MIT 6.840: Theory of Computation

Taught by Michael Sipser
Notes by Leonard Tang and Austin Li

Fall 2020

The course was taught by Michael Frederic Sipser. The lectures were on Tuesdays and Thursdays at 2:30–4pm EST. The course had biweekly problem sets, in-class check-in quizzes, one midterm, and one final. The course assistants were Fadi Atieh, Damian Barabonkov, Di-Chia Chueh, Alexander Dimitrakakis, Thomas Xiong, Abbas Zeitoun, and Emily Liu. Additional material can be found on the course website.

Contents

1 Regular Languages

1.1 Key Definitions

Definition 1.1. A **finite automaton** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where

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

Definition 1.2. If A is the set of all strings a machine M accepts, we say A is the **language** of M and write $L(M) = A$. We say M **recognizes** A or M **accepts** A. The **empty language** is the language of no strings, denoted $\varnothing$.

Definition 1.3. A language is called a **regular language** if some finite automaton recognizes it.

Definition 1.4. For languages A, B, the regular operations are:

Union: $A \cup B : \{x : x \in A \text{ or } x \in B\}$
Concatenation: $A \circ B : \{xy : x \in A \text{ and } y \in B\}$
Star: $A^* = \{x_1 x_2 \dots x_k : k \geq 0 \text{ and each } x_i \in A\}$.

Definition 1.5. A **nondeterministic finite automaton (NFA)** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where all are the same as in the deterministic case except

$$\delta : Q \times \Sigma_\varepsilon \to \mathcal{P}(Q), \qquad \Sigma_\epsilon = \Sigma \cup \{\varepsilon\}$$

and $\mathcal{P}(Q)$ the power set of Q.

Definition 1.6. Two machines M_1, M_2 are **equivalent** if they recognize the same language.

Definition 1.7. R is a **regular expression** if R is

1. a for some $a \in \Sigma$,
2. ε,
3. $\varnothing$,
4. $R_1 \cup R_2, R_1 \circ R_2$, or R_1^* for R_1, R_2 regular expressions.

Definition 1.8. A **generalized nondeterministic finite automaton (GNFA)** is a 5-tuple, $(Q, \Sigma, \delta, q_{start}, q_{accept})$ where all else same as DFA, NFA, transition function given by

$$\delta : (Q - \{q_{accept}) \times (Q - \{q_{start}) \to \mathbb{R}.$$

1.2 Key Results

Theorem 1.9. *Class of regular languages closed under union operation.*

Proof. Consider machines that recognize A_1, A_2 and construct M recognizing $A_1 \cup A_2$ with $Q = Q_1 \times Q_2, \Sigma = \Sigma_1 \cup \Sigma_2, \delta((r_1, r_2), a) = (\delta_1(r_1, a), \delta_2(r_2, a)), q_0 = (q_1, q_2), F = (F_1 \times Q_2) \cup (Q_1 \times F_2)$, keeping track of **pairs** of states.
Faster: Take two NFAs that recognize A_1, A_2, construct N recognizing $A_1 \cup A_2$ by creating new start state and sending ε-transitions to start states of N_1, N_2. □

Theorem 1.10. *Every NFA has an equivalent DFA.*

Proof. Massage states and transition function of an NFA N into the states and transition function of DFA M using sets. □

Corollary 1.11. *A language is regular* $\iff$ *some NFA recognizes it.*

Theorem 1.12. *Class of regular languages closed under concatenation.*

Proof. Use nondeterminism to guess where to make split by connecting accepting states of N_1 recognizing A_1 to start state of N_2 recognizing A_2 with ε-transitions. □

Theorem 1.13. *The class of regular languages is closed under the star operation.*

Proof. From N_1 recognizing A_1, create new start state q_0, connect to old start state via ε-transition, and connect all accepting states to old start state via ε-transitions. □

Theorem 1.14. *A language is regular* $\iff$ *some regular expression describes it.*

Proof. ($\Longleftarrow$) Convert R into NFA N. ($\Longrightarrow$) Convert DFA into GNFA into regular expression. The conversions are done by ripping out intermediate state and repairing all connections. □

Theorem 1.15 (Pumping lemma)**.** *If A a regular language, exists p (pumping length) where if $s \in A, |s| \geq p$, s can be divided into three pieces $s = xyz$:*

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

1.3 Proof Concepts and Examples

Example 1.16. Creating DFAs, NFAs to show languages regular, as if you are machine.

Example 1.17. Use ε-transitions to prove closure properties and build NFAs.

Example 1.18. Use **pumping lemma** to prove language nonregular:
Let $B = \{0^n1^n : n \geq 0\}$. WTS B nonregular. Consider string $0^p1^p \in B$. Use pumping lemma, $s = xyz$. Three cases, y contains only 0s or 1s. After pumped, there will be unequal amount. If y has both $0, 1$, after pumping, will be out of order, so a contradiction $\implies B$ nonregular.

1.4 Problem Set Results

Problem 1.19. *Class of regular languages closed under complement.*

Proof. Swap accept and nonaccept states of a DFA M. □

Problem 1.20. *Class of regular languages closed under reversal. For any language A, $A^{\mathbb{R}} = \{w^{\mathbb{R}} : w \in A\}$. A regular $\implies A^{\mathbb{R}}$ regular.*

Problem 1.21. *Class of **nonregular** languages is*

1. ***Not** closed under union*
2. ***Not** closed under concatenation*
3. ***Closed** under complementation.*

2 Context-Free Languages

2.1 Key Definitions

Definition 2.1. A **context-free grammar (CFG)** is a 4-tuple (V, Σ, R, S) where

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

Consider G_1 given by

$$\begin{aligned} S &\to 0S1 \mid B, \\ B &\to \# \end{aligned}$$

Here, S is the start variable, B is a variable, $0, 1, \#$ are terminals. A sequence of substitutions to obtain a string is a **derivation** and can be represented pictorially with a **parse tree**.

Definition 2.2. Any language that can be generated by some context-free grammar is called a **context-free language (CFL)**.

Definition 2.3. A string w is derived ambiguously in a CFG G if it has two or more leftmost derivations. Grammar G is **ambiguous** if it generates some string ambiguously.

Definition 2.4. A context-free grammar is in **Chomsky normal form** if every rule is of the form

$$\begin{aligned} A &\to BC \\ A &\to a \end{aligned}$$

where a is any terminal and A, B, C are any variables, with B, C not the start variable. We permit the rule $S \to \varepsilon$ where S the start variable.

Definition 2.5. A **pushdown automaton (PDA)** is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where Q, Σ, Γ, and F are all finite sets, where $Q, q_0, F \subseteq Q$ are the same as always with

1. Σ is the input alphabet,
2. Γ is the stack alphabet,
3. $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \mathcal{P}(Q \times \Gamma_\varepsilon)$ is the transition function.

PDAs are like NFAs with an extra component called a **stack**, that provides additional memory beyond finite control. A PDA can write on and read symbols on the stack. Writing is called **pushing** and removing a symbol is called **popping**.

Definition 2.6. A **deterministic pushdown automaton (DPDA)** is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$ where Q, Σ, Γ, F all finite sets with

$$\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to (Q \cup \Gamma_\varepsilon) \cup \{\varnothing\}$$

is the transition function satisfying: $\forall q \in Q, a \in \Sigma, x \in \Gamma$, exactly one of the values

$$\delta(q, a, x), \qquad \delta(q, a, \varepsilon), \qquad \delta(q, \varepsilon, x), \qquad \delta(q, \varepsilon, \varepsilon)$$

is **not** $\varnothing$. This conforms to the principle of determinism: at each step of computation, DPDA has at most one way to proceed according to transition function. The language of a DPDA is called a **deterministic context-free language (DCFL)**.

2.2 Key Results

Theorem 2.7. *Any context-free language is generated by a context-free grammar in Chomsky normal form.*

Proof. Convert any grammar G into Chomsky normal form. Add new start variable $S_0 \to S$, eliminate all ε-rules of form $A \to \varepsilon$ and eliminate all unit rules of the form $A \to B$ and patch up grammar to be sure that it generates the same language. □

Theorem 2.8. *A language is context-free* $\iff$ *some PDA recognizes it.*

Proof. ($\Longleftarrow$) Convert CFG G into PDA P by nondeterministically selecting one of the rules for A and substituting A by the string on RHS of the rule. If matches input, pop the part of string that matches and continue. ($\Longleftarrow$) Construct PDA P from CFG G. □

Theorem 2.9. *If a PDA recognizes some language, then it is context-free.*

Corollary 2.10. *Every regular language is context free.*

Theorem 2.11 (Pumping lemma for context-free languages)**.** *If A a CFL $\implies \exists p$ (pumping length) where if $s \in A : |s| \geq p$, s can be divided into five pieces $s = uvxyz$ satisfying conditions*

1. $\forall i \geq 0, uv^ixy^iz \in A$,
2. $|vy| > 0$, *and*
3. $|vxy| \leq p$.

Theorem 2.12. *Class of DCFLs is closed under complementation.*

2.3 Proof Concepts and Examples

Example 2.13. Use stack as additional memory and check for matches on input tape.

Example 2.14. Use pumping lemma to show language not context free.
Let $B = \{a^nb^nc^n : n \geq 0\}$. WTS B not context free.

2.4 Problem Set Results

Problem 2.15. *CFLs are **closed under** union, concatenation, and star.*

Problem 2.16. *CFL $\cap$ regular = CFL.*

Problem 2.17. *If G a CFG in Chomsky normal form, then for any string $w \in L(G)$ of length $n \geq 1$, exactly $2n - 1$ steps required for any derivation of w.*

3 The Church-Turing Thesis

3.1 Key Definitions

Definition 3.1. A **Turing machine (TM)** is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ where Q, Σ, Γ are all finite sets and

1. Σ is the input alphabet not containing the **blank symbol** $\sqcup$,
2. Γ is the tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$,
3. $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the transition function,
4. $q_0 \in Q$ is the start state,
5. $q_{accept} \in Q$ is the accept state, and
6. $q_{reject} \in Q$ is the reject state, $q_{reject} \neq q_{accept}$.

The transition function has $\{L, R\}$, meaning after reading state symbol and writing a symbol, it moves either left or right. As a Turing machine, computes, changes occur in the current state, the current tape contents, and the current head location. A setting of these three items is called a **configuration** of the Turing machine.
A Turing machine on an input may *accept*, *reject*, or *loop*. To **loop** means that the machine does not halt. Will use high-level descriptions to describe TMs.

Definition 3.2. The collection of strings that M accepts is the **language of** M, or the **language recognized by** M, denoted $L(M)$.

Definition 3.3. A language **Turing-recognizable** if some Turing machine recognizes it.

Definition 3.4. A language **Turing-decidable** or simply **decidable** if some Turing machine decides it. Deciders always make a decision to accept or reject, never halt.
Every decidable language is Turing-recognizable

Definition 3.5. A **multitape Turing machine** is an ordinary TM with several tapes. Each tape has its own head for reading and writing.

Definition 3.6. An **enumerator** is a Turing machine with an attached printer. The language enumerated by E is the collection of all the strings that it eventually prints out. E can generate the strings of the language in any order, possibly with repetitions.

3.2 Key Results

Theorem 3.7. *Every multitape TM has an equivalent single-tape TM.*

Proof. Convert multitape TM M into an equivalent single-tape TM S. $\forall a \in \Sigma$, add $\dot{a}$ to Σ to mark head positions of different tapes and separate different tape inputs by #. Simulate the M on S by writing all contents on tapes of M onto single-tape S and do what M does. □

Corollary 3.8. *A language is Turing-recognizable* $\iff$ *some multitape TM recognizes it.*

Theorem 3.9. *Every nondeterministic Turing machine has an equivalent deterministic Turing machine.*

Corollary 3.10. *A language is Turing-recognizable* $\iff$ *some nondeterministic TM recognizes it.*

Corollary 3.11. *A language is decidable* $\iff$ *some nondeterminsitic TM decides it.*

Theorem 3.12. *A language is Turing-recognizable* $\iff$ *some enumerator enumerates it.*

Proof. Show if M enumerates A, a TM M recognizes A. Create M such that it accepts all strings E prints. Create E such that it prints all strings that M accepts. □

Theorem 3.13 (Church-Turing thesis). *Intuitive notion of algorithms* $\cong$ *Turing machine algorithms.*

3.3 Proof Concepts and Examples

Example 3.14. Using high level descriptions for TM deciders and recognizers:
Let $A = \{\langle G\rangle : G$ is a connected undirected graph$\}$. Following high-level description of TM M that decides A.
$M =$ "on input $\langle G\rangle$, the encoding of a graph G:

1. Select first node of G and mark it.
2. Repeat the following stage until no new nodes are marked:

   For each node in G, mark if it is attached by an edge to a node that is already marked.

3. Scan all nodes of G to determine whether they all are marked. If they are, *accept*; otherwise, *reject*."

Example 3.15. Adding symbols to stack/tape alphabet to manipulate PDAs/to show equivalence.

Example 3.16. To show TM equivalence, need to show that operations can be simulated in both directions.

3.4 Problem Set Results

Problem 3.17. *A* ***deterministic queue automaton (DQA)*** *is like a push-down automaton with stack replaced by a queue. A* ***queue*** *is a tape allowing symbols to be written only on the left-hand side and read on the right-hand side. Each write operation (**push**) adds symbol to the left-hand end of the queue and each read operation (**pull**) reads and removes symbol on right-hand end. The input tape contains a cell with blank symbol to denote end of input.*
A language can be recognized by a DQA $\iff$ *language is Turing-recognizable.*

4 Decidability

4.1 Key Definitions

Definition 4.1. Let A, B sets. A function $f : A \to B$ is **one-to-one** or **injective** if $f(a) \neq f(b) \implies a \neq b$. f is **onto**, or **surjective**, if $\forall b \in B \exists a \in A : f(a) = b$. $|A| = |B|$ if $\exists$ a **bijection** $f : A \to B$, f is both injective and surjective.

Definition 4.2. A set A is **countable** if either it is finite or has the same size as $\mathbb{N}$.

4.2 Key Results

Theorem 4.3. *$A_{DFA} = \{\langle B, w\rangle : B$ is a DFA that accepts input string $w\}$ is decidable.*

Proof. Present a TM M deciding A_{DFA}: simulate B on w and *accept* if B accepts, *reject* otherwise. □

Theorem 4.4. *$A_{NFA} = \{\langle B, w\rangle : B$ is an NFA that accepts $w\}$ is decidable.*

Proof. Present NTM N deciding A_{NFA}: convert B into equivalent DFA C and simulate A_{DFA} on $\langle C, w\rangle$. *Accept* if A_{DFA} accepts, *reject* otherwise. □

Theorem 4.5. *$A_{REX} = \{\langle R, w\rangle : R$ is a regular expression that generates $w\}$ is decidable.*

Proof. TM P deciding A_{REX}: convert R into equivalent NFA A, run A_{NFA} on $\langle A, w\rangle$. *Accept* if A_{NFA} accepts and *reject* otherwise. □

Theorem 4.6. *$E_{DFA} = \{\langle A\rangle : A$ is a DFA and $L(A) = \varnothing\}$ is decidable.*

Proof. DFA accepts some string $\iff$ able to reach accept state from start state. Design marking algorithm for TM decider T: mark start state of A and continue to mark any state that has a transition coming into it from any state already marked until no new states get marked. If accept state is marked, *accept*. *Reject* otherwise. □

Theorem 4.7. *$EQ_{DFA} = \{\langle A, B\rangle : A$ and B are DFAs and $L(A) = L(B)\}$ is decidable.*

Proof. Consider the **symmetric difference** of $L(A), L(B)$ given by $L(C) = L(A)\Delta L(B)$. Then $L(C) = \varnothing \iff L(A) = L(B)$. Construct TM decider F: on input $\langle A, B\rangle$, construct C the symmetric difference and simulate E_{DFA} on $\langle C\rangle$. *Accept* if E_{DFA} accepts, *reject* if rejects. □

Theorem 4.8. *$A_{CFG} = \{\langle G, w\rangle : G$ is a CFG and generates $w\}$ is decidable.*

Proof. Recall that a grammar in Chomsky normal form can derive any string length n in at most $2n - 1$ steps. Then construct TM decider S: convert G into equivalent grammar in Chomsky normal form. List all derivations length $2n - 1 : n = |w|$. If any of these derivations generate w, *accept*; if not, *reject*. □

Theorem 4.9. *$E_{CFG} = \{\langle G\rangle : G$ is a CFG and $L(G) = \varnothing\}$ is decidable.*

Proof. Might want to use A_{CFG} to test membership in language, so in order to test $L(G) = \varnothing$, we can test all possible w's one by one, but this can be infinite. Different approach: need to test if start variable can generate a string of terminals via a marking procedure. TM decider R: mark all terminal symbols in G, mark all variables with rule $A \to U_1U_2 \dots U_k$ where each U_i already marked. If start variable marked, *accept*; *reject* otherwise. □

Theorem 4.10. *$EQ_{CFG} = \{\langle G, H\rangle : G$ and H are CFGs and $L(G) = L(H)\}$ is **not** decidable.*

Proof. Cannot use method used for EQ_{DFA} because class of CFLs are not closed under complement. Will prove in later section. □

Theorem 4.11. *Every CFL is decidable.*

Proof. Let G a CFG for A and design TM M_G deciding A: simulate A_{CFG} on $\langle G, w\rangle$. If A_{CFG} accepts, *accept*. *Reject* otherwise.
This establishes a relationship among classes of languages: regular $\subset$ CFL $\subset$ decidable $\subset$ Turing-recognizable. □

Theorem 4.12. *$A_{TM} = \{\langle M, w\rangle : M :$ is a TM and M accepts $w\}$ is undecidable.*

Proof. We first observe that A_{TM} is Turing-recognizable.
TM U recognizing A_{TM}: simulate M on w. If M ever enters accept state, *accept*. If M ever enters reject state, *reject*. This TM loops on $\langle M, w\rangle \implies$ doesn't decide A_{TM}.
Key idea: **diagonalization method**. AFTSOC TM H decides A_{TM}, construct new D that uses H as subroutine, but outputs the opposite of what H outputs. Run D on $\langle D\rangle$, but this outputs the opposite of what D does, a contradiction (D accepts $\langle D\rangle \iff D$ rejects $\langle D\rangle$). □

Theorem 4.13. *A language is decidable $\iff$ Turing-recognizable and co-Turing-recognizable.*

Proof. ($\implies$) Any decidable language Turing-recognizable and complement of decidable langauge is decidable. ($\impliedby$) If $A, \overline{A}$ both recognizable, let M_1, M_2 be recognizers. Construct TM decider M: run M_1, M_2 on w in parallel. If M_1 accepts, *accept*; if M_2 accepts, *reject*. □

Theorem 4.14. *$\overline{A_{TM}}$ **not** Turing-recognizable.*

Proof. A_{TM} is Turing-recognizable but **not** decidable. □

4.3 Proof Concepts and Examples

Example 4.15. Use old TMs to solve decidability problems (method for all theorems above except A_{TM}).

4.4 Problem Set Results

Problem 4.16. *A language is decidable $\iff$ some enumerator enumerates the language in **string order**. String order is the standard length-increasing, lexicographic order.*

Proof. There are two cases: if A is finite or infinite. If A finite, it is decidable. If A infinite, can create a decider as follows:
On input w, decider will use enumerator to enumerate all strings in A in string order until some string appears which is after w. If w has already appeared in the enumeration, *accept*; if it hasn't appeared yet, it never will, so *reject*. □

Problem 4.17. *$PUSHER = \{\langle P\rangle : P$ is a PDA that pushes a symbol on its stack on some branch of computation at some point on input $w \in \Sigma^*\}$ is decidable.*

5 Reducibility

5.1 Key Definitions

Definition 5.1. A **reduction** is a conversion from one problem to another such that a solution to the second problem can be used to solve the first. This is the primary method for proving that problems are computationally unsolvable.

Definition 5.2. Let M a Turing machine and w an input string. An **accepting computation history** for M on w is a sequence of configurations $C_1, C_2, \dots, C_l$ where C_1 the start configuration of M on w, C_l is an accepting configuration of M and each C_i legally follows from C_{i-1} according to rules of M. A **rejecting computation history** for M on w is defined similarly, except C_l is a rejecting configuration. Note: computation histories are **finite sequences**, e.g. must halt.

Definition 5.3. A **linear bounded automaton (LBA)** is a restricted Turing machine where the tape head cannot move off the portion of the tape containing the input, e.g. it has limited memory.

Definition 5.4. A function $f : \Sigma^* \to \Sigma^*$ is a **computable function** if some Turing machine M, on every input w, halts with just $f(w)$ on its tape.

Definition 5.5. 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, $w \in A \iff f(w) \in B$. The function f is called the **reduction** from A to B.
Allows us to convert membership testing in A to membership testing in B.

5.2 Key Results

Theorem 5.6. *$HALT_{TM} = \{\langle M, w\rangle : M$ is a TM and M halts on input $w\}$ is undecidable.*

Proof. AFTSOC TM R decides $HALT_{TM}$. Then we can construct a TM S deciding A_{TM}. $S =$ run TM R on $\langle M, w\rangle$. If R rejects, *reject*. If R accepts, simulate M on w until halts and output whatever M accepts $\implies A_{TM}$ decided. ☐

Theorem 5.7. *$E_{TM} = \{\langle M\rangle : M$ is a TM and $L(M) = \varnothing\}$ is undecidable.*

Proof. Similar to $HALT_{TM}$ proof, assume R decides E_{TM} and construct S deciding A_{TM} using R. Run R on modification of M : use $M_w : M$ only accepts w.
Then $S =$ use M, w to construct TM M_w. Simulate R on M_w. If R accepts, *reject*; if R rejects, *accept*. Observe that this decides A_{TM} because M_w empty $\iff$ M rejects w, M_w nonempty $\iff$ M accepts w. ☐

Theorem 5.8. *$REGULAR_{TM} = \{\langle M\rangle : M$ is a TM and $L(M)$ a regular language$\}$ is undecidable.*

Proof. Let R a TM that decides $REGULAR_{TM}$ and construct TM S to decide A_{TM}. S: on input $\langle M, w\rangle$, construct $M_w : M_w$ accepts x if x has form 0^n1^n and otherwise, runs M on w, accepting if M accepts. Now run R on $\langle M_w\rangle$. *Accept* if R accepts; *reject* otherwise.
Note that M_w recognizes the regular language Σ^* if M accepts w. ☐

Theorem 5.9. *$EQ_{TM} = \{\langle M_1, M_2\rangle : M_1, M_2$ are TMs and $L(M_1) = L(M_2)\}$ is undecidable.*

Proof. Reduction from E_{TM}. Let R decide EQ_{TM} construct S deciding E_{TM}. S: simulate R on $\langle M_1, M_2\rangle$ where M_1 rejects all inputs. If R accepts, *accept*; otherwise, *reject*. ☐

Lemma 5.10. *Let M be an LBA with q states and g symbols in the tape alphabet. There are exactly qng^n **distinct** configurations of M for a tape of length n.*

Proof. There are q possible states, n head positions, and g^n possible strings of tape symbols. ☐

Theorem 5.11. *$A_{LBA} = \{\langle M, w\rangle : M$ is an LBA that accepts string $w\}$ is decidable.*

Proof. Key idea: simulate M on w and spit out same result if halts. Looping is a problem, but we know there are only finitely many unique configurations for inputs length n, namely qng^n. □

Theorem 5.12. *$E_{LBA} = \{\langle M\rangle : M$ is an LBA where $L(M) = \varnothing\}$ is undecidable.*

Proof. Reduce from A_{TM} using computation histories. For a TM M and string w, can construct an LBA B that accepts all accepting computation histories for M on w. If $w \in L(M)$, there exists a computation history and so $L(B) \neq \varnothing$ and if $w \notin L(M) \implies L(B) = \varnothing$.
Construct B: B breaks up x according to delimiter into strings $C_1, C_2, \ldots, C_l$ and then determines if C_1 is the starting configuration, C_{i+1} follows legally from C_i, and C_l is an accepting configuration for M. Therefore we can decide A_{TM}. □

Theorem 5.13. *$ALL_{CFG} = \{\langle G\rangle : G$ is a CFG and $L(G) = \Sigma^*\}$ is undecidable.*

Proof. Proof by contradiction, reduction from A_{TM} using computation histories, but modify representation of C_is. We want G to generate all strings that do **not** start with C_1, do **not** end with an accepting configuration, or strings where C_i does not properly yield C_{i+1} under the rules of M.
Construct a PDA D (easier than designing CFG). Summary: writes the C_is in alternating order so we can pop off the stack and compare, where D accepts if the two histories do not follow M transition function. □

Theorem 5.14. *$EQ_{CFG} = \{\langle G_1, G_2\rangle : G_1$ and G_2 are equivalent CFGs$\}$ is undecidable.*

Theorem 5.15. *$PCP = \{\langle P\rangle : P$ is an instance of the **Post Correspondence Problem** with a **match**$\}$ is undecidable.*
*The Post Correspondence Problem is to determine where a collection of dominoes has a match, or a list of dominoes where the top and bottom symbols are the same. This problem is **unsolvable by algorithms**.*

Proof. Reduction from A_{TM} via accepting computation histories. Will create an instance of PCP where a match forces a simulation of M to occur. Slight modification to require that the match starts with the first domino:
$MPCP = \{\langle P\rangle : P$ is an instance of the PCP with a match that starts with the first domino$\}$.
The construction is quite tedious but can be found on pages 229-233. □

Theorem 5.16. *If $A \leq_m B$ and B is decidable, then A is decidable.*

Proof. Let M be the decider for B and f be reduction function from A to B. Describe decider N for A: compute $f(w)$, run M on $f(w)$ and output whatever M outputs. □

Corollary 5.17. *If $A \leq_m B$ and A is undecidable, B undecidable.*

Theorem 5.18. *If $A \leq_m B$ and B is Turing-recognizable, then A is Turing-recognizable. If $A \leq_m B$ and A **not** Turing-recognizable, then B is **not** Turing-recognizable.*

Theorem 5.19. *EQ_{TM} is neither Turing-recognizable nor co-Turing-recognizable.*

Proof. EQ_{TM} not Turing-recognizable: show $A_{TM} \leq_m \overline{EQ_{TM}}$. Give algorithm: on $\langle M, w\rangle$: construct M_1 that rejects on all inputs and M_2 that simulates M on w, accepting if it accepts. $w \notin L(M) \iff \langle M_1, M_2\rangle \in EQ_{TM}$.
$\overline{EQ_{TM}}$ not Turing-recognizable: show $A_{TM} \leq_m EQ_{TM}$. Give reduction similar to above, except M_1 accepts all inputs. It follows similarly that $w \in L(M) \iff \langle M_1, M_2\rangle \in EQ_{TM}$. □

5.3 Proof Concepts and Examples

Example 5.20. Reducing from A_{TM} to show undecidability.

Example 5.21. Use accepting computation histories for emptiness proofs!

Example 5.22 (Computable functions)**.** Arithmetic operations on integers are computable functions, e.g. can make machine that takes $\langle m, n \rangle$ and returns $m + n$, the sum of m and n.
Can also be transformations of machine descriptions. A computable function can take an input w and return the description of a TM $\langle M' \rangle$ if $w = \langle M \rangle$ is an encoding of a TM M.

Example 5.23 (Mapping reductions)**.** For $HALT_{TM}$, we can show $A_{TM} \leq_m HALT_{TM}$.
For the PCP problem, we showed that $A_{TM} \leq_m MPCP$ and then $MPCP \leq_m PCP$. Because mapping reducibility is transitive $\implies A_{TM} \leq_m PCP$.

5.4 Problem Set Results

Problem 5.24. $L_{TM} = \{\langle M, w \rangle : M$ *on input* w *ever moves its head left when its head is on the left-most tape cell*$\}$ *is undecidable.*

Proof. Reduction from A_{TM}. AFTSOC R decides L_{TM}. Construct TM S deciding A_{TM}:
$S =$ "on input $\langle M, w \rangle$:

1. Convert M to M' where M' first moves its input over one square to the right and writes a new symbol \$ on the leftmost tape cell. Then M' simulates M on the input. If M' ever sees \$ then M' moves its head one square right and remains in the same state. If M accepts, M' moves its head all the way to the left and then moves left off the leftmost tape cell.
2. Run R, the decider for L_{TM} on $\langle M', w \rangle$.
3. If R accepts, then *accept*. If it rejects, *reject*."

S decides A_{TM} because M' only moves left from leftmost tape cell when M accepts w. □

Problem 5.25. *The problem of whether a single-tape Turing machine ever writes a blank symbol over a non-blank symbol over course of computation on any input string is undecidable.*

Proof. Let $E = \{\langle M \rangle : M$ is a single-tape TM which ever writes a blank symbol over a nonblank symbol when it is run on any input$\}$. AFTSOC R decides E and construct TM S deciding A_{TM}.
$S =$ "on input $\langle M, w \rangle$:

1. Use M and w to construct TM T_w.

   $T_w =$ "on any input:

   i. Simulate M on w. Use new symbol ␣' instead of a blank when writing and treat like a blank when reading.
   ii. If M accepts, write a true blank symbol over a nonblank symbol."
2. Run R on $\langle T_w \rangle$ to determine if T_w ever writes a blank.
3. If R accepts, M accepts w and *accept*. Otherwise *reject*."

□

Problem 5.26. *A language A is Turing-recognizable* $\iff A \leq_m A_{TM}$. *A is decidable* $\iff A \leq_m 0^*1^*$.

Proof. Create a reduction function f: if $w \in A$, output 01. If $w \notin A$, output 10. □

Problem 5.27. $AMBIG_{CFG} = \{\langle G \rangle : G$ *is an ambiguous CFG*$\}$ *is undecidable.*

Proof. Reduce from an instance of PCP, where a match corresponds to two derivations of a string. □

Problem 5.28. *A variable A in CFG G is* ***redundant*** *if removing it and its associated rules leaves $L(G)$ unchanged.*
Let $REDUNDANT_{CFG} = \{\langle G, A\rangle : A$ is a redundant variable in $G\}$. $\overline{REDUNDANT_{CFG}}$ is Turing-recognizable and $REDUNDANT_{CFG}$ is undecidable.

Problem 5.29. *A* ***two-headed finite automaton (2DFA)*** *is a deterministic finite automaton that has two read-only, bidirectional heads that start at left-hand end of input tape and can be independent controlled to move in either direction. The tape of a 2DFA is finite and large enough to contain input and two blank tape cells on either end that serve as delimiters. A 2DFA accepts its input by entering special accept state.*
$A_{2DFA} = \{\langle M, x\rangle : M$ is a 2DFA and M accepts $x\}$ is decidable.
$E_{2DFA} = \{\langle M\rangle : M$ is a 2DFA and $L(M) = \varnothing\}$ is decidable.

6 Advanced Topics in Computability Theory

6.1 Key Definitions

Definition 6.1. If M is a Turing machine, we say the **length** of the description $\langle M \rangle$ is the number of symbols in the string describing M. Say that M is **minimal** if there is no Turing machine equivalent to M that has a shorter description.
Let $MIN_{TM} = \{\langle M \rangle : M \text{ is a minimal TM}\}$.

6.2 Key Results

Lemma 6.2. *There is a computable function $q : \Sigma^* \to \Sigma^*$ where if w is any string, $q(w)$ is a description of a Turing machine P_w that prints out w and then halts.*

Theorem 6.3 (Recursion theorem)**.** *Let T be a Turing machine that computes a function $t : \Sigma^* \times \Sigma^* \to \Sigma^*$. There is a Turing machine R that computes a function $f : \Sigma^* \to \Sigma^*$, where for every w, $r(w) = t(\langle R \rangle, w)$.*
*This theorem esentially states that **Turing machines can obtain their own description and then go on to compute with it**.*

Theorem 6.4. *A_{TM} is undecidable.*

Proof. We assume that a Turing machine H decides A_{TM}, for the purpose of obtaining a contradiction. Construct machine B:
$B =$ "on input w:

1. Obtain via the recursion theorem, own description $\langle B \rangle$.
2. Run H on input $\langle B, w \rangle$.
3. Do the opposite of what H says. *Accept* if H rejects and *reject* if H accepts."

□

Theorem 6.5. *MIN_{TM} is not Turing-recognizable.*

Proof. Assume that some TM E enumerates MIN_{TM} and obtain a contradiction. Construct TM C.
$C =$ "on input w:

1. Obtain via the recursion theorem $\langle C \rangle$.
2. Run enumerator E until a machine D appears with a longer description than C.
3. Simulate D on input w."

MIN_{TM} is infinite $\implies$ E's list must contain a TM with a longer description than C. Because C simulates D (description longer than C), C is equivalent to D $\implies$ D cannot be on the list (not minimal), a contradiction. □

6.3 Proof Concepts and Examples

6.4 Problem Set Results

7 Time Complexity

7.1 Key Definitions

Definition 7.1. Let f, g functions $f, g : \mathbb{N} \to \mathbb{R}^+$. We say that $f(n) = O(g(n))$ if $\exists c, n_0 : \forall n \geq n_0, f(n) \leq cg(n)$. We say $g(n)$ is an **asymptotic upper bound** for $f(n)$, suppressing constant factors.

Definition 7.2. Let $f, g : \mathbb{N} \to \mathbb{R}^+$. We say that $f(n) = o(g(n))$ if $\lim_{n\to\infty} f(n)/g(n) = 0$.

Definition 7.3. Let $t : \mathbb{N} \to \mathbb{R}^+$. Define the **time complexity class TIME(**$t(n)$**)** to be the collection of all languages that are decidable by an $O(t(n))$ time Turing machine.
$O(n)$ is called **linear time**.

Definition 7.4. Let N be a nondeterministic Turing machine 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 its computation on any input of length n.

Definition 7.5. All reasonable computational models are **polynomially equivalent**, that is, any one of them can simulate another with only a polynomial increase in running time.
P is the class of languages that are decidable in polynomial time on a deterministic single-tape Turing machine:

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

P roughly corresponds to the class of problems that are realistically solvable on a computer.

Definition 7.6. A **verifier** for a language A is an algorithm V where $A = \{w : V \text{ accepts } \langle w, c\rangle \text{ for some string } c\}$. We measure the time of the verifier only in terms of the length of w, so a **polynomial time verifier** runs in polynomial time in lenght of w. A is **polynomially verifiable** if it has a polynomial time verifier.
The verifier may use additional information to determine membership, called a **certificate**, denoted c.

Definition 7.7. NP is the class of languages with polynomial time verifiers.

Definition 7.8. NTIME($t(n)$**)** $= \{L : L$ is a language decided by an $O(t(n))$ time nondeterministic Turing machine$\}$ $\implies$

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

Definition 7.9. A **clique** is an undirected graph in a subgraph wherein every two nodes are connected by an edge. A **k-clique** is a clique that contains k nodes.

Definition 7.10. A **Boolean formula** is an expression involving Boolean variables and operations. A Boolean formula is **satisfiable** if some assignment of 0s and 1s to the variables makes the formula evaluate to 1. A **literal** is a Boolean variable of negated variable, e.g. $x, \overline{x}$. A **clause** is several literals connected with $\vee$s. A Boolean formula is in **conjunctive normal form**, called a **cnf-formula** if it comprises several clauses connected with $\wedge$s.

Definition 7.11. 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.

Definition 7.12. Language A is **polynomial time mapping reducible** or simply **polynomial time reducible** to language B, written $A \leq_p B$ if a polynomial time computable function $f : \Sigma^* \to \Sigma^*$ exists where for $\forall w, w \in A \iff f(w) \in B$. The function f is called the **polynomial time reduction** of A to B.

Definition 7.13. A language B is **NP-complete** if it satisfies two conditions:

1. $B \in$ NP,
2. $\forall A \in$ NP, $A \leq_p B$.

Definition 7.14. If G is an undirected graph, a **vertex cover** of G is a subset of the nodes where every edge of G touches one of those nodes.

7.2 Key Results

Theorem 7.15. *Every $t(n) \geq n$ time multitape TM has an equivalent $O(t^2(n))$ time single-tape machine.*

Proof. Simulating a step requires a $t(n)$ scan for each of k branches. The multi-tape TM takes $t(n)$ time/steps, so simulating takes $O(t(n)t(n)) = O(t^2(n))$ time. ☐

Theorem 7.16. *Every $t(n) \geq n$ nondeterministic single-tape TM has an equivalent $2^{O(t(n))}$ time deterministic single-tape TM.*

Proof. The single-tape essentially explores the NTM's computation tree via DFS (to simulate each branch of computation). There are at most b valid transitions at each NTM step, and the NTM runs in $t(n)$ time $\implies$ there are $O(b^{t(n)})$ leaves. The number of leaves in a tree is basically half the number of all nodes $\implies$ there are $O(b^{t(n)})$ nodes. Exploring each branch of computation is bounded by $t(n)$ so total time to simulate all branches is $O(t(n)b^{t(n)}) = O(2^{t(n)})$ ☐

Theorem 7.17. *$PATH \in P$.*

Proof. Doing BFS takes polynomial time. ☐

Theorem 7.18. *Let $RELPRIME = \{\langle x, y\rangle : x$ and y are relatively prime$\}$. $RELPRIME \in P$.*

Proof. Can't simply loop through all integers less than x, y since exponentially many (in length of representation). Instead, use Euclidean algorithm.
Define the algorithm $E =$ "On input $\langle x, y\rangle$:

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

Then just run E and check if it returns 1 or not. ☐

Theorem 7.19. *Every CFL is in P.*

Proof. Recall that CFGs can be converted to Chomsky Normal Form and all derivations of a Chomsky Normal Form grammar require only $2|w| - 1$ steps on input w (2.26 in the book). Naively, testing all derivations of length $|w|$ to see if they match could take exponential time, *so instead we use DP.*
The subproblems $DP(i, j)$ are whether $w_i \dots w_j$ can be generated by the CFG. The idea is if w is derivable, some sequence of substring splits must exist to get the string down to individual symbols.
There are n^2 such subproblems. Store the variable that generates string $w_i \dots w_j$ in a memo table at (i, j). So the base cases are $(i, i) = A$ for rules $A \to w_i$. For each subproblem, we need to loop through n split locations and then a constant r rules $A \to BC$ to check if some B and C form the desired split substrings (check memo table at left and right splits to see if they match B and C, store A at (i, j) if yes. Check if S is in memo position $(1, n)$ (if yes, then following S will eventually yield w). Yes $\to$ accept, reject otherwise. There are n^2 subproblems; looping through splits is $n \implies O(n^3)$ overall run time. ☐

Theorem 7.20. *Recall $HAMPATH = \{\langle G, s, t\rangle : G$ is a **directed** graph with a Hamiltonian path from s to $t\}$. Directed means all nodes are visited and each node is visited exactly once. HAMPATH is in NP, but $\overline{HAMPATH}$ is not in NP.*

Proof. A certificate for $HAMPATH$ is simply the path − verify that it visits all nodes once and that it goes from s to t.
It is difficult to provide a certificate to show that a graph *never* has a $HAMPATH$. ☐

Theorem 7.21. *$COMPOSITES = \{x : x = pq$, for $p, q \in \mathbb{Z}^+\}$ is in NP. It is also in P.*

Theorem 7.22. *$CLIQUE = \{\langle G, k\rangle | G$ is an **undirected** graph with a k-clique$\}$. $CLIQUE$ is in NP. It is unclear if $\overline{CLIQUE}$ is in NP.*

Proof. The $CLIQUE$ is the certificate. A poly-verifier can check if G contains all edges connecting nodes in the certificate clique. ☐

Theorem 7.23. *SUBSET-SUM $= \{\langle S, t\rangle : S = \{x_1, \ldots x_k\}$ and some subset sums to target $t\}$ is in NP. Also pseudo-polynomial in size of set by DP. It is unclear if $\overline{\text{SUBSET-SUM}}$ is in NP (how would you know for sure?)*

Theorem 7.24. *$SAT = \{\langle\phi\rangle : \phi$ is a satisfiable Boolean formula$\}$. It is not the language of assignments themselves. $SAT \in P \iff P = NP$ since SAT is NP-complete.*

Theorem 7.25. *If $A \leq_p B$ and $B \in P$ then $A \in P$. Proof is obvious (chain of polynomial computations).*

Theorem 7.26. *Recall $3SAT$ is an AND of OR clauses. $3SAT \leq_p CLIQUE$*

Proof. This is a crucial proof concept. We will convert formulas to graphs, where components of the graph mimic the function of the formula.
Given ϕ with k clauses, we poly-reduce with $f(\phi) = \langle G, k\rangle$ (so we are aiming to create a k-clique). The idea is to have triples of nodes encoding the behavior of each clause. All nodes are connected with edges barring two exceptions: 1) nodes that are contradictory (this helps with backward direction in particular) and 2) nodes from same triple cannot be connected (we need *exactly* k nodes in the clique).
($\implies$): If $\phi \in 3SAT$, then to form a k-clique in G, include a node corresponding to a true literal in each clause of ϕ (if more than once than potentially larger than k-clique). The edge conditions from above automatically create a k-clique, since nodes are not contradictory and also not from same clause.
($\impliedby$): If there is a k-clique in G, then make the literal corresponding to the included node of each triplet true. This satisfies ϕ. No problems arising from contradictory assignment since not allowed to be in clique by edge restrictions. ☐

Theorem 7.27. *If B is NP-complete and $B \leq_p C$ for C in NP, then C is NP-complete.*

Proof. Proof is fairly obvious; every problem must poly-reduce to C. Well all problems already poly-reduce to B, which poly reduce to C (chain of poly-reductions is poly). ☐

Theorem 7.28 (Cook-Levin)**.** *SAT is NP-complete. Remark: serves as the basis for many other NP-complete proofs. Review PCP for more precision. See problem 7.41 for practice.*

Proof. This is a pretty messy proof. Essentially, we need to convert input M, w to formula $\phi_{M,w}$ that tells us whether or not M accepts w. The idea is to use a $n^k \times n^k$ configuration *tableau* (one nondeterministic branch's configuration history; n^k rows for max time and n^k cells (width) since runtime n^k upper bounds cell usage). Note the tableau only contains *one* branch's history!
The idea is to *create a formula ϕ that tells us if a tableau is satisfiable (i.e. accepts some input)*. The ϕ is constructed as an AND of 4 parts (omitting details for key ideas):

ϕ_{cell} : Checks if all tableau cells has one and only one assignment

ϕ_{start} : Checks if cells of first row are precisely a start configuration

ϕ_{accept} : Checks if cells of last row are an accept configuration (row gets carried down if accept early)

ϕ_{move} : Checks all 2×3 windows across all positions (i, j) for valid variable assignments (i.e. legal move)

It follows that M accepts $w \iff \phi \in SAT$ (if M accepts w some configuration accepts, hence tableau is accepting, so ϕ is true. Reverse is similar). ϕ is basically a bunch of groupings of literals for each cell position, so it is indeed $O(n^{2k})$, a poly-reduction. It is also easy to check that $SAT \in NP$ (just use satisfying assignment as certificate).
Note that this proof would not work with a 2×2 window in ϕ_{move}. □

Theorem 7.29. *$3SAT$ is NP-complete.*

Proof. Again, assignment is certificate, so $3SAT \in NP$. For NP-hard, we slightly modify the previous proof. First, we convert each sub-ϕ to CNF form (really just ϕ_{move}, since others are already in CNF). To do so, note that an OR of ANDs can be written as an AND of ORs. Then the outer-AND is just now a regular AND over ORs, hence we have a CNF.
Now, to get each clause to have exactly 3 literals: for all clauses with less than 3, just duplicate literals until you get to 3. If more than 3 literals, then note you can split a clause into an AND of clauses with dummy variables (assigned at will), like so:

$$(a_1 \vee a_2 \vee \cdots \vee a_n) = (a_1 \vee a_2 \vee z_1) \wedge (\overline{z_1} \vee a_3 \vee z_2) \wedge (\overline{z_2} \vee a_4 \vee z_3) \wedge \cdots \wedge (\overline{z_{n-3}} \vee a_{n-1} \vee a_n)$$

□

Theorem 7.30. *$CLIQUE$ is NP-complete.*

Theorem 7.31. *$VERTEX\text{-}COVER = \{\langle G, k\rangle : G$ is an undirected graph that has a k-node vertex cover$\}$ is NP-complete.*

Proof. Show $3SAT \leq_p VERTEX\text{-}COVER$ using gadgets. □

Theorem 7.32. *$HAMPATH$ is NP-complete.*

Proof. Already know $HAMPATH \in$ NP, now show $3SAT \leq_p HAMPATH$ using zig-zag gadgets (from lecture). □

Theorem 7.33. *$UHAMPATH$, the undirected $HAMPATH$ problem, is NP-complete.*

Proof. Show $HAMPATH \leq_p UHAMPATH$. □

Theorem 7.34. *$SUBSET\text{-}SUM$ is NP-complete.*

Proof. $3SAT \leq_p SUBSET\text{-}SUM$. □

7.3 Proof Concepts and Examples

Example 7.35. Simulating multi-tape machine on single-tape machine.
A single tape TM S stores each of the multi-tape TM's tapes horizontally. There are special dotted tape symbols to represent a head position.
To simulate M, S scans across its tape to find where the dotted symbols are (representing M's tape heads). Makes another pass to update tape contents according to M. If one of the multi-tapes needs more space, S shifts all its tape contents right by one.
Each multi-tape TM step is simulated in is $O(t(n)) \cdot k$ time. Multi-tape runs in $O(t(n))$ time, i.e. steps. So $O(t^2(n))$ time to simulate.

Example 7.36. Use DP to bring exponential time problems down to poly-time.

Example 7.37. Converting into $3CNF$ form:
Duplicate literals (does not change satisfiability) to reach 3. To reduce to 3:

$$(a_1 \vee a_2 \vee \cdots \vee a_n) = (a_1 \vee a_2 \vee z_1) \wedge (\overline{z_1} \vee a_3 \vee z_2) \wedge (\overline{z_2} \vee a_4 \vee z_3) \wedge \cdots \wedge (\overline{z_{n-3}} \vee a_{n-1} \vee a_n)$$

Example 7.38. You can prevent two literals a, b from being simultaneously true by asserting:

$$(\overline{a} \vee \overline{b}), \text{ which is false when both } a \text{ and } b \text{ are true}$$

7.4 Problem Set Results

8 Space Complexity

8.1 Key Definitions

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 maximum number of tape cells that M scans on any input of length n.

Definition 8.2. The space complexity classes, SPACE($f(n)$) and NSPACE($f(n)$) are defined as:
SPACE($f(n)$) = $\{L : L$ is a language decided by an $O(f(n))$ space deterministic TM$\}$.
NSPACE($f(n)$) = $\{L : L$ is a language decided by an $O(f(n))$ space deterministic NTM$\}$.

Definition 8.3. **PSPACE** is the class of languages that are decidable in polynomial space on a deterministic Turing machine, PSPACE=$\bigcup_k$SPACE(n^k).

Definition 8.4. A language B is **PSPACE-complete** if it satisfies:

1. $B \in$ PSPACE,
2. $\forall A \in$ PSPACE, $A \leq_p B$.

If B only satisfies condition 2, we say B is **PSPACE-hard**.

Definition 8.5. Boolean formulas with quantifiers are called **quantified Boolean formulas**.

Definition 8.6. **L** is the class of languages decidable in logarithmic space on a deterministic TM, L=SPACE($\log n$).
NL is the class of languages that are decidable in logarithmic space on a NTM, NL=NSPACE($\log n$).

Definition 8.7. If M is a TM that has a separate read-only input tape and w is an input, a **configuration of M on w** is a setting of the state, work tape, and positions of the two tape heads.

Definition 8.8. A language B is **NL-complete** if

1. $B \in$ NL,
2. $\forall A \in$ NL, $A \leq_L B$, A is log space reducible to B.

8.2 Key Results

Theorem 8.9 (Savitch)**.** *For any function* $f : \mathbb{N} \to \mathbb{R}^+$ *where* $f(n) \geq n$, NSPACE($f(n)$) $\subseteq$ SPACE($f^2(n)$).

Theorem 8.10. $TQBF = \{\langle\phi\rangle : \phi$ *is a true fully quantified Boolean formula*$\}$ *is PSPACE-complete.*

Theorem 8.11. $FORMULA\text{-}GAME = \{\langle\phi\rangle :$ *player* E *has a winning strategy in the formula game associated with* $\phi\}$ *is PSPACE-complete.*

Proof. This is the same language as $TQBF$. □

Theorem 8.12. $GG = \{\langle G, b\rangle :$ *player* I *has a winning strategy for the generalized geography game played on a graph* G *starting at node* $b\}$ *is PSPACE-complete.*

Definition 8.13. If $A \leq_L B$ and $B \in$ L $\implies$ $A \in$ L.

Corollary 8.14. *If any NL-complete language is in L, then L=NL.*

Theorem 8.15. $PATH$ *is NL-complete.*

Proof. Proof idea: we know $PATH \in$ NL. To show hard, construct a graph that represents the computation of the nondeterministic log space TM machine for A. □

Corollary 8.16. $NL \subseteq P$.

Proof. Immediately follows theorem because $PATH \in$ P. □

Theorem 8.17. *NL=coNL*.

Proof. Proof idea: show $\overline{PATH} \in$ NL. □

8.3 Proof Concepts and Examples

Example 8.18. $O(f(n))$ space $\implies 2^{O(f(n))}$ time before machine loops (think about all possible different configurations machine could take on before repeating one).

Example 8.19. (Problem 8.8): We can test the equivalence of two regular expressions in polynomial space.

8.4 Problem Set Results

9 Intractability

9.1 Key Definitions

Definition 9.1. An **intractable** problem is one that can't be solved practically due to excessive time or space requirements.

Definition 9.2. A function $f : \mathbb{N} \to \mathbb{N}$ where $f(n) \geq O(\log n)$ is called **space constructible** if the function maps the string 1^n to the binary representation of $f(n)$ using only $O(f(n))$ space.
Example: $f(n) = k$ for constant k is not space constructible, as the computation requires deleting 1^n in $O(n)$ time before writing the binary representation $\log k$. To account for $f(n) < O(\log n)$ we use the same "read-only" tape idea as the log-space transducer.

Definition 9.3. **EXPSPACE** $= \bigcup_k SPACE(2^{n^k})$.

Definition 9.4. A function $t : \mathbb{N} \to \mathbb{N}$ where $t(n) \geq O(n \log n)$ is **time constructible** if t maps 1^n to the binary representation of $t(n)$ in $O(t(n))$ time.

Definition 9.5. A language B is **EXPSPACE-complete** if

1. $B \in$ EXPSPACE, and
2. $\forall A \in$ EXPSPACE, $A \leq_p B$.

Definition 9.6. An **oracle** for language A reports whether $w \in A$ in one step.

Definition 9.7. An **oracle Turing machine** M^A is a TM that can query an oracle for A via an oracle tape. P^A is class of languages decidable in poly-time with a TM with oracle A. NP^A is class of languages decidable in non-deterministic poly-time with a TM with oracle A.
Remark: NP^A means non-determnistically pick, then check with P^A.

9.2 Key Results

Theorem 9.8 (Space hierarchy)**.** *For any space constructible f, there exists language A that is decidable in $O(f(n))$ space but not $o(f(n))$ space. i.e. some language **requires** at least $f(n)$ space to be decided.*

Proof. Proceed by diagonalization. Basically, we want to describe a language by constructing an associated TM that does the exact "opposite" of all "smaller"-space TMs. This algorithm runs TMs on descriptions of TMs, doing the opposite of individual TMs that run in $o(f(n))$ space (no requirement to be different from TMs that run in more than $f(n)$ space), and rejecting otherwise.
Consider the following algorithm that decides A: Let $D =$ "On input w:

1. Let n be the length of w.
2. Compute $f(n)$ in $O(f(n))$ space (constructability). Mark off $f(n)$ cells – this is the maximum space that any simulated TM can use. *Reject* if more space is ever used.
3. Check if w is in the form $\langle M \rangle 10^*$ for some M. The trailing 0's are to allow asymptotic behavior to "kick in" for large enough n (D might run in more than $f(n)$ space for small n and miss an opportunity to contradict M running in $o(f(n))$ space). If not in this form *reject*.
4. Simulate M on w and count the number of steps used in simulation. D might loop, so we cap the steps at $2^{f(n)}$ since there are max $f(n)$ cells to use. Exceed cap $\implies$ loop $\implies$ *reject*.
5. If M accepts, *reject*. If M rejects, *accept*.

D is obviously a decider. It runs in $f(n) = O(f(n))$ space, so A decidable in $O(f(n))$ space. AFTSOC some M decides A in $o(f(n))$ space. Then on sufficiently long input $\langle M \rangle 10^{n_0}$, D runs in $f(n)$ space and does the opposite of M, so A cannot be decided by M. □

Corollary 9.9. *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* $\implies SPACE(f_1(n)) \subsetneq SPACE(f_2(n))$

Proof. Immediate corollary from Space Hierarchy − if some languages absolutely require $O(f(n))$ space, then the languages decidable in this space is larger than a smaller space. □

Corollary 9.10. *For any real numbers* $0 \le \epsilon_1 < \epsilon_2 \implies SPACE(n^{\epsilon_1}) \subsetneq SPACE(n^{\epsilon_2})$

Corollary 9.11. $NL \subsetneq PSPACE$. *Remark: this implies* $TQBF \notin NL$ *(since* $TQBF$ *is PSPACE-complete w.r.t. log-space reduction, then* $TQBF \in NL \implies$ *all problems in PSPACE log-space reducible to problem in* $NL \implies PSPACE \subseteq NL \implies NL = PSPACE$*.)*

Proof. $NL = NSPACE(\log n) \subseteq SPACE(\log^2 n)$ by Savitch. Then by Space Hierarchy, $SPACE(\log^2 n) \subsetneq SPACE(n^k) \subset PSPACE$. □

Corollary 9.12. $PSPACE \subsetneq EXPSPACE$

Theorem 9.13 (Time hierarchy)**.** *For any time constructible* $t(n)$*, there exists language* A *requiring* $O(t(n))$ *time to be decided (not decidable in* $o(t(n)/\log t(n))$ *time).*

Proof. Remark: note the weaker bound. This is because simulating M on $\langle M\rangle$ requires a logarithmic increase in time, rather than a constant increase in space (just use multiple D tape symbols to represent a larger alphabet of M).
For the actual proof, consider the following $O(t(n))$ time D deciding A:

1. Let n be the length of w.
2. Compute $t(n)$ (constructible in $O(t(n))$) and store binary representation of the counter $t(n)/\log t(n)$. Decrement counter for each simulation step of M on w. If counter hits 0, M has used $t(n)/\log t(n)$ time, meaning D has taken $t(n)$ time, so *reject*.
3. If w is not in form $\langle M\rangle 10^*$, *reject*.
4. Simulate M on w.
5. Do opposite of M.

The details of the log increase in simulation time for step 4 are omitted for the sake of sanity. □

Corollary 9.14. *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* $\implies TIME(t_1(n)) \subsetneq TIME(t_2(n))$.

Corollary 9.15. *For any two real numbers* $1 \le \epsilon_1 < \epsilon_2 \implies TIME(n^{\epsilon_1}) \subsetneq TIME(n^{\epsilon_2})$.

Corollary 9.16. $P \subsetneq EXPTIME$

Theorem 9.17. *Let* $EQ_{REX\uparrow} = \{\langle Q, R\rangle : Q$ *and* R *are equivalent regular expressions with exponentiation* $\}$*. Then we have that* $EQ_{REX\uparrow}$ *is EXPSPACE-complete.*

Proof. Reductions by computation histories (see page 220 in Section 5.1 for review).
First we show $EQ_{REX\uparrow} \in EXPSPACE$. Writing out the regular expressions as concatenations instead of exponentiation gives us exponential-length inputs. Converting regex to NFAs increases size linearly. Then test *in*-equivalence of NFA using NTM. To do so, non-deterministically pick one-by-one an input symbol to read for $2^{q_1} \cdot 2^{q_2} = 2^{q_1+q_2}$ steps (all possible subsets of states, more steps would guarantee a repeat of "state of states"). This takes linear time in length of input (recall encoding NFA encodes exponential transition function possibilities). A deterministic version takes n^2 time (Savitch), where n is exponential.
Now, we need to show $EQ_{REX\uparrow}$ is **EXPSPACE-hard** (all languages poly-reduce to it). We basically map $w \in A$ to regex R_1 and R_2. $R_1 = \Delta^*$ where $\Delta = \Gamma \cup Q \cup \#$. We construct R_2 to be all the computation histories that do not lead to a reject on input w. Clearly $w \in A \iff R_1 = R_2 \iff \langle R_1, R_2\rangle \in EQ_{REX\uparrow}$. Let $R_2 = R_{bad-start} \cup R_{bad-reject} \cup R_{bad-window}$. We describe each regex qualitatively:

$$R_{bad-start} = S_0 \cup \cdots \cup S_n \cup S_B \cup S_\#$$

Each regex S_i generates strings *not* including the i^{th} appropriate symbol of the starting configuration at the i^{th} location. Special case for S_B, since encompasses all missed trailing blank locations (location $n+2$ to $2^{(n^k)}$, could be expo). Instead: $S_B = \Delta^{n+1}(\Delta \cup \epsilon)^{2^{(n^k)}-n-2}\Delta_{-\llcorner}\Delta^*$.
The notation Δ_{-q_0} is shorthand for writing the union of all symboles in Δ **except** q_0.
Then $R_{bad-reject} = \Delta^*_{-q_{reject}}$ (straightforward).
Similar to Cook-Levin proof, we have:

$$R_{bad-window} = \bigcup_{\text{bad}(abc,def)} \Delta^* abc \Delta^{(2^{(n^k)}-2)} def \Delta^*.$$

This is the same 2×3 invalid window approach we've seen before. Note the $(2^{n^k}) - 2$ difference: this is the distance from c to d one configuration away (c to f is exactly 2^{n^k}, so subtract 2). ☐

Theorem 9.18. *There exists oracle A such that $P^A \neq NP^A$. Remark: this suggests we cannot solve $P = NP$ because that would imply $P^A = NP^A$ for all A.*

Proof. Consider language $L_A = \{w : \exists x \in A[|x| = |w|]\}$ for any oracle A. $L_A \in NP^A$ (to check if $w \in L_A$, guess the right x and check if $x \in A$ using oracle for A). We construct a *particular* A.
Consider $M_1, M_2 \ldots$ running in n^i time. At each i, we construct A so that M_i^A cannot decide L_A. At stage i, pick n that is greater than length of any string currently in A, *and* such that $2^n > n^i$.
Run M_i ☐

Theorem 9.19. *There exists oracle B such that $P^B = NP^B$. Remark: this suggests we cannot solve $P \neq NP$ because that would imply $P^B \neq NP^B$ for all B.*

Proof. Consider any PSPACE-complete problem, like $TQBF$. Then $NP^B \subseteq NPSPACE \subseteq PSPACE \subseteq P^B$. ☐

9.3 Proof Concepts and Examples

Example 9.20. $NP \subseteq P^{SAT}$ since all problems in NP reduce to SAT with some poly-time reduction, then we can check in one step if in SAT. It follows that $NP \subseteq coP^{SAT} \implies coNP \subseteq P^{SAT}$.

Example 9.21. It is unclear if $\overline{\textit{MIN-FORMULA}} \in NP$ (guess smaller formula, but would need to verify truthiness across potentially exponential inputs). However, we know $\overline{\textit{MIN-FORMULA}} \in NP^{SAT}$. First, we can decide in-equivalence of ϕ in NP (guess the right assignment), so equivalence is decidable in *coNP*. To decide *MIN-FORMULA*, guess the right smaller ϕ' then easily verify if $\phi' = \phi$ using P^{SAT} since $coNP \subseteq P^{SAT}$.

9.4 Problem Set Results

10 Advanced Topics in Complexity Theory

10.1 Key Definitions

Definition 10.1. A **probabilistic TM** M is a nondeterministic TM where each nondeterministic step is a **coin flip step** with two equally legal moves. The probability of following any branch of computation b is $\Pr[b] = 2^{-k}$, with k being the number of coin-flip steps on the branch. We define the probability of PTM M accepting w as $\Pr[M \text{ accepts } w] = \sum_{b \text{ accepting}} \Pr[b]$

Definition 10.2. A PTM decider M need not be correct *all the time*. Indeed, we say M **decides** A **with error probability** ϵ if the decider is wrong with probability ϵ, i.e.:

1. $w \in A \implies \Pr[M \text{ accepts } w] \geq 1 - \epsilon$
2. $w \notin A \implies \Pr[M \text{ rejects } w] \geq 1 - \epsilon$

Definition 10.3. BPP is the class of languages decidable by a probabilistic poly-time TM with $\epsilon = 1/3$ (sufficient by **amplification lemma**).

Definition 10.4. A **branching program** is a directed acylic graph that has **query nodes** labeled x_i having two outgoing edges labeled 0 or 1, two **output nodes** labeled 0 *and* 1 without any outgoing edges. One node is designated the start node. *Remark:* a BP describes a Boolean function $f : \{0,1\}^m \to \{0,1\}$ (following the BP path using assignment will lead you to a 0 or 1).

Definition 10.5. RP is the class of languages decided by probabilistic polynomial time TMs where inputs in the language are accepted with probability at least 1/2 and inputs not in the language are rejected with probability 1.

Definition 10.6. A **branching program** is a directed acyclic graph where all nodes are labeled by variables, except for two **output nodes** labeled 0, 1. Nodes that are labeled by nodes are called **query nodes**, each with two outgoing edges labeled 0 or 1. Output nodes have no outgoing edges.
A **read-once branching program** is one that can query each variable at most one time on every directed path from start to output node.

Definition 10.7. Graphs G, H are **isomorphic** if nodes of G can be reordered so that it is identical to H. Let $ISO = \{\langle G, H\rangle : G \cong H\}$, and $NONISO = \{\langle G, H\rangle : G \ncong H\}$.
Note $ISO \in$ NP but $NONISO$ not known to be in NP. Neither are known to be NP-hard.

Definition 10.8. Language A is in **IP** if some polynomial time computable function V exists such that for some (arbitrary) function P and for every (arbitrary) function $\widetilde{P}$ and for every string w:

1. $w \in A \implies P(V \leftrightarrow P \text{ accepts } w) \geq 2/3$,
2. $w \notin A \implies P(V \leftrightarrow \widetilde{P} \text{ accepts } w) \leq 1/3$.

If $w \in A$, some Prover P, an "honest" Prover, causes Verifier to accept with high probability. But if $w \notin A$, not even a "crooked" Prover $\widetilde{P}$ causes Verifier to accept with high probability.

Definition 10.9. The **counting problem** for satisfiability is the language $\#SAT = \{\langle \phi, k\rangle : \phi$ is a cnf-formula with exactly k satisfying assignments$\}$.

10.2 Key Results

Lemma 10.10 (Amplification)**.** *Let $0 < \epsilon < 1/2$ be a fixed constant. Then for any polynomial $p(n)$, a probabilistic polynomial time TM M_1 that operates with error probability ϵ has an equivalent probabilistic polynomial time TM M_2 that operates with error probability $2^{-p(n)}$.*

Theorem 10.11. *$PRIMES = \{n : n$ is a prime number in binary$\} \in BPP$.*

Theorem 10.12. *$COMPOSITES \in RP$.*

Theorem 10.13. $EQ_{ROBP} = \{\langle B_1, B_2 \rangle : B_1 \cong B_2\} \in BPP$.

Lemma 10.14. *For every* $d \geq 0$*, a degree-*d *polynomial* p *on a single variable* x *either has at most* d *roots or is everywhere equal to* 0.

Lemma 10.15. *Let* $\mathbb{F}$ *a finite field with* f *elements and let* p *be a nonzero polynomial on variables* $x_1, \ldots, x_m$ *where* $\deg x_i \leq d$*. If* $a_1, \ldots, a_m \in \mathbb{F}$ *selected randomly, then* $P(p(a_1, \ldots, a_m = 0) \leq md/f$.

Theorem 10.16. *IP=PSPACE*.

Lemma 10.17. $PSPACE \subseteq IP$.

Theorem 10.18. $\#SAT \in IP$.

Proof. Proof idea: V and P must exchange $\#\phi(r_1 \cdots r_n z \cdots)$ for arithmetized Boolean formulas (polynomials) and compare the number of satisfying assignments. Details ommited. □

10.3 Proof Concepts and Examples

10.4 Problem Set Results

Index