

   M_1 = On any input:

   (a) *Reject.*

   M_2 = On any input:

   (a) Run M on w. If it accepts, *accept.*

2. Output $\langle M_1, M_2 \rangle$.

M_1 accepts nothing. If M accepts w, M_2 accepts everything, and so the two machines are note equivalent. Conversely, if M doesn't accept w, M_2 accepts nothing, and they are equivalent.

To show that $\overline{EQ_{\text{TM}}}$ is not Turing-recognizable, we give a reduction from A_{TM} to the complement of $\overline{EQ_{\text{TM}}}$, which is EQ_{TM}. Hence we show that $A_{\text{TM}} \leq_m EQ_{\text{TM}}$.

G = On input $\langle M, w \rangle$:

1. Construct the following two machines, M_1 and M_2.

   M_1 = On any input:

   (a) *Accept.*

   M_2 = On any input:

   (a) Run M on w.
   (b) If it accepts, *accept.*

2. Output $\langle M_1, M_2 \rangle$.

□

ITOC - Rice's Theorem

Let P be any nontrivial property of the language of a Turing machine. The problem of determining whether a given Turing machine's language has property P is undecidable.
More formally, let P be a language consisting of Turing machine description where P fulfills two conditions. First P is nontrivial - it contains some, but not all, TM descriptions. Second, P is a property of the TM's language - whenever $L(M_1) = L(M_2)$, we have $\langle M_1 \rangle \in P$ iff $\langle M_2 \rangle \in P$. Here, M_1 and M_2 are any TMs. P is an undecidable language.

5.3 Exercises

5.4 If $A \leq_m B$ and B is a regular language, does that imply that A is a regular language?

No. If $A \leq_m B$ and B is regular, it does not necessarily imply that A is also regular. Because the reduction function can be more than the power of a DFA. Reduction functions are required only to halt on all inputs but can perform very sophisticated computations.
For example, let $A = \{0^n 1^n \mid n \leq 0\}$, which is not regular. Reduce A to $B = \{11\}$ using function f computed by the TM M. B has only one string, so B is regular.
M outputs 11 if it accepts w and outputs 00 if it rejects. So f maps instances of A to the only instance of B and maps instances of $\overline{A}$ to $00 \in \overline{B}$. So B is regular, while A is not.

5.5 Show that A_{TM} is not mapping reducible to E_{TM}.

Proof. Supposer for a contradiction that $A_{\text{TM}} \leq_m E_{\text{TM}}$ via reduction function f. Then by definition of reduction, $\overline{A_{\text{TM}}} \leq_m \overline{E_{\text{TM}}}$ via the same reduction function f. But $\overline{E_{\text{TM}}}$ is Turing-recognizable, so $\overline{A_{\text{TM}}}$ is also Turing-recognizable. But $\overline{A_{\text{TM}}}$ is not, a contradiction. □

5.6 Show that $\leq_m$ is a transitive relation.

Proof. Supposer $A \leq_m B$ and $B \leq_m C$. Then there are computable functions f and g such that $x \in A \Leftrightarrow f(x) \in B$ and $y \in B \Leftrightarrow g(y) \in C$. Consider the composition $h(x) = g(f(x))$. We build a TM that computes h as follows: First, simulate a TM for f on input x and call the output y. Then simulate a TM for g on y. The output is $h(x) = g(f(x))$. Therefore, h is a

computable function and $x \in A \Leftrightarrow h(x) \in C$. Hence $A \leq_m C$ via the reduction function h. □

5.7 Show that if A is Turing-recognizable and $A \leq_m \overline{A}$, then A is decidable.

Proof. Suppose that $A \leq_m \overline{A}$. Then $\overline{A} \leq_m A$ via the same mapping reduction. Since A is Turing-recognizable, $\overline{A}$ is also Turing-recognizable, then A is decidable. □

5.4 Problems

5.9 Let

$$WWR_{\text{TM}} = T = \{\langle M \rangle \mid M \text{ is a TM that accepts } w^R \text{ whenever it accepts } w\}$$

Show that T is undecidable.

Proof. Assume TM R decides T. Construct a TM S that uses R to decide A_{TM}.

S = On input $\langle M, w \rangle$, where M is a TM:

1. Construct a TM Q as follows:

   Q = On input x:

   (a) If x does not have the form 01 or 10, *reject*.

   (b) If x has the form 01, *accept*

   (c) Else if x has the form 10, run M on w and accept if M accepts w.

2. Run R on $\langle Q \rangle$.

3. Output whatever R outputs.

We see that if Q accepts both 01 and 10, then M accepts w. Contradiction. □

5.10 Consider the problem of determining whether a two-tape Turing machine ever writes a nonblank symbol on its second tape when it is run on input w. Formulate this problem as a language and show that it is undecidable.

Proof. Let

$$B = \{\langle M, w \rangle \mid M \text{ is a two-tape TM that writes a nonblank symbol on its second tape when it is run on } w\}$$

Show that A_{TM} reduces to B. Assume for the sake of contradiction that TM R decides B. Then construct a TM S that uses R to decide A_{TM}.

S = On input $\langle M, w \rangle$, where M is a TM and w a string:

1. Use M to construct the following two-tape TM T

   T = On input x:

(a) Simulate M on x using the first tape.

(b) If the simulation shows that M accept, write a nonblank symbol on the second tape.

2. Run R on $\langle T, w\rangle$ to determine whether T on input w writes a nonblank symbol on its second tape.

3. If R accepts, M accepts w, *accept*. Otherwise ,*reject*

□

5.11 Consider the problem of determining whether a two-tape Turing machine ever writes a nonblank symbol on its second tape during the course of its computation on any input string. Formulate this problem as a language and show that it is undecidable.

Proof. Let

$$C = \{\langle M\rangle \mid M \text{ is a two-tape TM that writes a nonblank symbol on its second tape when its is run on some input}\}$$

Show that A_{TM} reduces to C. Assume for the sake of contradiction that TM R decides C. Construct a TM S that uses R to decide A_{TM}.

S = On input $\langle M, w\rangle$, where M is a TM and w is a string:

1. Use M and w to construct the following two-tape TM T_w.

   T_w = On an input:

   (a) Simulate M on w using the first tape.

   (b) If the simulation shows that M accepts, write a nonblank symbol on the second tape.

2. Run R on $\langle T_w\rangle$ to determine whether T_w ever writes a nonblank symbol on its second tape.

3. If R accepts, M accepts w, so *accept*. Otherwise, *reject*

□

5.12 Consider the problem of determining whether a single-tape Turing machine ever writes a blank symbol over a nonblank symbol during the course of its computation on any input string. Formulate this problem as a language and show that it is undecidable.

Proof. Let

$$E = \{\langle M\rangle \mid M \text{ is a single-tape TM that writes a blank symbol over a non-blank symbol on some input}\}$$

Show that A_{TM} reduces to E. Assume for the sake of contradiction that TM R decides E. Construct a TM S that uses R to decide A_{TM}.

S = On input $\langle M, w\rangle$, where M is a TM and w is a string:

1. Use M and w to construct the following single-tape TM T_w

   T_w = On any input:

   (a) Simulate M on w. Use symbol $\sqcup'$ instead of $\sqcup$ when writing and treat it like $\sqcup$ when reading.

   (b) If M accepts w, write a true blank symbol $\sqcup$ over some nonblank symbol.

2. Run R on $\langle T_w\rangle$ to determine whether T_w ever writes a blank.

3. If R accepts, M accepts w, *accept*; Otherwise, *reject*.

□

5.13 A useless state in a Turing machine is one that is never entered on any input string. Consider the problem of determining whether a Turing machine has any useless states. Formulate this problem as a language and show that it is undecidable.

Proof. Let

$$USELESS_{\text{TM}} = \{\langle M, q\rangle \mid M \text{ is a TM with useless state } q\}$$

We show that E_{TM} reduces to $USELESS_{\text{TM}}$. Assume for the sake of contradiction that TM R decides $USELESS_{\text{TM}}$. Construct TM S that uses R to decide E_{TM}. For any TM M with accept state q_{accept}, q_{accept} is useless iff $L(M) = \emptyset$.

S = On input $\langle M\rangle$, where M is a TM:

1. Run R on $\langle M, q_{\text{accept}}\rangle$ to determine whether M has useless state q_{accept}.

2. If R accept, *accept*; if R reject, *reject*.

□

5.14 Consider the problem of determining whether a Turing machine M on an input w ever attempts to move its head left when its head is on the leftmost tape cell. Formulate this problem as a language and show that it is undecidable.

Proof. Let

$$L = \{\langle M, w\rangle \mid M \text{ attempts to move its head left when its head is on the leftmost tape cell}\}$$

Show that A_{TM} reduces to L. Assume for the sake of contradiction that TM R decides L. Construct a TM S that uses R to decide A_{TM}.

S = On input $\langle M, w\rangle$:

1. Convert M to M'

   M' = On input x:

(a) M' first moves it's input over one cell to the right, and writes a new symbol $\sqcup'$ on the leftmost tape cell.

(b) Simulates M on the input.

(c) If during the computation, the head sees $\sqcup'$, then move its head one square to the right and remains in the same state.

(d) If M accepts, move it's head all the way to the left and then moves left off the leftmost tape cell.

2. Run R on $\langle M', w\rangle$.

3. If R accepts, *accept*. If R rejects, *reject*.

□

5.15 Consider the problem of determining whether a Turing machine M on an input w ever attempts to move its head left at any point during its computation on w. Formulate this problem as a language and show that it is decidable.

Proof. □

5.22 Show that A is Turing-recognizable iff $A \leq_m A_{\text{TM}}$.

Proof. Assume $A \leq_m A_{\text{TM}}$, then since A_{TM} is Turing-recognizable, then A is Turing-recognizable.
Assume A is Turing-recognizable, then there exists a TM N that recognizes A.
$A = \{w \mid N \text{ accepts } w\}$. Let f be the reduction function such that $f(w) = \langle N, w\rangle$.
It is easy to see that $w \in A$ iff $f(w) = \langle N, w\rangle \in A_{\text{TM}}$. Therefore, $A \leq_m A_{\text{TM}}$. □

5.23 Show that A is decidable iff $A \leq_m 0^*1^*$.

Proof. Assume $A \leq_m 0^*1^*$, then since 0^*1^* is regular hence decidable, A is decidable.
Assume A is decidable, then there exists some TM R that decides A. We construct a TM Q computes the reduction function f.

Q = On input w:

1. Run R on w.

2. If R accepts, outputs 01; Otherwise, outputs 10.

It is easy to see that $w \in A \Leftrightarrow f(w) \in 0^*1^*$. Thus $A \leq_m 0^*1^*$. □

5.24 Let

$$J = \{w \mid \text{either } w = 0x \text{ for some } x \in A_{\text{TM}} \text{ or } w = 1y \text{ for some } y \in \overline{A_{\text{TM}}}\}$$

Show that neither J nor $\overline{J}$ is Turing-recognizable.

Proof. □

5.25 Give an example of an undecidable language B, where $B \leq_m \overline{B}$.

Proof. □

5.28 Rice's theorem Let P be any nontrivial property of the language of a TM. Prove that the problem of determining whether a given Turing machine's language has property P is undecidable. In more formal terms, let P be a language consisting of TM descriptions where P fulfills two conditions. First, P is nontrivial - it contains some but not all, TM descriptions. Second, P is a property of the TM's language - whenever $L(M_1) = L(M_2)$, we have $\langle M_1 \rangle \in P$ iff $\langle M_2 \rangle \in P$, where M_1 and M_2 are any TMs. Prove that P is an undecidable language.

Proof. Assume for the sake of contradiction that P is a decidable language satisfying the properties and let R_P be a TM that decides P. We show how to decide A_{TM} using R_P by constructing TM S. First, let $T_\emptyset$ be a TM that always rejects, so $L(T_\emptyset) = \emptyset$.
We could assume that $\langle T_\emptyset \rangle \notin P$ without loss of generality because we could proceed with $\overline{P}$ instead of P if $\langle T_\emptyset \rangle \in P$.
Because P is not trivial, there exists a TM T with $\langle T \rangle \in P$. Design S to decide A_{TM} using R_P's ability to distinguish between $T_\emptyset$ and T.
S = On input $\langle M, w \rangle$:

1. Use M and w to construct the following TM M_w.

   M_w = On input x:

   (a) Simulate M on w. If it halts and rejects, *reject*. If it accepts, proceed to stage 2.

   (b) Simulate T on x. If it accepts, *accept*.

2. Use TM R_P to determine whether $\langle M_w \rangle \in P$. Output whatever R_P outputs.

TM M_w simulates T if M accepts w. Hence $L(M_w) = L(T)$ if M accepts w, and $L(M_w) = \emptyset$ otherwise. Therefore, $\langle M_w \rangle \in P$ iff M accepts w.

□

5.29 Show that both conditions in Problem **5.28** are necessary for proving that P is undecidable.

Proof. □

5.30 Use Rice's theorem, prove the undecidability of each of the following languages
a. $INFINITE_{\text{TM}} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) \text{ is an infinite language.}\}$

Proof. $INFINITE_{\text{TM}}$ is a language of TM descriptions. It satisfies the two conditions of Rice's theorem. First, its is nontrivial because some TMs have infinite languages and others do not. Second, it depends only on the language. If two TMs recognize the same language, either both have the descriptions in $INFINITE_{\text{TM}}$ or neither do. Consequently, Rice's theorem

implies that $INFINITE_{\text{TM}}$ is undecidable. □

b. $\{\langle M \rangle \mid M \text{ is a TM and } 1011 \in L(M)\}$

Proof. See **5.30 a.**
$L(M_1) = \{0,1\}^*$ and $L(M_2) = \emptyset$ □

c. $ALL_{\text{TM}} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) = \Sigma^*\}$

Proof. See **5.30 a.** □

5.34 Let

$$X = \{\langle M, w \rangle \mid M \text{ is a single-tape TM that never modifies the portion of the tape that contains the input } w\}$$

Is X decidable? Prove your answer.

Proof. We show that X is undecidable by showing that $A_{\text{TM}} \leq X$. Assume R decides X, we build another TM S that uses R to decide A_{TM}.

S = On input $\langle M, w \rangle$, where M is a TM

1. Build another machine M' using M and w

   M' on input x

   (a) Add a special symbol \$ at the end of the input. Then copy the cells that contain w to the left of \$.

   (b) Simulate M on w, treating the special symbol as the left end of this simulation.

   (c) If M accepts w, go all the way to the first cell that contains w and write a different symbol on that cell.

2. Run R on $\langle M', w \rangle$.

3. If R accept, *reject*; if R reject, *accept*.

If M accepts w, then M' will modify the portion of the tape that contains the input w, then R will reject and hence S accept. If M doesn't accept w, then M' will never modify the portion of the tape that contains the input, then R will accept and hence S reject.

Since A_{TM} is undecidable, X is also undecidable. □

6 Advanced Topics in Computability Theory

6.1 A Definition of Information

We define the quantity of information contained in an object to be the size of that object's smallest representation or description.

6.1.1 Minimal Length Descriptions

We describe a binary string x with a Turing machine M and a binary input w to M. The length of the description is the combined length of representing M and w.
We define the string $\langle M, w\rangle$ to be $\langle M\rangle w$, where we simply concatenate the binary string w onto the end of the binary encoding of M.

ITOC - Definition 6.23

Let x be a binary string. The **minimal description** of x, written $d(x)$, is the shorted string $\langle M, w\rangle$ where TM M on input w halts with x on its tape. The **descriptive complexity** of x, written $K(x)$, is $K(x) = |d(x)|$.

$K(x)$ is the length of the minimal description of x.

ITOC - Theorem 6.24

$$\exists c\, \forall x\, [K(x) \leq |x| + c]$$

The descriptive complexity of a string is at most a fixed constant more than its length. The constant is a universal one, not dependent on the string.

Proof. Let M be a Turing machine that halts as soon as it is started. A description of x is simply $\langle M\rangle x$, c is the length of $\langle M\rangle$. □

The amount of information contained by a string cannot be (substantially) more than its length.

ITOC - Theorem 6.25

$$\exists c\, \forall x\, [K(xx) \leq K(x) + c]$$

Proof. Consider TM M, which expects an input of the form $\langle N, w\rangle$, where N is a TM and w is an input for it.
M = On input $\langle N, w\rangle$

1. Run N on w until it halts and produces an output string s.
2. Output string ss.

A description of xx is $\langle M\rangle d(x)$. The length of this description is $|\langle M\rangle| + |d(x)| = c + K(x)$, where c is the length of $\langle M\rangle$. □

ITOC - Theorem 6.25

$$\exists c\, \forall x, y\, [K(xy) \leq 2K(x)_K(y) + c]$$

can be improved to

$$K(xy) \leq 2\log_2(K(x)) + K(x) + K(y) + c$$

For any description language p, a fixed constant c exists that depends only on p where

$$\forall x\, [K(x) \leq K_p(x) + c]$$

6.1.2 Incompressible Strings and Randomness

ITOC - Definition 6.28

Let x be a string. Say that x is c-**compressible** if

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

If x is not c-compressible, say that x is **incompressible by** c.
If x is incompressible by 1, say that x is **incompressible**.

ITOC - Theorem 6.29

Incompressible strings of every length exist.

6.2 Exercises

6.3 Problems

6.21 Show how to compute the descriptive complexity of strings $K(x)$ with an oracle for A_{TM}.

Proof. Show how to compute $K(x)$ with an oracle for A_{TM}.

M = On input x:

1. Compute length $|x|+c$, where c is the length of TM that halts immediately upon starting.
2. Start testing all the strings s up to $|x| + c$, since $K(x) \leq |x| + c$. And all the strings up to that length are potential description of x.
3. If s is well formed as $\langle M, w\rangle$ from all binary strings in standard string order, then use oracle tell whether M accepts w.
4. If oracle says yes, then run M on w, compare the result with x. If they are the same,

output $K(x) = |d(x)|$ and halt; otherwise got to the next string.

5. If oracle says no, just go to the next string.

□

6.22 Use the result of Problem 6.21 to give a function f that is computable with an oracle for A_{TM}, where for each n, $f(n)$ is an incompressible string of length n.

Proof. For a string x, if x does not have any description shorter than itself, then x is incompressible.
Let M be a TM that uses the oracle A_{TM} and computes the description length of any given string. Build a TM F that computes f.

F = On input n

1. Enumerate all strings x of length n in string order.
2. For each x, use M to compute descriptive complexity $K(x)$.
3. As soon as we find a string that is incompressible, i.e. $K(x) \geq |x|$, output x.

□

6.23 Show that the function $K(x)$ is not a computable function.

Proof. Assume for the sake of contradiction that $K(x)$ is a computable function.
Then we can use the TM F in problem 6.22 to compute the first incompressible string of length n. Thus $d(s) = \langle F \rangle b_n$, where b_n is the binary string of integer n. Hence $K(x) = |d(s)| = |\langle F \rangle b_n| = |\langle f \rangle| + |b_n| = c + \log n$. By picking s long enough, we have $|s| = n > \log n + c = |\langle F \rangle b_n| = K(s)$ which is $K(s) < |s|$, this contradicts the fact that s is incompressible. Thus $K(x)$ is not a computable function. □

6.24 Show that the set of incompressible strings is undecidable.

Proof. Assume for the sake of contradiction that the set of incompressible strings is decidable. Since incompressible string of every length exists, this set is infinite. From this we know that there exists an enumerator E, that enumerates strings in this language in standard string order. We then build a TM M that computes the first incompressible string of length n.

M = On input b, where b is the binary string of integer n:

1. Run E, and output the first string of length n.

This machine always halts, because there are finitely many strings with length less than n. Let the output string be x. x is the first incompressible string of length n. Hence, $d(x) = \langle M \rangle b$, and $K(x) = |\langle M \rangle b| = \log n + c$. We then pick n big enough such that $|x| = n > \log n + c = K(x)$, which means $K(x) < |x|$. This contradicts the fact that x is incompressible. Hence our earlier assumption that the set of incompressible strings is decidable is wrong. □

6.25 Show that the set of incompressible strings contains no infinite subset that is Turing-recognizable.

Proof. Assume for the sake of contradiction that the set of incompressible strings contains an infinite subset that is Turing-recognizable. Call this set S. Since S is Turing-recognizable, there is an enumerator E that enumerates elements of S.
We then build a TM N that computes incompressible string of length at least n.

N = On input b, where b is binary string of integer n:

1. Run E, and for each string x compare its length with n.
2. If $|x| \geq n$, halt with x.

Since x is incompressible, $d(x) = |\langle N \rangle b| = \log n + c$, we can then pick n big enough such that $|x| = n > \log n + c = K(x)$, thus contradicts the fact that x is incompressible. Hence our earlier assumption that the setup incompressible strings contains an infinite Turing-recognizable subset is wrong. □

6.27 Let $S = \{\langle M \rangle \mid M \text{ is a TM and } L(M) = \{\langle M \rangle\}\}$. show that neither S nor $\overline{S}$ is Turing-recognizable.

Proof. To show that S is not Turing-recognizable, we reduce A_{TM} to $\overline{S}$. The reduction function works as follow:

F = On input $\langle M, w \rangle$:

1. Create a TM N.

   N = On input x:

   (a) If $x = \langle N \rangle$, accept.
   (b) If $x = 0$, run M on w, accept if M accepts;
   (c) For any other input x, reject.

This is a mapping reduction from A_{TM} to $\overline{S}$. If M accepts w, then $L(N) = \{\langle N \rangle, 0\}$, thus $\langle N \rangle \in \overline{S}$. If M does not accept w, then $L(N) = \{\langle N \rangle\}$, thus $\langle N \rangle \notin \overline{S}$. Therefore, $\langle M, w \rangle \in A_{\text{TM}}$ iff $f(\langle M, w \rangle) \in \overline{S}$. Because $A_{\text{TM}} \leq_m \overline{S}$, we have $\overline{A_{\text{TM}}} \leq_m S$. Since A_{TM} is not Turing-recognizable, S is also not Turing-recognizable.

Similarly, we can show that $A_{\text{TM}} \leq_m S$, or equivalently $\overline{A_{\text{TM}}} \leq_m \overline{S}$, thus showing $\overline{S}$ is not Turing-recognizable. □

7 Complexity Theory

7.1 Measuring Complexity

For simplicity, we compute the running time of an algorithm purely as a function of the length of the string representing the input and don't consider any other parameters.

In **worst-case analysis**, we consider the longest running time of all inputs of a particular length.

ITOC - Definition 7.1

Let M be a DTM 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. Say that M runs in time $f(n)$ and that M is an $f(n)$ time TM.

7.1.1 Big-O and Small-O Notation

In **asymptotic analysis**, we seek to understand the running time of the algorithm when it is run on large inputs

ITOC - Definition 7.2

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 for every integer $n \geq n_0$, $f(n) \leq cg(n)$.
When $f(n) = O(g(n))$, $g(n)$ is an **upper bound** for $f(n)$, or that $g(n)$ is an **asymptotic upper bound** for $f(n)$ to emphasize that we are suppressing constant factors.

Intuitively, $f(n) = O(g(n))$ means that f is less than or equal to g if we disregard difference up to a constant factor. You may think of O as a representing a suppressed constant.
Changing the value of base b changes the value of $\log_b n$ by a constant factor, since

$$\log_b n = \frac{\log_2 n}{\log_2 b} = \frac{\log_a n}{\log_a b}$$

Thus when we write $f(n) = O(\log n)$, specifying the base is no longer necessary because we are suppressing constant factors anyway.
When Big-O notation appears in arithmetic expressions, each occurrence of O symbol represents a different suppressed constant. $f(n) = O(n^2) + O(n)$, the $O(n^2)$ term dominates the $O(n)$ term, that expression is equivalent to $f(n) = O(n^2)$.
When the O symbol occurs in an exponent, $f(n) = 2^{O(n)}$, it means an upper bound of 2^{cn} for some constant c.
For $f(n) = 2^{O(\log n)}$, using the identity $n = 2^{\log_2 n}$ and thus $n^c = 2^{c \log_2 n}$, we see that $2^{O(\log n)}$ represents an upper bound of n^c for some c.
Bounds of the form n^c for $c > 0$ are called **polynomial bounds**. Bounds of the form $2^{(n^\delta)}$ are called **exponential bounds** for $\delta > 0$.
small-o notation: Big-O notation says that one function is asymptotically *no more than* another. Small-o notation says one function is asymptotically *less than* another.

ITOC - Definition 7.5

Let f and g be functions $f, g : \mathbb{N} \to \mathbb{R}^+$. Say that $f(n) = o(g(n))$ if

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

$f(n) = o(g(n))$ means that for any real number $c > 0$, a number n_0 exists, where $f(n) < cg(n)$ for all $n \geq n_0$.

$f(n)$ is never $o(f(n))$, $f(n)$ is always $O(f(n))$.

7.1.2 Analyzing Algorithms

ITOC - Definition 7.7

Let $t : \mathbb{N} \to \mathbb{R}^+$ 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.

The sequence of parities always gives the reverse of the binary representation. If all parities agree, the binary representations of the number of 0s and of 1s agree, and so the two numbers are equal.
Any language that can be decided in $o(n \log_n)$ time on a single-tape TM is regular.
In computability theory, the Church-Turing thesis implies that all reasonable models of computation are equivalent, they all decide the same class of language. In complexity theory, the choice of model affects the time complexity of languages.

7.1.3 Complexity Relationships Among Models

ITOC - Theorem 7.8

Let $t(n)$ be a function, where $t(n) \geq n$. Then every $t(n)$ time multitape TM has an equivalent $O(t^2(n))$ time single-tape TM.

Proof. Show that simulating each step of the multitape TM uses at most $O(t(n))$ steps on the single-tape TM.

Let M be a k-tape TM that runs in $t(n)$ time. Construct a single-tape TM S that runs in $O(t^2(n))$ time.

Machine S operates by simulating M. Initially, S puts its tape into the format that represents all the tapes of M and then simulates M's steps. To simulate one step, S scans all the information stored on its tape to determine the symbols under M's tape heads. Then S makes another pass over its tape to update the tape contents and head positions. If one of M's heads moves rightward onto the previously unread portion of its tape, S must increase the amount of space allocated to this tape.

For each step of M, S makes two passes over the active portion of its tape. The first obtains the information necessary to determine the next move and the second carries it out. Each the active portion of S's tape has length at most $t(n)$ because M uses $t(n)$ tape cells in $t(n)$ steps if the head moves rightward at every step. Thus the scan of the active portion of S's tape uses $O(t(n))$ steps.

To simulate each of M's steps, S performs two scans and possibly up to k rightward shifts. Each uses $O(t(n))$ time, so the total time for S to simulate one of M's steps is $O(t(n))$.

Therefore the entire simulation of M uses $O(n) + O(t^2(n)) = O(t^2(n))$ steps. □

Recall that an NDTM is a decide if all its computation branches halt on all inputs.

ITOC - Definition 7.9

Let N be a NDTM that is a decider. The **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 tis computation on any input of length n.

ITOC - Theorem 7.11

Let $t(n) \geq n$ be a function. Then every $t(n)$ time single-tape NDTM has an equivalent $2^{O(t(n))}$ time single-tape DTM.

Proof. Let N be an NDTM in $t(n)$ time. Construct a DTM D that simulates N by searching N's nondeterministic computation tree.

On an input of length n, every branch of N's nondeterministic computation tree has a length of at most $t(n)$. Every node in the tree can have at most b children (b is the maximum branching factor). Thus the total number of leaves in the tree is at most $b^{t(n)}$.

The simulation proceeds by exploring this tree breadth first. It visits all nodes at depth d before going on to any of the nodes at depth $d + 1$. The total number of nodes in the tree is less than twice the maximum number of leaves, so bound it by $O(b^{t(n)})$. The time it takes to start from the root and travel down to a node is $O(t(n))$.Therefore, the running time of D is $O(t(n)b^{t(n)}) = 2^{O(t(n))}$.

$$O(t(n)b^{t(n)}) = ct(n)b^{t(n)} = 2^{\log_2(ct(n))+t(n)\log_2 b} = 2^{O(t(n))}$$

TM D has 3 tapes. Converting to a single-tape TM at mosts squares the running time. Thus, the running time of the single-tape simulator is $(2^{O(t(n))})^2 = 2^{O(2t(n))} = 2^{O(t(n))}$. □

7.2 The Class P

7.2.1 Polynomial Time

Polynomial differences in running time are considered to be small, whereas exponential differences are considered to be large.
Exponential time algorithms typically arise when solving problems by exhaustively searching through a space of solutions, called **brute-force search**.
All reasonable deterministic computational models are **polynomially equivalent**. Any one of them can simulate another with only a polynomial increase in running time.

ITOC - Definition 7.12

P is the class of languages that are **decidable** in polynomial time on a single-tape DTM.

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

7.2.2 Examples of Problems in P

When analyzing an algorithm:

1. Give a polynomial upper bound on the number of stages that the algorithm uses when it runs on an input of length n.
2. Examine the individual stages in the description of the algorithm to be sure that each can be implemented in polynomial time on a reasonable deterministic model.

In reasonable graph representations, the size of the representation is a polynomial in the number of nodes.

ITOC - Theorem 7.14

$PATH \in$ P

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

Proof. A potential path is a sequence of nodes in G having a length of at most m, where m is the number of nodes in G. A Polynomial time algorithm M for $PATH$ operates as follows.

$M =$ On input $\langle G, s, t\rangle$ where G is a directed graph with nodes s and t:

1. Place a mark on node s.
2. Repeat the following until no additional nodes are marked:
3. Scan all the edges of G. If an edge (a, b) is found going from a marked node a to an unmarked node b, mark node b.
4. If t is marked, *accept*. Otherwise, *reject*.

Stage 1 and 4 are executed only once.
Stage 3 runs at most m times because each time except the last it marks an additional node in G. Thus the total number of stages is at most $1 + 1 + m$, which is polynomial in the size of G. □

ITOC - Theorem 7.15

$RELPRIME \in$ P.

$$RELPRIME = \{\langle x, y\rangle \mid x \text{ and } y \text{ are relatively prime}\}$$

Proof. The magnitude of a number represented in binary, or in any other base k notation for $k \geq 2$, is exponential in the length of its representation. Brute-force algorithm searches through an exponential number of potential divisors and has an exponential running time.

The Euclidean algorithm E is

E = On input $\langle x, y\rangle$ where x and y are natural numbers in binary:

1. Repeat until $y = 0$:
2. Assign $x \leftarrow x \mod y$.
3. Exchange x and y.
4. Output x.

Algorithm R solves $RELPRIME$, using E as a subroutine.

R = On input $\langle x, y\rangle$:

1. Run E on $\langle x, y\rangle$.
2. If the result is 1, *accept*. Otherwise, *reject*.

□

7.3 The Class NP

A **Hamiltonian path** in a directed graph G is a directed path that goes through each node exactly once.

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

A certificate for a string $\langle G, s, t\rangle \in HAMPATH$ is a Hamiltonian path from s to t.
A NDTM N_1 that decides $HAMPATH$.

N_1 = On input $\langle G, s, t\rangle$, where G is a directed graph with nodes s and t:

1. Write a list of m numbers, $p_1, \ldots, p_m$, where m is the number of nodes in G. Each number in the list is nondeterministically selected to be between 1 and m.
2. Check for repetitions in the list. If any are found, *reject*.

3. Check whether $s = p_1$ and $t = p_m$. If either fail, *reject*.

4. For each i between 1 and $m - 1$, check whether (p_i, p_{i+1}) is an edge of G. If any are not, *reject*. Otherwise, all tests have been passed, *accept*.

Verifying the existence of a hamiltonian path may be much easier than determining its existence.

$$COMPOSITES = \{x \mid x = pq, \text{for integers } p, q > 1\}$$

A certificate for the composite number x is one of its divisors.

Some problems may not be polynomially verifiable. For example, $\overline{HAMPATH}$, the complement of the $HAMPATH$ problem.

ITOC - Definition 7.18

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

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

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 $|w|$. A language A is **polynomially verifiable** if it has a polynomial time verifier

c is called a **certificate**, or **proof**, of membership in A. For polynomial verifiers, the certificate has polynomial length because that is all the verifier can access in its time bound.

ITOC - Definition 7.19

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

ITOC - Theorem 7.20

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

Proof. Show how to convert polynomial time verifier to an equivalent polynomial time NTM and vice versa. The NDTM simulates the verifier by guessing the certificate. The verifier simulates the NDTM by using the accepting branch as the certificate.

Let $A \in$ NP and show that A is decided by a polynomial time NTM N. Let V be the polynomial time verifier for A that exists by the definition of NP. Assume that V is a TM that runs in polynomial time and construct N as follows.

$N =$ On input w of length n:

1. Nondeterministically select string c of length at most n^k.

2. Run V on input $\langle w, c \rangle$.

3. If V accepts, *accept*; otherwise, *reject*.

Assume that A is decided by a poly time NDTM N and construct a poly time verifier V as follows.

V = On input $\langle w, c \rangle$:

1. Simulate N on input w, treating each symbol of c as a description of the nondeterministic choice to make at each step.

2. If this branch of N's computation accepts, *accept*; otherwise, *reject*.

□

ITOC - Definition 7.21

$$\textbf{NTIME}(t(n)) = \{L \mid L \text{ is a language decided by an } O(t(n)) \text{ time NDTM}\}$$

ITOC - Corollary 7.22

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

7.3.1 Examples of Problems in NP

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

ITOC - Theorem 7.24

$CLIQUE \in$ NP.

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

Proof. The clique is the certificate. The following is a verifier V for $CLIQUE$.

V = On input $\langle \langle G, k \rangle, c \rangle$, where G is an undirected graph

1. Test whether c is a subgraph with k nodes in G.

2. Test whether G contains all edges connecting nodes in c.

3. If both pass, *accept*; otherwise, *reject*.

Construct an NDTM N that decides $CLIQUE$.

N = On input $\langle G, k\rangle$, where G is an undirected graph:

1. Nondeterministically select a subset c of k nodes of G.
2. Test whether G contains all edges connecting nodes in c.
3. If yes, *accept*; otherwise, *reject*.

□

ITOC - Theorem 7.25

$SUBSETSUM \in$ NP.

$$SUBSETSUM = \{\langle S, t\rangle \mid S = \{x_1, \ldots, x_k\}, \text{and for some } \{y_1, \ldots, y_k\} \subseteq \{x_1, \ldots, x_k\}\}, \sum y_i = t\}$$

Proof. The subset is the certificate. The following is a verifier V for $SUBSETSUM$.

V = On input $\langle\langle S, t\rangle, c\rangle$:

1. Test whether c is a collection of numbers that sum to t.
2. Test whether S contains all the numbers in c.
3. If both pass, *accept*; otherwise, *reject*.

Construct an NDTM N that decides $SUBSETSUM$.

N = On input $\langle S, t\rangle$:

1. Nondeterministically select a subset c of the numbers in S.
2. Test whether c is a collection of numbers that sum to t.
3. If yes, *accept*, otherwise, *reject*.

□

The complements of these sets, $\overline{CLIQUE}$ and $\overline{SUBSETSUM}$ are not obviously members of NP. Verifying that something is *not* present seems to be more difficult than verifying that it *is* present. **coNP** contains the languages that are complements of languages in NP. We don't know whether coNP is different from NP.

7.3.2 The P Versus NP Question

P = the class of languages for which membership can be decide quickly.
NP = the class of languages for which membership can be verified quickly.
If P = NP, any polynomially verifiable problem would be polynomially decidable.

$$\text{NP} \subseteq \text{EXPTIME} = \bigcup_k \text{TIME}(2^{n^k})$$

7.4 NP-Completeness (NPC)

The first NPC problem is the **satisfiability problem**. A Boolean formula is **satisfiable** if some assignment of 0s and 1s to the variables makes the formula evaluate to 1.

ITOC - Theorem 7.27

$SAT \in$ P iff P = NP.

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

7.4.1 Polynomial Time Reducibility

ITOC - Definition 7.28

A function $f : \Sigma^* \to \Sigma^*$ is a **polynomial time computable function** if some polynomial time Turing machine M exists that halts with just $f(w)$ on its tape, when started on any input w.

ITOC - Definition 7.29

Language A is **polynomial time mapping reducible**, to language B, written $A \leq_\text{P} B$, if a polynomial time computable function $f : \Sigma^* \to \Sigma^*$ exists, where for every w,

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

The function f is called the **polynomial time reduction** of A to B.

ITOC - Theorem 7.31

If $A \leq_\text{P} B$ and $B \in$ P, then $A \in$ P.

Proof. Let M be the polynomial time algorithm deciding B and f be the polynomial time reduction from A to B. Construct a polynomial time TM N deciding A as follows.
$N =$ On input w:

1. Compute $f(w)$.
2. Run M on $f(W)$ and output whatever M outputs.

N runs in poly time because each of its two stages runs in poly time. □

A **literal** is a Boolean variable or a negated Boolean variable (x or $\overline{x}$). A **clause** is several literals connected with $\vee$s ($x_1 \vee \overline{x_2} \vee x_3$). A Boolean formula is in **conjunctive normal form**, if it comprises several clauses connected with $\wedge$s. It is a **3cnf-formula** if all the clauses have three literals. If an assignment satisfies a cnf-formula, each clause must contain at least one literal that evaluates to 1.

ITOC - Theorem 7.32

$3SAT$ is polynomial time reducible to $CLIQUE$.

$$3SAT = \{\langle\phi\rangle \mid \phi \text{ is a satisfiable 3cnf-formula}\}$$

Proof. The polynomial time reduction f converts formulas to graphs. In the constructed graphs, cliques of a specified size correspond to satisfying assignments of the formula.
The reduction f generates the string $\langle G, k\rangle$, where G is an undirected graph defined as follows. The nodes in G are organized into k groups of three nodes each called the **triples**, $t_1, \ldots, t_k$. Each triple corresponds to one of the clauses in ϕ, and each node in a triple corresponds to a literal in the associated clause. Label each node of G with its corresponding literal in ϕ.
No edge between nodes in the same triple, and no edge between two nodes with contradictory labels.
Show that ϕ is satisfiable iff G has a k-clique.
Suppose that ϕ has a satisfying assignment. At least one literal is true in every clause. In each triple of G, select one node corresponding to a true literal in the satisfying assignment. If more than one literal is true in a particular clause, chose arbitrarily among them. The nodes selected form a k-clique. Each pair of selected nodes is joined by an edge because no pair fits one of the exceptions described previously.
Suppose that G has a k-clique. No two of the clique's nodes occur in the same triple. Each of the k triples contains exactly one of the k clique nodes. Assign truth values to the variables of ϕ so that each literal labeling a clique node is make true. This assignment satisfies ϕ because each triple contains a clique node and each clause contains a literal that is assigned 1. □

ITOC - Definition 7.34

A language B is **NP-complete** if it satisfies two conditions:

1. $B \in$ NP.
2. $\forall A \in$ NP, $A \leq_{\mathrm{P}} B$.

ITOC - Theorem 7.35

If $B \in$ NPC and $B \in$ P, then P = NP.

Proof. Use polynomial time reduction. □

ITOC - Theorem 7.36

If $B \in$ NPC and $B \leq_{\mathrm{P}} C$ for $C \in$ NP, then $C \in$ NPC.

Proof. Already know that $C \in$ NP, just need to show that every $A \in$ NP is poly time reducible to B. Since $B \in$ NPC, every language in NP is poly time reducible to B, and B is poly time reducible to C. Sine poly time reductions compose, hence every language in NP is poly time reducible to C and $C \in$ NPC. □

7.4.2 The Cook-Levin Theorem

ITOC - Theorem 7.37

SAT is NP-complete.

Proof. It is easy to show that $SAT \in$ NP. An NDTM in poly time can guess an assignment to a given formula ϕ and accept if it satisfies ϕ.

Then show that any language in NP is poly time reducible to SAT. Construct a poly time reduction for each language A in NP to SAT. The reduction for A takes a string w and produces a Boolean formula ϕ that simulates the NP machine for A on input w.

The formula ϕ is the AND of four parts: $\phi_{\text{cell}} \wedge \phi_{\text{start}} \wedge \phi_{\text{move}} \wedge \phi_{\text{accept}}$.
Let $C = Q \cup \Gamma \cup \{\#\}$.
Formula ϕ_{cell} ensures that the assignment turns on exactly one variable for each cell.

$$\phi_{\text{cell}} = \bigwedge_{1 \leq i,j \leq n^k} \left[\left(\bigvee_{s \in C} x_{ijs} \right) \wedge \left(\bigwedge_{s,t \in C, s \neq t} (\overline{x_{ijs}} \vee \overline{x_{ijt}}) \right) \right]$$

The first part says that at least one variable is turned on in the corresponding cell. The second part says that no more than one variable is turned on (in each pair of variables, both cannot be on at the same time).

Formula ϕ_{start} ensures that the first row of the table is the starting configuration of N on w.

$$\begin{aligned}\phi_{\text{start}} =& x_{11\#} \wedge x_{12q_0} \wedge \\ & x_{13w_1} \wedge \ldots \wedge x_{1(n+2)w_n} \wedge \\ & x_{1(n+3)\sqcup} \wedge \ldots \wedge x_{1(n^k-1)\sqcup} \wedge x_{1n^k\#}\end{aligned}$$

Formula ϕ_{accept} guarantees that an accepting configuration occurs in the tableau.

$$\phi_{\text{accept}} = \bigvee_{1 \le i,j \le n^k} x_{ijq_{\text{accept}}}$$

Formula ϕ_{move} guarantees that each row of the tableau corresponds to a configuration that legally follows the preceding row's configuration according to N's rules. It does so by ensuring that each 2×3 window of cells is legal.

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

The tableau is $n^k \times n^k$ so it contains n^{2k} cells. Each cell has l variables where $l = |C|$. l depends only on the TM N but not on the length of the input n, the total number of variables is $O(n^{2k})$.
Formula ϕ_{cell} contains a fixed size fragment of the formula for each cell, so its size is $O(n^{2k})$.
Formula ϕ_{start} has a fragment for each cell in the top row, so its size is $O(n^k)$.
Formula ϕ_{move} and ϕ_{accept} each contain a fixed-size fragment of the formula for each cell, so there size is $O(n^{2k})$.
Thus, ϕ's total size is $O(n^{2k})$.
Each component of the formula is composed of many nearly identical fragments, which differ only at the indices. We may easily construct a reduction that produces ϕ in poly time from the input w. □

NP-completeness can usually be proved with a poly time reduction from a language that is already known to be NP-complete. It is usually easier to reduce from $3SAT$ instead of SAT.

ITOC - Corollary 7.42

$3SAT$ is NP-complete.

Proof. Obviously $3SAT \in$ NP. We could show that $SAT \le_{\text{P}} 3SAT$. Instead, modify the proof of Cook-Levin Theorem so that it directly produces a formula in CNF with 3 literals per clause.
First convert each formula to CNF. Only ϕ_{move} is not in CNF, we can replace an OR of ANDs with an equivalent AND of ORs. Doing so my increase the size of each subformula, but only increase the total size by a constant factor.
In each clause that currently has one or two literals, we replicate one of the literals until the total number is three. In each clause that has more than three literals, we split it into several

clauses and add additional variables to preserve the original. □

7.5 Additional NP-Complete Problems

When constructing a poly time reduction from $3SAT$ to a language, we look for structures in that language that can simulate the variables and clauses in Boolean formulas. Such structures are called **gadgets**.

ITOC - Corollary 7.43

$CLIQUE$ is NP-complete.

7.5.1 The Vertex Cover Problem

ITOC - Theorem 7.44

$VERTEXCOVER$ is NP-complete.

$$VERTEXCOVER = \{\langle G, k\rangle \mid G \text{ is an undirected graph that has a k-node vertex cover}\}$$

Proof. $VERTEXCOVER \in$ NP, a certificate is simply a vertex cover of size k.

Then show that $3SAT \leq_{\text{P}} VERTEXCOVER$. The reduction converts a 3cnf-formula ϕ into a graph G and a number k, so that ϕ is satisfiable iff G has a vertex cover with k nodes.

The variable gadget contains two nodes connected by an edge. One of these nodes must appear in the vertex cover. For each variable x in ϕ, produce an edge connecting two nodes x and $\overline{x}$. Setting x to TRUE will select the node x for the vertex cover.
The clause gadget contains three nodes and additional edges so that any vertex cover must include at least two of the nodes, or possibly all three. Nodes in the gadget are labeled with the three literals of the clause. These three nodes are connected to each other and to the nodes in the variable gadgets that have the identical labels.
Finally chose k so that vertex cover has one node per variable gadget and two nodes per clause gadget. Total number of nodes is $2m + 3l$ and $k = m + 2l$.

To show this reduction works. Start with a satisfying assignment. First put the nodes of the variable gadgets that correspond to the true literals in the assignment into the vertex cover. Then, select one true literal in every clause and put the remaining two nodes from every clause gadget into the vertex cover. They cover all edges because every variable gadget edge is covered, all three edges within every clause gadget are covered, and all edges between variable and clause gadgets are covered.
If G has a vertex cover with k nodes, show that ϕ is satisfiable. Take the nodes of the variable gadgets that are in the vertex cover and assign TRUE to the corresponding literals. This

satisfies ϕ because each of the three edges connecting the variable gadgets with each clause gadget is covered and only two nodes of the clause are in the vertex cover. Therefore, one of the edges must be covered by a node from a variable gadget and so that satisfies the corresponding clause. □

7.5.2 The Hamiltonian Path Problem

ITOC - Theorem 7.46

$HAMPATH$ is NP-complete.

Proof. Show that $3SAT \leq_{\mathrm{P}} HAMPATH$.
Convert 3cnf-formulas to graphs in which Hamiltonian paths correspond to satisfying assignments of the formula. The variable gadget is a diamond structure that can be traversed in either of two ways, corresponding to the two truth settings. The clause gadget is a node.

Represent each variable x_i with a diamond-shaped structure that contains a horizontal row of nodes. □

ITOC - Theorem 7.55

$UHAMPATH$ is NP-complete.

Proof. Show that $HAMPATH \leq_{\mathrm{P}} UHAMPATH$.

The reduction takes a directed graph G with nodes s and t, and constructs and undirected graph G' with nodes s' and t'. Graph G has a Hamiltonian path s to t iff G' has a Hamiltonian path from s' to t'.

Each node u of G, except for s and t is replaced by a triple of nodes $u^{\mathrm{in}}, u^{\mathrm{mid}}, u^{\mathrm{out}}$ in G'. Nodes s and t in G are replaced by nodes $s^{\mathrm{out}} = s'$ and $t^{\mathrm{in}} = t'$ in G'. Edges of two types appear in G'. First, edges connect u^{mid} with u^{in} and u^{out}. Second, an edge connects u^{out} with v^{in} if $(u, v) \in E_G$.

Assume $s, u_1, u_2, \ldots, u_k, t$ is a Hamiltonian path P in G, then the corresponding Hamiltonian path P' in G' is $s^{\mathrm{out}}, u_1^{\mathrm{in}}, u_1^{\mathrm{mid}}, u_1^{\mathrm{mid}}, \ldots, t^{\mathrm{in}}$.
To show the other direction, claim that any Hamiltonian path in G' from s^{out} to t^{in} must go from a triple of nodes to a triple of nodes, except for the start and finish. Any such path has a corresponding Hamiltonian path in G. starting at node s^{out}. Observe that the next node in the path must be u_i^{in} for some i because only those nodes are connected to s^{out}. The next node must be u_i^{mid} because no other way is available to include u_i^{mid} in the Hamiltonian path. After u_i^{mid} comes u_i^{out} because that is the only other node to which u_i^{mid} is connected. The next node must be u_j^{in} for some j because no other available node is connected to u_i^{out}. The

argument then repeats until t^{in} is reached. □

7.5.3 The Subset Sum Problem

ITOC - Theorem 7.56

$SUBSETSUM$ is NP-complete.

7.6 Exercises

7.1 a. $2n = O(n)$.

True. $c = 2$.

b. $n^2 = O(n)$

False.

c. $n^2 = O(n \log^2 n)$

False.

d. $n \log n = O(n^2)$

True. Since $\log n = O(n)$, $n \log n = nO(n) = O(n^2)$.

e. $3^n = 2^{O(n)}$

True. $3^n = 2^{n \log_2 3} = 2^{O(n)}$.

f. $2^{2^n} = O(2^{2^n})$

True. $f(n) = O(f(n))$.

7.2 a. $n = o(2n)$

False. $\lim_{n\to\infty} \frac{n}{2n} = 0.5$.

b. $2n = O(n^2)$

True. $\lim_{n\to\infty} \frac{2n}{n^2} = 0$.

c. $2^n = o(3^n)$

True.

$$\lim_{n\to\infty} \frac{2^n}{3^n} = \lim_{n\to\infty} \left(\frac{2}{3}\right)^n = 0$$

d. $1 = o(n)$

True.

e. $n = o(\log n)$

False. $\lim_{n\to\infty} \frac{n}{\log n} = \infty$.

f. $1 = o(1/n)$

False. $\lim_{n\to\infty} \frac{1}{1/n} = \infty$.

7.6 Show that P is closed under union, concatenation, and complement.

Let M_1 decides L_1 and M_2 decides L_2 in polynomial time.

a. union

Proof. M' decides $L_1 \cup L_2$ in poly time.

M' = On input $\langle w \rangle$:

1. Run M_1 on w. If it accepts, *accept*.
2. Run M_2 on w. If it accepts, *accept*. If it rejects, *reject*.

M' accepts w iff either M_1 or M_2 accepts w. Since both are deciders, M' is also a decider and decides $L_1 \cup L_2$. Since both M_1 and M_2 run in polynomial time, M' runs in polynomial time. □

b. Concatenation

Proof. M' decides $L_1 \circ L_2$ in polynomial time.

M' = On input w:

1. For each possible way, split w into two substring w_1 and w_2.
2. Run M_1 on w_1 and M_2 on w_2. If both accept, *accept*.
3. If w is not accepted after trying all possible splits, *reject*.

M' accepts w iff $w = w_1 w_2$ such that M_1 accepts w_1 and M_2 accepts w_2. Therefore, M' decides $L_1 L_2$. Since step 2 runs in polynomial time and is repeated at most n times, M' runs in polynomial time. □

c. Complement

Proof. M' decides $\overline{L}$ in polynomial time.

M' = On input w:

1. Run M on w and output the opposite of whatever M outputs.

M' decides $\overline{L}$. Since M runs in polynomial time, M' also runs in polynomial time. □

7.7 Show that NP is closed under union and concatenation.

Let M_1 decides L_1 and M_2 decides L_2 in nondeterministic polynomial time.

a. Union

Proof. M' is an NDTM and decides $L_1 \cup L_2$ in polynomial time.

M' = On input w:

1. Nondeterministically decide whether to check if w is a member of L_1 or L_2.
2. If L_1 was chosen, run M_1 on w and output whatever M_1 outputs. If L_2 was chosen, run M_2 on w and output whatever M_2 outputs.

M' will run in $O(1) + \max(O(M_1), O(M_2)) = \max(O(M_1), O(M_2))$, which is still polynomial time. Thus NP is closed under union. □

b. Concatenation

Proof. M' is an NDTM and decides L_1L_2 in polynomial time.

M' = On input w:

1. Nondeterministically divide w into w_1 and w_2.
2. Run M_1 on w_1.
3. Run M_2 on w_2.
4. If both accept, *accept*. Otherwise, *reject*.

M' runs in $\max(O(M_1), O(M_2))$, which is polynomial time. Thus, $L_1L_2 \in$ NP. □

7.8 Let $CONNECTED = \{\langle G \rangle \mid G \text{ is a connected undirected graph}\}$. A graph is connected if every node can be reached from every other node by traveling along the edges of the graph. Analyze the algorithm given on page 185 to show that this language is in P.

Proof. The following TM M decides $CONNECTED$.

M = On input $\langle G \rangle$ where G is a graph:

1. Select and mark the first node of G.
2. Repeat the following stage until no new nodes are marked:
3. For each node in G, mark it if it is attached by an edge to a node that is already marked.
4. Scan all the nodes of G to determine whether they are all marked. If yes, *accept*; otherwise, *reject*.

The algorithm runs in $O(n^3)$ time.
Stage 1 takes at most $O(n)$ steps to locate and mark the start node.
Stage 2 causes at most $n - 1$ repetitions.
Stage 3 uses at most $O(n^2)$ steps because G contains at most n to be checked an for each checked node, examining all adjacent nodes to see whether any have been marked uses at most $O(n^2)$ steps.
Therefore stage 2 and 3 take $O(n^3)$ time.
Stage 4 takes $O(n)$ steps to scan all nodes.
Therefore, the algorithm runs in $O(n^3)$ and $CONNECTED \in$ P. □

7.9 A triangle in an undirected graph is a 3-clique, show that $TRIANGLE \in$ P, where

$$TRIANGLE = \{\langle G \rangle \mid G \text{ contains a triangle}\}$$

Proof. Construct a TM M that decides $TRIANGLE$ in poly time.

M = On input $\langle G \rangle$ where G is a graph:

1. For each triple of vertices v_1, v_2, v_3 in G:
2. If edges $(v_1, v_2), (v_1, v_3), (v_2, v_3)$ are all edges of G, *accept*.
3. No triangle has been found in G, *reject*.

A graph with m vertices has $\binom{m}{3} = O(m^3)$ triples of vertices. Therefore, stage 2 will repeated at most $O(m^3)$ times. In addition, each stage can be implemented to run in polynomial time.
Therefore, $TRIANGLE \in$ P. □

7.10 Show that ALL_{DFA} is in P.

Proof. A DFA accepts Σ^* iff all reachable states are accepting. Construct a TM M to decide ALL_{DFA}.

M = On input $\langle A \rangle$ where A is a DFA:

1. Perform breadth-first search from start state

2. If any state is not accepting, *reject*.

3. If all states are accepting, *accept*

Since breadth-first search is polynomial time, M also runs in polynomial time. Thus $ALL_{\text{DFA}} \in P$. □

7.11 In both parts, provide an analysis of the time complexity of your algorithm.
a. Show that $EQ_{\text{DFA}} \in$ P.

Proof. □

b. Say that a language A is **star-closed** if $A = A^*$. Give a polynomial time algorithm to test whether a DFA recognizes a star-closed language

Proof. □

7.12 Call graphs G and H **isomorphic** if the nodes of G may be reordered so that it is identical to H. Let $ISO = \{\langle G, H\rangle \mid G \text{ and } G \text{ are isomophic graphs}\}$. Show that $ISO \in$ NP.

Proof. **manual 7.11**
An NDTM N for ISO works as follows:

N = On input $\langle G, H\rangle$ where G and H are undirected graphs:

1. Let m be the number of nodes of G. Check if H has m nodes, if not, *reject*.

2. Nondeterministically select a permutation π of m elements.

3. For each pair of nodes x and y of G check that (x, y) is an edge of G iff $(\pi(x), \pi(y))$ is an edge of H. If all agree, *accept*. If any differ, *reject*.

Stage 2 can be implemented in polynomial time nondeterministically.
Stage 3 takes polynomial time.
Therefore $ISO \in$ NP. □

7.7 Problems

7.13 Let

$$MODEXP = \{\langle a, b, c, p\rangle \mid a, b, c, p \text{ are positive binary integers such that } a^b = c(\mod p)\}$$

Show that $MODEXP \in$ P. (Note that the most obvious algorithm doesn't run in polynomial time)

Proof. **manual 7.12**
The most obvious algorithm is to carry out a^b explicitly, which will take $b - 1$ multiplication. Assume that b has n bits and b is a power of 2, then $b = 2^{n-1}$. Let x be the number of bits of a^b. We have $x = \log_2 a^{2^{n-1}} = 2^{n-1} \log_2 a$, which is exponential in the length of b. Thus, the

obvious algorithm needs exponential space, which also runs in exponential time.

We build a TM that uses repeated squaring to calculate $a^b \mod p$ to decide $MODEXP$.

M = On input $\langle a, b, c, p \rangle$:

1. Let $b = b_1 \cdots b_k$.
2. Calculate $d_i = (a^{2^{(i-1)}} \mod p)$ for $i = 1, \ldots, k$ using repeated squaring.
3. Let $d = 0$, for $i = 1, \ldots, k$
4. If $b_i = 1$, $d \leftarrow d + d_i$.
5. If $d \mod p = c \mod p$, *accept*; otherwise, *reject*.

Note that $(a^{2^{(i+1)}} \mod p) = (a^{2^i} \mod p * a^{2^i} \mod p) \mod p$. Since modular operation is linear time, step 2 is polynomial time. And thus M runs in polynomial time.

Reference: https://www.khanacademy.org/computing/computer-science/cryptography/modarithmetic/a/fast-modular-exponentiation □

7.14 A **permutation** on the set $\{1, \ldots, k\}$ is a one-to-one, onto function on this set. When p is a permutation, p^t means the composition of p with itself t times. Let

$$PERMPOWER = \{\langle p, q, t \rangle \mid p = q^t \text{ where } p \text{ and } q \text{ are permutations on } \{1, \ldots, k\} \text{ and } t \text{ is a binary integer}\}$$

Show that $PERMPOWER \in$ P. (Note that the most obvious algorithm doesn't run with polynomial time.

Proof. □

7.15 Show that P is closed under the star operation. (Hint: Use dynamic programming. On input $y = y_1 \cdots y_n$ for $y_i \in \Sigma$, build a table indicating for each $i \leq j$ whether the substring $y_i \cdots y_j \in A^*$ for any $A \in$ P)

Proof. □

7.16 Show that NP is closed under the star operation.

Proof. Let $A \in$ NP and an NDTM M decide A in nondeterministic poly time. Construct an NDTM N to decide A^* in nondeterministic poly time.

N = On input w:

1. If $w = \epsilon$, *accept*.
2. Nondeterministically choose k, where $1 \leq k \leq n$.
3. Nondeterministically select strings $w_1, w_2, \ldots, w_k$, each of length at most n.
4. If $w \neq w_1 \ldots w_k$, then *reject* (on this branch).

5. Simulate (using nondeterminism) M on each w_i.

6. If M accepts each w_i, *accept*; otherwise *reject*.

Each stage takes poly time, so N runs in poly time. □

7.17 Let $UNARYSSUM$ be the subset sum problem in which all numbers are represented in unary. Why does the NP-completeness proof of $SUBSETSUM$ fail to show $UNARYSSUM$ is NP-complete? Show that $UNARYSSUM \in$ P.

Proof. **Manual 7.15**
The reduction fails because it would result in an exponential unary instance, and therefore would not be a polynomial time reduction. □

7.18 Show that if P = NP, then every language $A \in$ P, except $A = \emptyset$ and $A = \Sigma^*$, is NP-complete. Why cant $\emptyset$ and Σ^* be NP-complete?

Proof. Let A be any language in P and $A \neq \emptyset$ and $A \neq \Sigma^*$. To show that A is NP-complete, we need to show.

1. $A \in$ NP.

   Since P = NP, $A \in$ NP.

2. Every B in NP is polynomial time reducible to A.

   Because $A \neq \emptyset$ and $A \neq \Sigma^*$, there exists $x \in A$ and $y \notin A$. For a language $B \in$ NP, B is also in P. So the reduction function will check in polynomial time whether an input $w \in B$. If $w \in B$, then $f(w) = x \in A$. Else if $w \notin B$, then $f(w) = y \notin A$. Thus, we have $w \in B$ iff $f(w) \in A$. Therefore, $B \leq_{\mathrm{P}} A$.

□

7.19 Show that $PRIMES = \{m \mid m \text{ is a prime number in binary}\} \in$ NP. (Hint: For $p > 1$, the multiplicative group $Z_p^* = \{x \mid x \text{ is relatively prime to } p \text{ and } 1 \leq x < p\}$ is both cyclic and of order $p - 1$ iff p is prime.

Proof. □

7.20 We generally believe that $PATH$ is not NP-complete. Explain the reason behind this belief. Show that proving $PATH$ is not NP-complete would prove P $\neq$ NP.

Proof. If $PATH$ is NP-complete, then $\forall L \in$ NP, $L \leq_{\mathrm{P}} PATH$. But since $PATH \in$ P, this implies that P = NP, which we believes is not true.

Want to show that if $PATH$ is not NP-complete, then P $\neq$ NP. We show the contrapositive, which is P = NP then $PATH$ is NP-complete. First $PATH \in$ P, so $PATH \in$ NP. Then $\forall L \in$ NP, since P = NP, $L \leq_{\mathrm{P}} PATH$, so $PATH$ is NP-complete. □

7.21 Let G represent an undirected graph. Also let

$$SPATH = \{\langle G, a, b, k\rangle \mid G \text{ contains a simple path of length at most } k \text{ from } a \text{ to } b\}$$
$$LPATH = \{\langle G, a, b, k\rangle \mid G \text{ contains a simple path of length at least } k \text{ from } a \text{ to } b\}$$

a. Show that $SPATH \in$ P.

Proof. **Manual 7.16**
Follow the marking algorithm for recognizing $PATH$ while additionally keeping track of the length of the shortest paths discovered. The following TM M decides $SPATH$.

M = On input $\langle G, a, b, k\rangle$ where G is a directed graph with m nodes and has nodes a and b:

1. Place a mark with label 0 on node a.
2. For each i from 0 to m:
3. Scan all the edges of G. If an edge (s, t) is found going from a node s marked with i to an unmarked node b, mark node t with $i + 1$.
4. If t is marked with a value of at most k accept. Otherwise, *reject*.

□

b. Show that $LPATH \in$ NPC.

Proof. **Manual 7.16**
First $LPATH \in$ NP because an NDTM can guess and verify a simple path of length at least k from a to b.
Next, $UHAMPATH \leq_P LPATH$, because the following TM F computes the reduction f.

F = On input $\langle G, a, b\rangle$, where G is an undirected graph and has nodes a and b:

1. Let k be the number of nodes in G.
2. Output $\langle G, a, b, k\rangle$.

If $\langle G, a, b\rangle \in UHAMPATH$, G contains a Hamiltonian path of length k from a to b, which is at least k, thus $\langle G, a, b, k\rangle \in LPATH$.
If $\langle G, a, b, k\rangle \in LPATH$, G contains a simple path of length k from a to b. Since G has k nodes, this path is Hamiltonian. Thus $\langle G, a, b\rangle \in UHAMPATH$. □

7.22 Let

$$DOUBLESAT = \{\langle \phi\rangle \mid \phi \text{ has at least two satisfying assignments}\}$$

Show that $DOUBLESAT$ is NP-complete.

Proof. **manual 7.19**
On input ϕ, a poly time NDTM can guess two assignments and accept if both assignments satisfy ϕ. Thus $DOUBLESAT \in$ NP.
We show that $SAT \leq_P DOUBLESAT$. The following TM F computes the polynomial time

reduction f.

F = On input $\langle \phi \rangle$ where ϕ is a Boolean formula with variables $x_1, x_2, \ldots, x_m$

1. Let ϕ' be $\phi \wedge (x \vee \overline{x})$, where x is a new variable.
2. Output $\langle \phi' \rangle$.

If $\langle \phi \rangle \in SAT$, ϕ' has at least two satisfying assignments because we can obtain two assignments from the original assignment of ϕ by changing the value of x. If $\langle \phi' \rangle \in DOUBLESAT$, ϕ is also satisfiable, because x does not appear in ϕ. Therefore $\langle \phi \rangle \in SAT$ iff $f(\langle \phi \rangle) \in DOUBLESAT$. □

7.23 Let

$$HALFCLIQUE = \{\langle G \rangle \mid G \text{ is an undirected graph having a complete subgraph with at least } m/2 \text{ nodes, where } m = |G|\}$$

Show that $HALFCLIQUE$ is NP-complete.

Proof. **Manual 7.21**
$HALFCLIQUE \in$ NP because an NDTM can guess and verify a clique with at least $m/2$ nodes in polynomial time.

We show that $CLIQUE \leq_{\text{P}} HALFCLIQUE$. The following TM F computes the reduction f.

F = On input $\langle G, k \rangle$ where G is an undirected graph $|G| = m$ and k an integer:

1. If $k = m/2$, output $\langle G \rangle$.
2. If $k < m/2$, construct G' by adding a complete graph with $m - 2k$ nodes and connecting them to all nodes in G. Output $\langle G' \rangle$.
3. If $k > m/2$, construct G'' by adding $2k - m$ isolated nodes. Output $\langle G'' \rangle$.

When $k < m/2$, if G has a k-clique, G' has a clique of size $k + (m - 2k) = (2m - 2k)/2$, thus $G' \in HALFCLIQUE$. If G' has $k+(m-2k)$-clique, then at most $m-2k$ of them come from the new nodes, thus G must contain a k-clique. When $k > m/2$, if G has a k-clique, G'' has a clique of size $(m + 2k - m)/2 = k$, thus $G'' \in HALFCLIQUE$. If $G'' \in HALFCLIQUE$, G must contain a k-clique because the half-clique cannot include any of the new isolated nodes. □

7.24 Let

$CNF_k = \{\langle \phi \rangle \mid \phi \text{ is a satisfiable cnf-formula where each variable appears in at most } k \text{ places}\}$

a. Show that $CNF_2 \in$ P.

Proof. □

b. Show that CNF_3 is NP-complete.

Proof. Show that $3SAT \leq_{\mathrm{P}} CNF_3$. □

7.25 Let

$$CNF_{\mathrm{H}} = \{\langle\phi\rangle \mid \phi \text{ is a satisfiable cnf-formula where each clause contains two literals, but at most one negated literal}\}$$

Show that $CNF_{\mathrm{H}} \in \mathrm{P}$.

Proof. □

7.26 Let ϕ be a 3cnf-formula. An **$\neq$-assignment** to the variables of ϕ is one where each clause contains two literals with unequal truth values. In other words, an $\neq$-assignment satisfies ϕ without assigning three true literals in any clause.
a. Show that the negation of any $\neq$-assignment to ϕ is also an $\neq$-assignment.

Proof. **manual 7.22**
By definition of an $\neq$-assignment, not all three literals in a clause can be true, which means there's at least one true literal and one false literal. Therefore, for the negation of such an $\neq$-assignment, there will also be at least one true literal and one false literal, and it also satisfies ϕ. Thus, the negations of any $\neq$-assignment to ϕ is also an $\neq$-assignment. □

b. Let $\neq SAT$ be the collection of 3cnf-formulas that have an $\neq$-assignment. Show that we obtain a polynomial time reduction from $3SAT$ to $\neq SAT$ by replacing each clause c_i of $(y_1 \vee y_2 \vee y_3)$ with the two clauses $(y_1 \vee y_2 \vee z_i)$ and $(\overline{z_i} \vee y_3 \vee b)$, where z_i is a new variable for each clause c_i, and b is a single additional new variable.

Proof. We want to show that $\phi \in 3SAT$ iff $f(\phi) \in \neq SAT$. Or ϕ has a satisfiable assignment iff $f(\phi)$ has an $\neq$-assignment.

First, given that ϕ is satisfiable, we show that the reduction gives an $\neq$-assignment. If ϕ is satisfiable, then at least one of y_1, y_2, y_3 is true. We see that the following assignments are all $\neq$-assignments, provided that (y_1, y_2, y_3) is a satisfiable assignment to any given clause of $3SAT$.

$y_1 \vee y_2$	z_i	$\overline{z_i}$	y_3	b
0	1	0	1	0
1	0	1	0,1	0

We will always assign $b = 0$, and if $y_1 \vee y_2 = 0$, then $z_i = 1$, otherwise, $z_i = 0$. And each clause clearly satisfies the restriction of an $\neq$-assignment.

Second, given that $f(\phi)$ has an $\neq$-assignment, we show that ϕ is satisfiable. We can always let $b = 0$ using **(a)** because the negation of an $\neq$-assignment is also an $\neq$-assignment. Note that if $b = 0$, then in order for$(y_1 \vee y_2 \vee z_i) \wedge (\overline{z_i} \vee y_3 \vee b)$ to be 1, we cannot have y_1, y_2, y_3 to be all zeros, otherwise the second clause will be all zeros. Thus, y_1, y_2, y_3 must contain at least one true variable, which means that ϕ is satisfiable. □

c. Conclude that $\neq SAT$ is NP-complete.

Proof. $\neq SAT$ is NP-complete, because

1. $\neq SAT \in$ NP, and
2. $3SAT \leq_{\mathrm{P}} \neq SAT$ and $3SAT$ is NP-complete.

□

7.30 Let

$SETSPLITTING = \{\langle S, C\rangle \mid S$ is a finte set and $C = \{C_1, \ldots, C_k\}$ is a collection of subsets of S for some $k > 0$, such that elements of S can be colored red or blue so that no C_i has all its elements colored with the same color$\}$

7.51 This problem investigates **resolution**, a method for proving the unsatisfiability of cnf-formulas. Let $\phi = C_1 \wedge C_2 \wedge \cdots \wedge C_m$ be a formula in cnf, where the C_i are its clauses. Let $\mathcal{C} = \{C_i \mid C_i$ is a clause of $\phi\}$. In a *resolution step*, we take two clauses C_a and C_b in $\mathcal{C}$, which both have some variable x occurring positively in one of the clauses and negatively in the other. Thus, $C_a = (x \wedge y_1 \wedge y_2 \wedge \cdots \wedge y_k)$ and $C_b = (\overline{x} \wedge z_1 \wedge z_2 \wedge \cdots \wedge z_l)$, where y_i and z_i are literals. We form the new clause $y_1 \wedge y_2 \wedge \cdots \wedge y_k \wedge z_1 \wedge z_2 \wedge \cdots \wedge z_l)$ and remove repeated literals. Add this new clause to $\mathcal{C}$. Repeat the resolution stps until no additional clauses can be obtained. If the empty clause () is in $\mathcal{C}$, then declare ϕ unsatisfiable.
Say that resolution is **sound** if it never declares satisfiable formulas to be unsatisfiable. Say that resolution is **complete** if all unsatisfiable formulas are declared to be unsatisfiable.
a. Show that resolution is sound and complete.

Proof. □

b. Use part (a) to show that $2SAT \in$ P.

Proof. □

8 Space Complexity

ITOC - Definition 8.1

Let M be a deterministic Turing machine that halts on all inputs. The **space complexity** of M is the function $f : \mathbb{N} \to \mathbb{N}$, where $f(n)$ is the maximum number of tape cells that M scans on any input of length n.
If M is an NDTM wherein all branches halt on all inputs, we define its space complexity $f(n)$ to be the maximum number of tape cells that M scans on **any branch** of its computation for any input of length n.

ITOC - Definition 8.2

Let $f : \mathbb{N} \to \mathbb{R}^+$ be a function. The **space complexity classes**, **SPACE($f(n)$)** and **NSPACE($f(n)$)**, are defined as follows.

$$\text{SPACE}(f(n)) = \{L \mid L \text{ is a language decided by an } O(f(n)) \text{ space DTM}\}$$
$$\text{NSPACE}(f(n)) = \{L \mid L \text{ is a language decided by an } O(f(n)) \text{ space NDTM}\}$$

ITOC - Example 8.3

$SAT \in$ PSPACE.

Space appears to be more powerful than time because space can be reused, whereas time cannot.

M_1 = On input $\langle \phi \rangle$, where ϕ is a Boolean formula:

1. For each truth assignment to the variables $x_1, \ldots, x_m$ of ϕ:
2. Evaluate ϕ on that truth assignment.
3. If ϕ ever evaluated to 1, *accept*; if not, *reject*.

M_1 clearly runs in linear space because each iteration of the loop can reuse the same portion of the tape. Only needs to store the current truth assignment.

ITOC - Example 8.4

$ALL_{\text{NFA}} \in$ PSPACE

$$ALL_{\text{NFA}} = \{\langle A \rangle \mid A \text{ is an NFA and } L(A) = \Sigma^*\}$$

Proof. Give a nondeterministic linear space algorithm that decides $\overline{ALL_{\text{NFA}}}$. Use nondeterminism to guess a string that is rejected by the NFA and to use linear space to keep track of which states the NFA could be in at a particular time.

N = On input $\langle M \rangle$, where M is an NFA:

1. Place a marker on the start state of the NFA.
2. Repeat 2^q times, where q is the number of states of M:
3. Nondeterministically select an input symbol and change the positions of the markers on M's states to simulate reading that symbol.

4. *Accept* if state 2 and 3 reveal some string that M rejects; that is, if at some point none of the markers lie on accept states of M. Otherwise, *reject*.

If M rejects any strings, it must reject one of length at most 2^q because in any longer string that is rejected, the locations of the markers described in the preceding algorithm would repeat. The only space needed by this algorithm is for storing the location of the markers and the repeat loop counter, and that can be done with linear space. □

8.1 Savitch's Theorem

Savitch's theorem shows that DTM can simulate NDTM by using a small amount of space. For time complexity, such a simulation seems to require an exponential increase in time. For space complexity, Savitch's theorem shows that nay NDTM that uses $f(n)$ space can be converted to a DTM that uses only $f^2(n)$ space.

ITOC - Theorem 8.5

Savitch's theorem
For any function $f : \mathbb{N} \to \mathbb{R}^+$, where $f(n) \geq n$, $\text{NSPACE}(f(n)) \subseteq \text{SPACE}(f^2(n))$

Savitch's theorem also holds when $f(n) \geq \log n$.

Proof. Need to simulate an $f(n)$ space NDTM deterministically. A naive approach is to proceed by trying all the branches of the NDTM's computation, one by one. The simulation needs to keep track of which branch it is currently trying. But a branch that uses $f(n)$ space may run for $2^{O(f(n))}$ steps and each step may be a nondeterministic choice. Exploring the branches sequentially would require recording all the choices used on a particular branch in order to be able to find the next branch. Therefore, this approach may use $2^{O(f(n))}$ space.

Give a deterministic, recursive algorithm that solves the yieldability problem. Given two configurations of the NDTM, c_1 and c_2 with a number t, test whether the NTM can get from c_1 to c_2 within t steps using only $f(n)$ space.
The algorithm operates by searching for an intermediate configuration c_m, and recursively testing whether (1) c_1 can get to c_m within $t/2$ steps and (2) whether c_m can get to c_2 within $t/2$ steps.
This algorithm needs space for storing the recursion stack. Each level of the recursion uses $O(f(n))$ space to store a configuration. The depth of the recursion is $\log t$, where t is the maximum time that the NDTM may use on any branch. We have $t = 2^{O(f(n))}$, so $\log t = O(f(n))$ and the simulation the deterministic simulation uses $O(f^2(n))$ space.

Let N be an NDTM deciding A in space $f(n)$. Construct a DTM M deciding A. M uses the procedure CANYIELD which tests whether one of N's configurations can yield another within a specified number of steps.
Let w be an input to N. For c_1 and c_2 of N, and integer t, CANYIELD(c_1, c_2, t) outputs *accept* if N can go from c_1 to c_2 in t or fewer steps along some nondeterministic path. If not, CANYIELD outputs *reject*. Assume that t is a power of 2

CANYIELD = On input c_1, c_2, and t:

1. If $t = 1$, then test directly whether $c_1 = c_2$ or whether c_1 yields c_2 in one step according to the rules of N. If either test succeeds, *accept*; otherwise, *reject*.
2. If $t > 1$, then for each configuration c_m of N using space $f(n)$:
3. Run CANYIELD$(c_1, c_m, t/2)$.
4. Run CANYIELD$(c_m, c_2, t/2)$.
5. If 3 and 4 both accept, then *accept*.
6. If haven't yet accepted, *reject*.

Modify N so that when it accepts, it clears its tape and moves the head to the leftmost cell, entering a configuration called c_{accept}. Let c_{start} be the start configuration of N on w. Select a constant d so that N has no more than $2^{df(n)}$ configurations using $f(n)$ tape.

M = On input w:

1. Output the result of CNAYIELD$(c_{\text{start}}, c_{\text{accept}}, 2^{df(n)})$

Whenever CANYIELD invokes itself recursively, it stores the current stage number and the values of c_1, c_2, t on a stack so that these values may be restored upon return from the recursive invocation. Each level of the recursion thus uses $O(f(n))$ additional space. Further more, each level of the recursion divides the size of t in half. Initially t starts out equal to $2^{df(n)}$, so the depth of the recursion is $O(\log 2^{df(n)})$ or $O(f(n))$. Therefore the total space used is $O(f^2(n))$. □

8.2 The Class PSPACE

ITOC - Theorem 8.6

PSPACE is the class of languages that are decidable in polynomial space on a deterministic Turing machine.

$$\text{PSPACE} = \bigcup_k \text{SPACE}(n^k)$$

We define NPSPACE in terms of the NSPACE classes. However, PSPACE = NPSPACE by Savitch's theorem because the square of any polynomial is still a polynomial.
P $\subseteq$ PSPACE because a machine that runs quickly cannot use a great deal of space. More precisely, for $t(n) \geq n$, any machine that operates in time $t(n)$ can use at most $t(n)$ space because a machine can explore at most one new cell at each of its computation.
Similarly NP $\subseteq$ NPSPACE and so NP $\subseteq$ PSPACE. Also because any problem in NP has a certificate that can be checked in poly time and so must not take up more than poly space. Thus, NP $\subseteq$ PSPACE.
Conversely, we can bound the time complexity of a TM in terms of its space complexity. For $f(n) \geq n$, a TM that uses $f(n)$ space can have at most $f(n)2^{O(f(n))}$ different configurations. A

TM computation that halts may not repeat a configuration. Therefore, a TM that uses space $f(n)$ must run in time $f(n)2^{O(f(n))}$, so PSPACE $\subseteq EXPTIME = \bigcup_k \text{TIME}(2^{n^k})$.

$$\text{P} \subseteq \text{NP} \subseteq \text{PSPACE} = \text{NPSPACE} \subseteq \text{EXPTIME}$$

We don't know whether any of these containments is actually an equality.
If a problem is in SPACE($s(n)$) for any function $s(n)$, then the problem is decidable. In particular, for $s(n) \geq n$, it has a TM bounded by time $O(2^{ks(n)})$.

8.3 PSPACE-Completeness

ITOC - Definition 8.8

A language B is **PSPACE-complete** if it satisfies two conditions:

1. $B \in$ PSPACE.
2. $\forall A \in$ PSPACE, $A \leq_{\text{P}} B$.

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

8.3.1 The TQBF Problem

It is convenient to require that all quantifiers appear at the beginning of the statement and that each quantifiers scope is everything following it. Such statements are said to be in **prenex normal form**.
When each variable of a formula appears within the scope of some quantifier, the formula is said to be fully **quantified**.

ITOC - Theorem 8.9

$TQBF$ is PSPACE-complete.

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

Proof. First we give a polynomial space algorithm deciding $TQBF$.

$T =$ On input $\langle \phi \rangle$, where ϕ is a fully quantified Boolean formula:

1. If ϕ contains no quantifiers, then it is an expression with only constants, so evaluate ϕ and *accept* if it is true; otherwise, *reject*.
2. If ϕ equals $\exists x\ \psi$, recursively call T on ψ, first with 0 substituted for x and then with 1 substituted for x. If either result is accept, then *accept*; otherwise, *reject*.
3. If ϕ equals $\forall x\ \psi$, recursively call T on ψ, first with 0 substituted for x and then with 1 substituted for x. If both results are accept, then *accept*; otherwise, *reject*.

T obviously decides $TQBF$. The depth of the recursion is at most the number of variables. At each level we need only store the value of one variable, so the total space used is $O(m)$, where m is the number of variables that appear in ϕ. Therefore, T runs in linear space.

Next, show that $TQBF$ is PSPACE-hard.
TODO □

8.3.2 Winning Strategies for Games

Let $\phi = \exists x_1 \, \forall x_2 \, \exists x_3 \cdots Q x_k[\psi]$ be a quantified Boolean formula in prenex normal form. Q represents either a $\forall$ or an $\exists$ quantifier. Player A selects values for the variables that are bound to $\forall$ quantifiers, and Player E selects values for the variables that are bound to $\exists$ quantifiers. At the end of play, use the values that the players have selected for the variables and declare that Player E has won if ψ is TURE, and Player A has won if ψ is FALSE.
A player has a **winning strategy** for a game if that player wins when both sides play optimally.

ITOC - Theorem 8.11

$FORMULAGAME$ is PSPACE-complete.

$$FORMULAGAME = \{\langle \phi \rangle \mid \text{Player E has a winning strategy in the formula game associated with } \phi\}$$

Proof. $FORMULAGAME$ is the same as $TQBF$. □

8.3.3 Generalized Geography

In **generalized geography**, we take an arbitrary directed graph with a designated start node.

ITOC - Theorem 8.14

GG is PSPACE-complete.

$$GG = \{\langle G, b \rangle \mid \text{Player I has a winning strategy for the generalized geography game played on graph } G \text{ starting at node } b\}$$

Proof. First we give an polynomial space algorithm that decides GG.
M = On input $\langle G, b \rangle$, where G is a directed graph and b is a node of G:

1. If b has outdegree 0, *reject* because Player I loses immediately.
2. Remove node b and all connected arrow a new graph G'.
3. For each of the nodes $b_1, b_2, \ldots, b_k$ that b originally pointed at, recursively call M on

$\langle G', b_i \rangle$.

4. If all of these accept, Player II has a winning strategy in the original game, so *reject*.
5. If all of these accept, Player II has a winning strategy in the original game, *reject*. otherwise, *accept*.

The only space required by this algorithm is for storing the recursion stack. Each level of the recursion adds a single node to the stack and at most $m \leq |G|$ levels occur. Hence the algorithm runs in linear space.

To show that GG is PSPACE-hard, show that $FORMULAGAME \leq_P GG$.
TODO □

8.4 The Classes L and NL

Introduce a new TM with two tapes: a read-only input tape and a read/write work tape. Only the cells scanned on the work tape contribute to the space complexity of this type of TM.
For space bounds that are at least linear, the two-tape TM model is equivalent to the standard one-tape model. For sublinear space bounds, we use only the two-tape model.

ITOC - Definition 8.17

L is the class of languages that are decidable in logarithmic space on a DTM.

$$\mathrm{L} = \mathrm{SPACE}(\log n)$$

NL is the class of languages that are decidable in logarithmic space on a NDTM.

$$\mathrm{NL} = \mathrm{NSPACE}(\log n)$$

Pointers into the input may be represented in logarithmic space. So one way to think about the power of log space algorithms is to consider the power of a fixed number of input pointers.
$A = \{0^k 1^k \mid k \geq 0\} \in \mathrm{L}$. Use a TM that decides A by zig-zagging back and forth across the input, crossing off the 0s and 1s as they are matched. The log space TM for A counts the number of 0s and the number of 1s in binary on the work tape. The only space required is that used to record the two counters.

$PATH \in \mathrm{P}$.

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

$PATH \in \mathrm{NL}$.

The nondeterministic log space TM deciding $PATH$ operates by starting at node s and nondeterministically guessing the nodes of a path from s to t. The machine records only the position of the current node at each step on the work tape. The machine nondeterministically selects the next node from among those pointed at by the current node. It repeats this action until it reaches node t and *accepts*, or until it has gone on for $m = |G|$ steps and *rejects*.

Our earlier claim that any $f(n)$ space bounded TM also runs in time $2^{O(f(n))}$ is no longer true for very small space bounds.

ITOC - Definition 8.20

IF M is a two-tape TM and w is an input, a **configuration of M on w** is a setting of the state, the work tape, and the positions of the two tape heads. The input w is not a part of the configuration of M on w.

If M runs in $f(n)$ space and w is an input of length n, the number of configurations of M on w is $n2^{O(f(n))}$. Say that M has c states and g tape symbols. The number of strings that can appear on the work tape is $g^{f(n)}$. The input head can be in one of n positions, and the work tape head can be in one of $f(n)$ positions. Therefore, the total number of configurations of M on w, which is an upper bound on the running time of M on w, is $cnf(n)g^{f(n)}$.
We can extend Savitch's theorem to hold for sublinear space bounds down to $f(n) \geq \log n$.

8.5 NL-Completeness

Analogous to the question of whether P = NP, we have the question of whether L = NL. The NL-complete languages are examples of languages that are the most difficult languages in NL. If L and NL are different, all NL-complete languages don't belong to L.
We define an NL-complete language to be one that is in NL and to which any other language in NL is reducible. We don't use polynomial time reducibility since all problems in NL are solvable in polynomial time. Every two problems in NL except $\emptyset$ and Σ^* are polynomial time reducible to one another.

ITOC - Definition 8.21

A **log space transducer** is a TM with a read-only input tape, a write-only output tape, and a read/write work tape. The head on the output tape cannot move leftward, so it cannot read what it has written. The work tape may contain $O(\log n)$ symbols. A log space transducer M computes a function $f : \Sigma^* \to \Sigma^*$, where $f(w)$ is the string remaining on the output tape after M halts when it is started with w on its input tape. We call f a **log space computable function**. Language A is **log space reducible** to language B, $A \leq_{\mathrm{L}} B$, if A is mapping reducible to B by means of a log space computable function f.

Any f computed by a log space transducer can be computed in polynomial time.
A transducer can never repeat a configuration otherwise it will not halt. There are only $2^{O(\log n)} = O(n^k)$ configurations.

ITOC - Definition 8.22

A language B is **NL-complete** if

1. $B \in \text{NL}$, and
2. $\forall A \in \text{NL}$, $A \leq_{\text{L}} B$.

ITOC - Theorem 8.23

If $A \leq_{\text{L}} B$ and $B \in \text{L}$, then $A \in \text{L}$.

Proof. M_A computes individual symbols of $f(w)$ as requested by M_B. In the simulation, M_A keeps track of where M_B's input head would be on $f(w)$. Every time M_B moves, M_A restarts the computation of f on w from the beginning and ignores all the output except for the desired location of $f(w)$. □

ITOC - Theorem 8.25

$PATH$ is NL-complete.

Proof. Show that $PATH$ is NL-hard. Construct a graph that represents the computation of the nondeterministic log space TM for A. The reduction maps a string w to a graph whose nodes correspond to the configurations of the NTM on input w.
NTM M decides A in $O(\log n)$ space. Given an input w construct $\langle G, s, t\rangle$ in log space, where G is a directed graph that contains a path from s to t iff M accepts w.
The nodes of G are the configurations of M on w. For configurations c_1 and c_2 of M on w, the pair (c_1, c_2) is an edge of G if c_2 is one of the possible next configurations of M starting from c_1. Node s is the start configuration of M on w. M is modified to have a unique accepting configuration, and we designate this configuration to be node t.
This mapping reduces A to $PATH$ because whenever M accepts its input, some branch of its computation accepts, which corresponds to a path from the start configuration s to the accepting configuration t in G. Conversely, if some path exists from s to t in G, some computation branch accepts when M runs on input w and M accepts w.
We then give a log space transducer that outputs $\langle G, s, t\rangle$ on input w by listing its nodes and edges. Each node is a configuration of M on w and can be represented in $c \log n$ space for some constant c. The transducer sequentially goes through all possible strings of length $c \log n$, tests whether each is a legal configuration of M on w and outputs those that pass the test. The transducer lists the edges similarly. Log space is sufficient for verifying that a configuration c_1 of M on w can yield configuration c_2 because the transducer only needs to examine the actual tape contents under the head locations given in c_1 to determine that M' transition function

would give configuration c_2 as a result. The transducer tries all pairs (c_1, c_2) in turn to find which qualify as edges of G. Those that do are added to the output tape. □

Graph G has one node for each possible configuration. $(c_1, c_2) \in E(G)$ if M can go from c_1 to c_2 in a single step.
Given an initial configuration s and accept configuration t (assume a unique accepting configuration), produce $\langle G, s, t \rangle$ using log space transducer. The log space transducer works as follows:

1. Running through counter and output valid configuration (as nodes of graph)
2. Try c_1, c_2 and if c_1 yield c_2 according to TM's transition function, write (c_1, c_2).

ITOC - Corollary 8.26

NL $\subseteq P$.

Proof. Theorem 8.25 shows that any language in NL is log space reducible to $PATH$. Recall that a TM 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 in turn is in P, by Theorem 7.14. We know that every language that is polynomial time reducible to a language in P is also in P, so the proof is complete. □

8.6 NL Equals coNL

The classes NP and coNP are generally believed to be different.
A language $A \in$ coNL if $\overline{A} \in$ NL.
A language $B \in$ NL, if there's an NDTM M in log space, such that given $w \in B$, some path of M accepts.
A language $B \in$ coNL, if $w \in B$ all branch accepts, $w \notin B$ some paths reject.
$\overline{PATH} = \{\langle G, s, t \rangle \mid \text{no path from } s \text{ to } t\}$.

ITOC - Theorem 8.27

NL = coNL.

Proof. Show that $\overline{PATH} \in$ NL and establish that every problem in coNL is also in NL, because $PATH$ is NLC. If $\overline{PATH} \in$ NL, then $PATH \in$ coNL and since $PATH$ is NLC, then coNL $\subseteq$ NL. The NL algorithm M for $\overline{PATH}$ must have an accepting computation whenever the input graph G does not contain a path from s to t.

Let c be the number of nodes in G that are reachable from s. Assume that c is given and show how to use c to solve $\overline{PATH}$.
Given G, s, t and c, the machine M operates as follows. One by one, M goes though all the m nodes of G and nondeterministically guesses whether each one is reachable from s. Whenever a node u is guessed to be reachable, M attempts to verify this guess by guessing a path of length m or less from s to u. If a computation branch fails to verify this guess, it rejects. In

addition, if a branch guesses that t is reachable, it rejects.M counts the number of nodes that have been verified to be reachable. When a branch has gone through all of G's nodes, it checks that the number of nodes that it verified to be reachable from s equals c, rejects if not.
M nondeterministically selects exactly c nodes reachable from s, not including t, and proves that each is reachable from s by guessing the path, M knows that the remaining nodes, including t, are not reachable, so it can accept.

Next, describe a ND log space procedure whereby at least one computation branch has the correct value for c and all other branches reject.
Define A_i for $i = 0, \ldots, m$ to be the collection of nodes that are at a distance of i or less from s. So $A_0 = \{s\}$, each $A_i \subseteq A_{i+1}$ and A_m contains all nodes that are reachable from s. Lt c_i be the number of nodes in A_i. Then calculate c_{i+1} from c_i.
The algorithm goes through all the nodes of G, determines whether each is a member of A_{i+1} and counts the members.
To determine whether a node v is in A_{i+1}, use an inner loop to go through all the nodes of G and guess whether each node is in A_i. Each positive guess is verified by guessing the path of length at most i from s. For each node u verified to be in A_i, the algorithm tests whether (u, v) is an edge of G. If it is an edge, v is in A_{i+1}. The number of nodes verified to be in A_i is counted. At the completion of the inner loop, if the total number of nodes verified to be in A_i is not c_i, all A_i have not been found, so this branch rejects. If the count equals c_i and v has not yet been shown to be in A_{i+1}, then $v \notin A_{i+1}$. Then go on to the next v in the outer loop.

NL = coNL
$\overline{PATH} \in$ NL, therefore $PATH \in$ coNL.
$\forall A \in$ NL, since $PATH$ is NL-complete, $A \leq_{\mathrm{L}} PATH$, and $\overline{A} \leq_{\mathrm{L}} \overline{PATH}$. But since $\overline{PATH} \in$ NL, $\overline{A} \in$ NL, and $A \in$ coNL. Thus $\forall A \in$ NL, $A \in$ coNL, and consequently NL $\subseteq$ coNL.
$\forall A \in$ coNL, $\overline{A} \in$ NL, thus $\overline{A} \leq_{\mathrm{L}} PATH$, and $A \leq_{\mathrm{L}} \overline{PATH}$. But $\overline{PATH} \in$ NL. Thus, $\forall A \in$ coNL, $A \in$ NL, and consequently coNL $\subseteq$ NL.
Together NL = coNL. □

8.7 Exercises

8.1 Show that for any function $f : \mathbb{N} \to \mathbb{R}^+$, where $f(n) \geq n$, the space complexity class SPACE($f(n)$) is the same whether you define the class by using the single tape TM model or the two-tape read-only input TM model.

Proof. □

8.4 Show that PSPACE is closed under the operations union, complementation, and star.

Proof. □

8.5 Show that $A_{\mathrm{DFA}} \in L$.

Proof. Construct a TM M to decide A_{DFA}. When M receives input $\langle A, w \rangle$, a DFA and a string, M simulates A on w by keeping track of As current state and its current head location, and updating them appropriately. The space required to carry out this simulation is $O(\log n)$

because M can record each of these values by storing a pointer into its input. □

8.6 Show that any PSPACE-hard language is also NP-hard.

Proof. First we must show that the language is not in NP. This is trivial since NP is a subset of PSPACE, and therefore, anything outside of PSPACE is also outside of NP.
Then we must show that any problem in NP can be reduced to any PSPACE-hard language. This is also fairly simple, since any SAT problem can be reduced to a TQBF problem by simply appending "there exists x_n" to the front of the SAT expression for each variable x_n and then solving it using the TQBF algorithm. Then the TQBF problem can be reduced to any PSPACE-hard problem by the definition of PSPACE-hard because TQBF is PSPACE-complete. Thus, any PSPACE-hard problem is also NP-hard. □

8.7 Show that NL is closed under the operations union, concatenation, and star.

Proof. Let A_1 and A_2 be languages that are decided by NL-machines N_1 and N_2. Construct three Turing machines: $N_\cup$ deciding $A_1 \cup A_2$; $N_\circ$ deciding $A_1 \circ A_2$; and N_* deciding A_1^*. Each of these machines operates as follows.

Machine $N_\cup$ nondeterministically branches to simulate N_1 or to simulate N_2. In either case, $N_\cup$ accepts if the simulated machine accepts.

Machine $N_\circ$ nondeterministically selects a position on the input to divide it into two substrings. Only a pointer to that position is stored on the work tape (insufficient space is available to store the substrings themselves). Then $N_\circ$ simulates N_1 on the first substring, branching nondeterministically to simulate N_1s nondeterminism. On any branch that reaches N_1s accept state, $N_\circ$ simulates N_2 on the second substring. On any branch that reaches N_2s accept state, $N_\circ$ accepts.

Machine N_* has a more complex algorithm, so we describe its stages.

N_* = On input w:

1. Initialize two input position pointers p_1 and p_2 to 0, the position immediately preceding the first input symbol.
2. Accept if no input symbols occur after p_2.
3. Move p2 forward to a nondeterministically selected position.
4. Simulate N_1 on the substring of w from the position following p_1 to the position at p_2, branching nondeterministically to simulate N_1s nondeterminism.
5. If this branch of the simulation reaches N_1s accept state, copy p_2 to p_1 and go to stage 2. If N_1 rejects on this branch, *reject*.

□

8.8 Problems

8.8 Let $EQ_{\text{REX}} = \{\langle R, S\rangle \mid R \text{ and } S \text{ are equivalent REX}\}$. Show that $EQ_{\text{REX}} \in$ PSPACE.

Proof. □

8.11 Show that if every NP-hard language is also PSAPCE-hard, then PSPACE = NP.

Proof. To show that PSPACE = NP, we need to show that PSPACE ⊆ NP and NP ⊆ PSPACE. We already know from the book that NP ⊆ PSPACE. Thus we only need to show that PSPACE ⊆ NP, given that every NP-hard language is also PSPACE-hard.
By definition of NP-hardness, every NP-complete language is also NP-hard, thus by the assumption, is also PSPACE-hard. And because SAT is NP-complete, therefore we know that SAT is also PSPACE-hard.
By definition of PSPACE-hardness, for any language $A \in$ PSPACE, $A \leq_{\mathrm{P}} SAT$ (because SAT is PSPACE-hard). We then want to show that for any such language $A \in$ PSPACE, A is also in NP (PSPACE ⊆ NP).
By definition of polynomial time mapping reduction, if $A \leq_{\mathrm{P}} SAT$, then there exists a TM M which computes function f on w such that $w \in A$ iff $f(w) \in SAT$.
We then build an NDTM N to decide A in polynomial time (to show $A \in$ NP).

N = on input w:

1. Call M to compute $f(w)$.
2. Use decider (NDTM) for SAT to decide $f(w)$.

We see that N decides A, since $w \in A$ iff $f(w) \in SAT$.
Step 1 is polynomial time, step 2 calls an NDTM which is also polynomial time ($SAT \in$ NP), therefore we claim that $A \in$ NP. Therefore, PSPACE ⊆ NP and together with NP ⊆ PSPACE, we get PSPACE = NP, given the assumption. □

8.12 Show that $TQBF$ restricted to formulas where the part following the quantifiers is in conjunctive normal form is still PSPACE-complete.

Proof. □

8.15 Consider the following two-person version of the language $PUZZLE$ that was described in Problem 7.28. Each player starts with an ordered stack of puzzle cards. The players take turns placing the cards in order in the box and may choose which side faces up. Player I wins if all hole positions are blocked in the final stack, and Player II wins if some hole position remains unblocked. Show that the problem of determining which player has a winning strategy for a given starting configuration of the cards is PSPACE-complete.

Proof. □

8.16 Read the definition of $MINFORMULA$ in Problem 7.46.
a. Show that $MINFORMULA \in$ PSPACE.

Proof. □

b. Explain why this argument fails to show that $MINFORMULA \in$ coNP:
If $\phi \notin MINFORMULA$, then ϕ has a smaller equivalent formula. An NTM can verify that $\phi \in MINFORMULA$ by guessing that formula.

Proof. □

8.17 Let A be the language of properly nested parentheses. For example, (()) and (()(()))() is in B but)(is not. Show that A is in L.

Proof. The following TM M decides A in logspace

M = On input w:

1. Initialize a counter to 0 and start scanning from left to right.
2. If the next symbol is (, add 1 to counter. If the next symbol is), subtract 1 from counter.
3. If counter value ever goes below zero, reject.
4. If after reading all symbols, counter is not 0, reject.
5. Otherwise, accept.

The only space needed is the counter, thus $A \in$ L. □

8.18 Let B be the language of properly nested parentheses and brackets. For example, ([()()]()[]) is in B but (]) is not. Show that B is in L.

Proof. The following machine M decides B in logspace

M = On input w:

1. If $w = \epsilon$, accept. Otherwise initialize

□

8.20 Let $MULT = \{a\#b\#c \mid a, b, c$ are binary natural numbers and $a \times b = c\}$. Show that $MULT \in$ L.

Proof. Let us assume, with out loss of generality, that $a = a_n a_{n-1} \cdots a_2 a_1$ and $b = b_n b_{n-1} \cdots b_2 b_1$ where each a_i and b_i can be 0 or 1. We show that the following algorithm uses only log space.

1. Initialize a counter $i = 1$, and start counting from 1 to n.
2. Initialize another counter $k = 1$, which point to a bit in c.

3. For each i, calculate the corresponding bit of $a \times b$ using the formula

$$\sum_{j=1}^{i} a_j b_{i+1-j} = a_1 b_i + a_2 b_{i-1} \cdots + a_i b_1$$

This is binary addition of single bits, and we just need log space to store the counter j, record the carry d and update it every time.

4. While doing the above steps, check the corresponding bit in c. If any bit disagree, *reject*.

5. Once i reaches n, start decreasing n back to 1. This time use a different formula $\sum_{j=1}^{i} a_{n+1-j} b_{n-i+j}$. Again, update the carry d, and check corresponding bit in c.

6. If all bits agree with c, *accept*

This algorithm uses log space because we just need a few counters, each in log space. we also need some space to store the carry which is also log space. Finally we need a single bit to store the resulting bit of the addition. Therefore, $MULT \in$ L. □

8.21 For any positive integer x, let x^R be the integer whose binary representation is the reverse of the binary representation of x. (Assume no leading 0s in the binary representation of x.) Define the function $R^+ : \mathbb{N} \to \mathbb{N}$ where $R^+(x) = x + x^R$.
a. Let $A_2 = \{\langle x, y\rangle \mid R^+(x) = y\}$. Show $A_2 \in$ L.

Proof. □

b. Let $A_3 = \{R^+(R^+(x)) = y\}$. Show $A_3 \in$ L.

Proof. □

8.22
a. Let $ADD = \{\langle x, y, z\rangle \mid x, y, z > 0$ are binary integers and $x + y = z\}$. Show that $ADD \in$ L.

Proof. The obvious binary addition algorithm working from least significant bit to most significant bit can be implemented in log-space. On the worktape we just need to keep track of which bit of x and u we are adding next and what the carry from the previous bits is. □

8.23 Define

$$UCYCLE = \{\langle G\rangle \mid G \text{ is an undirected graph that contains a simple cycle}\}$$

Show that $UCYCLE \in$ L. (Note: G may be a graph that is not connected.)

Proof. We can try to search the tree by always traversing the edges incident on a vertex in lexicographic order i.e. if we come in through the ith edge, we go out through the $(i+1)$ mod d-th edge, where d is the degree of the vertex. This process performs a DFS on a tree and we always come back to a vertex through the edge we went out on (if it doesn't contain a cycle). However, if the graph contains a cycle, there must exist at least one vertex u and at

least one starting edge (u, v) such that if we start the traversal through (u, v), we will come back to u through an edge different than (u, v). Hence, we enumerate all the vertices and all the edges incident on them and start a traversal through each one of them. If we come back to the starting vertex through an edge different than the one we started on, declare that the graph contains a cycle. Since we can enumerate all vertices and edges in log-space and also remember the starting vertex and edge using log-space, $UCYCLE \in$ L. □

8.25 An undirected graph is **bipartite** if its nodes may be divided into two sets so that all edges go from a node in one set to a node in the other set. Show that a graph is bipartite if and only if it does not contain a cycle that has an odd number of nodes. Let $BIPARTITE = \{\langle G \rangle \mid G \text{ is a bipartite graph}\}$. Show that $BIPARTITE \in$ NL.

Proof. First show that if a graph is bipartite, it must not contain a cycle with an odd number of nodes. Suppose it did contain such a cycle. Label the nodes $n_1, n_2, \ldots, n_{2k}, n_{2k+1}$. Clearly, if n_1 is in some set A, then n_2 must be in set B, so n_3 must be in set A, etc. By induction, all the nodes with an odd subscript must be in set A, and all those with an even subscript must be in set B. But this implies that n_1 and n_{2k+1} are both in set A, a contradiction because they are connected.

Second, we show that if a graph contains no cycles with an odd number of nodes, then the graph is bipartite. Suppose a graph does not contain any odd cycles. Pick a node, and label it A. Label all of its neighbors B. Label all of their unlabeled neighbors A, etc, until all nodes are labeled. Suppose that this construction caused two adjacent nodes x and y to have the same label. Then that would mean that both x and y were reached by taking an odd number of steps from the start node, or both were reached by taking an even number of steps. In either case, the total number of nodes traversed in getting to x from the start node, and getting to y from the start node (excluding the start node itself), is even. But adding the start node in makes the total number of nodes odd, contradicting the hypothesis that there were no odd cycles. Thus the construction succeeds in properly dividing the nodes, so the graph is bipartite.

Finally, we show that $BIPARTITE \in$ NL. Since NL = coNL, it suffices to demonstrate that $\overline{BIPARTITE} \in$ NL, where

$$\overline{BIPARTITE} = \{\langle G \rangle \mid G \text{ is an undirected graph that contains an odd cycle }\}$$

The following machine M works.

M = On input $\langle G \rangle$, where G is a graph

1. Initialize a counter to 0.
2. Nondeterministically select a starting node v.
3. Nondeterministically select an odd length $l \leq |V(G)|$.
4. Nondeterministically select the node in the cycle at each step.
5. If at the end of l steps we come back to v, then we have found an odd cycle, *accept*.

6. If no such cycle found, *reject*.

□

8.26 Define $UPATH$ to be the counterpart of $PATH$ for undirected graphs. $UPATH = \{\langle G, s, t\rangle \mid G$ is an undirected graph that has a path from s to $t\}$. Show that $\overline{BIPARTITE} \leq_{\mathrm{L}} UPATH$.

Proof.

$$\overline{BIPARTITE} = \{\langle G\rangle \mid G \text{ is an undirected graph that contains an odd cycle }\}$$

□

8.27 A directed graph is **strongly connected** if every two nodes are connected by a directed path in each direction. Let

$$STRCON = \{\langle G\rangle \mid G \text{ is a strongly connected graph}\}$$

Show that $STRCON$ is NL-complete

Proof. First we show that $STRCON \in$ NL. Since NL = coNL, we consider the language $\overline{STRCON} = \{\langle G\rangle \mid G$ is not a strongly connected graph$\}$, this means that there exists two nodes such that they are not connected by a directed path. The following NDTM M decides $\overline{STRCON}$.

$M =$ On input $\langle G\rangle$, where G is a directed graph,

1. Nondeterministically select two nodes a and b
2. Run decider for $PATH$ on $\langle G, a, b\rangle$. If it rejects, then the graph is not strongly connected, *accept*; Otherwise, *reject*.

Since storing two nodes a and b takes log-space and $PATH \in$ NL, so $\overline{STRCON} \in$ NL. Again, since NL = coNL, $STRCON \in$ NL.

Next we show that $STRCON$ is NL-hard, which means every language in NL is log-space reducible to $STRCON$. We do this by showing that $PATH \leq_{\mathrm{L}} STRCON$, since $PATH$ is NL-complete. The following log space transducer that computes the reduction function f.

$N =$ On input $\langle G, s, t\rangle$:

1. Copy all of G onto the output tape.
2. For each node v in G, add edges (v, s) and (t, v).

If there is a path from s to t, the constructed graph is strongly connected, because every node can now get to every other node by going through the $s - t$ path. If there is not a path from s to t, then the constructed graph is not strongly connected because the only additional edges in the constructed graph go into s and out of t, so there can be no new ways of getting from s to t. □

8.28 Let

$$BOTH_{\text{NFA}} = \{\langle M_1, M_2\rangle \mid M_1 \text{ and } M_2 \text{ are NFAs where } L(M_1) \cap L(M_2) \neq \emptyset\}$$

Show that $BOTH_{\text{NFA}} \in$ NLC.

Proof. □

8.29 Show that A_{NFA} is NL-complete.

Proof. First show that $A_{\text{NFA}} \in$ NL. The following machine M decides A_{NFA} in nondeterministic logspace.

$M =$ On input $\langle N, w\rangle$.

1. Simulating N on w via reading each symbol of w one by one.
2. Put a marker on the start state.
3. On reading each symbol, nondeterministically select which transition from the current state. Move the marker correspondingly.
4. If the marker is every put on an accepting state, *accept*; otherwise, *reject*

Then only space required is the space to store the marker, hence it is in logspace.

Then show that $PATH \leq_{\text{L}} A_{\text{NFA}}$. Hence it is NL-complete.
Given an instance $\langle G, s, t\rangle \in PATH$, the reduction computes $\langle M, w\rangle \in A_{\text{NFA}}$. The reduction works as follow:
Modify N so that there is only a single accepting state, this can be done by adding a new final states and add ϵ-transition from the old final states to the new final states. Call this new NFA N'.
Then map nodes of G to states of N', edges of G to transitions between states, s is the start state and t is the final state. This reduction works in logspace, since it can just output nodes and edges onto the output tape of the logspace transducer.
Then if G has a path from s to t, then there will be a computation path from the start state to the final state of N' and N will accept w. On the other hand, if G has no path from s to t, then there will be no computation path from the start state to the final state, hence N will not accept w. □

8.30 Show that E_{DFA} is NL-complete.

Proof. Since NL = coNL, it suffices to show that E_{DFA} is coNL-complete, or $\overline{E_{\text{DFA}}}$ is NL-complete.
First show that $\overline{E_{\text{DFA}}} \in$ NL. The following machine M decides $\overline{E_{\text{DFA}}}$ in nondeterministic logspace.

$M =$ On input $\langle D\rangle$:

1. Nondeterministically guess a string w that is accepted by D.

2. Verify this by simulating D on w, if simulation agrees, *accept*.

3. If no such string founds, *reject*.

This machine works in logspace because simulating DFA on a string takes only logspace, since we only need a few pointers to keep track of the current state and symbol.

Then show that $\overline{E_{\text{DFA}}}$ is NL-complete, by showing that $\overline{PATH} \leq_{\text{L}} E_{\text{DFA}}$. The reduction works as follow:
Given a directed graph G and vertices s and t, construct a DFA D, such that states of the DFA is the vertices of G, the start state is s and the final state is t. The alphabet is $\{1, 2, \ldots, d\}$ where d is the maximum out-degree of G. Edges in G will be transitions in D, if a vertex in G has less than d out-going edges, the rest will just loop back to the same state (vertex).
Now if there is no path from s to t in G, then no computation path in D ends in the final states, which means $L(D) = \emptyset$. On the other hand, if there exists a path from s to t in G, then there will be a computation path in D ends in the final states, and $L(D) \neq \emptyset$. □

8.31 Show that $2SAT$ is NL-complete.

Proof. We want to show that $2SAT$ is NL-complete.

$$2SAT = \{\langle\phi\rangle \mid \phi \text{ is a 2cnf satisfiable Boolean formula}\}$$

Since NL = coNL, then it suffice to show that $\overline{2SAT}$ is NL-complete.

We first show that $\overline{2SAT} \in$ NL. Given $\phi \in \overline{2SAT}$, construct a directed graph $G = (V, E)$. For each variable x_i in ϕ, G will have two vertices x_i and $\overline{x_i}$. And G has an edge (u, v), if $(\overline{u} \vee v)$ is a clause in ϕ. For $u \neq \overline{w}$, if there is a path from u to w in G, then there is also a path from $\overline{w}$ to $\overline{u}$ by reversing the direction of each edge along the path and replacing the literals of each vertex by their complements.
To show that this construction of the graph works, we want to show that $\phi \in \overline{2SAT}$ iff there exists a variable x in ϕ, such that there is a path in G from x_i to $\overline{x_i}$ and one from $\overline{x_i}$ to x_i.
First, if such x_i does exists, then it means that $x \Rightarrow \overline{x}$ and $\overline{x} \Rightarrow x$, this is a contradiction so ϕ is not satisfiable.
Second, if ϕ is not satisfiable, we can find a variable x_i such that there are paths from x_i to $\overline{x_i}$ and from $\overline{x_i}$ to x_i. **TODO: how to prove this?**.
With this graph construction, we give an NDTM N that decides $\overline{2SAT}$ in log space.

N = On input $\langle\phi\rangle$ where ϕ is a 2cnf Boolean formula:

1. Construct G as described above. In fact, we don't need to construct the graph explicitly, we could use ϕ as some sort of representation of the graph.

2. Nondeterministically select a vertex u in G.

3. Use NL decider for $PATH$ to decide $\langle G, \overline{u}, u\rangle$ and $\langle G, u, \overline{u}\rangle$.

4. If both accept, *accept*; otherwise, *reject*

This shows that $\overline{2SAT} \in$ NL.

We then show that $\overline{2SAT}$ is NL-hard, by showing $PATH \leq_{\mathrm{L}} \overline{2SAT}$. To this end, we wish to show a log-space computable function f such that $\langle G, s, t\rangle \in PATH$ iff $f(\langle G, s, t\rangle) \in \overline{2SAT}$. The log-space transducer works as follows:
Given an instance $\langle G, s, t\rangle \in PATH$, where G has $m+2$ vertices $\{s, t, v_1, \ldots, v_m\}$. The resulting 2cnf Boolean formula ϕ will have variables $\{x, y_1, \ldots y_m\}$, where we label s with x, t with $\overline{x}$, and each v_i with y_i. Similar to the previous graph construction, ϕ will have a clause $(\overline{u} \vee v)$ if (u, v) is an edge of G. It will also have an extra clause $(x \vee x)$ to force $x = 1$.
To show that this construction works, if $\langle G, s, t\rangle \in PATH$, that is, there exists a path from s to t. Then this path is of the form $(s, u_1, \ldots, u_k, t)$. Per the above construction, we know that $\phi = (x \vee x) \wedge (\overline{x} \vee u_1) \wedge (\overline{u_1} \vee u_2) \wedge \cdots \wedge (\overline{u_k} \vee \overline{x})$. This formula is obviously not satisfiable for both $x = 0$ and $x = 1$. Therefore, $\langle \phi\rangle \in \overline{2SAT}$.
For the other direction, we want to show that if $\langle \phi\rangle \in \overline{2SAT}$, then $\langle G, s, t\rangle \in PATH$. To this end, we show the contrapositive which is if $\langle G, s, t\rangle \in \overline{PATH}$, then $\langle \phi\rangle \in 2SAT$. Suppose that G has no path from s to t. Let V_s be the set of vertices that is reachable from s and V_t be the set of vertices from which t is reachable, and U be the rest of vertices in G. And they are pairwise disjoint. A satisfiable assignment would then assign true to all vertices in $V_s \cup U$ and false to all vertices in V_t.
Reference: http://cs.brown.edu/people/jes/book/Page.363.364.pdf □

8.34 Define

$$CYCLE = \{\langle G\rangle \mid G \text{ is a directed graph that contains a directed cycle}\}$$

Show that $CYCLE \in$ NLC.

Proof. First show that $CYCLE \in$ NL, the following NDTM M decides $CYCLE$ in log space.

$M =$ On input $\langle G\rangle$, where G is a directed graph:

1. Nondeterministically select a node v.
2. Nondeterministically select the next node from among those pointed at by the current node.
3. Repeat this action until it reaches node v again and *accepts*, or until it has gone on for $n = |G|$ steps and *rejects*.

Next, we show that $PATH \leq_{\mathrm{L}} CYCLE$. □

9 Intractability

9.1 Hierarchy Theorems

ITOC - Definition 9.1

A function $f : \mathbb{N} \to \mathbb{N}$, where $f(n)$ is at least $O(\log n)$, is called **space constructible** if the function that maps the string 1^n to the binary representation of $f(n)$ is computable in space $O(f(n))$.

f is space constructible if some $O(f(n))$ space TM exists that always halts with the binary representation of $f(n)$ on its tape when started on input 1^n.
All commonly occurring functions that are at least $O(\log n)$ are space constructible, including the functions $\log_2 n, n \log 2_n, n^2$.
If $f(n)$ and $g(n)$ are two space bounds, where $f(n)$ is asymptotically larger than $g(n)$, we would expect a machine to be able to decide more languages in $f(n)$ space than in $g(n)$ space.

ITOC - Theorem 9.3

Space hierarchy theorem - For any space constructible function $f : \mathbb{N} \to \mathbb{N}$, a language A exists that is decidable in $O(f(n))$ space but not in $o(f(n))$ space.

ITOC - Corollary 9.4

For any two functions $f_1, f_2 : \mathbb{N} \to \mathbb{N}$, where $f_1(n)$ is $o(f_2(n))$ and f_2 is space constructible, $\text{SPACE}(f_1(n)) \subset \text{SPACE}(f_2(n))^2$

ITOC - Corollary 9.5

For any two real numbers $0 \leq \epsilon_1 < \epsilon_2$, $\text{SPACE}(n^{\epsilon_1}) \subset \text{SPACE}(n^{\epsilon_2})$.

ITOC - Corollary 9.6

NL $\subset$ PSPACE.

Proof. Savitch's theorem shows that $\text{NL} \subseteq \text{SPACE}(\log^2 n)$, and the space hierarchy theorem show that $\text{SPACE}(\log^2 n) \subset \text{SPACE}(n)$. Hence, NL $\subset$ PSPACE. □

$\text{SPACE}(n^k) \subset \text{SPACE}(n^{\log_n}) \subset \text{SPACE}(2^n)$

ITOC- Corollary 9.7

PSPACE $\subset$ EXPSPACE.

ITOC - Definition 9.8

A function $t : \mathbb{N} \to \mathbb{N}$, where $t(n)$ is at least $O(n \log n)$, is called **time constructible** if the function that maps the string 1^n to the binary representation of $t(n)$ is computable in time $O(t(n))$.

t is time constructible if some $O(t(n))$ time TM exists that always halts with the binary representation of $t(n)$ on its tape when started on input 1^n.
All commonly occurring functions that are at least $n \log n$ are time constructible, including the functions $n \log n, n\sqrt{n}, n^2, 2^n$.

ITOC - Theorem 9.10

Time hierarchy theorem - For any time constructible function $t : \mathbb{N} \to \mathbb{N}$, a language A exists that is decidable in $O(t(n))$ time but not decidable in time $o(t(n)/\log t(n))$.

ITOC - COROLLARY 9.11

For any two functions $t_1, t_2 : \mathbb{N} \to \mathbb{N}$, where $t_1(n)$ is $o(t_2(n)/\log t_2(n))$ and t_2 is time constructible, $\text{TIME}(t_1(n)) \subset \text{TIME}(t_2(n))$.

ITOC - COROLLARY 9.12

For any two real numbers $1 \leq \epsilon_1 < \epsilon_2$, we have $\text{TIME}(n^{\epsilon_1}) \subset \text{TIME}(n^{\epsilon_2})$.

9.1.1 Exponential Space Completeness

ITOC - Definition 9.14

A language B is **EXPSPACE-complete** if

1. $B \in$ EXPSPACE, and
2. $\forall A \in$ EXPSPACE, $A \leq_{\text{P}} B$.

9.2 Exercises

9.1 Prove that $\text{TIME}(2^n) = \text{TIME}(2^{n+1})$.

Proof. $\text{TIME}(t(n))$ is the collection of languages that are decidable by $O(t(n))$ TM. Since $O(2^{n+1}) = O(2^n)$, $\text{TIME}(2^n) = \text{TIME}(2^{n+1})$. □

9.2 Prove that $\text{TIME}(2^n) \subset \text{TIME}(2^{2n})$.

Proof. TIME(2^n) $\subseteq$ TIME(2^{2n}) because $2^n \leq 2^{2n}$.
The function 2^{2n} is time constructible because a TM can write the number 1 followed by 2n 0s in $O(2^{2n})$ time.
$t_1(n) = 2^n, t_2(n) = 2^{2n}$, want to show that $t_1(n) = o(t_2(n)/\log t_2(n))$.

$$\lim_{n\to\infty} \frac{t_1(n)}{t_2(n)/\log t_2(n)} = \lim_{n\to\infty} \frac{2^n}{2^{2n}/\log 2^{2n}}$$
$$= \lim_{n\to\infty} \frac{2^n \log_2 2^{2n}}{2^{2n}} = 0$$

So $t_1(n) = o(t_2(n)/\log t_2(n))$ and thus TIME(2^n) $\subset$ TIME(2^{2n}). □

9.3 Prove that NTIME(n) $\subset$ PSPACE.

Proof. NTIME(n) $\subset$ NSPACE(n) because any TM that operates in time $t(n)$ on every computation branch can use at most $t(n)$ tape cells on every branch. Furthermore, NSPACE(n) $\subseteq$ SPACE(n^2) due to Savitch's theorem. However, SPACE(n^2) $\subset$ SPACE(n^3) because of the space hierarchy theorem. And SPACE(n^3) $\subseteq$ PSPACE. Thus NTIME(N) $\subset$ PSPACE. □

9.3 Problems

9.12 Describe the error in the following fallacious "proof" that P $\neq$ NP. Assume that P = NP and obtain a contradiction. If P = NP, then $SAT \in$ P and so for some k, $SAT \in$ TIME(n^k). Because every language in NP is polynomial time reducible to SAT, we have NP $\subseteq$ TIME(n^k). Therefore, P $\subseteq$ TIME(n^k). But by the time hierarchy theorem, TIME(n^{k+1}) contains a language that isn't int TIME(n^k), which contradicts P $\subseteq$ TIME(n^k). Therefore, P $\neq$ NP.

Proof. In the step where every language in NP is polynomial time reducible to SAT, the time of the reduction is not taken into account. □

9.13 Consider the function $pad : \Sigma^* \times \mathbb{N} \to \Sigma^*\#^*$ that is defined as follows. Let $pad(s, l) = s\#^j$, where $j = \max(0, l - m)$ and m is the length of s. Thus $pad(s, l)$ simply adds enough copies of the new symbol # to the end of s so that the length of the result is at least l. For any language A and function $f : \mathbb{N} \to \mathbb{N}$, define the elanguage $pad(A, f)$ as

$$pad(A, f) = \{pad(s, f(m)) \mid \text{where } s \in A \text{ and } m \text{ is the length of } s\}$$

prove that if $A \in$ TIME(n^6), then $pad(A, n^2) \in$ TIME(n^3).

Proof. Let M decides A in $O(n^6)$ time. We build a TM M' that decides $pad(A, n^2)$ in $O(n^3)$ time.

M' = On input w, where $|w| = n$

1. Check if w is of the form $pad(s, |s|^2)$, where s is some string. If not, reject.
2. Run M on s, and output whatever M outputs.

Step 1 will take one pass of the input so it will take $O(n)$ time.
Step 2 will take $O(|s|^6)$ time, but since $|s|^2 = n$, this takes $O(n^3)$ times.
Thus the total running time is $O(n^3)$ and thus $pad(A, f) \in \text{TIME}(n^3)$. □

9.14 Prove that if NEXPTIME $\neq$ EXPTIME, then P $\neq NP$. You may find the function *pad*, defined in Problem 9.13, to be helpful.

Proof. We prove the contrapositive, if P = NP, then NEXPTIME = EXPTIME.
Let $A \in$ NEXPTIME, then there is some positive integer c, such that $A \in \text{NTIME}(2^{n^c})$. And also $pad(A, 2^{n^c}) \in$ NP. since P = NP, $pad(A, 2^{n^c}) \in$ P. So $A \in \text{TIME}(2^{n^c}) \subseteq$ EXPTIME.
Thus EXPTIME = NEXPTIME. □

9.16 Prove that $TQBF \notin \text{SPACE}(n^{1/3})$.

Proof. We prove this by contradiction. Assume $TQBF \in \text{SPACE}(n^{1/3})$. According to space hierarchy theorem, there is some language A, where $A \notin \text{SPACE}(n)$, but $A \in \text{SPACE}(n^{1+\epsilon})$ where $\epsilon > 0$.
Since $TQBF$ is PSPACE-complete, by definition, any language in PSPACE reduces to $TQBF$ in poly time or in log space. Obviously A is such a language and $A \leq_{\text{P}} TQBF$ or $A \leq_{\text{L}} TQBF$. Let the output of this reduction on A take n^k space. Since $TQBF \in \text{SPACE}(n^{1/3})$, we have $A \in \text{SPACE}(n^{k/3})$. If $k = 3$, then $A \in \text{SPACE}(n)$, a contradiction. Thus $TQBF \notin \text{SPACE}(n^{1/3})$. □

10 Advanced Topics in Complexity Theory

10.1 Approximation Algorithms

10.2 Probabilistic Algorithms

10.2.1 The Class BPP

ITOC - Definition 10.3

A **probabilistic Turing machine** M is a type of nondeterministic Turing machine in which each nondeterministic step is called a **coin-flip step** and has two legal next moves. We assign a probability to each branch b of M's computation on input w as follows.
Define the probability of branch b to be $\Pr[b] = 2^{-k}$, where k is the number of coin-flip steps that occur on branch b.
Define the probability that M accepts w to be

$$\Pr[M \text{ accepts } w] = \sum_{b \text{ is accepting}} \Pr[b]$$

$$\Pr[M \text{ rejects } w] = 1 - \Pr[M \text{ accepts } w]$$

When a PTM decides a language, it must accept all strings in the language and reject all string out of the language as usual, except that we allow a small probability of error. For $0 \leq \epsilon < 0.5$, say that M **decides language** A **with error probability** ϵ if

1. $w \in A \rightarrow \Pr[M \text{ accepts } w] \leq 1 - \epsilon$, and
2. $w \notin A \rightarrow \Pr[M \text{ rejects } w] \leq 1 - \epsilon$

ITOC- Definition 10.4

BPP is the class of languages that are decided by probabilistic polynomial time Turing machines with an error probability of $\frac{1}{3}$.

Any constant error probability would yield an equivalent definition as long as it is strictly between 0 and $\frac{1}{2}$ by virtue of the **amplification lemma**.

ITOC - Lemma 10.5

Let ϵ be a fixed constant strictly between 0 and $\frac{1}{2}$. Then for any polynomial $p(n)$ a PPTM M_1 that operates with error probability ϵ has an equivalent PPTM M_2that operates with an error probability of $2^{-p(n)}$.

Proof. M_2 simulates M_1 by running it a polynomial number of times and taking the majority vote of the outcomes. The probability of error decreases exponentially with the number of runs of M_1 made.

Given TM M_1 deciding a language with an error probability of $\epsilon < \frac{1}{2}$ and a polynomial $p(n)$, we construct a TM M_2 that decides the same language with an error probability of $2^{-p(n)}$.

M_2 = On input x;

1. Calculate k.
2. Run $2k$ independent simulations of M_1 on input x.
3. If most runs of M_1 accept, then *accept*; otherwise, *reject*

Stage 2 yields a sequence of $2k$ results from simulating M_1. If most of these results are correct, M_2 gives the correct answer. We bound the probability that at least half of these results are wrong.
Let S be any sequence of results that M_2 might obtain in stage 2.Let P_S be the probability M_2 obtains S. Say that S has c correct results and w wrong results, so $c + w = 2k$. If $c \leq w$ and $M-2$ obtains S, then M_2 is incorrect, call such an S a bad sequence. Let ϵ_x be the probability that M_1 is wrong on x. If S is any bad sequence, then $P_S \leq (\epsilon_x)^w(1-\epsilon_x)^c$, which is at most $\epsilon^w(1-\epsilon)^c$, because $\epsilon_x \leq \epsilon \leq \frac{1}{2}$, and $c \leq w$. Furthermore, $\epsilon^w(1-\epsilon)^c$ is at most $\epsilon^k(1-\epsilon)^k$ because $k \leq w$ and $\epsilon \leq 1 - \epsilon$.

Summing P_S for all bad sequences S gives the probability that M_2 is incorrect. We have at most 2^{2k} bad sequences because 2^{2k} is the umber of all sequences. Hence

$$\Pr[M_2 \text{ incorrect on } x] = \sum_{\text{bad } S} P_S \leq 2^{2k} \cdot \epsilon^k (1-\epsilon)^k = (4\epsilon(1-\epsilon))^k$$

Since $\epsilon < \frac{1}{2}$, $4\epsilon(1-\epsilon) < 1$. Therefore the above probability decreases exponentially in k so does M_2's error probability. To calculate a specific value of k that allows us to bound M_2's error probability by 2^{-t} for any $t \geq 1$, let $\alpha = -\log_2(4\epsilon(1-\epsilon))$ and choose $k \geq t/\alpha$. Then we an error probability of $2^{-p(n)}$ within polynomial time. □

The probabilistic primality algorithm has **one-sided error**. When the algorithm outputs *reject*, the input must be composite. When the output is *accept*, the input could be prime or composite. Thus, an incorrect answer can only occur when the input is a composite number.

ITOC - Definition 10.10
RP is the class of language that are decided by PPTM where inputs in the language are accepted with a probability of at least $\frac{1}{2}$, and inputs not in the language are rejected with a probability of 1.

P ⊆ RP ⊆ BPP.

Give $L \in$ RP, run M for L twice, get an algorithm for BPP. Since $\frac{1}{2^2} = \frac{1}{4} < \frac{1}{3}$.

RP ⊆ NP.
$L \in \text{RP} \Rightarrow L \in \text{NP}$.

M_1 is a PTM, $x \in L$, M_1 accepts with probability $\geq \frac{1}{2}$, some sequences of coin-flips that causes M_1 to accept. and the sequence of coin-flips is a certificate for x.
If $x \in L$, no coin-flips causes M_1 to accept, therefore no certificate cause an NDTM to accept.

10.3 Alternation

10.4 Interactive Proof Systems

Interactive proof systems provide a way to define a probabilistic analog of the class NP, much like probabilistic polynomial time algorithms provide a probabilistic analog to P.
A Prover that finds the proofs of membership, and a Verifier that checks them.

10.4.1 Graph Non-isomorphism

Call graphs G and H **isomorphic** if the nodes of G maybe reordered so that it is identical to H. Let

$$ISO = \{\langle G, H\rangle \mid G \cong H\}$$

$ISO \in$ NP.

$$NONISO = \{\langle G, H\rangle \mid G \ncong H\}$$

The Verifier randomly selects either G_1 or G_2 and then randomly reorders its nodes to obtain a graph H. The Verifier sends H to the Prover. The Prover must respond by declaring whether $_1$ or G_2 was the source of H.
If G_1 and G_2 were nonisomorphic, the Prover could always carry out the protocol because the Prover could identify whether H came from G_1 or G_2. However, if the graphs were isomorphic, H might have come from either G_1 or G_2. So even with unlimited computational power, the Prover would have no better than a 50-50 chance of getting the correct answer. Thus, if the Prover is able to answer correctly consistently (say in 100 repetitions of the protocol), the Verifier has convincing evidence that the graphs are actually nonisomorphic.

Definition of the Model

ITOC - Definition 10.28

Say that language A is in **IP** if some poly time computable function V exists such that for some (arbitrary) function P and for every (arbitrary) function $\tilde{P}$ and for every string w,

1. $w \in A$ implies $\Pr[V \leftrightarrow P \text{ accepts } w] \geq \frac{2}{3}$, and
2. $w \notin A$ implies $\Pr[V \leftrightarrow \tilde{P} \text{ accepts } w] \leq \frac{1}{3}$.

If $w \in A$ then some Prover P (an "honest" Prover) causes the Verifier to accept with high probability; but if $w \notin A$ then no Prover (not even a "crooked" Prover $\tilde{P}$) causes the Verifier to accept with high probability.
We may amplify the success probability of an interactive proof system by repetition, as we did in Lemma 10.5, to make the error probability exponentially small. NP $\subseteq$ IP, BPP $\subseteq$ IP and $NONISO \in$ IP.

10.4.2 IP = PSPACE

ITOC - Theorem 10.29

IP = PSPACE.

ITOC - Lemma 10.30

IP $\subseteq$ PSPACE.

ITOC - Lemma 10.32

PSPACE $\subseteq$ IP.

ITOC- Theorem 10.33

$\#SAT \in$ IP.

$$\#SAT = \{\langle \phi, k \rangle \mid \phi \text{ is a cnf-formula with exactly } k \text{ satisfying assignments}\}$$

10.5 Exercises

10.3 Prove that if $A \leq_{\mathrm{L}} B$ and B is in NC, then A is in NC.

Proof. The circuit for A on an input w works as follows: It first computes the string y that is the result of applying the reduction. As we saw, L is contained in NC^2 and hence this reduction can be done by a polynomial-sized circuit of depth $\log^2 n$. We then use the NC circuit for B on the output of this reduction, resulting in an overall circuit whose depth is poly-logarithmic (which is just a way of saying that it is some constant power of the log), and whose size is polynomial. □

10.7 Show that BPP $\subseteq$ PSPACE.

Proof. Let M be a PPTM, M can be modified so that it makes exactly n^k coin flips. Then there will be a total of 2^{n^k} computation paths. The problem of determine whether M accepts or not is to count the number of accepting branches B and compare it to $P = \frac{2}{3}(2^{n^k})$. If $B \geq P$, accept; otherwise, reject. This takes only polynomial space because we can simulate all M's computation paths sequentially, using the same amount of space. Each path has n^k configurations and therefore takes only polynomial space. □

10.6 Problems

10.11 Let M be a PPTM, and let C be a language where for some fixed $0 < \epsilon_1 < \epsilon_2 < 1$,

1. $w \notin C$ implies $\Pr[M \text{ accepts } w] \leq \epsilon_1$, and
2. $w \in C$ implies $\Pr[M \text{ accepts } w] \geq \epsilon_2$.

Show that $C \in$ BPP. (Hint: Use Lemma 10.5)

Proof. Obviously, M might not be a BPP machine. Thus we construct another machine N that runs M multiple times and we show that N decides C in BPP.

For real numbers $\epsilon_1 < \epsilon_2$, there exists $c = \frac{\epsilon_1 + \epsilon_2}{2}$, such that $\epsilon_1 < c < \epsilon 2$. N will run M k times. Let A_k be the number of times that M says accept. If $\frac{A_k}{k} \geq c$, N accepts; otherwise N rejects. Thus we have

$$\begin{aligned} w \in C, &\quad \Pr[N \text{ accepts } w] = \Pr[A_k \geq ck] \\ w \notin C, &\quad \Pr[N \text{ rejects } w] = \Pr[A_k < ck] \end{aligned}$$

or looking at the probability of error

$$w \in C, \quad \Pr[N \text{ rejects } w] = \Pr[A_k < ck]$$
$$w \notin C, \quad \Pr[N \text{ accepts } w] = \Pr[A_k \geq ck]$$

To show that $N \in$ BPP, we want to show that when $w \in C$, $\Pr[A_k < ck]$ is strictly less than $\frac{1}{2}$ and when $w \notin C$, $\Pr[A_k \geq ck]$ is also strictly less than $\frac{1}{2}$.

First look at $w \in C$. Let A_k be a random variable, then it is the sum of k random variables $X_1, \ldots X_k$ (each is the output of a call to M), each with a mean $\mu_k \geq \epsilon_2$. Then A_k has a mean $c_2 \geq \epsilon_2$. We also know that $\Pr[A_k < ck] \leq \Pr[|\frac{A_k}{k} - c_2| > c_2 - c]$, and since $c_2 - c \leq \epsilon_2 - c > 0$, we know by Chebyshev inequality that $\lim_{k\to\infty} \Pr[|\frac{A_k}{k} - c_2| > c_2 - c] = 0$. So there exists a k such that this probability is strictly less than $\frac{1}{2}$. Then same with the case when $w \notin C$. Thus by amplification lemma, $N \in$ BPP. □

10.19 Show that if NP $\subseteq$ BPP, then NP = RP.

Proof. First show that RP $\subseteq$ NP.
If $L \in$ RP, then there exists a PPTM M such that if $w \in L$, then M accepts w with probability $\frac{1}{2}$, which means there exists a sequence of coin-flips that causes M to accept w. This sequence of coin-flips is a certificate for w. On the other hand, if $w \notin L$, then M rejects w with probability of 1, which means no sequence of coin-flips exists to cause M to accept w. Therefore, RP $\subseteq$ NP.

Then we show that if NP $\subseteq$ BPP, then NP $\subseteq$ RP. We do this by showing that $SAT \in$ RP. Since SAT is NP-complete, then NP $\subseteq$ RP.
Since $SAT \in$ NP and given that NP $\subseteq$ BPP, $SAT \in$ BPP. Then there exists a PPTM M that decides SAT with two-sided error with error probability $\frac{1}{3}$.
We will construct an RP machine N, such that if $\langle\phi\rangle \in SAT$, then N accepts $\langle\phi\rangle$ with a probability of at least $\frac{1}{2}$, and if $\langle\phi\rangle \notin SAT$, N rejects $\langle\phi\rangle$ with a probability of 1.

The proposed machine N works as follow:
N = On input $\langle\phi\rangle$, where ϕ is a Boolean formula:

1. Run M on $\langle\phi\rangle$, if M rejects, *reject*.
2. Otherwise, let $x_1, \ldots, x_m$ be variables of ϕ and try to find a satisfying assignment for each x_i, starting with $i = 1$.
3. Set $x_i = 0$ and the resulting Boolean formula is ϕ_i.
4. Run M on $\langle\phi_i\rangle$, if M accepts, confirm this assignment; otherwise, set set $x_i = 1$.
5. After all assignments are confirmed, check that if $\phi(x_1, \ldots, x_m) = 1$, if yes, *accept*; if no, *reject*

This machine runs in polynomial time, because it calls M at most $m + 1$ times and $m \leq n$, where n is the number of literals in ϕ. Also check satisfiability is polynomial time. Hence, since M is a PPTM, N is also a PPTM.

This algorithm is correct. If ϕ is not satisfiable, then either step 1 says reject and N reject, or step 1 says accept, and N proceed to try to find a satisfying assignment (and failed to find one) and reject in the end. So the probability that N rejects $\langle\phi\rangle$ when $\langle\phi\rangle \notin SAT$ is 1.

On the other hand, if ϕ is satisfiable, we show that N accepts $\langle\phi\rangle$ with a probability of at least $\frac{1}{2}$. If ϕ is satisfiable, and if M always being correct by saying accept to each $\langle\phi_i\rangle$, then we will get a satisfying assignment in the end. If M ever makes a mistake in any of these m calls, then in the end, N will check this assignment and find out that it is not satisfiable, so it will reject, thus making a mistake itself.

Here we have to use amplification lemma to show that if M is a PPTM with error probability of $\epsilon \in (0, 1/2)$, then there exists another PPTM M' with error probability 2^{-n}, where n is the number of literals in ϕ. Now we replace M with M' in the previous discussion since they are equivalent.

Again, for $\langle\phi\rangle \in SAT$, the probability that M' makes a mistake by outputting reject on $\langle\phi_i\rangle$ is thus 2^{-n}. The probability of such event (N making a mistake resulting from M making at least 1 mistakes) happening is at most $\frac{m}{2^n}$. Again since $m \leq n$, $\frac{m}{2^n} \leq \frac{n}{2^n} \leq \frac{1}{2}$. Consequently, for $\langle\phi\rangle \in SAT$, $\Pr[N \text{ accepts } \langle\phi\rangle] \geq \frac{1}{2}$. Therefore N is an RP machine and $SAT \in$ RP. □

hw6 (3) Suppose M is a Probabilistic Turing Machine that recognizes a language L. On any input, M runs in expected polynomial time, but always gives the right answer. In other words, on some branches of its computation M could take a very long time, but the expected run-time over all its branches is polynomial. Does this imply that the language L is in BPP? Prove your answer. It may be helpful to look at Markov's inequality in elementary probability theory.

Proof. □

10.20 Define a ZPP-machine to be a probabilistic Turing machine that is permitted three types of output on each of its branches: accept, reject, and ?. A ZPP-machine M decides a language A if M outputs the correct answer on every input string w (accept if $w \in A$ and reject if $w \notin A$) with probability at least $\frac{2}{3}$, and M never outputs the wrong answer. On every input, M may output ? with probability at most $\frac{1}{3}$. Furthermore, the average running time over all branches of M on w must be bounded by a polynomial in the length of w. Show that RP $\cap$ coRP = ZPP, where ZPP is the collection of languages that are recognized by ZPP-machines.

Proof. For a ZPP-machine

$$\begin{aligned} w \in A, \quad & \Pr[M \text{ accepts } w] \geq \frac{2}{3} \\ & \Pr[M \text{ outputs ?}] \leq \frac{1}{3} \\ & \Pr[M \text{ rejects } w] = 0 \\ w \notin A, \quad & \Pr[M \text{ accepts } w] = 0 \\ & \Pr[M \text{ outputs ?}] \leq \frac{1}{3} \\ & \Pr[M \text{ rejects } w] \geq \frac{2}{3} \end{aligned}$$

To show that ZPP = RP $\cap$ coRP, we need to show ZPP $\subseteq$ RP $\cap$ coRP and RP $\cap$ coRP $\subseteq$ ZPP.

To show that ZPP $\subseteq$ RP $\cap$ coRP, first show that ZPP $\subseteq$ RP. Let $A \in$ ZPP, then there exists a PTM M that decides A in expected polynomial time, with the probabilities defined above. We construct an RP machine M' using M. M' will run M for at most $2t$ steps, where t is the expected running time of M. If M halts within $2t$ steps, then M' outputs whatever M outputs. If M doesn't halt after $2t$ steps or outputs ?, M' rejects. Hence when $w \in A$, $\Pr[M' \text{ rejects } w] \leq \frac{1}{3}$ and when $w \notin A$, $\Pr[M \text{ rejects } w] = 1$. So $L \in$ RP. By similar arguments, it is easy to show that ZPP $\subseteq$ coRP. Also by amplification lemma, we can change the error probability from $\frac{1}{3}$ to $\frac{1}{2}$. Thus ZPP $\subseteq$ RP $\cap$ coRP.

Then show RP $\cap$ coRP $\subseteq$ ZPP. Let $A \in$ RP $\cap$ coRP. There exists an RP machine M_1 that decides A in polynomial time and a coRP machine M_2 that decides A in polynomial time.

$$\begin{aligned} w \in A, \quad & \Pr[M_1 \text{ accepts } w] \geq \frac{1}{2} \\ & \Pr[M_2 \text{ rejects } w] = 1 \\ w \notin A, \quad & \Pr[M_1 \text{ rejects } w] = 1 \\ & \Pr[M_2 \text{ accepts } w] \geq \frac{1}{2} \end{aligned}$$

We then build a ZPP machine N that decides A in expected polynomial time. N will run both M_1 and M_2 on input w. If M_1 accepts, then it must be the case that $w \in A$, so N accept. If M_2 accepts, then it must be the case that $w \notin A$, so N rejects. Otherwise output ?. Obviously N never makes mistakes, therefore $N \in$ ZPP and hence RP $\cap$ coRP $\subseteq$ ZPP. $\square$

hw6 (7) Let $\phi = (x_1 + x_2 + \overline{x}_3)(\overline{x}_1 + \overline{x_2} + x_3)(\overline{x}_1 + x_2 + x_3)$.
(a) What is the polynomial that results from arithmetizing ϕ?
(b) ϕ has 5 satisfying assignments. What is the true polynomial $\tilde{p}_1(z)$?
(c) Suppose a prover is trying to prove that ϕ has 4 satisfying assignments. What would happen if she sent the true polynomial?
(d) What might be another polynomial that the prover could send to hide the fact that she lied in the first round? For what random integer values do the true polynomial and the provers polynomial

that you constructed have the same value?

Proof. (a)

$$\begin{aligned}
p^1(x_1,x_2,x_3) =& (x_1 \vee x_2 \vee \overline{x}_3) = 1-(1-x_1)(1-x_2)x_3 \\
p^2(x_1,x_2,x_3) =& (\overline{x}_1 \vee \overline{x}_2 \vee x_3) = 1-x_1x_2(1-x_3) \\
p^3(x_1,x_2,x_3) =& (\overline{x}_1 + x_2 + x_3) = 1-x_1(1-x_2)(1-x_3) \\
p(x_1,x_2,x_3) =& p^1(x_1,x_2,x_3)p^2(x_1,x_2,x_3)p^3(x_1,x_2,x_3) \\
=& (1-(1-x_1)(1-x_2)x_3)(1-x_1(1-x_2)(1-x_3))(1-x_1(1-x_2)(1-x_3))
\end{aligned}$$

□

11 Language Membership

$$\mathrm{L} \subseteq \mathrm{NL} = \mathrm{coNL} \subseteq \mathrm{SPACE}(\log^2 n) \subseteq \mathrm{P} \subseteq \mathrm{NP} \subseteq \mathrm{PSPACE} = \mathrm{NSPACE} \subseteq \mathrm{EXPTIME}$$

11.1 Decidable

Closure, union, concatenation, star, complementation, intersection.
A_{DFA}, on input $\langle D, w\rangle$, simulate D on w.
A_{NFA}, on input $\langle N, w\rangle$, convert NFA N to DFA D, then use A_{DFA} decider to decide $\langle D, w\rangle$.
A_{REX}, on input $\langle R, w\rangle$, convert REX R to NFA N, then use A_{NFA} decider to decide $\langle N, w\rangle$.
E_{DFA}, marking algorithm, on input $\langle D\rangle$, mark the start state, then keep marking states that has a transition from a marked state.
EQ_{DFA}, on input $\langle A, B\rangle$, construct C such that $L(C)$ is the symmetric difference of $L(A)$ and $L(B)$, then use E_{DFA} decider to decide $\langle C\rangle$, $L(C) = \emptyset$ iff $L(A) = L(B)$.
$EQ_{\mathrm{DFA,REX}}$, on input $\langle D, R\rangle$, convert REX R to DFA D_R, then use EQ_{DFA} decider to decide $\langle D, D_R\rangle$.
ALL_{DFA}, on input $\langle D\rangle$, construct $\overline{D}$, then use E_{DFA} decider to decide $\langle \overline{D}\rangle$.
$INFINITE_{\mathrm{DFA}}$, on input $\langle A\rangle$, let k be number of states in A, construct DFA D that accepts strings $|w| \geq k$, then construct DFA M such that $L(M) = L(D) \cap L(A)$. Use E_{DFA} decider to on $L(M)$, output opposite.
WWR_{DFA}, on input $\langle M\rangle$, construct DFA N that recognizes $L(M)^R$. Use EQ_{DFA} decider to decide $\langle M, N\rangle$.

11.2 Undecidable

A_{TM}, diagonalization method, assume H is a decider for A_{TM}. Build D using H, on input $\langle M\rangle$, output the opposite of $H(\langle M, \langle M\rangle\rangle)$. Then run D on $\langle D\rangle$, D accepts $\langle D\rangle$ iff D rejects $\langle D\rangle$.
$HALT_{\mathrm{TM}}$, reduction $A_{\mathrm{TM}} \leq HALT_{\mathrm{TM}}$, assume R decides $HALT_{\mathrm{TM}}$, construct S using R to decide A_{TM}. S on input $\langle M, w\rangle$, run R on $\langle M, w\rangle$ to test whether M halts on w. If yes, run M on w to test whether M accepts w, accept if yes. Contradiction.
E_{TM}, reduction $A_{\mathrm{TM}} \leq E_{\mathrm{TM}}$, assume R decides E_{TM}, construct S using R to decide A_{TM}. S on input $\langle M, w\rangle$, construct Q that only accept w if M accepts w, otherwise accepts noting (if $x \neq w$,

reject; if $x = w$, run M on w and accept if M accepts). Run R on $\langle Q \rangle$ to test whether $L(Q) = \emptyset$. Output the opposite of R's output. Contradiction.
$REGULAR_{\text{TM}}$, reduction $A_{\text{TM}} \leq REGULAR_{\text{TM}}$, assume R decides $REGULAR_{\text{TM}}$, construct S using R to decide A_{TM}. S on input $\langle M, w \rangle$, construct Q that $L(Q) = \Sigma^*$ if M accepts w and $L(Q) = 0^n 1^n$ if M doesn't accept w (If x is $0^n 1^n$, accept; otherwise, run M on w and accept if M accepts). Run R on $\langle Q \rangle$ to test whether $L(Q)$ is regular. Output whatever R outputs. Contradiction.
EQ_{TM}, reduction $E_{\text{TM}} \leq EQ_{\text{TM}}$, assume R decides EQ_{TM}, construct S using R to decide E_{TM}. S on input $\langle M \rangle$, construct Q where $L(Q) = \emptyset$. Run R on $\langle M, Q \rangle$ to test whether $L(M) = L(Q) = \emptyset$. Output whatever R outputs. Contradiction.
WWR_{TM}, reduction $A_{\text{TM}} \leq WWR_{\text{TM}}$, assume R decides WWR_{TM}, construct S using R to decide A_{TM}. S on input $\langle M, w \rangle$, construct Q where on input x, if x is not 01 or 10, reject; if x is 01, accept; else if x is 10, accept if M accepts w. Run R on $\langle Q \rangle$, output whatever R outputs.
$USELESS_{\text{TM}}$, **TODO**.

11.3 Turing-recognizable

Closure, union, concatenation, star, intersection.
A_{TM}, by definition of Turing-recognizable.
$\overline{E_{\text{TM}}}$, on input $\langle M \rangle$, use dovetailing to run M on all strings in string order, accept if M accept any of them.

11.4 Not Turing-recognizable

$\overline{A_{\text{TM}}}$, A_{TM} is undecidable and Turing-recognizable, thus $\overline{A_{\text{TM}}}$ cannot be Turing-recognizable.
EQ_{TM}, mapping reduction, $\overline{A_{\text{TM}}} \leq_{\text{m}} EQ_{\text{TM}}$ iff $A_{\text{TM}} \leq_{\text{m}} \overline{EQ_{\text{TM}}}$. The reduction function f is a TM F on input $\langle M, w \rangle$, construct M_1 that accepts nothing ($L(M_1) = \emptyset$), then construct M_2 that accepts inputs only when M accepts w. Output $\langle M_1, M_2 \rangle$. Thus, $\langle M, w \rangle \in A_{\text{TM}}$ iff $\langle M_1, M_2 \rangle \in \overline{EQ_{\text{TM}}}$.
$\overline{EQ_{\text{TM}}}$, mapping reduction, $\overline{A_{\text{TM}}} \leq_{\text{m}} \overline{EQ_{\text{TM}}}$ iff $A_{\text{TM}} \leq_{\text{m}} EQ_{\text{TM}}$. The reduction function g is a TM G on input $\langle M, w \rangle$, construct M_1 that accepts anything ($L(M_1) = \Sigma^*$), then construct M_2 that accepts inputs only when M accepts w. Output $\langle M_1, M_2 \rangle$. Thus, $\langle M, w \rangle \in A_{\text{TM}}$ iff $\langle M_1, M_2 \rangle \in EQ_{\text{TM}}$.

11.5 P

Closure, union, concatenation, and complement.
$PATH$, Breadth-first search with marking algorithm.
$RELPRIME$, on input $\langle x, y \rangle$ use Euclidean algorithm, each stage cuts x or y by at least half, thus the maximum number of time executed is the lesser of $2 \log_2 x$ and $2 \log_2 y$, which runs in $O(n)$.
$CONNECTED$, marking algorithm, mark the first node, then repeatedly mark nodes that can be reached from a node that is already marker. If all nodes are marked, then accept, otherwise reject.
$TRIANGLE$, for each possible triplet of nodes, check whether their edges are in G. If yes, accept; If no such triplet found, reject.
ALL_{DFA}, perform breadth-first search from start state, if any node is not accepting, reject. If all nodes are accepting, accept.
$MODEXP$, **TODO**

$PERMPOWER$, **TODO**
$UNARYSUM$, **TODO**

11.6 NP

Closure, union, concatenation, star.
$HAMPATH$, a certificate of the path.
$COMPOSITES$, a certificate of the factors.
$CLIQUE$, a certificate of the k-clique. Or on input $\langle G, k\rangle$, nondeterministically select a subset c of k nodes in G and test whether G contains all edges connecting nodes in c.
$SUBSETSUM$, a certificate of the set of numbers which sum to t.
$VERTEXCOVER$, a certificate is a vertex cover of size k.

11.7 NP-Complete

SAT, Cook-Levin Theorem, simulate NDTM's configuration history with a big cnf-formula. $\phi = \phi_{\text{cell}} \wedge \phi_{\text{start}} \wedge \phi_{\text{accept}} \wedge \phi_{\text{move}}$.
$3SAT$, $SAT \leq_{\text{P}} 3SAT$,
$CLIQUE$, $3SAT \leq_{\text{P}} CLIQUE$, make each clause a triplet of nodes, connect all nodes to each other except nodes within a triplet and nodes have contradicting labels. If ϕ is satisfiable, pick one node from each triplet that corresponds to a true literal in the satisfying assignment, the resulting graph is a k-clique. If G has a k-clique, assign truth values to the variables of ϕ so that each literal labeling a clique node is made true.
$VERTEXCOVER$, $3SAT \leq_{\text{P}} VERTEXCOVER$,
$HAMPATH$, **TODO**
$BDST$, Bounded-degree spanning tree.

11.8 PSPACE

Closure, union, complementation, and star.
SAT, Try all possible truth assignments.
ALL_{NFA}, **TODO**

11.9 PSPACE-Complete

11.10 L

$MULT$, **TODO**
ADD, on input $\langle x, y, z\rangle$, use the addition algorithm, keep tracking of which bit we are adding and the carry from previous bits.
$UCYCLE$, on input $\langle G\rangle$, perform DFS on G for each node and each edge of that node, if we come back to that node through a different edge, the graph has a cycle. Need log-space for each node, its starting edge, the current node and current edge.

11.11 NL

Closure, union, concatenation, and star.
$PATH$, on input $\langle G, s, t \rangle$, nondeterministically guess the nodes of a path from s to t. Records only the position of the current node at each step. Nondeterministically selects the next node. Repeats until it reaches t and accepts, or until $m = |G|$ steps and reject.
$BIPARTITE$, **TODO**

11.12 NL-Complete

$STRONGLYCONNECTED$, **TODO**