Instructor's Solutions Manual for Introduction to the Theory of Computation third edition

Michael Sipser

Mathematics Department

MIT

Preface

This Instructor's Manual is designed to accompany the textbook, *Introduction to the Theory of Computation, third edition*, by Michael Sipser, published by Cengage, 2013. It contains solutions to almost all of the exercises and problems in Chapters 0–9. Most of the omitted solutions in the early chapters require figures, and producing these required more work that we were able to put into this manual at this point. A few problems were omitted in the later chapters without any good excuse.

Some of these solutions were based on solutions written by my teaching assistants and by the authors of the Instructor's Manual for the first edition.

This manual is available only to instructors.

Chapter 0

0.1
a. The odd positive integers.
b. The even integers.
c. The even positive integers.
d. The positive integers which are a multiple of 6.
e. The palindromes over $\{\mathtt{0},\mathtt{1}\}$.
f. The empty set.

0.2
a. $\{1, 10, 100\}$.
b. $\{n |\ n > 5\}$.
c. $\{1, 2, 3, 4\}$.
d. $\{\mathtt{aba}\}$.
e. $\{\varepsilon\}$.
f. $\emptyset$.

0.3
a. No.
b. Yes.
c. A.
d. B.
e. $\{(\mathtt{x},\mathtt{x}), (\mathtt{x},\mathtt{y}), (\mathtt{y},\mathtt{x}), (\mathtt{y},\mathtt{y}), (\mathtt{z},\mathtt{x}), (\mathtt{z},\mathtt{y})\}$.
f. $\{\emptyset, \{\mathtt{x}\}, \{\mathtt{y}\}, \{\mathtt{x},\mathtt{y}\}\}$.

0.4 $A \times B$ has ab elements, because each element of A is paired with each element of B, so $A \times B$ contains b elements for each of the a elements of A.

0.5 $\mathcal{P}(C)$ contains 2^c elements because each element of C may either be in $\mathcal{P}(C)$ or not in $\mathcal{P}(C)$, and so each element of C doubles the number of subsets of C. Alternatively, we can view each subset S of C as corresponding to a binary string b of length c, where S contains the ith element of C iff the ith place of b is $\mathtt{1}$. There are 2^c strings of length c and hence that many subsets of C.

0.6
a. $f(2) = 7$.
b. The range $= \{6, 7\}$ and the domain $= \{1, 2, 3, 4, 5\}$.
c. $g(2, 10) = 6$.
d. The range $= \{1, 2, 3, 4, 5\} \times \{6, 7, 8, 9, 10\}$ and the domain $= \{6, 7, 8, 9, 10\}$.
e. $f(4) = 7$ so $g(4, f(4)) = g(4, 7) = 8$.

0.7 The underlying set is $\mathcal{N}$ in these examples.

a. Let R be the "within 1" relation, that is, $R = \{(a, b) | \ |a - b| \leq 1\}$.

b. Let R be the "less than or equal to" relation, that is, $R = \{(a, b) | \ a \leq b\}$.

c. Finding a R that is symmetric and transitive but not reflexive is tricky because of the following "near proof" that R cannot exist! Assume that R is symmetric and transitive and chose any member x in the underlying set. Pick any other member y in the underlying set for which $(x, y) \in R$. Then $(y, x) \in R$ because R is symmetric and so $(x, x) \in R$ because R is transitive, hence R is reflexive. This argument fails to be an actual proof because y may fail to exist for x.

Let R be the "neither side is 1" relation, $R = \{(a, b) | \ a \neq 1 \text{ and } b \neq 1\}$.

0.10 Let G be any graph with n nodes where $n \geq 2$. The degree of every node in G is one of the n possible values from 0 to $n - 1$. We would like to use the pigeon hole principle to show that two of these values must be the same, but number of possible values is too great. However, not all of the values can occur in the same graph because a node of degree 0 cannot coexist with a node of degree $n - 1$. Hence G can exhibit at most $n - 1$ degree values among its n nodes, so two of the values must be the same.

0.11 The error occurs in the last sentence. If H contains at least 3 horses, H_1 and H_2 contain a horse in common, so the argument works properly. But, if H contains exactly 2 horses, then H_1 and H_2 each have exactly 1 horse, but do not have a horse in common. Hence we cannot conclude that the horse in H_1 has the same color as the horse in H_2. So the 2 horses in H may not be colored the same.

0.12 **a.** *Basis:* Let $n = 0$. Then, $S(n) = 0$ by definition. Furthermore, $\frac{1}{2}n(n + 1) = 0$. So $S(n) = \frac{1}{2}n(n + 1)$ when $n = 0$.

Induction: Assume true for $n = k$ where $k \geq 0$ and prove true for $n = k + 1$. We can use this series of equalities:

$$\begin{aligned} S(k+1) &= 1 + 2 + \cdots + k + (k+1) && \text{by definition} \\ &= S(k) + (k+1) && \text{because } S(k) = 1 + 2 + \cdots + k \\ &= \tfrac{1}{2}k(k+1) + (k+1) && \text{by the induction hypothesis} \\ &= \tfrac{1}{2}(k+1)(k+2) && \text{by algebra} \end{aligned}$$

b. *Basis:* Let $n = 0$. Then, $C(n) = 0$ by definition, and $\frac{1}{4}(n^4 + 2n^3 + n^2) = 0$. So $C(n) = \frac{1}{4}(n^4 + 2n^3 + n^2)$ when $n = 0$.

Induction: Assume true for $n = k$ where $k \geq 0$ and prove true for $n = k + 1$. We can use this series of equalities:

$$\begin{aligned} C(k+1) &= 1^3 + 2^3 + \cdots + k^3 + (k+1)^3 && \text{by definition} \\ &= C(k) + (k+1)^3 && C(k) = 1^3 + \cdots + k^3 \\ &= \tfrac{1}{4}(n^4 + 2n^3 + n^2) + (k+1)^3 && \text{induction hypothesis} \\ &= \tfrac{1}{4}((n+1)^4 + 2(n+1)^3 + (n+1)^2) && \text{by algebra} \end{aligned}$$

0.13 Dividing by $(a - b)$ is illegal, because $a = b$ hence $a - b = 0$ and division by 0 is undefined.

Chapter 1

1.12 Observe that $D \subseteq \mathtt{b}^*\mathtt{a}^*$ because D doesn't contain strings that have $\mathtt{ab}$ as a substring. Hence D is generated by the regular expression $(\mathtt{aa})^*\mathtt{b}(\mathtt{bb})^*$. From this description, finding the DFA for D is more easily done.

1.14 **a.** Let M' be the DFA M with the accept and non-accept states swapped. We show that M' recognizes the complement of B, where B is the language recognized by M. Suppose M' accepts x. If we run M' on x we end in an accept state of M'. Because M and M' have swapped accept/non-accept states, if we run M on x, we would end in a non-accept state. Therefore, $x \notin B$. Similarly, if x is not accepted by M', then it would be accepted by M. So M' accepts exactly those strings not accepted by M. Therefore, M' recognizes the complement of B.
Since B could be any arbitrary regular language and our construction shows how to build an automaton to recognize its complement, it follows that the complement of any regular language is also regular. Therefore, the class of regular languages is closed under complement.

b. Consider the NFA in Exercise 1.16(a). The string a is accepted by this automaton. If we swap the accept and reject states, the string a is still accepted. This shows that swapping the accept and non-accept states of an NFA doesn't necessarily yield a new NFA recognizing the complementary language. The class of languages recognized by NFAs is, however, closed under complement. This follows from the fact that the class of languages recognized by NFAs is precisely the class of languages recognized by DFAs which we know is closed under complement from part (a).

1.18 Let $\Sigma = \{0, 1\}$.

a. $1\Sigma^*0$
b. $\Sigma^*1\Sigma^*1\Sigma^*1\Sigma^*$
c. $\Sigma^*0101\Sigma^*$
d. $\Sigma\Sigma0\Sigma^*$
e. $(0 \cup 1\Sigma)(\Sigma\Sigma)^*$
f. $(0 \cup (10)^*)^*1^*$
g. $(\varepsilon \cup \Sigma)(\varepsilon \cup \Sigma)(\varepsilon \cup \Sigma)(\varepsilon \cup \Sigma)(\varepsilon \cup \Sigma)$
h. $\Sigma^*0\Sigma^* \cup 1111\Sigma^* \cup 1 \cup \varepsilon$
i. $(1\Sigma)^*(1 \cup \varepsilon)$
j. $0^*(100 \cup 010 \cup 001 \cup 00)0^*$
k. $\varepsilon \cup 0$
l. $(1^*01^*01^*)^* \cup 0^*10^*10^*$

m. $\emptyset$
n. Σ^+

1.20 **a.** ab, ε; ba, aba
b. ab, abab; ε, aabb
c. ε, aa; ab, aabb
d. ε, aaa; aa, b
e. aba, aabbaa; ε, abbb
f. aba, bab; ε, ababab
g. b, ab; ε, bb
h. ba, bba; b, ε

1.21 In both parts we first add a new start state and a new accept state. Several solutions are possible, depending on the order states are removed.
a. Here we remove state 1 then state 2 and we obtain
$\mathtt{a}^*\mathtt{b}(\mathtt{a} \cup \mathtt{b}\mathtt{a}^*\mathtt{b})^*$
b. Here we remove states 1, 2, then 3 and we obtain
$\varepsilon \cup ((\mathtt{a} \cup \mathtt{b})\mathtt{a}^*\mathtt{b}((\mathtt{b} \cup \mathtt{a}(\mathtt{a} \cup \mathtt{b}))\mathtt{a}^*\mathtt{b})^*(\varepsilon \cup \mathtt{a}))$

1.22 **b.** $\mathtt{/\#}(\mathtt{\#}^*(\mathtt{a} \cup \mathtt{b}) \cup \mathtt{/})^*\mathtt{\#}^+\mathtt{/}$

1.24 **a.** q_1, q_1, q_1, q_1; 000.
b. q_1, q_2, q_2, q_2; 111.
c. q_1, q_1, q_2, q_1, q_2; 0101.
d. q_1, q_3; 1.
e. q_1, q_3, q_2, q_3, q_2; 1111.
f. $q_1, q_3, q_2, q_1, q_3, q_2, q_1$; 110110.
g. q_1; ε.

1.25 A *finite state transducer* is a 5-tuple $(Q, \Sigma, \Gamma, \delta, q_0)$, where
i) Q is a finite set called the *states*,
ii) Σ is a finite set called the *alphabet*,
iii) Γ is a finite set called the *output alphabet*,
iv) δ: $Q \times \Sigma \longrightarrow Q \times \Gamma$ is the *transition function*,
v) $q_0 \in Q$ is the *start state*.

Let $M = (Q, \Sigma, \Gamma, \delta, q_0)$ be a *finite state transducer*, $w = w_1 w_2 \cdots w_n$ be a string over Σ, and $v = v_1 v_2 \cdots v_n$ be a string over the Γ. Then M *outputs* v if a sequence of states $r_0, r_1, \ldots, r_n$ exists in Q with the following two conditions:
i) $r_0 = q_o$
ii) $\delta(r_i, w_{i+1}) = (r_{i+1}, v_{i+1})$ for $i = 0, \ldots, n-1$.

1.26 **a.** $T_1 = (Q, \Sigma, \Gamma, \delta, q_1)$, where
i) $Q = \{q_1, q_2\}$,
ii) $\Sigma = \{0, 1, 2\}$,
iii) $\Gamma = \{0, 1\}$,
iv) δ is described as

	0	1	2
q_1	(q_1,0)	(q_1,0)	(q_2,1)
q_2	(q_1,0)	(q_2,1)	(q_2,1)

v) q_1 is the start state.

b. $T_2 = (Q, \Sigma, \Gamma, \delta, q_1)$, where
i) $Q = \{q_1, q_2, q_3\}$,
ii) $\Sigma = \{\mathtt{a}, \mathtt{b}\}$,
iii) $\Gamma = \{\mathtt{0}, \mathtt{1}\}$,
iv) δ is described as

	a	b
q_1	(q_2,1)	(q_3,1)
q_2	(q_3,1)	(q_1,0)
q_3	(q_1,0)	(q_2,1)

v) q_1 is the start state.

1.29 **b.** Let $A_2 = \{www |\ w \in \{\mathtt{0},\mathtt{1}\}^*\}$. We show that A_2 is nonregular using the pumping lemma. Assume to the contrary that A_2 is regular. Let p be the pumping length given by the pumping lemma. Let s be the string $\mathtt{a}^p\mathtt{b}\mathtt{a}^p\mathtt{b}\mathtt{a}^p\mathtt{b}$. Because s is a member of A_2 and s has length more than p, the pumping lemma guarantees that s can be split into three pieces, $s = xyz$, satisfying the three conditions of the lemma. However, condition 3 implies that y must consist only of as, so $xyyz \notin A_2$ and one of the first two conditions is violated. Therefore A_2 is nonregular.

1.30 The error is that $s = \mathtt{0}^p\mathtt{1}^p$ *can* be pumped. Let $s = xyz$, where $x = \mathtt{0}$, $y = \mathtt{0}$ and $z = \mathtt{0}^{p-2}\mathtt{1}^p$. The conditions are satisfied because
i) for any $i \geq 0$, $xy^iz = \mathtt{0}\mathtt{0}^i\mathtt{0}^{p-2}\mathtt{1}^p$ is in $\mathtt{0}^*\mathtt{1}^*$.
ii) $|y| = 1 > 0$, and
iii) $|xy| = 2 \leq p$.

1.31 We construct a DFA which alternately simulates the DFAs for A and B, one step at a time. The new DFA keeps track of which DFA is being simulated. Let $M_1 = (Q_1, \Sigma, \delta_1, s_1, F_1)$ and $M_2 = (Q_2, \Sigma, \delta_2, s_2, F_2)$ be DFAs for A and B. We construct the following DFA $M = (Q, \Sigma, \delta, s_0, F)$ for the perfect shuffle of A and B.
i) $Q = Q_1 \times Q_2 \times \{1, 2\}$.
ii) For $q_1 \in Q_1$, $q_2 \in Q_2$, $b \in \{1, 2\}$, and $a \in \Sigma$:

$$\delta((q_1, q_2, b), a) = \begin{cases} (\delta_1(q_1, a), q_2, 2) & b = 1 \\ (q_1, \delta_1(q_2, a), 1) & b = 2. \end{cases}$$

iii) $s_0 = (s_1, s_2, 1)$.
iv) $F = \{(q_1, q_2, 1) |\ q_1 \in F_1 \text{ and } q_2 \in F_2\}$.

1.32 We construct an NFA which simulates the DFAs for A and B, nondeterministically switching back and forth from one to the other. Let $M_1 = (Q_1, \Sigma, \delta_1, s_1, F_1)$ and $M_2 = (Q_2, \Sigma, \delta_2, s_2, F_2)$ be DFAs for A and B. We construct the following NFA $N = (Q, \Sigma, \delta, s_0, F)$ for the shuffle of A and B.
i) $Q = Q_1 \times Q_2$.
ii) For $q_1 \in Q_1$, $q_2 \in Q_2$, and $a \in \Sigma$:

$$\delta((q_1, q_2), a) = \{(\delta_1(q_1, a), q_2), (q_1, \delta_2(q_2, a))\}.$$

iii) $s_0 = (s_1, s_2)$.
iv) $F = \{(q_1, q_2) |\ q_1 \in F_1 \text{ and } q_2 \in F_2\}$.

1.33 Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA that recognizes A. Then we construct NFA $N = (Q', \Sigma, \delta', q_0', F')$ recognizing *DROP-OUT*(A). The idea behind the construction is that N simulates M on its input, nondeterministically guessing the point at which the

dropped out symbol occurs. At that point N guesses the symbol to insert in that place, without reading any actual input symbol at that step. Afterwards, it continues to simulate M.

We implement this idea in N by keeping two copies of M, called the top and bottom copies. The start state is the start state of the top copy. The accept states of N are the accept states of the bottom copy. Each copy contains the edges that would occur in M. Additionally, include ε edges from each state q in the top copy to every state in the bottom copy that q can reach.

We describe N formally. The states in the top copy are written with a T and the bottom with a B, thus: (T, q) and (B, q).

i) $Q' = \{\mathrm{T}, \mathrm{B}\} \times Q$,
ii) $q_0' = (\mathrm{T}, q_0)$,
iii) $F' = \{\mathrm{B}\} \times F$,
iv) $\delta'((\mathrm{T}, q), a) = \begin{cases} \{(\mathrm{T}, \delta(q, a))\} & a \in \Sigma \\ \{(\mathrm{B}, \delta(q, b)) \mid b \in \Sigma\} & a = \varepsilon \end{cases}$

$$\delta'((\mathrm{B}, q), a) = \begin{cases} \{(\mathrm{B}, \delta(q, a))\} & a \in \Sigma \\ \emptyset & a = \varepsilon \end{cases}$$

1.35 Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA that recognizes A. We construct a new DFA $M' = (Q, \Sigma, \delta, q_0, F')$ that recognizes A/B. Automata M and M' differ only in the sets of accept states. Let $F' = \{r \mid$ starting at r and reading a string in B we get to an accept state of $M\}$. Thus M' accepts a string w iff there is a string $x \in B$ where M accepts wx. Hence M' recognizes A/B.

1.36 For any regular language A, let M_1 be the DFA recognizing it. We need to find a DFA that recognizes A^R. Since any NFA can be converted to an equivalent DFA, it suffices to find an NFA M_2 that recognizes A^R.

We keep all the states in M_1 and reverse the direction of all the arrows in M_1. We set the accept state of M_2 to be the start state in M_1. Also, we introduce a new state q_0 as the start state for M_2 which goes to every accept state in M_1 by an ϵ-transition.

1.39 The idea is that we start by comparing the most significant bit of the two rows. If the bit in the top row is bigger, we know that the string is in the language. The string does not belong to the language if the bit in the top row is smaller. If the bits on both rows are the same, we move on to the next most significant bit until a difference is found. We implement this idea with a DFA having states q_0, q_1, and q_2. State q_0 indicates the result is not yet determined. States q_1 and q_2 indicate the top row is known to be larger, or smaller, respectively. We start with q_0. If the top bit in the input string is bigger, it goes to q_1, the only accept state, and stays there till the end of the input string. If the top bit in the input string is smaller, it goes to q_2 and stays there till the end of the input string. Otherwise, it stays in state q_0.

1.40 Assume language E is regular. Use the pumping lemma to a get a pumping length p satisfying the conditions of the pumping lemma. Set $s = \left[\begin{smallmatrix}0\\1\end{smallmatrix}\right]^p \left[\begin{smallmatrix}1\\0\end{smallmatrix}\right]^p$. Obviously, $s \in E$ and $|s| \geq p$. Thus, the pumping lemma implies that the string s can be written as xyz with $x = \left[\begin{smallmatrix}0\\1\end{smallmatrix}\right]^a$, $y = \left[\begin{smallmatrix}0\\1\end{smallmatrix}\right]^b$, $z = \left[\begin{smallmatrix}0\\1\end{smallmatrix}\right]^c \left[\begin{smallmatrix}1\\0\end{smallmatrix}\right]^p$, where $b \geq 1$ and $a + b + c = p$. However, the string $s' = xy^0z = \left[\begin{smallmatrix}0\\1\end{smallmatrix}\right]^{a+c} \left[\begin{smallmatrix}1\\0\end{smallmatrix}\right]^p \notin E$, since $a + c < p$. That contradicts the pumping lemma. Thus E is not regular.

1.41 For each $n \geq 1$, we build a DFA with the n states $q_0, q_1, \ldots, q_{n-1}$ to count the number of consecutive a's modulo n read so far. For each character a that is input, the counter increments by 1 and jumps to the next state in M. It accepts the string if and only if the machine stops at q_0. That means the length of the string consists of all a's and its length is a multiple of n.

More formally, the set of states of M is $Q = \{q_0, q_1, \ldots, q_{n-1}\}$. The state q_0 is the start state and the only accept state. Define the transition function as: $\delta(q_i, \mathtt{a}) = q_j$ where $j = i + 1 \bmod n$.

1.42 By simulating binary division, we create a DFA M with n states that recognizes C_n. M has n states which keep track of the n possible remainders of the division process. The start state is the only accept state and corresponds to remainder 0.

The input string is fed into M starting from the most significant bit. For each input bit, M doubles the remainder that its current state records, and then adds the input bit. Its new state is the sum modulo n. We double the remainder because that corresponds to the left shift of the computed remainder in the long division algorithm. If an input string ends at the accept state (corresponding to remainder 0), the binary number has no remainder on division by n and is therefore a member of C_n.

The formal definition of M is $(\{q_0, \ldots, q_{n-1}\}, \{\mathtt{0}, \mathtt{1}\}, \delta, q_0, \{q_0\})$. For each $q_i \in Q$ and $b \in \{\mathtt{0}, \mathtt{1}\}$, define $\delta(q_i, b) = q_j$ where $j = (2i + b) \bmod n$.

1.43 Use the same construction given in the proof of Theorem 1.39, which shows the equivalence of NFAs and DFAs. We need only change F', the set of accept states of the new DFA. Here we let $F' = \mathcal{P}(F)$. The change means that the new DFA accepts only when *all* of the possible states of the all-NFA are accepting.

1.44 Let $A_k = \Sigma^* \mathtt{0}^{k-1} \mathtt{0}^*$. A DFA with k states $\{q_0, \ldots, q_{k-1}\}$ can recognize A_k. The start state is q_0. For each i from 0 to $k-2$, state q_i branches to q_{i+1} on 0 and to q_0 on 1. State q_{k-1} is the accept state and branches to itself on 0 and to q_0 on 1.

In any DFA with fewer than k states, two of the k strings 1, 10, ... , 10^{k-1} must cause the machine to enter the same state, by the pigeon hole principle. But then, if we add to both of these strings enough 0s to cause the longer of these two strings to have exactly $k-1$ 0s, the two new strings will still cause the machine to enter the same state, but one of these strings is in A_k and the other is not. Hence, the machine must fail to produce the correct accept/reject response on one of these strings.

1.45 **b.** Let $M = (Q, \Sigma, \delta, q_0, F)$ be an NFA recognizing A, where A is some regular language. We construct $M' = (Q', \Sigma, \delta, q_0', F')$ recognizing $NOEXTEND(A)$ as follows:

i) $Q' = Q$
ii) $\delta' = \delta$
iii) $q_0' = q_0$
iv) $F' = \{q \mid q \in F$ and there is no path of length ≥ 1 from q to an accept state$\}$.

1.47 To show that $\equiv_L$ is an equivalence relation we show it is reflexive, symmetric, and transitive. It is reflexive because no string can distinguish x from itself and hence $x \equiv_L x$ for every x. It is symmetric because x is distinguishable from y whenever y is distinguishable from x. It is transitive because if $w \equiv_L x$ and $x \equiv_L y$, then for each z, $wz \in L$ iff $xz \in L$ and $xz \in L$ iff $yz \in L$, hence $wz \in L$ iff $yz \in L$, and so $w \equiv_L y$.

1.49 **a.** F is not regular, because the nonregular language $\{\mathtt{ab}^n\mathtt{c}^n \mid n \geq 0\}$ is the same as $F \cap \mathtt{ab}^*\mathtt{c}^*$, and the regular languages are closed under intersection.

b. Language F satisfies the conditions of the pumping lemma using pumping length 2. If $s \in F$ is of length 2 or more we show that it can be pumped by considering four cases, depending on the number of a's that s contains.

i) If s is of the form $\mathtt{b}^*\mathtt{c}^*$, let $x = \varepsilon$, y be the first symbol of s, and let z be the rest of s.

ii) If s is of the form $\mathtt{ab}^*\mathtt{c}^*$, let $x = \varepsilon$, y be the first symbol of s, and let z be the rest of s.

iii) If s is of the form $\mathtt{aab}^*\mathtt{c}^*$, let $x = \varepsilon$, y be the first two symbols of s, and let z be the rest of s.

iv) If s is of the form $\mathtt{aaa}^*\mathtt{b}^*\mathtt{c}^*$, let $x = \varepsilon$, y be the first symbol of s, and let z be the rest of s.

In each case, the strings xy^iz are members of F for every $i \geq 0$. Hence F satisfies the conditions of the pumping lemma.

c. The pumping lemma is not violated because it states only that regular languages satisfy the three conditions, and it doesn't state that nonregular languages fail to satisfy the three conditions.

1.50 The objective of this problem is for the student to pay close attention to the exact formulation of the pumping lemma.

c. This language is that same as the language in in part (b), so the solution is the same.

e. The minimum pumping length is 1. The pumping length cannot be 0, as in part (b). Any string in $(\mathtt{01})^*$ of length 1 or more contains $\mathtt{01}$ and hence can be pumped by dividing it so that $x = \varepsilon$, $y = \mathtt{01}$, and z is the rest.

f. The minimum pumping length is 1. The pumping length cannot be 0, as in part (b). The language has no strings of length 1 or more so 1 is a pumping length. (the conditions hold vacuously).

g. The minimum pumping length is 3. The string $\mathtt{00}$ is in the language and it cannot be pumped, so the minimum pumping length cannot be 2. Every string in the language of length 3 or more contains a $\mathtt{1}$ within the first 3 symbols so it can be pumped by letting y be that $\mathtt{1}$ and letting x be the symbols to the left of y and z be the symbols to the right of y.

h. The minimum pumping length is 4. The string $\mathtt{100}$ is in the language but it cannot be pumped (down), therefore 3 is too small to be a pumping length. Any string of length 4 or more in the language must be of the form xyz where x is $\mathtt{10}$, y is in $\mathtt{11}^*\mathtt{0}$ and z is in $(\mathtt{11}^*\mathtt{0})^*\mathtt{0}$, which satisfies all of the conditions of the pumping lemma.

i. The minimum pumping length is 5. The string $\mathtt{1011}$ is in the language and it cannot be pumped. Every string in the language of length 5 or more (there aren't any) can be pumped (vacuously).

j. The minimum pumping length is 1. It cannot be 0 as in part (b). Every other string can be pumped, so 1 is a pumping length.

1.51 **a.** Assume $L = \{\mathtt{0}^n\mathtt{1}^m\mathtt{0}^n \mid m, n \geq 0\}$ is regular. Let p be the pumping length given by the pumping lemma. The string $s = \mathtt{0}^p\mathtt{10}^p \in L$, and $|s| \geq p$. Thus the pumping lemma implies that s can be divided as xyz with $x = \mathtt{0}^a$, $y = \mathtt{0}^b$, $z = \mathtt{0}^c\mathtt{10}^p$, where $b \geq 1$ and $a + b + c = p$. However, the string $s' = xy^0z = \mathtt{0}^{a+c}\mathtt{10}^p \notin L$, since $a + c < p$. That contradicts the pumping lemma.

c. Assume $C = \{w |\ w \in \{0,1\}^* \text{ is a palindrome}\}$ is regular. Let p be the pumping length given by the pumping lemma. The string $s = 0^p10^p \in C$ and $|s| \geq p$. Follow the argument as in part (a). Hence C isn't regular, so neither is its complement.

1.52 One short solution is to observe that $\overline{Y} \cap 1^*\#1^* = \{1^n\#1^n |\ n \geq 0\}$. This language is clearly not regular, as may be shown using a straightforward application of the pumping lemma. However, if Y were regular, this language would be regular, too, because the class of regular languages is closed under intersection and complementation. Hence Y isn't regular.

Alternatively, we can show Y isn't regular directly using the pumping lemma. Assume to the contrary that Y is regular and obtain its pumping length p. Let $s = 1^{p!}\#1^{2p!}$. The pumping lemma says that $s = xyz$ satisfying the three conditions. By condition 3, y appears among the left-hand 1s. Let $l = |y|$ and let $k = (p!/l)$. Observe that k is an integer, because l must be a divisor of $p!$. Therefore, adding k copies of y to s will add $p!$ additional 1s to the left-hand 1s. Hence, $xy^{1+k}z = 1^{2p!}\#1^{2p!}$ which isn't a member of Y. But condition 1 of the pumping lemma states that this string is a member of Y, a contradiction.

1.53 The language D can be written alternatively as $0\Sigma^*0 \cup 1\Sigma^*1 \cup 0 \cup 1 \cup \varepsilon$, which is obviously regular.

1.54 The NFA N_k guesses when it has read an a that appears at most k symbols from the end, then counts $k-1$ more symbols and enters an accept state. It has an initial state q_0 and additional states q_1 thru q_k. State q_0 has transitions on both a and b back to itself and on a to state q_1. For $1 \leq i \leq k-1$, state q_i has transitions on a and b to state q_{i+1}. State q_k is an accept state with no transition arrows coming out of it. More formally, NFA $N_k = (Q, \Sigma, \delta, q_0, F)$ where

i) $Q = \{q_0, \ldots, q_k\}$

ii) $\delta(q,c) = \begin{cases} \{q_0\} & q = q_0 \text{ and } c = \mathtt{a} \\ \{q_0, q_1\} & q = q_0 \text{ and } c = \mathtt{b} \\ \{q_{i+1}\} & q = q_i \text{ for } 1 \leq i < k \text{ and } c \in \Sigma \\ \emptyset & q = q_k \text{ or } c = \varepsilon \end{cases}$

iii) $F = \{q_k\}$.

1.55 Let M be a DFA. Say that w *leads* to state q if M is in q after reading w. Notice that if w_1 and w_2 lead to the same state, then w_1p and w_2p also lead to the same state, for all strings p.

Assume that M recognizes C_k with fewer than 2^k states, and derive a contradiction. There are 2^k different strings of length k. By the pigeonhole principle, two of these strings w_1 and w_2 lead to the same state of M.

Let i be the index of the first bit on which w_1 and w_2 differ. Since w_1 and w_2 lead M to the same state, $w_1\mathtt{b}^{i-1}$ and $w_2\mathtt{b}^{i-1}$ lead M to the same state. This cannot be the case, however, since one of the strings should be rejected and the other accepted. Therefore, any two distinct k bit strings lead to different states of M. Hence M has at least 2^k states.

1.60 **a.** We construct M' from M by converting each transition that is traversed on symbol $a \in \Sigma$ to a sequence of transitions that are traversed while reading string $f(a) \in \Gamma^*$. The formal construction follows.

Let $M = (Q, \Sigma, \delta, q_0, F)$. For each $a \in \Sigma$ let $z^a = f(a)$ and let $k_a = |z^a|$. We write $z^a = z_1^a z_2^a \ldots z_{k_a}^a$ where $z_i^a \in \Gamma$. Construct $M' = (Q', \Gamma, \delta', q_0, F)$. $Q' = Q \cup \{q_i^a |\ q \in Q, a \in \Sigma, 1 \leq i \leq k_a\}$

For every $q \in Q$,

$$\delta'(q, b) = \begin{cases} \{r |\ \delta(q, a) = r \text{ and } z^a = \varepsilon\} & b = \varepsilon \\ \{q_1^a |\ b = z_1^a\} & b \neq \varepsilon \end{cases}$$

$$\delta'(q_i^a, b) = \begin{cases} \{q_{i+1}^a |\ b = z_{i+1}^a\} & 1 \leq i < k_a \text{ and } b \neq \varepsilon \\ \{r |\ \delta(q, a) = r\} & i = k_a \text{ and } b = \varepsilon \end{cases}$$

b. The above construction shows that the class of regular languages is closed under homomorphism. To show that the class of non-regular languages is not closed under homomorphism, let $B = \{\mathtt{0}^m\mathtt{1}^m |\ m \geq 0\}$ and let $f\colon \{\mathtt{0},\mathtt{1}\} \longrightarrow \{\mathtt{2}\}$ be the homomorphism where $f(\mathtt{0}) = f(\mathtt{1}) = \mathtt{2}$. We know that B is a non-regular language but $f(B) = \{\mathtt{2}^{2m} |\ m \geq 0\}$ is a regular language.

1.61 **a.** $RC(A) = \{w_{i+1} \cdots w_n w_1 \cdots w_i |\ w = w_1 \cdots w_n \in A \text{ and } 1 \leq i \leq n\}$, because we can let $x = w_1 \cdots w_i$ and $y = w_{i+1} \cdots w_n$. Then $RC(RC(A))$ gives the same language because if $st = w_{i+1} \cdots w_n w_1 \cdots w_i$ for some strings s and t, then $ts = w_{j+1} \cdots w_n w_1 \cdots w_j$ for some j where $1 \leq j \leq n$.

b. Let A be a regular language that is recognized by DFA M. We construct a NFA N that recognizes $RC(A)$. In the idea behind the construction, N guesses the cut point nondeterministically by starting M at any one of its states. Then N simulates M on the input symbols it reads. If N finds that M is in one of its accept states then N may nondeterministically reset M back to it's start state prior to reading the next symbol. Finally, N accepts its input if the simulation ends in the same state it started in, and exactly one reset occurred during the simulation. Here is the formal construction. Let $M = (Q, \Sigma, \delta, q_0, F)$ recognize A and construct $N = (Q', \Sigma, \delta', r, F')$ to recognize $RC(A)$.

Set $Q' = (Q \times Q \times \{1, 2\}) \cup \{r\}$. State (q, r, i) signifies that N started the simulation in M's state q, it is currently simulating M in state r, and if $i = 1$ a reset hasn't yet occurred whereas if $i = 2$ then a reset has occurred.

Set $\delta'(r, \varepsilon) = \{(q, q, 1) |\ q \in Q\}$. This starts simulating M in each of its states, nondeterministically.

Set $\delta'((q, r, i), a) = \{(q, \delta(r, a), i)\}$ for each $q, r \in Q$ and $i \in \{1, 2\}$. This continues the simulation.

Set $\delta'((q, r, 1), \varepsilon) = \{(q, q_0, 2)\}$ for $r \in F$ and $q \in Q$. This allows N to reset the simulation to q_0 if M hasn't yet been reset and M is currently in an accept state.

We set δ' to $\emptyset$ if it is otherwise unset.

$F' = \{(q, q, 2) |\ q \in Q\}$.

1.62 Assume to the contrary that ADD is regular. Let p be the pumping length given by the pumping lemma. Choose s to be the string $\mathtt{1}^p\mathtt{=0+1}^p$, which is a member of ADD. Because s has length greater than p, the pumping lemma guarantees that s can be split into three pieces, $s = xyz$, satisfying the conditions of the lemma. By the third condition in the pumping lemma have that $|xy| \leq p$, it follows that y is $\mathtt{1}^k$ for some $k \geq 1$. Then xy^2z is the string $\mathtt{1}^{p+k}\mathtt{=0+1}^p$, which is not a member of ADD, violating the pumping lemma. Hence ADD isn't regular.

1.63 Let $A = \{2^k | \ k \geq 0\}$. Clearly $B_2(A) = \mathtt{1}\mathtt{0}^*$ is regular. Use the pumping lemma to show that $B_3(A)$ isn't regular. Get the pumping length p and chose $s \in B_3(A)$ of length p or more. We show s cannot be pumped. Let $s = xyz$. For string w, write $(w)_3$ to be the number that w represents in base 3. Then

$$\lim_{i\to\infty} \frac{(xy^iz)_3}{(xy^i)_3} = 3^{|z|} \quad \text{and} \quad \lim_{i\to\infty} \frac{(xy^{i+1}z)_3}{(xy^i)_3} = 3^{|y|+|z|}$$

so

$$\lim_{i\to\infty} \frac{(xy^{i+1}z)_3}{(xy^iz)_3} = 3^{|y|}.$$

Therefore for a sufficiently large i,

$$\frac{(xy^{i+1}z)_3}{(xy^iz)_3} = 3^{|y|} \pm \alpha$$

for some $\alpha < 1$. But this fraction is a ratio of two members of A and is therefore a whole number. Hence $\alpha = 0$ and the ratio is a power of 3. But the ration of two members of A also is a power of 2. No number greater than 1 can be both a power of 2 and of 3, a contradiction.

1.64 Given an NFA M recognizing A we construct an NFA N accepting $A_{\frac{1}{2}\_}$ using the following idea. M keeps track of two states in N using two "fingers". As it reads each input symbol, N uses one finger to simulate M on that symbol. Simultaneously, M uses the other finger to run M backwards from an accept state on a guessed symbol. N accepts whenever the forward simulation and the backward simulation are in the same state, that is, whenever the two fingers are together. At those points we are sure that N has found a string where another string of the same length can be appended to yield a member of A, precisely the definition of $A_{\frac{1}{2}\_}$.

In the formal construction, Exercise 1.11 allows us to assume for simplicity that the NFA M recognizing A has a single accept state. Let $M = (Q, \Sigma, \delta, q_0, q_{\text{accept}})$. Construct NFA $N = (Q', \Sigma, \delta', q_0', F')$ recognizing the first halves of the strings in A as follows:

i) $Q' = Q \times Q$.
ii) For $q, r \in Q$ define
$\delta'((q, r), a) = \{(u, v) | \ u \in \delta(q, a) \text{ and } r \in \delta(v, b) \text{ for some } b \in \Sigma\}$.
iii) $q_0{}' = (q_0, q_{\text{accept}})$.
iv) $F' = \{(q, q) | \ q \in Q\}$.

1.65 Let $A = \{\mathtt{0}^*\mathtt{\#}\mathtt{1}^*\}$. Thus, $A_{\frac{1}{3}-\frac{1}{3}} \cap \{\mathtt{0}^*\mathtt{1}^*\} = \{\mathtt{0}^n\mathtt{1}^n | \ n \geq 0\}$. Regular sets are closed under intersection, and $\{\mathtt{0}^n\mathtt{1}^n | \ n \geq 0\}$ is not regular, so $A_{\frac{1}{3}-\frac{1}{3}}$ is not regular.

1.66 If M has a synchronizing sequence, then for any pair of states (p, q) there is a string $w_{p,q}$ such that $\delta(p, w_{p,q}) = \delta(q, w_{p,q}) = h$, where h is the home state. Let us run two copies of M starting at states p and q respectively. Consider the sequence of pairs of states (u, v) that the two copies run through before reaching home state h. If some pair appears in the sequence twice, we can delete the substring of $w_{p,q}$ that takes the copies of M from one occurrence of the pair to the other, and thus obtain a new $w_{p,q}$. We repeat the process until all pairs of states in the sequence are distinct. The number of distinct state pairs is k^2, so $|w_{p,q}| \leq k^2$.

Suppose we are running k copies of M and feeding in the same input string s. Each copy starts at a different state. If two copies end up at the same state after some step,

they will do exactly the same thing for the rest of input, so we can get rid of one of them. If s is a synchronizing sequence, we will end up with one copy of M after feeding in s. Now we will show how to construct a synchronizing sequence s of length at most k^3.

i) Start with $s = \epsilon$. Start k copies of M, one at each of its states. Repeat the following two steps until we are left with only a single copy of M.
ii) Pick two of M's remaining copies (M_p and M_q) that are now in states p and q after reading s.
iii) Redefine $s = sw_{p,q}$. After reading this new s, M_p and M_q will be in the same state, so we eliminate one of these copies.

At the end of the above procedure, s brings all states of M to a single state. Call that state h. Stages 2 and 3 are repeated $k - 1$ times, because after each repetition we eliminate one copy of M. Therefore $|s| \leq (k-1)k^2 < k^3$.

1.67 Let $C = \Sigma^* B \Sigma^*$. Then C is the language of all strings that contain some member of B as a substring, If B is regular then C is also regular. We know from the solution to Problem 1.14 that the complement of a regular language is regular, and so $\overline{C}$ is regular. It is the language of all strings that do not contain some member of B as a substring. Note that A *avoids* $B = A \cap \overline{C}$. Therefore A *avoids* B is regular because we showed that the intersection of two regular languages is regular.

1.68 **a.** Any string that doesn't begin and end with 0 obviously cannot be a member of A. If string w does begin and end with 0 then $w = 0u0$ for some string u. Hence $A = 0\Sigma^*0$ and therefore A is regular.

b. Assume for contradiction that B is regular. Use the pumping lemma to get the pumping length p. Letting $s = 0^p10^p$ we have $s \in B$ and so we can divide up $s = xyz$ according to the conditions of the pumping lemma. By condition 3, xy has only 0s, hence the string $xyyz$ is 0^l10^p for some $l > p$. But then 0^l10^p isn't equal to 0^k1u0^k for any u and k, because the left-hand part of the string requires $k = l$ and the right-hand part requires $k \leq p$. Both together are impossible, because $l > p$. That contradicts the pumping lemma and we conclude that B isn't regular.

1.69 **a.** Let s be a string in U whose length is shortest among all strings in U. Assume (for contradiction) that $|s| \geq \max(k_1, k_2)$. One or both of the DFAs accept s because $s \in U$. Say it is M_1 that accepts s. Consider the states $q_0, q_1, \ldots, q_l$ that M_1 enters while reading s, where $l = |s|$. We have $l \geq k_1$, so $q_0, q_1, \ldots, q_l$ must repeat some state. Remove the portion of s between the repeated state to yield a shorter string that M_1 accepts. That string is in U, a contradiction. Thus $|s| < \max(k_1, k_2)$.

b. Let s be a string in $\overline{U}$ whose length is shortest among all strings in $\overline{U}$. Assume (for contradiction) that $|s| \geq k_1k_2$. Both of the DFAs reject s because $s \in \overline{U}$. Consider the states $q_0, q_1, \ldots, q_l$ and $r_0, r_1, \ldots, r_l$ that M_1 and M_2 enter respectively while reading s, where $l = |s|$. We have $l \geq k_1k_2$, so in the sequence of ordered pairs $(q_0, r_0), (q_1, r_1), \ldots, (q_l, r_l)$, some pair must repeat. Remove the portion of s between the repeated pair to yield a shorter string that both M_1 and M_2 reject. That shorter string is in $\overline{U}$, a contradiction. Thus $|s| < k_1k_2$.

1.70 A PDA P that recognizes $\overline{C}$ operates by nondeterministically choosing one of three cases. In the first case, P scans its input and accepts if it doesn't contain exactly two #s. In the second case, P looks for a mismatch between the first two strings that are separated by #. It does so by reading its input while pushing those symbols onto the stack until it reads #. At that point P continues reading input symbols and matching

them with symbols that are popped off the stack. If a mismatch occurs, or if the stack empties before P reads the next #, or if P reads the next # before the stack empties, then it accepts. In the third case, P looks for a mismatch between the last two strings that are separated by #. It does so by reading its input until it reads # and the it continues reading input symbols while pushing those symbols onto the stack until it reads a second #. At that point P continues reading input symbols and matching them with symbols that are popped off the stack. If a mismatch occurs, or if the stack empties before P reaches the end of the input or if P reaches the end of the input before the stack empties, then it accepts.

Alternatively, here is a CFG that generates $\overline{C}$.

$$
\begin{aligned}
A &\to YDY\#Y \mid Y\#YDY \mid Y \mid Y\#Y \mid Y\#Y\#Y\#Z \\
D &\to XDX \mid 0E1 \mid 1E0 \\
E &\to XEX \mid \# \\
X &\to 0 \mid 1 \\
Y &\to XY \mid \varepsilon \\
Z &\to Y\#Z \mid \varepsilon
\end{aligned}
$$

1.71 **a.** Observe that $B = 1\Sigma^*1\Sigma^*$ and thus is clearly regular.

b. We show C is nonregular using the pumping lemma. Assume C is regular and let p be its pumping length. Let $s = 1^p01^p$. The pumping lemma says that $s = xyz$ satisfying the three conditions. Condition three says that y appears among the left-hand 1s. We pump down to obtain the string xz which is not a member of C. Therefore C doesn't satisfy the pumping lemma and hence isn't regular.

1.72 **a.** Let $B = \{0110\}$. Then $CUT(B) = \{0110, 1010, 0011, 1100, 1001\}$ and $CUT(CUT(B)) = \{0110, 1010, 0011, 1100, 1001, 0101\}$.

b. Let A be a regular language that is recognized by DFA M. We construct a NFA N that recognizes $CUT(A)$. This construction is similar to the construction in the solution to Problem 1.61. Here, N begins by nondeterministically guessing two states q_1 and q_2 in M. Then N simulates M on its input beginning at state q_1. Whenever the simulation reaches q_2, nondeterministically N may switch to simulating M at its start state q_0 and if it reaches state q_1, it again nondeterministically may switch to state q_2. At the end of the input, if N's simulation has made both switches and is now in one of M's accept states, it has completed reading an input yxz where M accepts xyz, and so it accepts.

1.73 **a.** The idea here is to show that a DFA with fewer than 2^k states must fail to give the right answer on at least one string. Let $A = (Q, \Sigma, \delta, q_0, F)$ be a DFA with m states. Fix the value of k and let $W = \{w \mid w \in \Sigma^k\}$ be the set of strings of length k.

For each of the 2^k strings $w \in W$, let q_w be the state that A enters after it starts in state q_0 and then reads w. If $m < 2^k$ then two different strings s and t must exist in W where A enters the same state, i.e., $q_s = q_t$. The string ss is in WW so A must enter an accepting state after reading ss. Similarly, $st \notin WW$ so A must enter a rejecting string after reading st. But $q_s = q_t$, so A enters the same state after reading ss or st, a contradiction. Therefore $m \geq 2^k$.

b. We can give an NFA $N = (Q, \Sigma, \delta, c_0, F)$ with $4k + 4$ states that recognizes $\overline{WW}$. The NFA N branches into two parts. One part accepts if the inputs length isn't $2k$. The other part accepts if the input contains two unequal symbols that are k separated.

Formally, let $Q = \{c_0, \ldots, c_{2k+1}, r, y_1, \ldots, y_k, z_1, \ldots, z_k\}$ and let $F = \{c_0, \ldots, c_{2k-1}, c_{2k+1}, y_k, z_k\}$.

For $0 \leq i \leq 2k$ and $a \in \Sigma$, set $\delta(c_i, a) = \{c_{i+1}\}$ and $\delta(c_{k+1}, a) = \{c_{k+1}\}$.
Additionally, $\delta(c_0, \varepsilon) = \{r\}$, $\delta(r, 0) = \{r, y_1\}$ and $\delta(r, 1) = \{r, z_1\}$.
Finally, for $0 \leq i < k - 1$, set $\delta(y_i, a) = \{y_{i+1}\}$ and $\delta(z_i, a) = \{z_{i+1}\}$,
$\delta(y_{k-1}, 1) = \{y_k\}$, $\delta(y_k, a) = \{y_k\}$, and
$\delta(z_{k-1}, 1) = \{z_k\}$, $\delta(z_k, a) = \{z_k\}$.

Chapter 2

2.1 Here are the derivations (but not the parse trees).

a. $E \Rightarrow T \Rightarrow F \Rightarrow \mathtt{a}$

b. $E \Rightarrow E\mathtt{+}T \Rightarrow T\mathtt{+}T \Rightarrow F\mathtt{+}T \Rightarrow F\mathtt{+}F \Rightarrow \mathtt{a+}F \Rightarrow \mathtt{a+a}$

c. $E \Rightarrow E\mathtt{+}T \Rightarrow E\mathtt{+}T\mathtt{+}T \Rightarrow T\mathtt{+}T\mathtt{+}T \Rightarrow F\mathtt{+}T\mathtt{+}T \Rightarrow F\mathtt{+}F\mathtt{+}T \Rightarrow F\mathtt{+}F\mathtt{+}F \Rightarrow \mathtt{a+}F\mathtt{+}F \Rightarrow \mathtt{a+a+}F \Rightarrow \mathtt{a+a+a}$

d. $E \Rightarrow T \Rightarrow F \Rightarrow \mathtt{(}E\mathtt{)} \Rightarrow \mathtt{(}T\mathtt{)} \Rightarrow \mathtt{(}F\mathtt{)} \Rightarrow \mathtt{((}E\mathtt{))} \Rightarrow \mathtt{((}T\mathtt{))} \Rightarrow \mathtt{((}F\mathtt{))} \Rightarrow \mathtt{((a))}$

2.2 **a.** The following grammar generates A:

$$\begin{aligned} S &\rightarrow RT \\ R &\rightarrow \mathtt{a}R \mid \varepsilon \\ T &\rightarrow \mathtt{b}T\mathtt{c} \mid \varepsilon \end{aligned}$$

The following grammar generates B:

$$\begin{aligned} S &\rightarrow TR \\ T &\rightarrow \mathtt{a}T\mathtt{b} \mid \varepsilon \\ R &\rightarrow \mathtt{c}R \mid \varepsilon \end{aligned}$$

Both A and B are context-free languages and $A \cap B = \{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 0\}$. We know from Example 2.36 that this language is not context free. We have found two CFLs whose intersection is not context free. Therefore the class of context-free languages is not closed under intersection.

b. First, the context-free languages are closed under the union operation. Let $G_1 = (V_1, \Sigma, R_1, S_1)$ and $G_2 = (V_2, \Sigma, R_2, S_2)$ be two arbitrary context free grammars. We construct a grammar G that recognizes their union. Formally, $G = (V, \Sigma, R, S)$ where:

i) $V = V_1 \cup V_2$

ii) $R = R_1 \cup R_2 \cup \{S \rightarrow S_1,\ S \rightarrow S_2\}$
(Here we assume that R_1 and R_2 are disjoint, otherwise we change the variable names to ensure disjointness)

Next, we show that the CFLs are not closed under complementation. Assume, for a contradiction, that the CFLs are closed under complementation. Then, if G_1 and G_2 are context free grammars, it would follow that $\overline{L(G_1)}$ and $\overline{L(G_2)}$ are context free. We previously showed that context-free languages are closed under union and so $\overline{L(G_1)} \cup \overline{L(G_1)}$ is context free. That, by our assumption, implies that $\overline{\overline{L(G_1)} \cup \overline{L(G_1)}}$ is context free. But by DeMorgan's laws, $\overline{\overline{L(G_1)} \cup \overline{L(G_1)}} = L(G_1) \cap L(G_2)$. However, if G_1 and

G_2 are chosen as in part (a), $\overline{\overline{L(G_1)} \cup \overline{L(G_1)}}$ isn't context free. This contradiction shows that the context-free languages are not closed under complementation.

2.4 **b.** $S \to \mathtt{0}R\mathtt{0} \mid \mathtt{1}R\mathtt{1} \mid \varepsilon$
$R \to \mathtt{0}R \mid \mathtt{1}R \mid \varepsilon$
c. $S \to \mathtt{0} \mid \mathtt{1} \mid \mathtt{00}S \mid \mathtt{01}S \mid \mathtt{10}S \mid \mathtt{11}S$
e. $S \to \mathtt{0}S\mathtt{0} \mid \mathtt{1}S\mathtt{1} \mid \mathtt{0} \mid \mathtt{1} \mid \varepsilon$
f. $S \to S$

2.5 **a.** This language is regular, so the PDA doesn't need to use its stack. The PDA reads the input and uses its finite control to maintain a counter which counts up to 3. It keeps track of the number of 1s it has seen using this counter. The PDA enters an accept state and scans to the end of the input if it has read three 1s.
b. This language is regular. The PDA reads the input and keeps track of the first and last symbol in its finite control. If they are the same, it accepts, otherwise it rejects.
c. This language is regular. The PDA reads the input and keeps track of the length (modulo 2) using its finite control. If the length is 1 (modulo 2) it accepts, otherwise it rejects.
d. The PDA reads the input and pushes the symbols onto the stack. At some point it nondeterministically guesses where the middle is. It looks at the middle symbol. If that symbol is a 1, it rejects. If it is a 0 the PDA reads the rest of the string, and for each character read, it pops one element off of its stack. If the stack is empty when it finishes reading the input, it accepts. If the stack is empty before it reaches the end of the input, or nonempty when the input is finished, it rejects.
e. The PDA reads the input and pushes each symbol onto its stack. At some point it nondeterministically guesses when it has reached the middle. It also nondeterministically guesses whether string has odd length or even length. If it guesses even, it pushes the current symbol it's reading onto the stack. If it guesses the string has odd length, it goes to the next input symbol without changing the stack. Then it reads the rest of the input, and it compares each symbol it reads to the symbol on the top of the stack. If they are the same, it pops the stack, and continues reading. If they are different, it rejects. If the stack is empty when it finishes reading the input, it accepts. If the stack is empty before it reaches the end of the input, or nonempty when the input is finished, it rejects.
f. The PDA never enters an accept state.

2.6 **b.**

$$\begin{aligned} S &\to X\mathtt{b}X\mathtt{a}X \mid T \mid U \\ T &\to \mathtt{a}T\mathtt{b} \mid T\mathtt{b} \mid \mathtt{b} \\ U &\to \mathtt{a}U\mathtt{b} \mid \mathtt{a}U \mid \mathtt{a} \\ X &\to \mathtt{a}X \mid \mathtt{b}X \mid \varepsilon \end{aligned}$$

d.

$$\begin{aligned} S &\to M\#P\#M \mid P\#M \mid M\#P \mid P \\ P &\to \mathtt{a}P\mathtt{a} \mid \mathtt{b}P\mathtt{b} \mid \mathtt{a} \mid \mathtt{b} \mid \varepsilon \mid \# \mid \#M\# \\ M &\to \mathtt{a}M \mid \mathtt{b}M \mid \#M \mid \varepsilon \end{aligned}$$

Note that we need to allow for the case when $i = j$, that is, some x_i is a palindrome. Also, ε is in the language since it's a palindrome.

2.9 A CFG G that generates A is given as follows:

$G = (V, \Sigma, R, S)$, $V = \{S, E_{\mathtt{ab}}, E_{\mathtt{bc}}, C, A\}$, and $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$. The rules are:

$$\begin{aligned} S &\rightarrow E_{\mathtt{ab}}C \mid AE_{\mathtt{bc}} \\ E_{\mathtt{ab}} &\rightarrow \mathtt{a}E_{\mathtt{ab}}\mathtt{b} \mid \varepsilon \\ E_{\mathtt{bc}} &\rightarrow \mathtt{b}E_{\mathtt{bc}}\mathtt{c} \mid \varepsilon \\ C &\rightarrow C\mathtt{c} \mid \varepsilon \\ A &\rightarrow A\mathtt{a} \mid \varepsilon \end{aligned}$$

Initially substituting $E_{\mathtt{ab}}C$ for S generates any string with an equal number of a's and b's followed by any number of c's. Initially substituting $E_{\mathtt{bc}}$ for S generates any string with an equal number of b's and c's prepended by any number of a's.

The grammar is ambiguous. Consider the string ε. On the one hand, it can be derived by choosing $E_{\mathtt{ab}}$C with each of $E_{\mathtt{ab}}$ and C yielding ε. On the other hand, ε can be derived by choosing $AE_{\mathtt{bc}}$ with each of A and $E_{\mathtt{bc}}$ yielding ε. In general, any string $\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k$ with $i = j = k$ can be derived ambiguously in this grammar.

2.10

1. Nondeterministically branch to either Stage 2 or Stage 6.
2. Read and push a's.
3. Read b's, while popping a's.
4. If b's finish when stack is empty, skip c's on input and *accept*.
5. Skip a's on input.
6. Read and push b's.
7. Read c's, while popping b's.
8. If c's finish when stack is empty, *accept*.

2.11

1. Place \$ and E on the stack.
2. Repeat the following steps forever.
3. If the top of stack is the variable E, pop it and nondeterministically push either E+T or T into the stack.
4. If the top of stack is the variable T, pop it and nondeterministically push either T×F or F into the stack.
5. If the top of stack is the variable F, pop it and nondeterministically push either (*E*) or a into the stack.
6. If the top of stack is a terminal symbol, read the next symbol from the input and compare it to the terminal in the stack. If they match, repeat. If they do not match, reject on this branch of the nondeterminism.
7. If the top of stack is the symbol \$, enter the accept state. Doing so accepts the input if it has all been read.

2.12 Informal description of a PDA that recognizes the CFG in Exercise 2.3:

1. Place the marker symbol \$ and the start variable R on the stack.
2. Repeat the following steps forever.
3. If the top of stack is the variable R, pop it and nondeterministically push either XRX or S into the stack.
4. If the top of stack is the variable S, pop it and nondeterministically push either aTb or bTa into the stack.
5. If the top of stack is the variable T, pop it and nondeterministically push either XTX, X or ε into the stack.
6. If the top of stack is the variable X, pop it and nondeterministically push either a or b into the stack.

7. If the top of stack is a terminal symbol, read the next symbol from the input and compare it to the terminal symbol in the stack. If they match, repeat. If they do not match, reject on this branch of the nondeterminism.
8. If the top of stack is the symbol $, enter the accept state. Doing so accepts the input if it has all been read.

2.13 **a.** $L(G)$ is the language of strings of 0s and #s that either contain exactly two #s and any number of 0s, or contain exactly one # and the number of 0s on the right-hand side of the # is twice the number of 0s on the left-hand side of the #.

b. Assume $L(G)$ is regular and obtain a contradiction. Let $A = L(G) \cap \mathtt{0}^*\mathtt{\#}\mathtt{0}^*$. If $L(G)$ is regular, so is A. But we can show $A = \{\mathtt{0}^k\mathtt{\#}\mathtt{0}^{2k} \mid k \geq 0\}$ is not regular by using a standard pumping lemma argument.

2.14

$$\begin{aligned} S_0 &\to AB \mid CC \mid BA \mid BD \mid BB \mid \varepsilon \\ A &\to AB \mid CC \mid BA \mid BD \mid BB \\ B &\to CC \\ C &\to \mathtt{0} \\ D &\to AB \end{aligned}$$

2.16 Let $G_1 = (V_1, \Sigma, R_1, S_1)$ and $G_2 = (V_2, \Sigma, R_2, S_2)$ be CFGs.

Construct a new CFG $G_\cup$ where $L(G_\cup) = L(G_1) \cup L(G_2)$. Let S be a new variable that is neither in V_1 nor in V_2 and assume these sets are disjoint.

Let $G_\cup = (V_1 \cup V_2 \cup \{S\},\ \Sigma,\ R_1 \cup R_2 \cup \{r_0\},\ S)$, where r_0 is $S \to S_1 \mid S_2$.

We construct CFG $G_\circ$ that generates $L(G_1) \circ L(G_2)$ as in $G_\cup$ by changing r_0 in $G_\cup$ into $S \to S_1 S_2$.

To construct CFG G_* that generates the language $L(G_1)^*$, let S' be a new variable not in V_1 and make it the starting variable. Let r_0 be $S' \to S'S_1 \mid \varepsilon$ be a new rule in G_*.

2.17 Let A be a regular language generated by regular expression R. If R is one of the atomic regular expressions b, for $b \in \Sigma_\varepsilon$, construct the equivalent CFG $(\{S\}, \{b\}, \{S \to b\}, S)$. If R is the atomic regular expressions $\emptyset$, construct the equivalent CFG $(\{S\}, \{b\}, \{S \to S\}, S)$. If R is a regular expression composed of smaller regular expressions combined with a regular operation, use the result of Problem 2.16 to construct an equivalent CFG out of the CFGs that are equivalent to the smaller expressions.

2.18 S can generate a string of Ts. Each T can generate strings in $\{\mathtt{a}^m b^m \mid m \geq 1\}$. Here are two different leftmost derivations of ababab.

$S \Rightarrow SS \Rightarrow SSS \Rightarrow TSS \Rightarrow \mathtt{ab}SS \Rightarrow \mathtt{ab}TS \Rightarrow \mathtt{abab}S \Rightarrow \mathtt{abab}T \Rightarrow \mathtt{ababab}$.

$S \Rightarrow SS \Rightarrow TS \Rightarrow \mathtt{ab}S \Rightarrow \mathtt{ab}SS \Rightarrow \mathtt{ab}TS \Rightarrow \mathtt{abab}S \Rightarrow \mathtt{abab}T \Rightarrow \mathtt{ababab}$.

The ambiguity arises because S can generate a string of Ts in multiple ways. We can prevent this behavior by forcing S to generate each string of Ts in a single way by changing the first rule to be $S \to TS$ instead of $S \to SS$. In the modified grammar, a leftmost derivation will repeatedly expand T until it is eliminated before expanding S, and no other option for expanding variables is possible, so only one leftmost derivation is possible for each generated string.

2.19 Let A be a CFG that is recognized by PDA P and construct the following PDA P' that recognizes $RC(A)$. For simplicity, assume that P empties its stack when it accepts its

input. The new PDA P' must accept input yx when P accepts xy. Intuitively, would like P' to start by guessing the state and the stack contents that P would be in after reading x, so that P' can simulate P on y, and then if P ends up in an accept state after reading y, we can simulate P on x to check that it ends up with the initially guessed start and stack contents. However, P' has no way to store the initially guessed stack contents to check that it matches the stack after reading x. To avoid this difficulty, P' guesses these stack symbols only at the point when P would be popping them while reading y. It records these guesses by pushing a specially marked copy of each guessed symbol. When P' guesses that it has finished reading y, it will have a copy of the stack that P would have when it finishes reading x but in reverse order, with marked symbols instead of regular symbols. Then, while P' simulates P reading x, whenever P pushes a symbol, P' may guess that it is one of the symbols that would have been popped while P read y, and if so P' pops the stack and matches the symbol P would push with the marked symbol P' popped.

2.20 Let $B = \{1^m 2^n 3^n 4^m | \ n, m \geq 0\}$ which is a CFL. If $CUT(B)$ were a CFL then the language $CUT(B) \cap 2^*1^*3^*4^*$ would be a CFL because Problem 2.30 shows that the intersection of a CFL and a regular language is a CFL. But that intersection is $\{2^n 1^m 3^n 4^m | \ n, m \geq 0\}$ which is easily shown to be a non-CFL using the pumping lemma.

2.21 We show that an ambiguous CFG cannot be a DCFG. If G is ambiguous, G derives some string s with at least two different parse trees and therefore s has at least two different rightmost derivations and at least two different leftmost reductions. Compare the steps of two of these different leftmost reductions and locate the first string where these reduction differ. The preceding string must be the same in both reductions, but it must have two different handles. Hence G is not a DCFG.

2.23 **a.** Let $B = \{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k | \ i \neq j \text{ for } i, j, k \geq 0\}$ and $C = \{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k | \ j \neq k \text{ for } i, j, k \geq 0\}$. The following DPDA recognizes B. Read all a's (if any) and push them onto the stack. Read b's (if any) while popping the a's. If the b's finish while a's remain on the stack or if b's remain unread when the stack becomes empty, then scan to the end of the input and *accept*, assuming that the input is in $\mathtt{a}^*\mathtt{b}^*\mathtt{c}^*$ which was checked in parallel. Otherwise, *reject*.

A similar DPDA recognizes C. Thus B and C are DCFLs. However we have shown in the text that $A = B \cup C$ isn't a DCFL. Therefore the class of DCFLs is not closed under union.

b. We have shown in the text that the class of DCFLs is closed under complement. Thus, if the class of DCFLs were closed under intersection, the class would also be closed under union because $B \cup C = \overline{B} \cap \overline{C}$, contradicting Part **a** of this problem. Hence the class of DCFLs is not closed under intersection.

c. Define B and C as in Part **a**. Let $B' = (\mathtt{01} \cup \mathtt{0})B$ and $C' = (\mathtt{10} \cup \mathtt{1})C$. Then $D = B' \cup C'$ is a DCFL because a DPDA can determine which test to use by reading the first few symbols of the input. Let $E = (\mathtt{1} \cup \varepsilon)D$ which is $(\mathtt{101} \cup \mathtt{10} \cup \mathtt{01} \cup \mathtt{0})B \cup (\mathtt{110} \cup \mathtt{11} \cup \mathtt{10} \cup \mathtt{1})C$. To see that E isn't a DCFL, take $F = E \cap \mathtt{01}(\mathtt{abc})^*$. The intersection of a DCFL and a regular language is a DCFL so if E were a DCFL then F would be a DCFL but $F = \mathtt{01}A$ which isn't a DCFL for the same reason A isn't a DCFL.

d. Define C and F as in Part **c**. The language $H = \mathtt{1} \cup \varepsilon \cup D$ is easily seen to be a DCFL. However H^* is not a DCFL because $H^* \cap \mathtt{01}(\mathtt{abc})^* = F \cup \mathtt{1} \cup \varepsilon$ and the latter isn't a DCFL for the same reason that F isn't a DCFL.

e. Define B and C as in Part **a**. Let $K = \mathtt{0}B^{\mathcal{R}} \cup \mathtt{1}C^{\mathcal{R}}$. First, show that K is a DCFL by giving a DPDA which operates as follows. After reading the first symbol, the DPDA follows the deterministic procedure to recognize membership in $B^{\mathcal{R}}$ or in $C^{\mathcal{R}}$ depending on whether the first symbol is 0 or 1. Next, show that $K^{\mathcal{R}}$ is not a DCFL by showing how to convert a DPDA P for $K^{\mathcal{R}}$ into a DPDA P' for the language $A = B \cup C$ which the text shows isn't a DCFL. Modify P so that it recognizes the endmarked version of $K^{\mathcal{R}}$. Thus $L(P) = K^{\mathcal{R}}\dashv = B\mathtt{0}\dashv \cup C\mathtt{1}\dashv$. It is enough to design P' to recognize $A\dashv = B\dashv \cup C\dashv$ because $A\dashv$ is a DCFL iff A is a DCFL. It simulates P but also keeps additional information on the stack which says what P would do if its next input symbol were 0 or it it were 1. Then when P' reads $\dashv$, it can use this information to accept when P would have accepted if the prior input symbol had been either a 0 or a 1.

2.24 **a.** Let $E = \{w |\ w$ has an equal number of a's and b's$\}$. It is enough to show that $T \stackrel{*}{\Rightarrow} w$ iff $w \in E$. The forward direction is straightforward. If $T \stackrel{*}{\Rightarrow} w$ then $w \in E$ because every substitution adds an equal number of a's and b's. The reverse direction is the following claim.

Claim: If $w \in E$ then $T \stackrel{*}{\Rightarrow} w$.

Proof by induction on $|w|$, the length of w.

Basis, $|w| = 0$: Then $w = \varepsilon$ and the rule $T \to \varepsilon$ shows that $T \stackrel{*}{\Rightarrow} w$.

Induction step: Let $k > 0$ and assume the claim is true whenever $|w| < k$. Prove the claim is true for $|w| = k$.

Take $w = w_1 \cdots w_k \in E$ where $|w| = k$ and each w_i is a or b. Let $x = w^{\mathcal{R}}$. For $1 \leq i \leq k$, let a_i (b_i) be the number of a's (b's) that appear among the first i symbols of x and let $c_i = a_i - b_i$. In other words, c_i is the running count of the excess number of a's over b's across x. Because $w \in E$ and E is closed under reversal, we know that $c_k = 0$. Let m be the smallest i for which $c_i = 0$. Consider the case where $x_1 = \mathtt{a}$. Then $c_1 = 1$. Moreover, $c_i > 0$ from $i = 1$ to $m - 1$ and $c_{m-1} = 1$. Thus, $x_2 \cdots x_{m-1} \in E$ and $x_m = \mathtt{b}$ and then also $x_{m+1} \cdots x_k \in E$. We can write $x = \mathtt{a}x_2 \cdots x_{m-1}\mathtt{b}x_{m+1} \cdots x_k$ and so $w = (x_{m-1} \cdots x_k)^{\mathcal{R}}\mathtt{b}(x_2 \cdots x_{m-1})^{\mathcal{R}}\mathtt{a}$. The induction assumption implies that $T \stackrel{*}{\Rightarrow} (x_{m-1} \cdots x_k)^{\mathcal{R}}$ and $T \stackrel{*}{\Rightarrow} (x_2 \cdots x_{m-1})^{\mathcal{R}}$. Therefore the rule $T \to T\mathtt{b}T\mathtt{a}$ shows that $T \stackrel{*}{\Rightarrow} w$. A similar argument works for the case where $x_1 = \mathtt{b}$.

b. Omitted

c. The DPDA reads the input and performs the following actions for each symbol read. If the stack is empty or if the input symbol is the same as the top stack symbol, the DPDA pushes the input symbol. Otherwise, if the top stack symbol differs from the input symbol, it pops the stack. When it reads $\dashv$, it accepts if the stack is empty, and otherwise it doesn't accept.

2.26 Following the hint, the modified P would accept the strings of the form $\{\mathtt{a}^m\mathtt{b}^m\mathtt{c}^m |\ m \geq 1\}$. But that is impossible because this language is not a CFL.

2.27 Assume that DPDA P recognizes B and construct a modified DPDA P' which simulates P while reading a's and b's. If P enters an accept state, P' checks whether the next input symbol is a c, and if so it simulates P on bc (pretending it has read an extra b) and then continues to simulate P on the rest of the input. It accepts only when P enters an accept state after reading c's. Then P' recognizes the non-CFL $\{\mathtt{a}^m\mathtt{b}^m\mathtt{c}^m |\ m \geq 1\}$, an impossibility.

2.28 Assume to the contrary that DPDA P recognizes C. For a state q and a stack symbol x, call (q, x) a *minimal pair* if when P is started q with x on the top of its stack, P never pops its stack below x, no matter what input string P reads from that point on. In that case, the contents of P's stack at that point cannot affect its subsequent behavior, so P's subsequent behavior can depend only on q and x. Additionally, call (q, ε) a minimal pair. It corresponds to starting P in state q with an empty stack.

Claim: Let y be any input to P. Then y can be extended to $z = ys$ for some $s \in \{0,1\}^*$ where P on z enters a minimal pair.

To prove this claim, observe that if P on input y enters a minimal pair then we are done immediately, and if P on input y enters a pair (q, x) which isn't minimal then some input s_1 exists on which P pops its stack below x. If P on input ys_1 then enters a minimal pair, we are again done because we can let $z = ys_1$. If that pair still isn't minimal then some input s_2 exists which take P to an even lower stack level. If P on ys_1s_2 enters a minimal pair, we are done because we can let $z = ys_1s_2$. This procedure must terminate with a minimal pair, because P's stack shrinks at every step and will eventually become empty. Thus, the claim is proved.

Let $k = 1 + |Q| \times (|\Gamma| + 1)$ be any value which is greater than the total number of minimal pairs. For every $i \leq k$, let $y_i = 10^i1$. The strings y_i are all distinct, and no string y_i is a prefix of some other string y_j. The claim shows that we can extend these to $z_1, \ldots, z_k \in \{0,1\}^*$ where each z_i takes P to a minimal pair. Because k exceeds the number of minimal pairs, two of these strings, z_i and z_j, lead to the same minimal pair. Observe that P accepts both $z_iz_i^{\mathcal{R}}$ and $z_jz_j^{\mathcal{R}}$ because these strings are members of C. But because z_i and z_j lead to the same minimal pair, P's will behave identically if we append the same string to either of them. Thus P accepts input $z_iz_j^{\mathcal{R}}$ because it accepts input $z_jz_j^{\mathcal{R}}$. But $z_iz_j^{\mathcal{R}} \notin C$, a contradiction.

2.31 The grammar generates all strings not of the form $\mathtt{a}^k\mathtt{b}^k$ for $k \geq 0$. Thus the complement of the language generated is $\overline{L(G)} = \{\mathtt{a}^k\mathtt{b}^k \mid k \geq 0\}$. The following grammar generates $\overline{L(G)}$: $\{\{S\}, \{\mathtt{a}, \mathtt{b}\}, \{S \to \mathtt{a}S\mathtt{b} \mid \varepsilon\}, S\}$.

2.32 Let P be a PDA that recognizes A. We construct a PDA P' that recognizes A/B as follows. It operates by reading symbols and simulating P. Whenever P branches nondeterministically, so does P'. In addition, at every point P' nondeterministically guesses that it has reached the end of the input and refrains from reading any more input symbols. Instead, it guesses additional symbols as if they were input and continues to simulate P on this guessed input, while at the same time using its finite control to simulate a DFA for B on the guessed input string. If P' simultaneously reaches accept states in both of these simulations, P' enters an accept state.

2.33 The following CFG G generates the language C of all strings with twice as many a's as b's. $S \to \mathtt{b}S\mathtt{aa} \mid \mathtt{aa}S\mathtt{b} \mid \mathtt{a}S\mathtt{b}S\mathtt{a} \mid SS \mid \varepsilon$.

Clearly G generates only strings in C. We prove inductively that every string in C is in $L(G)$. Let s be a string in C of length k.

Basis If $k = 0$ then $S = \varepsilon \in L(G)$.

Induction step If $k > 0$, assume all strings in C of length less than k are generated by G and show s is generated by G.

If $s = s_1 \ldots s_k$ then for each i from 0 to k let c_i to be the number of a's minus twice the number of b's in $s_1 \ldots s_i$. We have $c_0 = 0$ and $c_k = 0$ because $s \in L(G)$. Next we consider two cases.

i) If $c_i = 0$ for some i besides 0 and k, we divide into two substrings $s = tu$ where t is the first i symbols and u is the rest. Both t and u are members of C and hence are in $L(G)$ by the induction hypothesis. Therefore the rule $S \to SS$ show that $s \in L(G)$.

ii) If $c_i \neq 0$ for all i besides 0 and k, we consider three subcases.

   i. If s begins with `b`, then all c_i from $i = 1$ to $k-1$ are negative, because $c_1 = -2$ and jumping from a negative c_i to a positive c_{i+1} isn't possible because the c values can increase only by 1 at each step. Hence s ends with `aa`, because no other ending would give $c_k = 0$. Therefore $s =$ `btaa` where $t \in C$. By the induction hypothesis, $t \in L(G)$ and so the rule $S \to$ `b`S`aa` shows that $s \in L(G)$.

   ii. If s begins with `a` and all c_i are non-negative, then $s_2 =$ `a` and $s_k =$ `b`. Therefore $s =$ `aatb` where $t \in C$. By the induction hypothesis, $t \in L(G)$ and so the rule $S \to$ `aa`S`b` shows that $s \in L(G)$.

   iii. If s begins with `a` and some c_i is negative, then select the lowest i for which a negative c_i occurs. Then $c_{i-1} = +1$ and $c_i = -1$, because the c values can decrease only by 2 at each step. Hence $s_i =$ `b`. Furthermore s ends with `a`, because no other ending would give $c_k = 0$. If we let t be $s_2 \cdots s_{i-1}$ and u be $s_{i+1} \cdots s_{k-1}$ then $s =$ `a`t`b`u`a` where both t and u are members of C and hence are in $L(G)$ by the induction hypothesis. Therefore the rule $S \to$ `a`S`b`S`a` shows that $s \in L(G)$.

2.34 We construct a PDA P that recognizes C. First it nondeterministically branches to check either of two cases: that x and y differ in length or that they have the same length but differ in some position. Handling the first case is straightforward. To handle the second case, it operates by guessing corresponding positions on which the strings x and y differ, as follows. It reads the input at the same time as it pushes some symbols, say `1`s, onto the stack. At some point it nondeterministically guesses a position in x and it records the symbol it is currently reading there in its finite memory and skips to the `#`. Then it pops the stack while reading symbols from the input until the stack is empty and checks that the symbol it is now currently reading is different from the symbol it had recorded. If so, it accepts.

Here is a more detailed description of P's algorithm. If something goes wrong, for example, popping when the stack is empty, or getting to the end of the input prematurely, P rejects on that branch of the computation.

1. Nondeterministically jump to either 2 or 4.
2. Read and push these symbols until read `#`. *Reject* if `#` never found.
3. Read and pop symbols until the end of the tape. *Reject* if another `#` is read or if the stack empties at the same time the end of the input is reached. Otherwise *accept*.
4. Read next input symbol and push 1 onto stack.
5. Nondeterministically jump to either 4 or 6.
6. Record the current input symbol a in the finite control.
7. Read input symbols until `#` is read.
8. Read the next symbol and pop the stack.
9. If stack is empty, go to 10, otherwise go to 8.
10. *Accept* if the current input symbol isn't a. Otherwise *reject*.

2.35 We construct a PDA P recognizing D. This PDA guesses corresponding places on which x and y differ. Checking that the places correspond is tricky. Doing so relies on the

observation that the two corresponding places are $n/2$ symbols apart, where n is the length of the entire input. Hence, by ensuring that the number of symbols between the guessed places is equal to the number other symbols, the PDA can check that the guessed places do indeed correspond, Here we give a more detailed description of the PDA algorithm. If something goes wrong, for example, popping when the stack is empty, or getting to the end of the input prematurely, P rejects on that branch of the computation.

1. Read next input symbol and push 1 onto the stack.
2. Nondeterministically jump to either 1 or 3.
3. Record the current input symbol a in the finite control.
4. Read next input symbol and pop the stack. Repeat until stack is empty.
5. Read next input symbol and push 1 onto the stack.
6. Nondeterministically jump to either 5 or 7.
7. *Reject* if current input symbol equals a.
8. Read next input symbol and pop the stack. Repeat until stack is empty.
9. *Accept* if input is empty.

Alternatively we can give a CFG for this language as follows.

$$\begin{aligned} S &\to AB \mid BA \\ A &\to XAX \mid \mathtt{0} \\ B &\to XBX \mid \mathtt{1} \\ X &\to \mathtt{0} \mid \mathtt{1} \end{aligned}$$

2.38 Consider a derivation of w. Each application of a rule of the form $A \to BC$ increases the length of the string by 1. So we have $n-1$ steps here. Besides that, we need exactly n applications of terminal rules $A \to a$ to convert the variables into terminals. Therefore, exactly $2n-1$ steps are required.

2.39 **a.** To see that G is ambiguous, note that the string

`if condition then if condition then a:=1 else a:=1`

has two different leftmost derivations (and hence parse trees):

1. ⟨STMT⟩
→ ⟨IF-THEN⟩
→ `if condition then` ⟨STMT⟩
→ `if condition then` ⟨IF-THEN-ELSE⟩
→ `if condition then if condition then` ⟨STMT⟩ `else` ⟨STMT⟩
→ `if condition then if condition then a:=1 else` ⟨STMT⟩
→ `if condition then if condition then a:=1 else a:=1`

2. ⟨STMT⟩
→ ⟨IF-THEN-ELSE⟩
→ `if condition then` ⟨STMT⟩ `else` ⟨STMT⟩
→ `if condition then` ⟨IF-THEN⟩ `else` ⟨STMT⟩
→ `if condition then if condition then` ⟨STMT⟩ `else` ⟨STMT⟩
→ `if condition then if condition then a:=1 else` ⟨STMT⟩
→ `if condition then if condition then a:=1 else a:=1`

b. The ambiguity in part a) arises because the grammar allows matching an `else` both to the nearest and to the farthest `then`. To avoid this ambiguity we construct a new grammar that only permits the nearest match, by disabling derivations which introduce an ⟨IF-THEN⟩

before an `else`. This grammar has two new variables: ⟨E-STMT⟩ and ⟨E-IF-THEN-ELSE⟩, which work just like their non-⟨E⟩ counterparts except that they cannot generate the dangling ⟨IF-THEN⟩. The rules of the new grammar are the same as for the old grammar except we remove the ⟨IF-THEN-ELSE⟩ rule and add the following new rules:

$$\begin{aligned} \langle \text{E-STMT} \rangle &\rightarrow \langle \text{ASSIGN} \rangle | \langle \text{E-IF-THEN-ELSE} \rangle \\ \langle \text{E-IF-THEN-ELSE} \rangle &\rightarrow \texttt{if condition then } \langle \text{E-STMT} \rangle \texttt{ else } \langle \text{E-STMT} \rangle \\ \langle \text{IF-THEN-ELSE} \rangle &\rightarrow \texttt{if condition then } \langle \text{E-STMT} \rangle \texttt{ else } \langle \text{STMT} \rangle \end{aligned}$$

2.40 I found these problems to be surprisingly hard.

a. Define an *a-string* to be a string where every prefix has at least as many `a`'s as `b`'s. The following grammar G generates all `a`-strings unambiguously.

$$\begin{aligned} M &\rightarrow A\texttt{a}M \mid A \\ A &\rightarrow \texttt{a}A\texttt{b}A \mid \varepsilon \end{aligned}$$

Here is a proof sketch. First we claim that A generates all balanced `a`-strings unambiguously. Let $w = w_1 \cdots w_n$ be any string. Let c_i be the number of `a`'s minus the number of `b`'s in positions 1 through i in w. The *mate* of `a` at position i in w is the `b` at the lowest position $j > i$ where $c_j < c_i$. It is easy to show inductively that for any balanced `a`-string w and any parse tree for w generated from A, the rule $A \rightarrow \texttt{a}A\texttt{b}A$ generates the mated pairs at the same time, hence it divides the string in a unique way and consequently grammar is unambiguous.

Next we claim that M generates all `a`-strings that have an excess of `a`'s. Say that `a` is *unmated* if it has no mate. We can show inductively that for any `a`-string, the M rule $M \rightarrow A\texttt{a}M$ generates the unmated `a`'s and the A rules generates the mated pairs. The generation can be done in only one way so the grammar is unambiguous.

b.

$$\begin{aligned} E &\rightarrow \texttt{a}A\texttt{b}E \mid \texttt{b}B\texttt{a}E \mid \varepsilon \\ A &\rightarrow \texttt{a}A\texttt{b}A \mid \varepsilon \\ B &\rightarrow \texttt{b}B\texttt{a}B \mid \varepsilon \end{aligned}$$

Proof omitted.

c.

$$\begin{aligned} E &\rightarrow \texttt{a}A\texttt{b}E \mid \texttt{b}B\texttt{a}E \mid F \\ A &\rightarrow \texttt{a}A\texttt{b}A \mid \varepsilon \\ B &\rightarrow \texttt{b}B\texttt{a}B \mid \varepsilon \\ F &\rightarrow \texttt{a}AF \mid \varepsilon \end{aligned}$$

Proof omitted.

2.41 This proof is similar to the proof of the pumping lemma for CFLs. Let $G = (V, \Sigma, S, R)$ be a CFG generating A and let b be the length of the longest right-hand side of a rule in G. If a variable A in G generates a string s with a parse tree of height at most h, then $|s| \leq b^h$.

Let $p = b^{|V|+2}$ where $|V|$ is the number of variables in G. Let $s_1 = \texttt{a}^{2p!}\texttt{b}^{p!}\texttt{c}^{p!}$. Consider a parse tree for s_1 that has the fewest possible nodes. Take a symbol `c` in s_1 which has the longest path to S in the parse tree among all `c`'s. That path must be longer

than $|V| + 1$ so some variable B must repeat on this path. Chose a repetition that occurs among the lowest (nearest to the leaves) $|V| + 1$ variables. The upper B can have at most $b^{|V|+2} \leq p$ c's in the subtree below it and therefore it can have at most that number of b's or else the tree surgery technique would yield a derived string with unequal numbers of b's and c's. Hence the subtree below the upper B doesn't produce any a's. Thus the upper and lower B's yield a division of s_1 into five parts $uvxyz$ where v contains only b's and y contains only c's. Any other division would allow us to pump the result to obtain a string outside of A. Let $q = |v| = |y|$. Because $q \leq p$ we know that q divides $p!$. So we can pump this string up and get a parse tree for $\mathtt{a}^{2p!}\mathtt{b}^{2p!}\mathtt{c}^{2p!}$.

Next let $s_2 = \mathtt{a}^{p!}\mathtt{b}^{p!}\mathtt{c}^{2p!}$ and carry out the same procedure to get a different parse tree for $\mathtt{a}^{2p!}\mathtt{b}^{2p!}\mathtt{c}^{2p!}$. Thus this string has two different parse trees and therefore G is ambiguous.

2.42 **a.** Assume A is context-free. Let p be the pumping length given by the pumping lemma. We show that $s = \mathtt{0}^p\mathtt{1}^p\mathtt{0}^p\mathtt{1}^p$ cannot be pumped. Let $s = uvxyz$.
If either v or y contain more than one type of alphabet symbol, uv^2xy^2z does not contain the symbols in the correct order and cannot be a member of A. If both v and y contain (at most) one type of alphabet symbol, uv^2xy^2z contains runs of 0's and 1's of unequal length and cannot be a member of A. Because s cannot be pumped without violating the pumping lemma conditions, A is not context free.

d. Assume A is context-free. Let p be the pumping length from the pumping lemma. Let $s = \mathtt{a}^p\mathtt{b}^p\#\mathtt{a}^p\mathtt{b}^p$. We show that $s = uvxyz$ cannot be pumped. Use the same reasoning as in part (c).

2.43 Assume B is context-free and get its pumping length p from the pumping lemma. Let $s = \mathtt{0}^p\mathtt{1}^{2p}\mathtt{0}^p$. Because $s \in B$, it can be split $s = uvxyz$ satisfying the conditions of the lemma. We consider several cases.

i) If both v and y contain only 0's (or only 1's), then uv^2xy^2z has unequal numbers of 0s and 1s and hence won't be in B.

ii) If v contains only 0s and y contains only 1s, or vice versa, then uv^2xy^2z isn't a palindrome and hence won't be in B.

iii) If both v and y contain both 0s and 1s, condition 3 is violated so this case cannot occur.

iv) If one of v and y contain both 0s and 1s, then uv^2xy^2z isn't a palindrome and hence won't be in B.

Hence s cannot be pumped and contradiction is reached. Therefore B isn't context-free.

2.44 Assume C is context-free and get its pumping length p from the pumping lemma. Let $s = \mathtt{1}^p\mathtt{3}^p\mathtt{2}^p\mathtt{4}^p$. Because $s \in C$, it can be split $s = uvxyz$ satisfying the conditions of the lemma. By condition 3, vxy cannot contain both 1s and 2s, and cannot contain both 3s and 4s. Hence uv^2xy^2z doesn't have equal number of 1s and 2s or of 3s and 4s, and therefore won't be a member of C, so s cannot be pumped and contradiction is reached. Therefore C isn't context-free.

2.45 Assume to the contrary that F is a CFL. Let p be the pumping length given by the pumping lemma. Let $s = \mathtt{a}^{2p^2}\mathtt{b}^{2p}$ be a string in F. The pumping lemma says that we can divide $s = uvxyz$ satisfying the three conditions. We consider several cases. Recall that condition three says that $|vxy| \leq p$.

i) If either v or y contain two types of symbols, uv^2xy^2z contains some b's before a's and is not in F.

ii) If both v and y contain only a's, uv^2xy^2z has the form $\mathtt{a}^{2p^2+l}\mathtt{b}^{2p}$ where $l \leq p < 2p$. But $2p^2 + l$ isn't a multiple of $2p$ if $l < 2p$ so $uv^2xy^2z \notin F$.

iii) If both v and y contain only b's, uv^mxy^mz for $m = 2p^2$ has the form $\mathtt{a}^i\mathtt{b}^j$ where $i < j$ and so it cannot be in F.

iv) If v contains only a's and y contains only b's, let $t = uv^mxy^mz$. String t cannot be member of F for any sufficiently large value of m. Write s as $\mathtt{a}^{g+|v|}\mathtt{b}^{h+|y|}$. Then t is $\mathtt{a}^{g+m|v|}\mathtt{b}^{h+m|y|}$. If $t \in F$ then $g + m|v| = k(h + m|y|)$ for some integer k. In other words

$$k = \frac{g + m|v|}{h + m|y|}.$$

If m is sufficiently large, g is a tiny fraction of $m|v|$ and h is a tiny fraction of $m|y|$ so we have

$$k = \frac{m|v|}{m|y|} = \frac{|v|}{|y|}$$

because k is an integer. Moreover $k < p$. Rewriting the first displayed equation we have $g - hk = m(k|y| - |v|)$ which must be 0, due to the second displayed equation. But we have chosen s so that $g - hk \neq 0$ for $k < p$, so t cannot be member of F.

Thus none of the cases can occur, so the string s cannot be pumped, a contradiction.

2.46 $L(G)$ is the language of strings of 0s and #s that either contain exactly two #s and any number of 0s, or contain exactly one # and the number of 0s to the right of the # is twice the number of 0s to the left. First we show that three is not a pumping length for this language. The string 0#00 has length at least three, and it is in $L(G)$. It cannot be pumped using $p = 3$, because the only way to divide it into $uvxyz$ satisfying the first two conditions of the pumping lemma is $u = z = \varepsilon$, $v = 0$, $x = \#$, and $y = 00$, but that division fails to satisfy the third condition.

Next, we show that 4 is a pumping length for $L(G)$. If $w \in L(G)$ has length at least 4 and if it contains two #s, then it contains at least one 0. Therefore, by cutting w into $uvxyz$ where either v or y is the string 0, we obtain a way to pump w. If $w \in L(G)$ has length at least 4 and if it contains a single #, then it must be of the form $0^k\#0^{2k}$ for some $k \geq 1$. Hence, by assigning $u = 0^{k-1}$, $v = 0$, $x = \#$, $y = 00$, and $z = 0^{2k-2}$, we satisfy all three conditions of the lemma.

2.47 Assume G generates a string w using a derivation with at least 2^b steps. Let n be the length of w. By the results of Problem 2.38, $n \geq \frac{2^b+1}{2} > 2^{b-1}$.

Consider a parse tree of w. The right-hand side of each rule contains at most two variables, so each node of the parse tree has at most two children. Additionally, the length of w is at least 2^b, so the parse tree of w must have height at least $b+1$ to generate a string of length at least 2^b. Hence, the tree contains a path with at least $b + 1$ variables, and therefore some variable is repeated on that path. Using a surgery on trees argument identical to the one used in the proof of the CFL pumping lemma, we can now divide w into pieces $uvxyz$ where $uv^ixy^iz \in G$ for all $i \geq 0$. Therefore, $L(G)$ is infinite.

2.48 Let $F = \{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k\mathtt{d}^l \mid i, j, k, l \geq 0 \text{ and if } i = 1 \text{ then } j = k = l\}$. F is not context free because $F \cap \mathtt{ab}^*\mathtt{c}^*\mathtt{d}^* = \{\mathtt{ab}^n\mathtt{c}^n\mathtt{d}^n \mid n \geq 0\}$, which is not a CFL by the pumping lemma, and because the intersection of a CFL and a regular language is a CFL. However, F does satisfy the pumping lemma with $p = 2$. ($p = 1$ works, too, with a bit more effort.)

If $s \in F$ has length 2 or more then:

i) If $s \in \mathtt{b}^*\mathtt{c}^*\mathtt{d}^*$ write s as $s = rgt$ for $g \in \{\mathtt{b}, \mathtt{c}, \mathtt{d}\}$ and divide $s = uvxyz$ where $u = r$, $v = g$, $x = y = \varepsilon$, $z = t$.

ii) If $s \in \mathtt{ab}^*\mathtt{c}^*\mathtt{d}^*$ write s as $s = \mathtt{a}t$ and divide $s = uvxyz$ where $u = \varepsilon$, $v = \mathtt{a}$, $x = y = \varepsilon$, $z = t$.

iii) If $s \in \mathtt{aaa}^*\mathtt{b}^*\mathtt{c}^*\mathtt{d}^*$ write s as $s = \mathtt{aa}t$ and divide $s = uvxyz$ where $u = \varepsilon$, $v = \mathtt{aa}$, $x = y = \varepsilon$, $z = t$.

In each of the three cases, we may easily check that the division of s satisfies the conditions of the pumping lemma.

2.49 Let $G = (\Sigma, V, T, P, S)$ be a context free grammar for A. Define r to be the length of the longest string of symbols occurring on the right hand side of any production in G. We set the pumping length k to be $r^{2|V|+1}$. Let s be any string in A of length at least k and let T be a derivation tree for A with the fewest number of nodes. Observe that since $s \geq k$, the depth of T must be at least $2|V| + 1$. Thus, some path p in T has length at least $2|V| + 1$. By the pigeonhole principle, some variable V' appears at least three times in p. Label the last three occurrences of V' in p as V_1, V_2, and V_3. Moreover, define the strings generated by V_1, V_2, and V_3 in T as s_1, s_2, and s_3 respectively. Now, each of these strings is nested in the previous because they are all being generated on the same path. Suppose then that $s_1 = l_1 s_2 r_1$ and $s_2 = l_2 s_3 r_2$. We now have three cases to consider:

i) $|l_1 l_2| \geq 1$ and $|r_1 r_2| \geq 1$: In this case, there is nothing else to prove since we can simply pump in the usual way.

ii) $|l_1 l_2| = 0$: Since we defined T to be a minimal tree, neither r_1 nor r_2 can be the empty string. So, we can now pump r_1 and r_2.

iii) $|r_1 r_2| = 0$: This case is handled like the previous one.

2.51 Let A and B be defined as in the solution to Problem 2.50. Let D be the shuffle of A and B. Then let $E = D \cap ((\mathtt{0} \cup \mathtt{1})(\mathtt{a} \cup \mathtt{b}))^*$ The language E is identical to the language C in Problem 2.50 and was shown there to be non-context free. The intersection of a CFL and a regular language is a CFL, so D must not be context free, and therefore the class of CFLs isn't closed under the shuffle operation.

2.52 The language C is context free, therefore the pumping lemma holds. Let p be the pumping length and $s \in C$ be a string longer then p, so it can be split in $uvxyz$ such that $uv^i xy^i z \in C$ for all $i > 0$, where $|vy| > 0$. All prefixes are in C so all $uv^i \in C$ and thus the regular language $u(v)^* \subseteq C$. If $v \neq \varepsilon$, that language is infinite and we're done. If $v = \varepsilon$, then $y \neq \varepsilon$, and the infinite regular language $uvx(y)^* \subseteq C$.

2.53 **a.** Let $B = \{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k | \ i \neq j \text{ or } i \leq k\}$ which is a CFL. Let $s = \mathtt{a}^i\mathtt{b}^j\mathtt{c}^k \in B$. Assume $k \geq 2$. Let $s' = \mathtt{a}^i\mathtt{b}^j\mathtt{c}^{k-1}$ which is a prefix of s. Show that $s' \in B$ if $i \neq j$ or $i \neq k$.
1. If $i \neq j$ then $s' \in B$.
2. If $i < k$ then $i \leq (k-1)$ so $s' \in B$.
3. If $i > k$ then $i \neq j$ (because $s \in B$) so $s' \in B$.
If $i = j = k$ then s has no prefix in B because removing some of the c's would yield a string that fails both conditions of membership in B. Thus, $(\mathit{NOPREFIX}(B) \cap \mathtt{a}^*\mathtt{b}^*\mathtt{ccc}^*) \cup \mathtt{abc} \cup \varepsilon = \{\mathtt{a}^i\mathtt{b}^i\mathtt{c}^i | \ i \geq 0\}$ which isn't a CFL. Therefore, $\mathit{NOPREFIX}(B)$ cannot be a CFL.

b. Let $C = \{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k | \ i \neq j \text{ or } i \geq k\}$ which is a CFL. Let $s = \mathtt{a}^i\mathtt{b}^j\mathtt{c}^k \in C$. If $i \neq j$ or if $i > k$ then $s\mathtt{c} \in C$. But if $i = j = k$ then s has no extension in C. Therefore, $\mathit{NOEXTEND}(C) = \{\mathtt{a}^i\mathtt{b}^i\mathtt{c}^i | \ i \geq 0\}$ which isn't a CFL.

2.54 Assume Y is a CFL and let p be the pumping length given by the pumping lemma. Let $s = 1^{p+1}\#1^{p+2}\#\cdots\#1^{5p}$. String s is in Y but we show it cannot be pumped. Let $s = uvxyz$ satisfying the three conditions of the lemma. Consider several cases.

i) If either v or y contain #, the string uv^3xy^3z has two consecutive t_i's which are equal to each other. Hence that string is not a member of Y.

ii) If both v and y contain only 1s, these strings must either lie in the same run of 1s or in consecutive runs of 1s within s, by virtue of condition 3. If v lies within the runs from 1^{p+1} to 1^{3p} then uv^2xy^2z adds at most p 1s to that run so that it will contain the same number of 1s in a higher run. Therefore the resulting string will not be a member of Y. If v lies within the runs from 1^{3p+1} to 1^{5p} then uv^0xy^0z subtracts at most p 1s to that run so that it will contain the same number of 1s in a lower run. Therefore the resulting string will not be a member of Y.

The string s isn't pumpable and therefore doesn't satisfy the conditions of the pumping lemma, so a contradiction has occurred. Hence Y is not a CFL.

2.55 **a.** First note that a PDA can use its stack to simulate an unbounded integer counter. Next suppose $A \subseteq \{0,1\}^*$ is recognized by a DFA D. Clearly, a binary string x is in *SCRAMBLE*(A) iff x has the same length and same number of 1s as some w that is accepted by D. Thus, a PDA M for *SCRAMBLE*(A) operates on input x by nondeterministically simulating D on every possible $|x|$-long input w using its stack as a counter to keep track of the difference between the number of 1s in w and the number of 1s in x. A branch of M's computation accepts iff it reaches the end of x with the stack recording a difference of 0 and the simulation of D on an accepting state.

b. If $|\Sigma| > 2$, then the claim in part (a) is false. For example, consider the regular language $A = (\mathtt{abc})^*$ over the ternary alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$.

$$\textit{SCRAMBLE}(A) = \{x \in \Sigma^* \mid x \text{ has the same number of a's, b's and c's.}\}$$

This language is not context-free.

2.56 Let M_A be a DFA that recognizes A, and M_B be a DFA that recognizes B. We construct a PDA recognizing $A \diamond B$. This PDA simulates M_A on the first part of the string pushing every symbol it reads on the stack until it guesses that it has reached the middle of the input. After that it simulates M_B on the remaining part of the string popping the stack for every symbol it reads. If the stack is empty at the end of the input and both M_A and M_B accepted, the PDA accepts. If something goes wrong, for example, popping when the stack is empty, or getting to the end of the input prematurely, the PDA rejects on that branch of the computation.

2.57 Suppose A is context-free and let p be the associated pumping length. Let $s = 1^{2p}0^p1^p1^{2p}$ which is in A and longer than p. By the pumping lemma we know that $s = uvxyz$ satisfying the three conditions. We distinguish cases to show that s cannot be pumped and remain in A.

- If vxy is entirely in the last two thirds of s, then uv^2xy^2z contains 0s in its first third but not in its last third and so is not in A.
- Otherwise, vxy intersects the first third of s, and it cannot extend beyond the first half of s without violating the third condition.
  - If v contains both 1s and 0s, then uv^2xy^2z contains 0s in its first third but not in its last third and so is not in A.
  - If v is empty and y contains both 0s and possibly 1s, then again uv^2xy^2z contains 0s in its first third but not in its last third and is not in A.

– Otherwise, either v contains only 1s, or v is empty and y contains only 1s. In both cases, $uv^{1+6p}xy^{1+6p}z$ contains 0s in its last third but not in its first third and so is not in A.

A contradiction therefore arises and so A isn't a CFL.

2.58 a.

$$\begin{aligned} S &\to XSX \mid T1 \\ T &\to XT \mid X \\ X &\to 0 \mid 1 \end{aligned}$$

Here T is any nonempty string, and S is any string with $T1$ in the middle (so the 1 falls at least one character to the right of the middle).

b. Read the input until a 1 appears, while at the same time pushing 0s. Pop two 0s. Continue reading the input while popping the stack in an accept state until reach the end of the input. If the stack empties before reaching the end of the input, then do not read any more input (i.e., reject).

2.59 a.

$$\begin{aligned} S &\to T \mid 1 \\ T &\to XXTX \mid XTXX \mid XTX \mid X1X \\ X &\to 0 \mid 1 \end{aligned}$$

b. Assume that C_2 is a CFG and apply the pumping lemma to obtain the pumping length p. Let $s = 0^{p+2}10^p10^{p+2}$. Clearly, $s \in C_2$ so we may write $s = uvxyz$ satisfying the three conditions of the pumping lemma. If either v or y contains a 1, then the string uxz contains fewer than two 1s and thus it cannot be a member of C_2. By condition three of the pumping lemma, parts v and y cannot together contain 0s from both of the two outer runs of 0s. Hence in the string uv^2xy^2z the 1s cannot both remain in the middle third and so uv^2xy^2z is not in C_2.

Chapter 3

3.1 **a.** $q_1\texttt{0}$, $\sqcup q_2\sqcup$, $\sqcup\sqcup q_{\text{accept}}$

c. $q_1\texttt{000}$, $\sqcup q_2\texttt{00}$, $\sqcup\texttt{x}q_3\texttt{0}$, $\sqcup\texttt{x0}q_4\sqcup$, $\sqcup\texttt{x0}\sqcup q_{\text{reject}}$

d. $q_1\texttt{000000}$, $\sqcup q_2\texttt{00000}$, $\sqcup\texttt{x}q_3\texttt{0000}\sqcup$, $\sqcup\texttt{x0}q_4\texttt{000}$, $\sqcup\texttt{x0x}q_3\texttt{00}$, $\sqcup\texttt{x0x0}q_4\texttt{0}$, $\sqcup\texttt{x0x0x}q_3\sqcup$, $\sqcup\texttt{x0x0}q_5\texttt{x}\sqcup$, $\sqcup\texttt{x0x}q_5\texttt{0x}\sqcup$, $\sqcup\texttt{x0}q_5\texttt{x0x}\sqcup$, $\sqcup\texttt{x}q_5\texttt{0x0x}\sqcup$, $\sqcup q_5\texttt{x0x0x}\sqcup$, $q_5\sqcup\texttt{x0x0x}\sqcup$, $\sqcup q_2\texttt{x0x0x}\sqcup$, $\sqcup\texttt{x}q_2\texttt{0x0x}\sqcup$, $\sqcup\texttt{xx}q_3\texttt{x0x}\sqcup$, $\sqcup\texttt{xxx}q_3\texttt{0x}\sqcup$, $\sqcup\texttt{xxx0}q_4\texttt{x}\sqcup$, $\sqcup\texttt{xxx0x}q_4\sqcup$, $\sqcup\texttt{xxx0x}\sqcup q_{\text{reject}}$

3.2 **b.** $q_1\texttt{1}\#\texttt{1}$, $\texttt{x}q_3\texttt{\#1}$, $\texttt{x\#}\mathtt{q}_5\texttt{1}$, $\texttt{x}q_6\texttt{\#x}$, $q_7\texttt{x\#x}$, $\texttt{x}q_1\texttt{\#x}$, $\texttt{x\#}q_8\texttt{x}$, $\texttt{x\#x}q_8\sqcup$, $\texttt{x\#x}\sqcup q_{\text{accept}}$

c. $q_1\texttt{1\#\#1}$, $\texttt{x}q_3\texttt{\#\#1}$, $\texttt{x\#}\mathtt{q}_5\texttt{\#1}$, $\texttt{x\#\#}q_{\text{reject}}\texttt{1}$

d. $q_1\texttt{10\#11}$, $\texttt{x}q_3\texttt{0\#11}$, $\texttt{x0}q_3\texttt{\#11}$, $\texttt{x0\#}q_5\texttt{11}$, $\texttt{x0}q_6\texttt{\#x1}$, $\texttt{x}q_7\texttt{0\#x1}$, $q_7\texttt{x0\#x1}$, $\texttt{x}q_1\texttt{0\#x1}$, $\texttt{xx}q_2\texttt{\#x1}$, $\texttt{xx\#}q_4\texttt{x1}$, $\texttt{xx\#x}q_4\texttt{1}$, $\texttt{xx\#x1}q_{\text{reject}}\sqcup$

e. $q_1\texttt{10\#10}$, $\texttt{x}q_3\texttt{0\#10}$, $\texttt{x0}q_3\texttt{\#10}$, $\texttt{x0\#}q_5\texttt{10}$, $\texttt{x0}q_6\texttt{\#x0}$, $\texttt{x}q_7\texttt{0\#x0}$, $q_7\texttt{x0\#x0}$, $\texttt{x}q_1\texttt{0\#x0}$, $\texttt{xx}q_2\texttt{\#x0}$, $\texttt{xx}\#q_4\texttt{x0}$, $\texttt{xx\#x}q_4\texttt{0}$, $\texttt{xx\#}q_6\texttt{xx}$, $\texttt{xx}q_6\#x\texttt{x}$, $\texttt{x}q_7\texttt{x\#xx}$, $\texttt{xx}q_1\texttt{\#xx}$, $\texttt{xx\#}q_8\texttt{xx}$, $\texttt{xx\#x}q_8\texttt{x}$, $\texttt{xx\#xx}q_8\sqcup$, $\texttt{xx\#xx}\sqcup q_{\text{accept}}$.

3.4 An enumerator is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{print}}, q_{\text{halt}})$, where Q, Σ, Γ are all finite sets and

i) Q is the set of states,
ii) Γ is the work tape alphabet,
iii) Σ is the output tape alphabet,
iv) $\delta\colon Q \times \Gamma \longrightarrow Q \times \Gamma \times \{\text{L}, \text{R}\} \times \Sigma_\varepsilon$ is the transition function,
v) $q_0 \in Q$ is the start state,
vi) $q_{\text{print}} \in Q$ is the print state, and
vii) $q_{\text{halt}} \in Q$ is the reject state, where $q_{\text{print}} \neq q_{\text{halt}}$.

The computation of an enumerator E is defined as in an ordinary TM, except for the following points. It has two tapes, a work tape and a print tape, both initially blank. At each step, the machine may write a symbol from Σ on the output tape, or nothing, as determined by δ. If $\delta(q, a) = (r, b, \text{L}, c)$, it means that in state q, reading a, enumerator E enters state r, writes b on the work tape, moves the work tape head left (or right, if L had been R), writes c on the output tape, and moves the output tape head to the right if $c \neq \varepsilon$.

Whenever state q_{print} is entered, the output tape is reset to blank and the head returns to the left-hand end. The machine halts when q_{halt} is entered. $L(E) = \{w \in \Sigma^* \mid w$ appears on the work tape if q_{print} is entered$\}$.

3.6 In Stage 2 of this algorithm: "Run M on s_i", if M loops on a certain input s_i, E would not check any inputs after s_i. If that were to occur, E might fail to enumerate $L(M)$ as required.

3.7 The variables $x_1, \ldots, x_k$ have infinitely may possible settings. A Turing machine would required infinite time to try them all. But, we require that every stage in the Turing machine description be completed in a finite number of steps.

3.8 **b.** "On input string w:

1. Scan the tape and mark the first `0` which has not been marked. If there is no unmarked `0`, go to stage 5.
2. Continue scanning and mark the next unmarked `0`. If there is not any on the tape, *reject*. Otherwise, move the head to the front of the tape.
3. Scan the tape and mark the first `1` which has not been marked. If there is no unmarked `1`, *reject*.
4. Move the head to the front of the tape and repeat stage 1.
5. Move the head to the front of the tape. Scan the tape for any unmarked `1`s. If none, *accept*. Otherwise, *reject*."

c. "On input string w:

1. Scan the tape and mark the first `0` which has not been marked. If there is no unmarked `0`, go to stage 5.
2. Continue scanning and mark the next unmarked `0`. If there is not any on the tape, *accept*. Otherwise, move the head to the front of the tape.
3. Scan the tape and mark the first `1` which has not been marked. If there is no unmarked `1`, *accept*.
4. Move the head to the front of the tape and repeat stage 1.
5. Move the head to the front of the tape. Scan the tape for any unmarked `1`s. If none, *reject*. Otherwise, *accept*."

3.11 We give a DFA A that is equivalent to TM M. Design A in two stages. First, we arrange the states and transitions of A to simulate M on the read-only portion of the tape. Second, we assign the accept states of A to correspond to the action of M on the read/write portion of the tape to the right of the input. In the first stage, we construct A to store a function

$$F_s : (Q \cup \{\text{first}\}) \longrightarrow (Q \cup \{\texttt{acc}, \texttt{rej}\})$$

in its finite control where Q is the set of M's states and s is the string that it has read so far. Note that only finitely many such functions exist, so we assign one state for each possibility.

The function F_s contains information about the way M would compute on a string s. In particular, $F_s(\text{first})$ is the state that M enters when it is about to move off of the right end of s for the first time, if M was started in its starting state at the left end of s. If M accepts or rejects (either explicitly or by looping) before ever moving off of the right end of s, then $F_s(\text{first}) = \texttt{acc}$ or `rej` accordingly. Furthermore, $F_s(q)$ is the state M enters when it is about to move off of the right end of s for the first time, if M was started in state q at the right end of s. As before, if M accepts or rejects before ever moving off of the right end of s, then $F_s(q) = \texttt{acc}$ or `rej`.

Observe that, for any string s and symbol a in Σ, we can determine F_{sa} from a, F_s, and M's transition function. Furthermore, $F_{\boldsymbol{\varepsilon}}$ is predetermined because $F_{\boldsymbol{\varepsilon}}(\text{first}) = q_0$, the start state of M, and $F_{\boldsymbol{\varepsilon}}(q) = q$ (because TMs that attempt to move off the leftmost end of the tape just stay in the left-hand cell). Hence we can obtain A's start state and transition function. That completes the first stage of A's design.

To complete the second stage, we assign the accept states of M. For any two strings s and t, if $F_s = F_t$, then M must have the same output on s and t—either both s and

t are in $L(M)$ or both are out. Hence A's state at the end of s is enough to determine whether M accepts s. More precisely, we declare a state of A to be accepting if it was assigned to a function F that equals F_s for any string s that M accepts.

3.12 Let E be an enumerator for some infinite Turing-recognizable language B. We construct an enumerator D which outputs strings of an infinite language C in lexicographic order. The decidability of C follows from the solution to Problem 3.13. Enumerator D simulates E but only outputs a string if it is longer than the string previously outputted. Shorter strings are skipped. Because B is infinite, eventually some string that is longer than any outputted thus far must appear in E's output, so C will be infinite, too.

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

If A is enumerable in lexicographic order, we 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 appears which is lexicographically after w. 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 $reject$.

Note: Breaking into two cases is necessary to handle the possibility that the enumerator may loop without producing additional output when it is enumerating a finite language. As a result, we end up showing that the language is decidable but we do not (and cannot) algorithmically construct the decider for the language from the enumerator for the language. This subtle point is the reason for the star on the problem.

3.14 Let E be an enumerator for B. We construct an enumerator D which outputs the strings of C in lexicographic order. The decidability of C follows from the solution to Problem 3.13. Enumerator D simulates E. When E outputs the ith TM $\langle M_i \rangle$, enumerator D pads M_i by adding sufficiently extra useless states to obtain a new TM M_i' where the length of $\langle M_i' \rangle$ is greater than the length of $\langle M_{i-1}' \rangle$. Then E outputs $\langle M_i' \rangle$.

3.15 **b.** 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 NTM M' that recognizes the concatenation of L_1 and L_2:

"On input w:

1. Nondeterministically cut w into two parts $w = w_1 w_2$.
2. Run M_1 on w_1. If it halts and rejects, $reject$.
3. Run M_2 on w_2. If it accepts, $accept$. If it halts and rejects, $reject$."

If there is a way to cut w into two substrings such M_1 accepts the first part and M_2 accepts the second part, w belongs to the concatenation of L_1 and L_2 and M' will accept w after a finite number of steps.

c. For any Turing-recognizable language L, let M be the TM that recognizes it. We construct a NTM M' that recognizes the star of L:

"On input w:

1. Nondeterministically cut w into parts so that $w = w_1 w_2 \cdots w_n$.
2. Run M on w_i for all i. If M accepts all of them, $accept$. If it halts and rejects any of them, $reject$."

If there is a way to cut w into substrings such M accepts the all the substrings, w belongs to the star of L and M' will accept w after a finite number of steps.

d. 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 intersection of L_1 and L_2:
"On input w:

1. Run M_1 on w. If it halts and rejects, *reject*. If it accepts, go to stage 3.

2. Run M_2 on w. If it halts and rejects, *reject*. If it accepts, *accept*."

If both of M_1 and M_2 accept w, w belongs to the intersection of L_1 and L_2 and M' will accept w after a finite number of steps.

3.16 **b.** For any two decidable languages L_1 and L_2, let M_1 and M_2 be the TMs that decide them. We construct a NTM M' that decides the concatenation of L_1 and L_2:
"On input w:

1. For each way to cut w into two parts $w = w_1w_2$:

2. Run M_1 on w_1.

3. Run M_2 on w_2.

4. If both accept, *accept*. Otherwise, continue with next w_1, w_2.

5. All cuts have been tried without success, so *reject*."

We try every possible cut of w. If we ever come across a cut such that the first part is accepted by M_1 and the second part is accepted by M_2, w is in the concatenation of L_1 and L_2. So M' accept w. Otherwise, w does not belong to the concatenation of the languages and is rejected.

c. For any decidable language L, let M be the TM that decides it. We construct a NTM M' that decides the star of L:
"On input w:

1. For each way to cut w into parts so that $w = w_1w_2 \dots w_n$:

2. Run M on w_i for $i = 1, 2, \dots, n$.
If M accepts each of these string w_i, *accept*.

3. All cuts have been tried without success, so *reject*."

If there is a way to cut w into different substrings such that every substring is accepted by M, w belongs to the star of L and thus M' accepts w. Otherwise, w is rejected. Since there are finitely many possible cuts of w, M' will halt after finitely many steps.

d. For any decidable language L, let M be the TM that decides it. We construct a TM M' that decides the complement of L:
"On input w:

1. Run M on w. If M accepts, *reject*; if M rejects, *accept*."

Since M' does the opposite of whatever M does, it decides the complement of L.

e. For any two decidable languages L_1 and L_2, let M_1 and M_2 be the TMs that decide them. We construct a TM M' that decides the intersection of L_1 and L_2:
"On input w:

1. Run M_1 on w, if it rejects, *reject*.

2. Run M_2 on w, if it accepts, *accept*. Otherwise, *reject*."

M' accepts w if both M_1 and M_2 accept it. If either of them rejects, M' rejects w, too.

3.18 A TM with doubly infinite tape can simulate an ordinary TM. It marks the left-hand end of the input to detect and prevent the head from moving off of that end.

To simulate the doubly infinite tape TM by an ordinary TM, we show how to simulate it with a 2-tape TM, which was already shown to be equivalent in power to 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.

3.19 We simulate an ordinary TM with a reset TM that has only the RESET and R operations. 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.

3.20 This type of TM can recognize only regular languages.
Proof omitted.

3.21 First, we show that any queue automaton Q can be simulated by a 2-tape TM M. The first tape of M holds the input, and the second tape holds the queue. To simulate reading Q's next input symbol, M reads the symbol under the first head and moves it to the right. To simulate a push of a, M writes a on the leftmost blank square of the second tape. To simulate a pull, M reads the leftmost symbol on the second tape and shifts that tape one symbol leftward. Multi-tape TMs are equivalent to single tape TMs, so we can conclude that if a language can be recognized by a queue automaton, it is Turing-recognizable.

Now we show that any single-tape, deterministic TM M can be simulated by a queue automaton Q. For each symbol c of M's tape alphabet, the queue alphabet of Q has two symbols, c and $\dot{c}$. We use $\dot{c}$ to denote c with M's head over it. In addition, the queue alphabet has end-of-tape marker symbol $.

Automaton Q simulates M by maintaining a copy of M's tape in the queue. Q can effectively scan the tape from right to left by pulling symbols from the right-hand side of the queue and pushing them back on the left-hand side, until the $ is seen. When the dotted symbol is encountered, Q can determine M's next move, because Q can record M's current state in its control. Updating the queue so that it represents M's tape after the move requires another idea. If M's tape head moves leftward, then the updating is easily done by writing the new symbol instead of the old dotted symbol, and moving the dot one symbol leftward. If M's tape head moves rightward, then updating is harder because the dot must move right. By the time the dotted symbol is pulled from the queue, the symbol which receives the dot has already been pushed onto the queue. The solution is to hold tape symbols in the control for an extra move, before pushing them onto the queue. That gives Q enough time to move the dot rightward if necessary.

3.22 **a.** By Example 2.36, no PDA recognizes $B = \{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 0\}$. The following 2-PDA recognizes B. Push all the a's that appear in the front of the input tape to stack 1. Then push all the b's that follow the a's in the input onto stack 2. If it sees any a's at this stage, *reject*. When it sees the first c, pop stack 1 and stack 2 at the same time. After this, reject the input if it sees any a's or b's in the input. Pop stack 1 and stack 2 for each input character c it reads. If the input ends at the same time both stacks become empty, *accept*. Otherwise, *reject*.

b. We show that a 2-PDA can simulate a TM. Furthermore, a 3-tape NTM can simulate a 3-PDA, and an ordinary deterministic 1-tape TM can simulate a 3-tape NTM. Therefore a 2-PDA can simulate a 3-PDA, and so they are equivalent in power.

The simulation of a TM by a 2-PDA is as follows. Record the tape of the TM on two stacks. Stack 1 stores the characters on the left of the head, with the bottom of stack storing the leftmost character of the tape in the TM. Stack 2 stores the characters on the

right of the head, with the bottom of the stack storing the rightmost nonblank character on the tape in the TM. In other words, if a TM configuration is aq_ib, the corresponding 2-PDA is in state q_i, with stack 1 storing the string a and stack 2 storing the string b^R, both from bottom to top.

For each transition $\delta(q_i, c_i) = (q_j, c_j, \mathrm{L})$ in the TM, the corresponding PDA transition pops c_i off stack 2, pushes c_j into stack 2, then pops stack 1 and pushes the character into stack 2, and goes from state q_i to q_j. For each transition $\delta(q_i, c_i) = (q_j, c_j, \mathrm{R})$ in the TM, the corresponding PDA transition pops c_i off stack 2 and pushes c_j into stack 1, and goes from state q_i to q_j.

Chapter 4

4.2 Let $EQ_{\mathsf{DFA,REX}} = \{\langle A, R\rangle |\ A \text{ a DFA}, R \text{ a regular expression and } L(A) = L(R)\}$. The following TM E decides $EQ_{\mathsf{DFA,REX}}$.
$E =$ "On input $\langle A, R\rangle$:
1. Convert regular expression R to an equivalent DFA B using the procedure given in Theorem 1.54.
2. Use the TM C for deciding EQ_{DFA} given in Theorem 4.5, on input $\langle A, B\rangle$.
3. If R accepts, *accept*. If R rejects, *reject*."

4.3 Let $ALL_{\mathsf{DFA}} = \{\langle A\rangle |\ A \text{ is a DFA that recognizes } \Sigma^*\}$. The following TM L decides ALL_{DFA}.
$L =$ "On input $\langle A\rangle$ where A is a DFA:
1. Construct DFA B that recognizes $\overline{L(A)}$ as described in Exercise 1.14.
2. Run TM T from Theorem 4.4 on input $\langle B\rangle$, where T decides E_{DFA}.
3. If T accepts, *accept*. If T rejects, *reject*."

4.4 Let $A\varepsilon_{\mathsf{CFG}} = \{\langle G\rangle |\ G \text{ is a CFG that generates } \varepsilon\}$. The following TM V decides $A\varepsilon_{\mathsf{CFG}}$.
$V =$ "On input $\langle G\rangle$ where G is a CFG:
1. Run TM S from Theorem 4.7 on input $\langle G, \varepsilon\rangle$, where S is a decider for A_{CFG}.
2. If S accepts, *accept*. If S rejects, *reject*."

4.6
b. f is not onto because there does not exist $x \in X$ such that $f(x) = 10$.
c. f is not a correspondence because f is not one-to-one and onto.
e. g is onto.
f. g is a correspondence because g is one-to-one and onto.

4.7 Suppose $\mathcal{B}$ is countable and a correspondence $f\colon \mathcal{N} \longrightarrow \mathcal{B}$ exists. We construct x in $\mathcal{B}$ that is not paired with anything in $\mathcal{N}$. Let $x = x_1x_2\ldots$. Let $x_i = \mathtt{0}$ if $f(i)_i = \mathtt{1}$, and $x_i = \mathtt{1}$ if $f(i)_i = \mathtt{0}$ where $f(i)_i$ is the ith bit of $f(i)$. Therefore, we ensure that x is not $f(i)$ for any i because it differs from $f(i)$ in the ith symbol, and a contradiction occurs.

4.8 We demonstrate a one-to-one $f\colon T \longrightarrow \mathcal{N}$. Let $f(i, j, k) = 2^i3^j5^k$. Function f is one-to-one because if $a \neq b$, $f(a) \neq f(b)$. Therefore, T is countable.

4.9 Let $\sim$ be the binary relation "is the same size". In other words $A \sim B$ means A and B are the same size. We show that $\sim$ is an equivalence relation. First, $\sim$ is reflexive because

the identity function $f(x) = x, \forall x \in A$ is a correspondence $f\colon A \longrightarrow A$. Second, $\sim$ is symmetric because any correspondence has an inverse, which itself is a correspondence. Third, $\sim$ is transitive because if $A \sim B$ via correspondence f, and $B \sim C$ via correspondence g, then $A \sim C$ via correspondence $g \circ f$ (the composition of g and f). Because $\sim$ is reflexive, symmetric, and transitive, $\sim$ is an equivalence relation.

4.10 Here is an algorithm for F.

"On input $\langle P, q, x\rangle$:

1. Convert P to a PDA R which initially pushes a new stack symbol \$ and x and then simulates P starting in state q. If P ever sees \$ or it ever tries to read a new input symbol, then R accepts.
2. Test whether Q accepts input ε.
3. *Accept* if no and *reject* if yes."

4.11 Let $T = \{\langle G, A\rangle |\ G = (\Sigma, V, S, R)$ is a CFG and A is a usable variable$\}$. An algorithm for T is:

"On input $\langle G, w\rangle$:

1. Create a CFG H which is the same as G, except designate A to be the start variable.
2. Test whether $L(H)$ is empty and *reject* if it is.
3. Create a CFG K which is the same as G except designate A to be a terminal symbol and and remove all rules with A on the left-hand side. Call the new terminal alphabet $\Sigma_A = \Sigma \cup A$.
4. Test whether $L(K) \cap \Sigma_A^*$ is empty and *reject* if it is. Otherwise *accept*."

If A is usable in G, then $S \stackrel{*}{\Rightarrow} (\Sigma \cup V)^* A (\Sigma \cup V)^* \stackrel{*}{\Rightarrow} \Sigma^* A \Sigma^* \stackrel{*}{\Rightarrow} w \in \Sigma^*$ for some w. Hence $L(H)$ isn't empty because S derives a string that contains A and A derives a substring of w. Therefore the algorithm will accept. Conversely, if the algorithm accepts, then $L(K)$ isn't empty so S derives a string that contains A and $L(H)$ isn't empty so A derives some string over Σ. Hence S derives some string of terminals in G by way of A. Therefore A is usable.

4.12 Use the diagonalization method to obtain D. Let $\langle M_1\rangle, \langle M_2\rangle, \ldots$ be the output of an enumerator for A. Let $\Sigma = \{s_1, s_2, \ldots\}$ be a list of all strings over the alphabet. The algorithm for D is:

$D =$ "On input w:

1. Let i be the index of w on the list of all strings, that is, $s_i = w$.
2. Run $\langle M_i\rangle$ on input w.
3. If it accepts, *reject*. If it rejects, *accept*."

D is a decider because each M_i is a decider. But D doesn't appear on the enumeration for A, because it differs from M_i on input s_i.

4.13 Given CFG G and a number k, first find the pumping length of $L(G)$ using the expression in the proof of the pumping lemma for context-free languages. Use the test for $\mathit{INFINITE}_{\mathsf{PDA}}$ to determine whether G generates infinitely many strings. If yes, then accept if $k = \infty$ and reject if $k \neq \infty$. If $L(G)$ is not infinite and $k = \infty$ then reject. If $L(G)$ is not infinite and k is finite, find which strings up to length p are generated by G by testing each string individually, then count the number to see if it agrees with k. Then accept on agreement and reject otherwise.

4.14 We describe an algorithm for C. Given CFG G and string x, construct a PDA P_x which simulates the PDA P for $L(G)$, except that during the course of the simulation, it nondeterministically chooses a point to insert x into the symbols on which it is simulating P. After P has processed all of x, the simulation continues reading the rest of the actual input. Observe that P accepts some input string that contains x as a substring iff P_x accepts any string at all. So we can test whether $\langle G, x\rangle \in C$ by testing whether $L(P_x)$ is empty.

4.15 The language of all strings with more 1s than 0s is a context free language, recognized by a PDA P that keeps a (positive or negative) unary count on its stack of the number of 1s minus the number of 0s that have been seen so far on the input. Build a TM M for BAL_{DFA}, which operates as follows. On input $\langle B\rangle$, where B is a DFA, use B and P to construct a new PDA R that recognizes the intersection of the languages of B and P, by the solution to Problem 2.30. Test whether R's language is empty. If its language is empty, reject; otherwise accept.

4.16 The language of all palindromes is a CFL. Let P be a PDA that recognizes this language. Build a TM M for PAL_{DFA}, which operates as follows. On input $\langle B\rangle$, where B is a DFA, use B and P to construct a new PDA R that recognizes the intersection of the languages of B and P, as in the solution to Problem 2.30. Test whether R's language is empty. If its language is empty, reject; otherwise accept.

4.18 Let $U = \{\langle P\rangle |\ P \text{ is a PDA that has useless states}\}$. The following TM T decides U.
$T =$ "On input $\langle P\rangle$, where P is a PDA:

1. For each state q of P:
2. Modify P so that q is the only accept state.
3. Use the decider for E_{PDA} to test whether the modified PDA's language is empty. If it is, *accept*. If it is not, continue.
4. At this point, all states have been shown to be useful, so *reject*."

4.20 For any language A, let $A^{\mathcal{R}} = \{w^{\mathcal{R}} |\ w \in A\}$. Observe that if $\langle M\rangle \in S$ then $L(M) = L(M)^{\mathcal{R}}$. The following TM T decides S.
$T =$ "On input $\langle M\rangle$, where M is a DFA:

1. Construct DFA N that recognizes $L(M)^{\mathcal{R}}$ using the construction from Problem 1.36.
2. Run TM F from Theorem 4.5 on $\langle M, N\rangle$, where F is the Turing machine deciding EQ_{DFA}.
3. If F accepts, *accept*. If F rejects, *reject*."

4.21 We describe two solutions to this problem. In the first solution, we convert a given regular expression R to a DFA M. We test whether each accept state of M is reachable along a path starting from any accept state (including the same one). The answer is yes iff M's language is not prefix-free. This method doesn't work for CFGs because it would ignore the effect of the stack in the corresponding PDAs. For the second solution, we let regular expression $R' = R\Sigma^+$. Observe that R' generates all strings that are extensions of strings in $L(R)$, so $L(R)$ is prefix-free iff $L(R) \cap L(R')$ is empty. Testing whether that language is empty is possible, by converting it to a DFA and using the standard empty language test for DFAs. That method fails for CFGs because testing whether the intersection of two CFLs is empty isn't decidable.

4.22 Let A and B be two languages such that $A \cap B = \emptyset$, and $\overline{A}$ and $\overline{B}$ are recognizable. Let J be the TM recognizing $\overline{A}$ and K be the TM recognizing $\overline{B}$. We will show that the language decided by TM T separates A and B.
$T =$ "On input w:

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

The algorithm T terminates 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 accepted 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.

4.23 For an TM M and a string w, we write $\langle M, w\rangle$ to represent the encoding of M and w into a string. We can assume that this string is over the alphabet $\{0,1\}$. Let $D = \{\langle M, w\rangle 2^k |\ M \text{ accepts } w \text{ within } k \text{ steps}\}$be a language over the alphabet $\{0,1,2\}$. Language D is decidable, because we can run M on w for k steps to determine whether $\langle M, w\rangle 2^k \in D$. Define the homomorphism $f\colon \{0,1,2\} \longrightarrow \{0,1,2\}$ as follows: $f(0) = 0$, $f(1) = 1$, $f(2) = \varepsilon$. Then $f(D) = A_{\text{TM}}$.

4.24 We need to prove both directions. To handle the easier one first, assume that 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 continue searching.

For the other direction, assume that C is recognized by TM M. Define a language B to be $\{\langle x, y\rangle |\ M \text{ accepts } x \text{ within } |y| \text{ steps}\}$. Language B is decidable, and if $x \in C$ then M accepts x within some number of steps, so $\langle x, y\rangle \in B$ for any sufficiently long y, but if $x \notin C$ then $\langle x, y\rangle \notin C$ for any y.

4.25 For any DFAs A and B, $L(A) = L(B)$ if and only if 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. Obviously, if $L(A) = L(B)$, A and B will accept the same strings. If $L(A) \neq L(B)$, we 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 languages differ. Let l be the length of t. If $l \leq mn$ 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 of t 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.

4.26 The following TM X decides A.
$X =$ "On input $\langle R\rangle$ where R is a regular expression:

1. Construct DFA E that accepts $\Sigma^*117\Sigma^*$.
2. Construct DFA B such that $L(B) = L(R) \cap L(E)$.
3. Run TM T from Theorem 4.4 on input $\langle B\rangle$, where T decides E_{DFA}.
4. If T accepts, *reject*. If T rejects, *accept*."

4.29 We observe that $L(R) \subseteq L(S)$ if and only if $\overline{L(S)} \cap L(R) = \emptyset$. The following TM X decides A.

$X =$ "On input $\langle R, S \rangle$ where R and S are regular expressions:

1. Construct DFA E such that $L(E) = \overline{L(S)} \cap L(R)$.
2. Run TM T on $\langle E \rangle$, where T decides E_{DFA} from Theorem 4.4.
3. If T accepts, *accept*. If T rejects, *reject*."

4.31 We sketch the algorithm to decide $\mathit{INFINITE}_{\mathsf{PDA}}$. Given $\langle P \rangle$, a description of a PDA, convert it to a CFG G and compute G's pumping length p. PDA P accepts a string longer than p iff P's language is infinite. We show how to test whether P accepts such a string. Let T be the regular language containing all strings longer than p. Use the solution to Problem 2.30 to find a CFG H that generates $L(G) \cap T$. Then use Theorem 4.8 to test whether $L(H)$ is empty. Accept P if $L(H)$ nonempty; otherwise reject.

Chapter 5

5.1 Suppose for a contradiction that EQ_{CFG} were decidable. We construct a decider M for $ALL_{\mathsf{CFG}} = \{\langle G\rangle \mid G \text{ is a } \mathsf{CFG} \text{ and } L(G) = \Sigma^*\}$ as follows:
$M =$ "On input $\langle G\rangle$:

1. Construct a CFG H such that $L(H) = \Sigma^*$.
2. Run the decider for EQ_{CFG} on $\langle G, H\rangle$.
3. If it accepts, *accept*. If it rejects, *reject*."

M decides ALL_{CFG} assuming a decider for EQ_{CFG} exists. Since we know ALL_{CFG} is undecidable, we have a contradiction.

5.2 Here is a Turing Machine M which recognizes the complement of EQ_{CFG}:
$M =$ "On input $\langle G, H\rangle$:

1. For each string $x \in \Sigma^*$ in lexicographic order:
2. Test whether $x \in L(G)$ and whether $x \in L(H)$, using the algorithm for A_{CFG}.
3. If one of the tests accepts and the other rejects, *accept*; otherwise, continue."

5.3 Here is a match: $\left[\frac{\mathtt{ab}}{\mathtt{abab}}\right]\left[\frac{\mathtt{ab}}{\mathtt{abab}}\right]\left[\frac{\mathtt{aba}}{\mathtt{b}}\right]\left[\frac{\mathtt{b}}{\mathtt{a}}\right]\left[\frac{\mathtt{b}}{\mathtt{a}}\right]\left[\frac{\mathtt{aa}}{\mathtt{a}}\right]\left[\frac{\mathtt{aa}}{\mathtt{a}}\right]$.

5.4 No, it doesn't. For example, $\{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 0\} \leq_{\mathrm{m}} \{\mathtt{a}, \mathtt{b}\}$. The reduction first tests whether its input is a member of $\{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 0\}$. If so, it outputs the string $\mathtt{ab}$, and if not, it outputs the string $\mathtt{a}$.

5.9 We reduce PCP to $\mathit{AMBIG}_{\mathsf{CFG}}$, thereby proving that $\mathit{AMBIG}_{\mathsf{CFG}}$ is undecidable. We use the construction given in the hint in the text. We show that P has a solution iff the CFG is ambiguous.

If P has a match $t_{i_1}t_{i_2}\ldots t_{i_l} = b_{i_1}b_{i_2}\ldots b_{i_l}$, the string

$$t_{i_1}t_{i_2}\ldots t_{i_l}\mathtt{a}_{i_l}\ldots\mathtt{a}_{i_2}\mathtt{a}_{i_1} = b_{i_1}b_{i_2}\ldots b_{i_l}\mathtt{a}_{i_l}\ldots\mathtt{a}_{i_2}\mathtt{a}_{i_1}$$

has two different leftmost derivations, one from T and one from B. Hence the CFG is ambiguous.

Conversely, if the CFG is ambiguous, some string w has multiple leftmost derivations. All generated strings have the form $w = w_{start}\mathtt{a}_{i_l}\ldots\mathtt{a}_{i_2}\mathtt{a}_{i_1}$ where w_{start} contains only symbols from P's alphabet. Notice that once we choose the first step in the derivation of w (either $S \to T$ or $S \to B$), the following steps in the derivation are uniquely determined by the sequence $\mathtt{a}_{i_l}\ldots\mathtt{a}_{i_2}\mathtt{a}_{i_1}$. Therefore w has at most two leftmost derivations:

i) $S \to T \to t_{i_1} T \mathtt{a}_{i_1} \to t_{i_1} t_{i_2} T \mathtt{a}_{i_2} \mathtt{a}_{i_1} \to \cdots \to t_{i_1} t_{i_2} \ldots t_{i_l} \mathtt{a}_{i_l} \ldots \mathtt{a}_{i_2} \mathtt{a}_{i_1}$, and
ii) $S \to B \to b_{i_1} T \mathtt{a}_{i_1} \to b_{i_1} b_{i_2} T \mathtt{a}_{i_2} \mathtt{a}_{i_1} \to \cdots \to b_{i_1} b_{i_2} \ldots b_{i_l} \mathtt{a}_{i_l} \ldots \mathtt{a}_{i_2} \mathtt{a}_{i_1}$.

If w is ambiguous, $t_{i_1} t_{i_2} \ldots t_{i_l} \mathtt{a}_{i_l} \ldots \mathtt{a}_{i_2} \mathtt{a}_{i_1} = b_{i_1} b_{i_2} \ldots b_{i_l} \mathtt{a}_{i_l} \ldots \mathtt{a}_{i_2} \mathtt{a}_{i_1}$, and therefore $t_{i_1} t_{i_2} \ldots t_{i_l} = b_{i_1} b_{i_2} \ldots b_{i_l}$. Thus, the initial PCP instance P has a match.

5.10 If $A \leq_{\mathrm{m}} A_{\mathsf{TM}}$ then A is Turing-recognizable by virtue of Theorem 5.28. If A is Turing-recognizable, let M be a TM that recognizes it. We reduce A to A_{TM} by mapping each string x to $\langle M, x \rangle$. Then $x \in A \iff \langle M, x \rangle \in A_{\mathsf{TM}}$. In addition, this mapping is computable, so it gives a mapping reduction from A to A_{TM}.

5.11 If $A \leq_{\mathrm{m}} \mathtt{0}^*\mathtt{1}^*$ then A is decidable by virtue of Theorem 5.22. If A is decidable then we can reduce A to $\mathtt{0}^*\mathtt{1}^*$ with the function computed by the following TM F:
$F =$ "On input w:

1. Test whether $w \in A$ using A's decision procedure.
2. If $w \in A$ then output $\mathtt{01}$.
3. If $w \notin A$ then output $\mathtt{10}$."

5.12 To prove that J is not Turing-recognizable, we show that $\overline{A_{\mathsf{TM}}} \leq_{\mathrm{m}} J$. The reduction maps any string y to the string $\mathtt{1}y$. Then $y \in \overline{A_{\mathsf{TM}}} \iff \mathtt{1}y \in J$. To prove that $\overline{J}$ is not Turing-recognizable we show that $A_{\mathsf{TM}} \leq_{\mathrm{m}} J$ (note $A_{\mathsf{TM}} \leq_{\mathrm{m}} J \iff \overline{A_{\mathsf{TM}}} \leq_{\mathrm{m}} \overline{J}$). The reduction maps any string x to string $\mathtt{0}x$. Then $x \in A_{\mathsf{TM}} \iff \mathtt{0}x \in J$.

5.13 Use the language J from Problem 5.12. That problem showed that J is undecidable. Here we show that $J \leq_{\mathrm{m}} \overline{J}$. The reduction f maps ε to itself and for other strings x,

$$f(x) = \begin{cases} \mathtt{0}s & \text{if } x = \mathtt{1}s \text{ for some string } s \in \Sigma^* \\ \mathtt{1}s & \text{if } x = \mathtt{0}s \text{ for some string } s \in \Sigma^* \end{cases}$$

The function f is computable, and $x \in J \iff f(x) \in \overline{J}$. Thus, $J \leq_{\mathrm{m}} \overline{J}$.

5.14 **a.** A 2DFA M with s states, computing on an input x of size n, has at most $s(n+2)^2$ possible configurations. Thus, on input x, M either halts in $s(n+2)^2$ steps or loops forever. To decide A_{2DFA}, it is enough to simulate M on x for $s(n+2)^2$ steps and accept if M has halted and accepted x during this finite duration.

b. Suppose a TM D decides E_{2DFA}. We construct a TM E that decides E_{TM} as follows:
$E =$ "On input $\langle M \rangle$:

1. Construct a 2DFA M' that recognizes the language of accepting computation histories on M: $\{\langle c_1 \# c_2 \# \cdots \# c_k \rangle \mid c_i$ is a configuration of M, c_1 is a start configuration, c_k is an accepting configuration, c_{i+1} legally follows c_i for each $i\}$. This 2DFA checks whether c_1 is a start configuration of M by verifying that it begins with a start state q_0 and contains legal symbols from M's input alphabet. Similarly, it checks whether c_k is an accepting configuration by verifying that it contains q_{accept} and symbols from M's tape alphabet. These two steps use only one head. To check whether c_{i+1} legally follows c_i, the 2DFA keeps its two heads on c_i and c_{i+1}, and compares them symbol by symbol. Notice that $L(M') = \emptyset$ iff $L(M) = \emptyset$.
2. Run D on $\langle M' \rangle$. If D accepts, *accept*. If D rejects, *reject*."

Since E_{TM} is undecidable, E_{2DFA} is decidable.

5.15 We formulate the problem of testing 2DIM-DFA equivalence as a language: $EQ_{\text{2DIM-DFA}} = \{\langle M_1, M_2\rangle |\ M_1 \text{ is equivalent to } M_2\}$.
We show that if $EQ_{\text{2DIM-DFA}}$ is decidable, we can decide A_{TM} by reducing A_{TM} to $EQ_{\text{2DIM-DFA}}$. Therefore $EQ_{\text{2DIM-DFA}}$ is undecidable. Assume R is the TM deciding $EQ_{\text{2DIM-DFA}}$. We construct S that decides A_{TM} as follows:
$S =$ "On input $\langle A, w\rangle$:

1. Construct a 2DIM-DFA T that accepts only the accepting computation history of A on w. The input that T accepts should represent the computation history of A in the form of an $m \times n$ rectangle, where $(m-2)$ is the length of the longest configuration that A ever has during the computation and $(n-2)$ is the number of steps it takes to get to the accepting state. Each row of the inner cells contains a proper configuration of A during its computation. In order to check if the input is a computation history of A on w, T follows a procedure similar to that given the proof of Theorem 5.9. It first checks that the first inner row contains the configuration with input w and the start state at the left end. Then it checks whether the TM reaches the accept state at the last inner row. After that, it compares two consecutive rows of input to see if the two configurations follow the transition function of A. If the input passes all the three tests, the 2DIM-DFA accepts. Otherwise, it rejects.
2. Construct another 2DIM-DFA E which always rejects. Run R on $\langle T, E\rangle$ to see if T and E are equivalent. If R accepts, *reject* because there is no computation history for A on w. Otherwise, *accept*."

The TM S decides A_{TM}. However, we know that A_{TM} is undecidable. Therefore $EQ_{\text{2DIM-DFA}}$ is also undecidable.

5.17 First, let P be the language $\{\langle M\rangle |\ M \text{ is a TM with 5 states}\}$. P is non-trivial, and so it satisfies the second condition of Rice's Theorem but P can be easily decided by checking the number of states of the input TM. Second, let P be the empty set. Then it does not contain any TM and so it satisfies the first condition of Rice's Theorem, but P can be decided by a TM that always rejects. Therefore both properties are necessary for proving P undecidable.

5.18 **b.** Let $A = \{\langle M\rangle |\ M \text{ is a TM and } \texttt{1011} \in L(M)\}$. Clearly A is a language of TM descriptions. It satisfies the two conditions of Rice's theorem. First, it is nontrivial because some TMs accept `1011` and others do not. Second, membership in A depends only on the TM's language. If two TMs recognize the same language, either both accept `1011` or neither do. Thus, Rice's theorem implies that A is undecidable.

c. Show that ALL_{TM}, a language of TM descriptions, satisfies the conditions of Rice's theorem. First, it is nontrivial because some TMs recognize Σ^* and others do not. Second, membership in ALL_{TM} depends only on the TM's language. If two TMs recognize the same language, either both recognize Σ^* or neither do. Thus, Rice's theorem implies that A is undecidable.

5.19 Define TM S on input $\langle n\rangle$ where n is a natural number, to run the $3x+1$ procedure stating at n and *accept* if and when 1 is reached. (If n never reaches 1, then S will not halt.) Define TM T on input $\langle n\rangle$, to use H to determine whether or not S accepts $\langle n\rangle$, and then *accept* if it does, and halt and *reject* if it does not. Define TM P, which (on any input) runs T on input $\langle 1\rangle, \langle 2\rangle, \langle 3\rangle, \ldots$ and *accept* if T is found to reject any of these inputs. Otherwise P will not halt. Define TM Q, which (on any input) uses H

to determine whether P accepts 0 and outputs `conjecture is false` if P accepts 0, and `conjecture is true` if not. TM Q is the solution to the problem.

5.20 **a.** Reduce PCP to $\mathit{OVERLAP}_{\mathsf{CFG}}$, thereby proving that $\mathit{OVERLAP}_{\mathsf{CFG}}$ is undecidable. Modify the construction given in the hint to Problem 5.9 to map a PCP problem P to a pair of CFGs G and H so that P has a match iff G and H generate a string in common.
Grammar G is: $T \to t_1 T \mathtt{a}_1 \mid \cdots \mid t_k T \mathtt{a}_k \mid t_1 \mathtt{a}_1 \mid \cdots \mid t_k \mathtt{a}_k$.
Grammar H is: $B \to b_1 B \mathtt{a}_1 \mid \cdots \mid b_k B \mathtt{a}_k \mid b_1 \mathtt{a}_1 \mid \cdots \mid b_k \mathtt{a}_k$.
If P has a match $t_{i_1} t_{i_2} \ldots t_{i_l} = b_{i_1} b_{i_2} \ldots b_{i_l}$, the string

$$t_{i_1} t_{i_2} \ldots t_{i_l} \mathtt{a}_{i_l} \ldots \mathtt{a}_{i_2} \mathtt{a}_{i_1} = b_{i_1} b_{i_2} \ldots b_{i_l} \mathtt{a}_{i_l} \ldots \mathtt{a}_{i_2} \mathtt{a}_{i_1}$$

is in $L(G)$ and in $L(H)$. Hence the $\langle G, H \rangle \in \mathit{OVERLAP}_{\mathsf{CFG}}$. Conversely, if $\langle G, H \rangle \in \mathit{OVERLAP}_{\mathsf{CFG}}$ then some string $w \in L(G) \cap L(H)$. All generated strings in $L(G)$ and $L(H)$ have the form $w = x \mathtt{a}_{i_l} \ldots \mathtt{a}_{i_2} \mathtt{a}_{i_1}$ where x contains only symbols from P's alphabet. If we take the dominos in P corresponding to $i_1, i_2, \ldots, i_l$, then the string appearing along the tops of these dominos is $t_{i_1} t_{i_2} \ldots t_{i_l}$ which equals w_{start} because of the way G is designed. Similarly, the string appearing along the bottoms of these dominos is $b_{i_1} b_{i_2} \ldots b_{i_l}$ which also equals x because of the way H is designed. Thus the string along the tops equals the string along the bottoms, hence P has match.

b. Reduce $\mathit{OVERLAP}_{\mathsf{CFG}}$ to $\mathit{PREFIX\text{-}FREE}_{\mathsf{CFG}}$ using the following reduction f. Given $\langle G, H \rangle$ where G and H are CFGs, let $f(\langle G, H \rangle)$ be a CFG D which generates the language $L(G)\mathtt{\#} \cup L(H)\mathtt{\#\#}$. If $x \in L(G) \cap L(H)$, then $x\mathtt{\#} \in L(D)$ and $x\mathtt{\#\#} \in L(D)$ so D isn't prefix-free. Conversely, if D isn't prefix-free, then strings y and z are in $L(D)$ where y is a proper prefix of z. That can occur only if $y = x\mathtt{\#}$ and $z = x\mathtt{\#\#}$ for some string $x \in L(G) \cap L(H)$.

5.21 We shall show this problem is undecidable by reducing A_{TM} to it. Define

$$\mathit{WW}_{\mathsf{PDA}} = \{\langle P \rangle \mid \text{is a PDA and } L(P) \cap \{ww \mid w \in \{\mathtt{0},\mathtt{1}\}^*\} \neq \emptyset\}$$

Given an instance $\langle M, x \rangle$ of A_{TM}, construct $\langle P_{M,x} \rangle$, an instance of $\mathit{WW}_{\mathsf{PDA}}$, such that $\langle M, x \rangle \in A_{\mathsf{TM}}$ iff $\langle P_{M,x} \rangle \in \mathit{WW}_{\mathsf{PDA}}$. The only string of the form ww that $P_{M,x}$ accepts encodes the accepting computation history $\mathit{CH}_{M,x}$ of M on x. Here $\mathit{CH}_{M,x} = C_1 \# C_2^R \# C_3 \# C_4^R \ldots$ (alternate configurations are written in reverse order). The encoded accepting computation history is $CH_{M,x}\$CH_{M,x}\$$. A PDA $P_{M,x}$ can check that C_1 legally yields C_2 in a pair of configurations presented as $C_1 \# C_2^R$ (the reversal facilitates the check). Thus a PDA can check in a given configuration history $C_1 \# C_2^R \# C_3 \# C_4^R \ldots$, whether C_1 yields C_2, then whether C_3 yields C_4, then whether C_5 yields C_4 and so on, but checking the validity of consecutive steps C_1 yields C_2 and C_2 yields C_3 is problematic because it can read C_2 only once. However, because we are focusing on input strings of the form ww, the PDA effectively sees w twice, so if $w = \mathit{CH}_{M,x}$, the PDA can check the odd steps first and then the even steps. The PDA accepts only if all of these steps correspond to a legal move of M. Hence $P_{M,x}$ accepts a string of the form ww iff M accepts x.

5.22 Reduce A_{TM} to X to show that X is undecidable. Assume some machine R decides X. The following TM S decides A_{TM}:

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

1. Construct the following TM Q.
   $Q =$ "On input w:
   1. Move past the end of the input.
   2. Using the rest of the tape, run M on w.
   3. If M accepts then go to the beginning of the tape and write `good-bye`."
2. Run R on input $\langle Q, \texttt{hello}\rangle$.
3. If R accepts then *reject*. If it rejects, *accept*."

If $\langle M, w\rangle \in A_{\mathsf{TM}}$ then M accepts w. So, on input `hello` machine Q would write `good-bye` on the initial part of the tape. and R would reject $\langle Q, \texttt{hello}\rangle$ and so S would accept $\langle M, w\rangle$. Conversely, if S accepts $\langle M, w\rangle$, then R must reject $\langle Q, \texttt{hello}\rangle$ and so M doesn't accept w and thus $\langle M, w\rangle \notin A_{\mathsf{TM}}$.

5.23 **a.** Let $N = \{\langle G, A, w\rangle |\ A$ appears in every derivation of $w\}$. First show that N is decidable, in order to show that *NECESSARY*$_{\mathsf{CFG}}$ is T-recognizable. Let $\Sigma^* = \{w_1, w_2, \dots\}$ be a list of all strings.

"On input $\langle G, A, w\rangle$:

1. Let G_A be CFG G, except that rules involving A are removed.
2. For $i = 1, 2, \dots$:
3. If $w_i \in L(G)$ and $w_i \notin L(G_A)$ then *accept*."

b. To show *NECESSARY*$_{\mathsf{CFG}}$ is undecidable, reduce *ALL*$_{\mathsf{CFG}}$ to it as follows. Assume that TM R decides *NECESSARY*$_{\mathsf{CFG}}$ and construct TM S deciding *ALL*$_{\mathsf{CFG}}$.

$S =$ "On input $\langle G\rangle$:

1. Construct CFG G' which is a copy of G, except that it has the following new rules: $S_0 \to S \mid T$ and $T \to aT \mid \varepsilon$ for each $a \in \Sigma$. Here S_0 is a new start variable, S is the original start variable and T is a new variable.
2. Use R to test whether T is necessary in G'.
3. *Reject* if yes, and *accept* if no."

If G generates all strings then G' does too because G is embedded in G' and T isn't necessary for any string. However if G doesn't generate some string, then G' still generates all strings, but now T is necessary to generate the string missing from $L(G)$. Hence $\langle G\rangle \in$ *ALL*$_{\mathsf{CFG}}$ iff $\langle G', T\rangle \in$ *NECESSARY*$_{\mathsf{CFG}}$.

5.24 Part b of this problem is a bit too tricky.

a. The following algorithm recognizes *MIN*$_{\mathsf{CFG}}$. Let $\Sigma^* = \{w_1, w_2, \dots\}$ be a list of all strings.

"On input $\langle G\rangle$:

1. For each rule r in G, let G_r be the same grammar as G, but with r removed. Then perform the following:
2. For $i = 1, 2, \dots$
3. Test whether $w_i \in L(G)$ and whether $w_i \in L(G_r)$.
4. If the answers are different, then removing r changes G's language, so r is necessary. Continue with the next r. If the answers are equal, continue with the next i.
5. If reach this point then all r are necessary so *accept*."

b. To show *MIN*$_{\mathsf{CFG}}$ is undecidable, reduce *ALL*$_{\mathsf{CFG}}$ to it. The idea is to convert a CFG G to another CFG H where $L(G) = \Sigma^*$ iff H is non-minimal. The CFG H is is a modified

version of G plus a few rules which generate Σ^*. If $L(G) = \Sigma^*$ then these few rules can be removed from H without changing it's language, so H wouldn't be minimal. If $L(G) \neq \Sigma^*$ then removing these new rules from H would change its language, so H would be minimal if G were minimal. The following trick modifies G to guarantee it is minimal.

Let $G = (V, \Sigma, S, R)$. Create alphabet Σ' by adding new terminal symbols a_A and b_A for each variable A. Thus $\Sigma' = \Sigma \cup \{a_A |\ A \in V\} \cup \{b_A |\ A \in V\}$. Let r be a rule $A \to z$ in G where $z \in (V \cup \Sigma)^*$. Say that r is *redundant* if $A \stackrel{*}{\Rightarrow} z$ in G_r, where G_r is G with r removed. A redundant rule is clearly an unnecessary rule, but a rule may be unnecessary without being redundant. Conveniently, we can test algorithmically whether a rule is redundant. Construct new CFG $G' = (V', \Sigma', S', R')$ with a new start symbol S' and whose rules are a copy of G's rules and additional rules. Add the rule $S' \to S$ to G'. Furthermore, add the rules $S' \to a_A A$ and $A \to b_A$ to G' for each variable A in G. Let G'_r be G' with rule r removed. First determine whether each r is redundant in G' by testing whether $a_A z' \in L(G'_r)$ where r is $A \to z$ and z' is z where every variable B is substituted by terminal b_B. Clearly, if r is redundant then $a_A z' \in L(G'_r)$ and conversely if $a_A z' \in L(G'_r)$ then $A \stackrel{*}{\Rightarrow} z'$ which can happen only if $A \stackrel{*}{\Rightarrow} z$. If r is redundant, remove it before testing whether other rules are redundant.

Once all redundant rules are removed from G', all remaining rules are necessary because they are needed to generate the strings $a_A z'$. Thus, the grammar H is minimal iff $\Sigma^* \subsetneq L(G')$, which happens exactly when $L(G) \neq \Sigma^*$. Thus, if we can decide whether a CFG is minimal, we decide whether a CFG generates Σ^*, but the latter is undecidable.

5.25 We show that $A_{\mathsf{TM}} \leq_{\mathrm{m}} T$ by mapping $\langle M, w\rangle$ to $\langle M'\rangle$ where M' is the following TM:
$M' =$ **"**On input x:

1. If $x \neq \mathtt{01}$ and $x \neq \mathtt{10}$ then *reject*.
2. If $x = \mathtt{01}$ then *accept*.
3. If $x = \mathtt{10}$ simulate M on w. If M accepts w then *accept*; if M halts and rejects w then *reject*.**"**

If $\langle M, w\rangle \in A_{\mathsf{TM}}$ then M accepts w and $L(M') = \{\mathtt{01}, \mathtt{10}\}$, so $\langle M'\rangle \in T$. Conversely, if $\langle M, w\rangle \notin A_{\mathsf{TM}}$ then $L(M') = \{01\}$, so $\langle M'\rangle \notin T$. Therefore, $\langle M, w\rangle \in A_{\mathsf{TM}} \iff \langle M'\rangle \in T$.

5.28 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$\}$. Show that A_{TM} reduces to E Assume for the sake of contradiction that TM R decides E. Construct TM S that uses R to decide A_{TM}.
$S =$ **"**On input $\langle M, w\rangle$:

1. Use M and w to construct the following TM T_w.
   $T_w =$ **"**On any input:
   1. Simulate M on w. Use new symbol $\sqcup'$ instead of blank when writing, and treat it like a blank when reading it.
   2. If M accepts, write a true blank symbol over a 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, therefore *accept*. Otherwise *reject*.**"**

5.29 Let $USELESS_{\mathsf{TM}} = \{\langle M\rangle |\ M$ is a TM with one or more useless states$\}$. Show that A_{TM} reduces to $USELESS_{\mathsf{TM}}$. Assume for the sake of contradiction that TM R decides

$USELESS_{\mathsf{TM}}$. Construct TM S that uses R to decide A_{TM}. The new TM has a useless state exactly when M doesn't accept w. Doing so carefully can be messy. For convenience we use the universal TM.

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

T = "On input x:

1. Replace x on the input by the string $\langle M, w\rangle$.
2. Run the universal TM U that is described after the statement of Theorem 4.11 to simulate $\langle M, w\rangle$. Note that this TM was designed to use all of its states.
3. If U accepts, enter a special state q_A and *accept*."

1. Run R on $\langle T\rangle$ to determine whether T has any useless states.
2. If R rejects, M accepts w, therefore *accept*. Otherwise *reject*."

If M accepts w, then T enters all states, but if M doesn't accept w, then T avoids q_A. So T has a useless state—namely q_A—iff M doesn't accept w.

5.30 $L_{\mathsf{TM}} = \{\langle M, w\rangle |\ M \text{ on } w \text{ tries moving its head left from the leftmost tape cell, at some point in its computation}\}$. Assume to the contrary that TM R decides L_{TM}. Construct TM S that uses R to decide 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 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*."

TM S decides A_{TM} because M' only moves left from the leftmost tape cell when M accepts w. S never attempts that move during the course of the simulation because we put the \$ to "intercept" such moves if made by M.

5.31 Let $LM_{\mathsf{TM}} = \{\langle M, w\rangle |\ M \text{ ever moves left while computing on } w\}$. LM_{TM} is decidable. Let M_{LEFT} be the TM which on input $\langle M, w\rangle$ determines the number of states n_M of M and then simulates M on w for $|w| + n_M + 1$ steps. If M_{LEFT} discovers that M moves left during that simulation, M_{LEFT} accepts $\langle M, w\rangle$. Otherwise M_{LEFT} rejects $\langle M, w\rangle$.

The reason that M_{LEFT} can reject without simulating M further is as follows. Suppose M does make a left move on input w. Let $p = q_0, q_1, \ldots, q_s$ be the shortest computation path of M on w ending in a left move. Because M has been scanning only blanks (it's been moving right) since state $q_{|w|}$, we may remove any cycles that appear after this state and be left with a legal computation path of the machine ending in a left move. Hence p has no cycles and must have length at most $|w| + n_M + 1$. Hence M_{LEFT} would have accepted $\langle M, w\rangle$, as desired.

5.32 We show that if *BB* is computable then A_{TM} is decidable. Assume that TM F computes *BB*. We construct TM S that decides A_{TM} as follows.

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

1. Construct the following TM M_w which has tape alphabet Γ:

   M_w = "On any input:

   1. Simulate M on w while keeping a count c of the number of steps that are used in the simulation.

2. If and when M halts, write c 1s on the tape and halt.**"**

2. Use F to compute $b = BB(k)$ where k is the number of states of M_w.

3. Run M on w for b steps.

4. *Accept* if M has accepted within that number of steps; otherwise *reject*.**"**

If M accepts w then M_w will halt with c 1s on the tape where c is the number of steps that M ran on w. Moreover, $b \geq c$ because of the way *BB* is defined. Hence S will have run M long enough to see that M accepts and will therefore accept. If M doesn't accept w then S will never see that M accepts and therefore S will reject.

5.33 The PCP over a unary alphabet is decidable. We describe a TM M that decides unary PCP. Given a unary PCP instance

$$P = \left\{ \left[\frac{1^{a_1}}{1^{b_1}}\right], \ldots, \left[\frac{1^{a_n}}{1^{b_n}}\right] \right\}$$

$M =$ **"**On input $\langle P \rangle$:

1. Check if $a_i = b_i$ for some i. If so, *accept*.

2. Check if there exist i, j such that $a_i > b_i$ and $a_j < b_j$.
If so, *accept*. Otherwise, *reject*.**"**

In the first stage, M first checks for a single domino which forms a match. In the second stage, M looks for two dominos which form a match. If it finds such a pair, it can construct a match by picking $(b_j - a_j)$ copies of the i^{th} domino, putting them together with $(a_i - b_i)$ copies of the j^{th} domino. This construction has $a_i(b_j - a_j) + a_j(a_i - b_i) = a_i b_j - a_j b_i$ 1s on the top, and $b_i(b_j - a_j) + b_j(a_i - b_i) = a_i b_j - a_j b_i$ 1s on the bottom. If neither stages of M accept, the problem instance contains dominos with all the upper parts having more (or less) 1s than the lower parts. In such a case no match can exist. Therefore M rejects.

5.34 We construct a computable function that maps a PCP instance to a PCP instance over a binary alphabet ($BPCP$). Therefore $BPCP$ is undecidable.

For any PCP instance A over alphabet $\{a_1, \ldots, a_n\}$, the function encodes each character a_i into binary as 10^i. The function is a mapping reduction from PCP to *BPCP*. For any P in PCP, the converted instance P' is in *BPCP* because a match for P becomes a match for P' using the same set of dominos after conversion. Conversely, if P' is in *BPCP*, P is in PCP, because we can uniquely decode a match in P' to a match in P. Therefore, the function is a mapping reduction from PCP to *BPCP*.

5.35 Any match for *SPCP* starts with a domino that has two equal strings, and therefore is a match all by itself. So we need only check whether the input contains a domino that has two equal strings. If so, *accept*; otherwise, *reject*.

5.36 For this problem, we need to observe only that we can encode binary strings into unary. For example, if $\{0,1\}^* = \{w_1, w_2, w_3, \ldots\}$ then we can represent any w_i as 1^i. Using this encoding we can represent A_{TM} in unary to obtain a unary undecidable language.

Chapter 6

6.1 The following program is in LISP:

```
((LAMBDA (X) (LIST X (LIST (QUOTE QUOTE) X)))
(QUOTE (LAMBDA (X) (LIST X (LIST (QUOTE QUOTE) X)))))
```

6.2 The proof of Theorem 6.7 works for this stronger statement, too.

6.4 We modify the proof of Theorem 4.11 to give the answer to this exercise. Everywhere the original proof refers to a TM use here a TM with an A_{TM} oracle. Everywhere the original proof refers to A_{TM} use here the language A_{TM}'. We reach a similar contradiction in this way.

6.6 These theories are decidable because the universes are finite. Therefore the quantifiers can be tested by brute-force.

6.7 Let $J = \{0w |\ w \in A\} \cup \{1w |\ w \in B\}$. In other words $J = 0A \cup 1B$. Then $A \leq_{\mathrm{m}} J$ with the reduction $f(w) = 0w$ and $B \leq_{\mathrm{m}} J$ with the reduction $f(w) = 1w$.

6.8 Given A, we let $B = \{\langle M, w\rangle |$ oracle TM M with an oracle for A accepts $w\}$. Then $A \leq_{\mathrm{T}} B$ because we can test whether $w \in A$ by determining whether N accepts w where N is the oracle TM that checks whether its input is in its oracle. We can show that $B \not\leq_{\mathrm{T}} A$ by using an proof that is similar to the proof of Theorem 4.11, the undecidability of A_{TM}. Use TMs with an oracle for A in place of TMs in the original proof and use B in place of A_{TM}.

6.9 We build up two incomparable sets A and B to kill off all possible Turing-reductions, $R_1, R_2, \ldots$. The sets start off undefined.

First, extend A to be longer and longer initial portions of the empty set and B to be longer and longer initial portions of A_{TM}. Eventually, R_1 must get the wrong answer when trying to reduce B to A. Then extend A and B interchanging the empty set and A_{TM} until R_1 errs in reducing A to B. Continue with R_2, etc.

6.10 Consider the two languages A_{TM} and $R_{\mathsf{TM}} = \{\langle M, w\rangle |$ TM M halts and rejects input $w\}$. We show that these Turing-recognizable languages aren't separable by a decidable language, using a proof by contradiction. Assume that decidable language C separates A_{TM} and R_{TM}. Construct the following TM T:

$T =$ "On input w:

1. Obtain own description T.

2. Test whether $\langle T, w\rangle \in C$.
3. *Accept* if $\langle T, w\rangle \notin C$.
4. *Reject* if $\langle T, w\rangle \in C$."

Observe that T is a decider. If T accepts w then $\langle T, w\rangle \in A_{\mathsf{TM}} \subseteq C$, but then T rejects w. Similarly, if T rejects w then T accepts w. This contradiction shows that C cannot exist.

6.11 The following oracle TM recognizes $\overline{EQ_{\mathsf{TM}}}$ with an oracle for A_{TM}. Let $\Sigma^* = \{w_1, w_2, \dots\}$ be a listing of all strings over Σ.
"On input $\langle M_1, M_2\rangle$:
1. For $i = 1, 2, \dots$
2. Use the A_{TM} oracle to test whether M_1 accepts w_i and whether M_2 accepts w_i.
3. If the results of these two tests differ, *accept*. Otherwise continue with the next i value."

6.12 The collection of all oracle TMs is a countable set. Thus the collection of all languages that are Turing-recognizable relative to A_{TM} (or relative to any other particular language) is countable, because each such language is recognized by an oracle TM and not oracle TM recognizes more than one language (for any particular oracle). The set of all languages is uncountable, and is therefore larger than any countable set. Hence some language must not be Turing-recognizable relative to A_{TM}.

6.13 We consider the following TM M:
$M =$ "On input P, and instance of the *PCP*:
1. For each possible finite sequence of dominos of P:
2. *Accept* if the sequence forms a match. If not, continue."
To determine whether $P \in$ *PCP* we test whether $\langle M, P\rangle \mathcal{N} A_{\mathsf{TM}}$. If so, a match exists, so $P \in$ *PCP*. Otherwise no match exists, so $P \notin$ *PCP*.

6.14 To compute $\mathrm{K}(x)$ from x, run all descriptions of length up to $|x| + c$ where c is the constant in Theorem 6.27, using the A_{TM} oracle to avoid those which do not halt. The length of the shortest is $\mathrm{K}(x)$.

6.15 We know by Theorem 6.29 that incompressible strings exist of every length, so we only to to find them. To do so we compute $\mathrm{K}(x)$ for each x of length n, using the result of Problem 6.14, until we find an x that isn't compressible.

6.16 Assume to the contrary that $\mathrm{K}(x)$ is computable. Then we can construct TM M which on input n computes $\mathrm{K}(x)$ of all strings x of length n and outputs the first incompressible string it finds. However, for large n, that gives a short description of the incompressible string, a contradiction.

6.17 Assume to the contrary that the set of incompressible strings is decidable. Then we can construct TM M which on input n computes which strings of length n are incompressible and outputs the first such string it finds. However, for large n, that gives a short description of the incompressible string, a contradiction.

6.18 Assume to the contrary that an infinite set of incompressible strings is Turing-recognizable. Then we can construct TM M which on input n finds a string of length at

least n which is incompressible and outputs it. However, for large n, that gives a short description of the incompressible string, a contradiction.

6.19 Let w be an incompressible string of length n. Let z be the first $\lfloor \frac{1}{2} \log n \rfloor$ bits of w. Let the number j be the value of $1z$ as a number in binary. Let x be the first k bits of w where $k = \lfloor \frac{1}{2} \log n \rfloor + j$. Let y be the rest of w. Because $w = xy$ is incompressible $\mathrm{K}(xy) \geq n$. But, $\mathrm{K}(y) \leq (n-k)+c_1$ because $|y| \leq n-k$, and $\mathrm{K}(x) \leq j+c_2$ because we can describe x by giving only the trailing j bit of x and then computing j and hence z from these. Then,

$$\mathrm{K}(x) + \mathrm{K}(y) \leq (n-k) + j + c_1 + c_2 \leq n - (k-j) + c_1 + c_2 \leq n - \frac{\log n}{2} + c_1 + c_2$$

Both c_1 and c_2 are constant so $n - 1 \not\leq n - \frac{\log n}{2} + c_1 + c_2$, for large n. Thus $\mathrm{K}(xy) \not\leq \mathrm{K}(x) + \mathrm{K}(y) + c$.

6.20 First, assume S is Turing-recognizable and derive a contradiction. Let
$M_1 =$ "On input w:

1. Obtain own description $\langle M_1 \rangle$.
2. If $w = \langle M_1 \rangle$ then *accept*.
3. Run the S recognizer on input $\langle M_1 \rangle$.
4. If the recognizer accepts then *accept*."

If $M_1 \in S$ then $L(M_1) = \Sigma^*$. If $M_1 \notin S$ then $L(M_1) = \langle M_1 \rangle$. Either way yields a contradiction. Next, assume $\overline{S}$ is Turing-recognizable and derive a contradiction. Let
$M_2 =$ "On input w:

1. Obtain own description $\langle M_2 \rangle$.
2. Run the S recognizer on input $\langle M_2 \rangle$.
3. If the recognizer accepts and if $w = \langle M_2 \rangle$ then *accept*."

If $M_2 \in S$, then $L(M_2) = \langle M_1 \rangle$. If $M_2 \notin S$, then $L(M_2) = \emptyset$. Either way yields a contradiction.

6.21 **b.** R_1 is definable in $\mathrm{Th}(\mathcal{N}, +)$ by $\phi_1(x) = \forall y \forall z[y{+}z{=}x \rightarrow (\phi_0(y) \vee \phi_0(z))]$.
d. $R_<$ is definable in $\mathrm{Th}(\mathcal{N}, +)$ by $\phi_<(u, v) = \exists x[u{+}x{=}v] \wedge \neg\phi_=(u, v)$.

6.22 Using the terminology of the proof of Theorem 6.3, the recursion theorem, we let $M = q(\langle N \rangle)$ where N is the TM:
$N =$ "On any input:

1. Compute own description $\langle N \rangle$.
2. Output $q(\langle N \rangle)$."

Hence $M = P_{\langle N \rangle}$ prints $\langle N \rangle$ and N prints M. These two TMs are not equal because q outputs a string that is different from its input.

6.23 A fixed point for t is a TM that always loops. It never reaches the q_{accept} or q_{reject} states, so interchanging them has no effect on the machine's behavior.

6.24 Assume to the contrary that $EQ_{\mathsf{TM}} \leq_{\mathrm{m}} \overline{EQ_{\mathsf{TM}}}$ via reduction f. Construct TM M.
$M =$ "On input w:

1. Compute own description $\langle M \rangle$.
2. Compute $f(\langle M, N \rangle) = \langle M', N' \rangle$, where N is the TM that accepts everything.

3. Run both M' and N' on all inputs of length at most $n = |w|$ for n steps or until halt. For each such input, if one of the machines has accepted it within n steps, run the other machine on that input until it also accepts.

4. When (and if) the previous step completes, *accept*."

If $L(M') = L(N')$, then Stage 3 will always run to completion and M will always accept, so $L(M) = L(N)$. If $L(M') \neq L(N')$, then for some input x, one of these machines accepts it and the other doesn't, so Stage 3 won't run to completion on that input, and M won't always accept, so $L(M) \neq L(N)$. Either way, $\langle M, N\rangle \in EQ_{\mathsf{TM}}$ iff $\langle M', N'\rangle \in EQ_{\mathsf{TM}}$, but then f isn't reducing EQ_{TM} to $\overline{EQ_{\mathsf{TM}}}$.

6.27 In any model of ϕ_{eq} the relation R_1 is an equivalence relation. The additional parts of ϕ_{lt} requires that R_2 is a total order without maximal elements on the equivalence classes of R_1. The model $(\mathcal{N}, =, <)$ is a model of ϕ_{lt}. Note that any model of ϕ_{lt} must have an infinite universe.

Chapter 7

7.1 **a.** True
b. False
e. True
f. True

7.2 **a.** False
b. True
e. False
f. False

7.3 We can use the Euclidean algorithm to find the greatest common divisor.

a.

$$\begin{aligned} 10505 &= 1274 \times 8 + 313 \\ 1274 &= 313 \times 4 + 22 \\ 313 &= 22 \times 14 + 5 \\ 22 &= 5 \times 4 + 2 \\ 5 &= 2 \times 2 + 1 \\ 2 &= 1 \times 2 + 0 \end{aligned}$$

The greatest common divisor of 10505 and 1274 is 1. Therefore they are relatively prime.

b.

$$\begin{aligned} 8029 &= 7289 \times 1 + 740 \\ 7289 &= 740 \times 9 + 629 \\ 740 &= 629 \times 1 + 111 \\ 629 &= 111 \times 5 + 74 \\ 111 &= 74 \times 1 + 37 \\ 74 &= 37 \times 2 + 0 \end{aligned}$$

The greatest common divisor of 8029 and 7289 is 37. Therefore they are not relatively prime.

7.4 The table constructed by using the algorithm from Theorem 7.16 is:

	1	2	3	4
1	T	T, R	S	S, R, T
2		R	S	S
3			T	T, R
4				R

Because $table(1, 4)$ contains S, the TM accepts w.

7.5 The formula is not satisfiable. Each assignment of Boolean values to x and y falsifies one of the four clauses.

7.6 P is closed under union. For any two Planguages L_1 and L_2, let M_1 and M_2 be the TMs that decide them in polynomial time. We construct a TM M' that decides $L_1 \cup L_2$ in polynomial 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*. Otherwise, *reject*."

P is closed under concatenation. For any two P languages L_1 and L_2, let M_1 and M_2 be the TMs that decide them in polynomial time. We construct a TM M' that decides $L_1 L_2$ in polynomial time:
M' = "On input $\langle w \rangle$:

1. For each way to cut w into two substrings $w = w_1 w_2$:
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 the possible cuts, *reject*."

Stage 2 runs in polynomial time and is repeated at most $O(n)$ times, so the algorithm runs in polynomial time.

P is closed under complement. For any P language L, let M be the TM that decides it in polynomial time. We construct a TM M' that decides the complement of L in polynomial time:
M' = "On input $\langle w \rangle$:

1. Run M on w.
2. If M accepts, *reject*. If it rejects, *accept*."

7.7 NP is closed under union. For any two NP languages L_1 and L_2, let M_1 and M_2 be the NTMs that decide them in polynomial time. We construct a NTM M' that decides $L_1 \cup L_2$ in polynomial 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*. Otherwise, *reject*."

In both stages 1 and 2, M' uses its nondeterminism when the machines being run make nondeterministic steps. M' accepts w if and only if either M_1 and M_2 accept w. Therefore, M' decides the union of L_1 and L_2. Since both stages take polynomial time, the algorithm runs in polynomial time.

NP is closed under concatenation. For any two NP languages L_1 and L_2, let M_1 and M_2 be the NTMsthat decide them in polynomial time. We construct a NTM M' that decides $L_1 L_2$ in polynomial time:
M' = "On input $\langle w \rangle$:

1. For each way to cut w into two substrings $w = w_1 w_2$:

2. Run M_1 on w_1.
3. Run M_2 on w_2. If both accept, *accept*; otherwise continue with the next choice of w_1 and w_2.
4. If w is not accepted after trying all the possible cuts, *reject*."

In both stages 2 and 3, M' uses its nondeterminism when the machines being run make nondeterministic steps. M' accepts w if and only if w can be expressed as w_1w_2 such that M_1 accepts w_1 and M_2 accepts w_2. Therefore, M' decides the concatenation of L_1 and L_2. Stage 2 runs in polynomial time and is repeated for at most $O(n)$ time, so the algorithm runs in polynomial time.

7.8 Here we give a high-level analysis of the algorithm. This analysis is sufficient to allow us to conclude that it runs in polynomial time. A more detailed analysis would go into the head motion of the TM, but we do not do that here. 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, because each repetition except for the last repetition marks at least one additional node. Each execution of stage 3 uses at most $O(n^3)$ steps because G has at most n nodes and checking each node uses at most $O(n^2)$ steps by examining all adjacent nodes to see whether any have been marked. Therefore in total, stages 2 and 3 take $O(n^4)$ time. Stage 4 uses $O(n)$ steps to scan all nodes. Therefore, the algorithm runs in $O(n^4)$ time and *CONNECTED* is in P.

7.9 We construct a TM M that decides *TRIANGLE* in polynomial time.
$M =$ "On input $\langle G \rangle$ where G is a graph:
1. For each triple of nodes v_1, v_2, v_3 in G:
2. If (v_1, v_2), (v_1, v_3), and (v_2, v_3), are edges of G, *accept*.
3. No triangle has been found in G, so *reject*."

Graph G has at most n nodes so it has at most

$$\binom{m}{3} = \frac{m!}{3!(m-3)!} = O(m^3)$$

triples of nodes. Hence stage 2 will repeat at most $O(m^3)$ times. Each stage runs in polynomial time. Therefore, *TRIANGLE* $\in$ P.

7.10 Test whether a path exists from the start state to each non-accepting state and accept only if no such path exists.

7.11 **a.** The EQ_{DFA} algorithm in Chapter 4 runs in polynomial time. This algorithm takes DFAs A and B and constructs a new DFA C which recognizes the symmetric difference of $L(A)$ and $L(B)$ and then tests E_{DFA}. Thus $L(C) = (L(A) \cap \overline{L(B)}) \cup (\overline{L(A)} \cap L(B))$. The union and intersection constructions for DFAs run in polynomial time by using the Cartesian product procedure. The complementation construction for DFAs simply interchanges the accept and non-accept states and therefore it runs in polynomial time. The procedure for E_{DFA} uses the polynomial time *PATH* procedure to test whether a path exists from the start state to an accept state. Hence the entire procedure for EQ_{DFA} runs in polynomial time.

b. We know that $B \subseteq B^*$ for any language B so DFA A is star-closed iff $(L(A))^* \subseteq L(A)$ which in turn holds iff $(L(A))^* \cap \overline{L(A)} = \emptyset$. Use the polynomial time procedure for DFA complementation to construct a DFA that recognizes $\overline{L(A)}$ and the star construction to construct an NFA that recognizes $(L(A))^*$. Construct the NFA which recognizes

the intersection of these two languages by using the Cartesian product method. The procedure E_{NFA} is identical to the E_{DFA} procedure. Hence this entire algorithm runs in polynomial time.

7.12 A nondeterministic polynomial time algorithm for *ISO* operates as follows:
"On input $\langle G, H\rangle$ where G and H are undirected graphs:

1. Let m be the number of nodes of G and H. If they don't have the same number of nodes, $reject$.
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. Stages 1 and 3 takes polynomial time. Therefore *ISO* $\in$ NP.

7.13 **a. Soundness.** Observe that every resolution step preserves the satisfiability of ϕ, because the same satisfying assignment satisfies the added clause. A formula that contains the empty clause is unsatisfiable, so if such a formula is obtained through a sequence of resolution steps, the original formula must have been unsatisfiable.

Completeness. If ϕ on variables is unsatisfiable, we show that some sequence of resolution steps produces the empty clause. The sequence we have in mind performs all possible resolution steps involving variable x and its negation, then does the same for another variable, until all variables are done. The following claim shows that this works.

Claim: Let ϕ be a formula and select one of its variables x. Let ϕ' be the formula after all resolution steps involving the x and its negation have been performed. Let ψ be ϕ' with all clauses containing x or $xbar$ removed. The if ϕ is unsatisfiable, so is ψ.

The completeness follows because when all variables have been processed in this way, only two outcomes are possible: the empty formula containing no clauses, and the formula containing one empty clause. The former case is satisfiable, so it cannot occur if we started with an unsatisfiable formula. Hence the latter case occurs and resolution must have produced an empty clause. Next we prove the claim.

Suppose ψ is satisfiable. We show that ϕ is satisfiable by finding a way to extend psi's satisfying assignment to include a value for x. All clauses of ϕ that contain x or its negation are of the form $(x \vee \alpha_i)$ or $(xbar \vee \beta_j)$ where the α_i and β_j are conjunctions of literals. Take ψ's satisfying assignment and extend it by setting x to True. If that satisfies ϕ, then we are done. If not, then some α_i evaluates to False in this assignment. But ψ contains clauses $(\alpha_i \vee \beta_j)$ for every i and j, and therefore every β_j must be True. Otherwise some clause $(\alpha_i \vee \beta_j)$ in ψ won't be satisfied in this assignment. Thus we can assign $xbar$ to True (x to False) and obtain a satisfying assignment for ϕ. That completes the proof of the claim.

b. Every resolution step involving two clauses that each contain two literals, can produce a clause that contains at most two literals. Only $O(n^2)$ clauses with two literals can be formed so this process can continue for at most a polynomial number of steps if we start with a 2cnf.

7.14 Let *SAT-CHECK* $= \{\langle\phi\rangle a |\ a$ is a satisfying assignment for $\phi\}$. Here $\langle\phi\rangle$ is a string over $\{0,1\}$ and a is over alphabet $\{\mathtt{T},\mathtt{F}\}$. Obviously *SAT-CHECK* $\in$ P. Let nonerasing homomorphism f be the identity mapping on $\{0,1\} \cup \{\mathtt{T}\}$ and $f(\mathtt{F}) = \mathtt{T}$. Clearly f(*SAT-CHECK*) $= \{\langle\phi\rangle \mathtt{T}^m |\ \phi \in$ *SAT*$\} \in$ P iff *SAT* $\in$ P. Thus if P is closed under nonerasing homomorphism, f(*SAT-CHECK*) $\in$ P and so P $=$ NP. Conversely, if P $=$ NP then we show that P is closed under nonerasing homomorphism. Let f be a

nonerasing homomorphism and let $A \in \mathrm{P}$. First, $f(A) \in \mathrm{NP}$ because $y \in f(A)$ iff $y = f(x)$ for some $x \in A$. Because f is nonerasing, $|y| \geq |x|$, so $x \in A$ is the short certificate for $y \in f(A)$.

7.15 Follow the terminology described in the hint. We can assume that f maps its inputs to strings over A's alphabet. Because A is a unary language, f has at most polynomially many different output values for the (exponentially many) input formulas up to length n. Hence $f(\phi) = f(\psi)$ for many pairs of input strings ϕ and ψ where $\phi \neq \psi$. Moreover, if $f(\phi) = f(\psi)$ then ϕ is satisfiable iff ψ is satisfiable. This observation is the crux of the polynomial time algorithm for *SAT*.

For a formula ϕ on the n Boolean variables $x_1, \ldots, x_n$, consider the tree whose nodes are the reduced formula ϕ_s, where $s \in \{0,1\}^*$ and $|s| \leq n$. The root of the tree is $\phi = \phi_\varepsilon$. The leaves are expressions with only constants. The following recursive algorithm determines whether ϕ_s is satisfiable for any ϕ with n variables and any s.

"On input $\langle \phi_s \rangle$:

1. If $|s| = n$, compute whether the constant expression ϕ_s is satisfied (i.e., true), and return *satisfiable* if so and *unsatisfiable* if not.
2. Compute $f(\phi_s)$ and check whether any previously examined formula mapped to the same value under f. If so, return the recorded value.
3. Recursively test whether ϕ_{s0} and ϕ_{s1} are satisfiable. If either are satisfiable, then return *satisfiable* and otherwise return *unsatisfiable*. In both cases, record the returned value under $f(\phi_s)$.**"**

Every time the algorithm calls itself recursively, it records a value at one new output place of f. Hence it runs in polynomial time.

7.16 Given formula ϕ with variables $x_1, \ldots, x_m$ and clauses $c_1, \ldots, c_k$ construct graph G which has two nodes x_{true} and x_{false} for each variable x and one node c_{node} for each clause c. Place edges in G connecting x_{true} with x_{false}, connecting x_{true} with c_{node} if x appears positively in c, and connecting x_{false} with c_{node} if x appears negatively in c. Lastly place x_{true} and x_{false} in C for every node x.

Show that ϕ is satisfiable iff G with C satisfies the condition of the problem. If ϕ is satisfiable, consider some satisfying assignment. Orient the x_{true}, x_{false} edges to point toward x_{true} if x is assigned true, and toward x_{false} if x is assigned false. Orient the other edges of G according to the rules of the problem so that x_{true} and x_{false} nodes have indegree 0 or outdegree 0. It is easy to verify that every clause node c_{node} will have outdegree at least 1, because every clause has at least one true literal. Similarly, if G's nodes are orientable according to the conditions of the problem, assign x true iff the x_{true}, x_{false} edge points to x_{true}. Each edge incoming to a clause node corresponds to a true literal in the associated clause. Each clause node has indegree 1 or more, so the assignment satisfies every clause.

7.17 To show that $EQ_{\text{SF-REX}} \in \mathrm{coNP}$, we show that the complement of $EQ_{\text{SF-REX}}$ is in NP. Notice that if R is a star-free regular expression, then every string in $L(R)$ is of length at most the number of symbols in R. Thus if $L(R) \neq L(S)$ we can nondeterministically guess a witness w of length at most $\max\{|R|, |S|\}$ such that w is accepted by one of R and S and rejected by the other. This gives the following NP-algorithm for the complement of $EQ_{\text{SF-REX}}$.

$ST =$ **"**On input $\langle R, S \rangle$:

1. If R or S are not star-free regular expressions, *accept*.
2. Construct NFAs N_R and N_S respectively for R and S.

3. Nondeterministically guess w of length at most $\max\{|R|, |S|\}$.
4. Simulate N_R and N_S on w and *accept* if one of them accepts w and the other rejects w."

The conversion from regular expressions to NFAs can be done in linear time. To simulate an NFA N on input w we maintain the set of states that N could be in as N processes w. If at the end of scanning the input, that set contains an accept state of N, we accept and otherwise we reject. This simulation takes at most $O(q|w|)$ time where q is the number of states of the NFA N.

The above argument does not extend to general regular expressions because there exist expressions R and S such that the string of shortest length that belongs to exactly one of R and S is exponentially long in the length of the description of R and S.

7.18 First we show that $Z \in$ DP. Let *GPAIR* $= \{\langle G, k\rangle |\ G$ is a graph and $k \geq 3\}$. Consider the following two languages:

$$\begin{aligned} Z_1 &= \{\langle G_1, k_1, G_2, k_2\rangle |\ G_1 \text{ has a } k_1 \text{ clique}, G_2 \text{ is a graph and } k_2 \geq 3\} \\ &= \textit{CLIQUE} \times \textit{GPAIR} \\ Z_2 &= \{\langle G_1, k_1, G_2, k_2\rangle |\ G_2 \text{ has a } k_2 \text{ clique}, G_1 \text{ is a graph and } k_1 \geq 3\} \\ &= \textit{GPAIR} \times \textit{CLIQUE}, \end{aligned}$$

Clearly Z_1 and Z_2 are in NP, and $Z =$ *CLIQUE* $\times \overline{\textit{CLIQUE}} = Z_1 \cap \overline{Z_2}$, therefore $Z \in$ DP. To show Z is complete for DP we need to show that for all $A \in$ DP, $A \leq_{\mathrm{P}} Z$. But $A \in$ DP means $A = A_1 \cap \overline{A_2}$ for some NP languages A_1, A_2. Now by the NP-completeness of *CLIQUE* we have that $A_1, A_2 \leq_{\mathrm{P}} CLIQUE$; let f_1, f_2 denote the corresponding poly-time reduction mappings. We claim that $f = (f_1, f_2)$ is a poly-time reduction from A to Z.

Indeed, if $w \in A$ then $w \in A_1$, so $f_1(w) = \langle G_1, k_1\rangle \in$ *CLIQUE* and also $w \in \overline{A_2}$, so $f_2(w) = \langle G_2, k_2\rangle \in \overline{\textit{CLIQUE}}$. Hence $f(w) = (f_1(w), f_2(w)) \in Z$. Conversely, if $w \in \overline{A}$ then either $w \in \overline{A_1}$ and $f_1(w) = \langle G_1, k_1\rangle \notin$ *CLIQUE*, or $w \in A_2$ and $f_2(w) = \langle G_2, k_2\rangle \in$ *CLIQUE*. In either case we have $f(w) = (f_1(w), f_2(w)) = \langle G_1, k_1, G_2, k_2\rangle \notin Z$.

7.19 To show that *MAX-CLIQUE* is in DP, we define the language

$$\textit{LARGER-CLIQUE} = \{\langle G, k\rangle |\ G \text{ is a graph with a clique of size} > k\}$$

which is in NP. We have *MAX-CLIQUE* $=$ *CLIQUE* $\cap \overline{\textit{LARGER-CLIQUE}}$ therefore *MAX-CLIQUE* is in DP.

To show DP-completeness, show that $Z \leq_{\mathrm{P}}$ *MAX-CLIQUE* by giving a reduction that takes input $\langle G_1, k_1, G_2, k_2\rangle$ and output $\langle G, k\rangle$, where G has a clique of maximum size k iff G_1 has a k_1-clique and G_2 does not have a k_2-clique. We start with a few simple transformations that allow us to control clique sizes in graphs. Let G be any graph. First, adding l nodes that are connected to each other and to all of G's nodes yields a new graph $G + l$ where G has a k-clique iff $G + l$ has a $k + l$-clique. Second, replacing each of G's nodes by a pair of connected nodes and each of G's edges by four edges connecting the nodes in the pairs yields a new graph $2 \times G$ where G has a k-clique iff $2 \times G$ has a $2k$-clique. Third, replacing each of G's nodes x by b nodes $x_1, \ldots, x_b$ that are not connected to each other and each of G's edges (x, y) by edges (x_i, y_j) for all $i \neq j$ yields a new graph $G * b$ whose largest clique has size $\min(b, k)$ where k is the size of G's largest clique.

The reduction applies the third transformation to obtain the graphs $H_1 = G_1 * k_1$ and $H_2 = G_2 * (k_2 + 1)$ whose largest cliques have size k_1 and $k_2 + 1$ respectively. Next it applies the second transformation to obtain the $H_2' = 2 \times H_2$ whose largest clique can have size $2k_2$ or $2k_2 + 2$ or a smaller value (but not $2k_2 + 1$). Next it uses the first transformation obtain the graphs $J_1 = H_1 + 2k_2$ and $J_2 = H_2' + k_1$. Observe that J_1's maximum clique size is $k_1 + 2k_2$ iff G_1 has a k_1-clique, and that J_2's maximum clique size is $k_1 + 2k_2$ iff G_2 doesn't have a $k_2 + 1$-clique. Hence the graph $K = J_1 \cup J_2$ obtained by combining J_1 and J_2 into one graph without adding any new edges has maximum clique size $k_1 + 2k_2$ iff G_1 has a k_1-clique and G_2 doesn't have a $k_2 + 1$-clique. The reduction outputs $(K, k_1 + 2k_2)$.

7.20 Here is a sketch of the proof. If we run a TM on an input, a *crossing sequence* is the sequence of states that the machine enters at a given cell as the computation progresses. If a TM has all crossing sequences of a fixed length k or less, no matter how long the input is, an NFA simulating it is easily obtained (just have enough states to remember the current crossing sequence and update that information as each successive input symbol is read.) So assume the TM has arbitrarily long crossing sequences. We find some input for which the TM uses too much time.

Take some input that has a crossing sequence of length k for some large k. Say that sequence occurs at cell i. If the TM runs in $o(n \log n)$ time, at least half of the crossing sequences for this input will have length $o(\log n)$. For large n, three of these short crossing sequences will be equal, occurring at positions a, b, and c.

Obtain a new input by removing the portion between a and b, or between b and c, taking care not to remove position i, with the very long crossing sequence. When the TM is run on this new input, the very long crossing sequence still occurs. Repeat this paragraph until the input has length at most $\sqrt{(k)}$. But then the TM uses at least n^2 time, a contradiction.

7.21 First notice that the language of all pairs of Boolean formulas (ϕ_1, ϕ_2) such that ϕ_1 and ϕ_2 are not equivalent is an NP language. We can guess an assignment and check that the formulas differ on this assignment in polynomial time. Assuming P = NP, this language and its complement, the equivalence problem for formulas, are in P.

A formula is not minimal if there exists a smaller formula that is equivalent to it. Thus, given the above, $\overline{\textit{MIN-FORMULA}} \in$ NP because we can guess a smaller formula and check equivalence. Again, assuming P = NP, we have $\overline{\textit{MIN-FORMULA}} \in$ P. and hence *MIN-FORMULA* $\in$ P.

7.23 The NFA operates by guessing which clause is not satisfied by the assignment and then reading the input to check that the clause is indeed not satisfied. More precisely, the NFA contains $l(m + 1) + 1$ states, where l is the number of clauses and m is the number of variables in ϕ. Each clause c_i is represented by $m + 1$ states $q_{i,1}, \ldots, q_{i,m+1}$. The start state q_0 branches nondeterministically to each state $q_{i,1}$ on ε. Each state $q_{i,j}$ branches to $q_{i,j+1}$ for $j \leq m$. The label on that transition is 0 if x_j appears in clause c_i, and is 1 if $\overline{x_j}$ appears in c_i. If neither literals appear in c_i, the label on the transition is 0,1. Each state $q_{i,m+1}$ is an accept state.

If we could minimize this NFA in polynomial time, we would be able to determine whether it accepts all strings of length m, and hence whether ϕ is satisfiable, in polynomial time. That would imply P = NP.

7.24 The clause $(x \vee y)$ is logically equivalent to each of the expressions $(\overline{x} \rightarrow y)$ and $(\overline{y} \rightarrow x)$. We represent the 2cnf formula ϕ on the variables $x_1, \ldots, x_m$ by a directed graph G on $2m$ nodes labeled with the literals over these variables. For each clause in ϕ, place two edges in the graph corresponding to the two implications above. Additionally, place an edge from $\overline{x}$ to x if (x) is a clause and the reverse edge if $(\overline{x})$ is a clause. We show that ϕ is satisfiable iff G doesn't contain a cycle containing both x_i and $\overline{x_i}$ for any i. We'll call such a cycle an ***inconsistency cycle***. Testing whether G contains an inconsistency cycle is easily done in polynomial time with a marking algorithm, or a depth-first search algorithm.

We show that G contains an inconsistency cycle iff no satisfying assignment exists. The sequence of implications in any inconsistency cycle yields the logical equivalence $x_i \leftrightarrow \overline{x_i}$ for some i, and that is contradictory. Thus if G contains an inconsistency cycle, ϕ is unsatisfiable. Next we show the converse. Write $x \stackrel{*}{\rightarrow} y$ if G contains a path from node x to node y. Because G contains the two designated edges for each clause in ϕ, we have $x \stackrel{*}{\rightarrow} y$ iff $\overline{y} \stackrel{*}{\rightarrow} \overline{x}$. If G doesn't contain an inconsistency cycle, we construct ϕ's satisfying assignment.

Pick any variable x_i. We cannot have both $x_i \stackrel{*}{\rightarrow} \overline{x_i}$ and $\overline{x_i} \stackrel{*}{\rightarrow} x_i$ because G doesn't contain an inconsistency cycle. Select literal x_i if $x_i \stackrel{*}{\rightarrow} \overline{x_i}$ is false, and otherwise select $\overline{x_i}$. Assign the selected literal and all implied literals (those reachable along paths from the selected node) to be True. Note that we never assign both x_j and $\overline{x_j}$ True because if $x_i \stackrel{*}{\rightarrow} x_j$ and $x_i \stackrel{*}{\rightarrow} \overline{x_j}$ then $x_j \stackrel{*}{\rightarrow} \overline{x_i}$ and hence $x_i \stackrel{*}{\rightarrow} \overline{x_i}$ and we would not have selected literal x_i (similarly for $\overline{x_i}$). Then, we remove all nodes labeled with assigned literals or their complements from G, and repeat this paragraph until all variables are assigned. The resulting assignment satisfies every implication and hence every clause in. Thus ϕ is satisfiable.

7.25 Let us call two states q_i and q_j *indistinguishable* if for every string z, $\delta(q_i, z)$ and $\delta(q_j, z)$ are either both accept or both reject (i.e. starting in state q_i or q_j and following any string z will result in identical accept/reject behavior). The algorithm *minimize* puts edges between any two vertices v_i and v_j where following a string from the state corresponding to v_i will result in acceptance and following the same string from the state corresponding to v_j will result in rejection. At the end of the algorithm *minimize*, every two vertices in the graph G that are not adjacent correspond to two states in the original DFA that are indistinguishable. Thus, we can build a new DFA M' by grouping states which are all indistinguishable from one another. Our new DFA only needs to remember which group of indistinguishable states we are in.

First we prove that for any two vertices v_i and v_j, (v_i, v_j) is an edge in G if and only if there exists a string z taking v_i to accept and v_j to reject (or vice versa). The proof is by induction on when the edge was created in the algorithm $minimize$. If the edge was initially there then the string $z = \epsilon$. Assume the edge was added at stage k. Then at stage $k-1$ there is an edge between $\delta(v_i, a)$ and $\delta(v_j, a)$. By the induction hypothesis there must be a string y taking $\delta(v_i, a)$ to accept and $\delta(v_j, a)$ to reject (or vice versa). Thus the string ay takes v_i to accept and v_j to reject or vice versa. For the reverse implication we proceed by induction on the length of z. If z is empty then certainly v_i and v_j are connected since that is the initial condition of G. Assume z has length k. After reading one symbol of z, v_i is in state v_i' and v_j is in state v_j'. By induction, v_i' and v_j' must have an edge connecting them since the rest of z takes one state to accept and the other to reject. From state 6 of the algorithm, v_i must be connected to v_j. This proves that any two states q_i, $q_j \in [q]$ for any $[q]$ are indistinguishable.

We now prove that for any x, M accepts x if and only if M' accepts x. The proof is by induction on the length of x. We strengthen our induction hypothesis to say that if M transitions through states $q_0, \ldots, q_k$ on input x then the states M' transitions through are $[q_0], \ldots, [q_k]$. The base case, when the length of the string is 0, is trivial. Assume M accepts a string x of length k and ends up in state q_{accept}. Consider the state of M after reading the first $k-1$ symbols of x. Call this state q. Then after reading $k-1$ symbols of x, by the induction hypothesis, M' must be in state $[q]$. Since $[q]$ contains all of the states indistinguishable from q, M' will accept x if and only if M accepts x. Further, M' cannot transition to some state $[q'] \neq [q_{accept}]$ as this would violate the indistinguishability of states within $[q]$.

We now show that M' is minimal. From Problem 1.40 it is enough to show that the index of the language accepted by M is equal to the number of states of M'. We know that the language of M equals the language of M' so from part (a) of Problem 1.40, we know that the index of L is less than or equal to the size of M'. We need to show that the index is greater than or equal to the size of M'. Consider the set of strings S of size $|M'|$ where the ith element of S is a string that takes M' to state i from the start state. Since M has no useless states, such a set exists. By the construction of M' for any two states $v_i \neq v_j$ there must be a string z taking v_i to accept and v_j to reject (or vice versa). Hence S is pairwise distinguishable by L. Thus the index is at least $|M'|$ and thus equal to $|M'|$.

Finally we show that this algorithm runs in polynomial time. At each iteration we consider all pairs of nodes in G, this takes time $O(n^2)$. If we add no new edges we stop. We add at least 1 edge at each iteration so we can have at most $O(n^2)$ iterations. Thus the algorithm works in polynomial time.

7.26 In the proof of the Cook–Levin theorem, we argue that if every every 2×3 window is legal, the entire table corresponds to a computation of the NTM, i.e., it is a tableau. If we had used 2×2 windows instead, we would not have been able reach that conclusion, because all such windows may be legal yet the table isn't a tableau. For example, if the table corresponds to a situation where the TMs has 2 nondeterministic possibilities, one moving rightward and the other moving leftward, but both are simultaneously represented by having the state corresponding to the head position "split" into 2 states in the next row, then all 2×2 windows may be legal but obviously the table isn't. Note that the 2×3 window whose upper center cell is at the state before the split occurs would not be legal.

7.27 *MCP* $= \{\langle G, N\rangle \mid G$ is a graph (V, E), N is a function $N\colon V \longrightarrow \{*, 0, 1, 2, \ldots\}$, and a placement of mines on starred nodes is possible, so that all numbers are consistent$\}$.

MCP is in NP because we can guess an assignment of mines and verify that it is consistent in poly-time. We reduce *3SAT* to *MCP* in poly-time. Given a Boolean formula ϕ, convert it to a minesweeper configuration as follows. For each variable x, add nodes called $x, \overline{x}, x_{\text{mid}}$, and edges $(x, x_{\text{mid}}), (x_{\text{mid}}, \overline{x})$. Set $N(x_{\text{mid}}) = 1$ and $N(x) = N(\overline{x}) = *$. For each clause c add nodes c_1, c_2, c_3 and edges $(c_1, c_2), (c_1, c_3)$. Set $N(c_1) = 3$ and $N(c_2) = N(c_3) = *$. Connect c_1 to the nodes corresponding to the literals in clause c.

Given a satisfying assignment for ϕ, a consistent mine assignment can be constructed as follows. For each variable x assigned True (False), put a mine on node x ($\overline{x}$) and no mine on node $\overline{x}$ (x). This ensures that exactly one neighbor of x_{mid} has a mine. For every clause c, the number of mine-carrying neighbors of c_1 is equal to the number of true literals in c. Since ϕ is satisfied, this number is 1, 2, or 3. Assign

mines to c_2 and c_3 if necessary to bring the number of mine-carrying neighbors of c_1 to 3.

Given a consistent mine assignment, a satisfying assignment for ϕ can be constructed as follows. For each variable x, exactly one of nodes x and $\overline{x}$ carries a mine because $N(x_{\text{mid}}) = 1$ is satisfied. Assign x True (False) if x ($\overline{x}$) carries a mine. Each clause c is satisfied because c_1 has three neighbors with mines. At least one of them is neither c_2 nor c_3. Hence, it has to be a node that corresponds to a literal in clause c. That literal is set to True, and therefore the clause is satisfied.

7.29 *SET-SPLITTING* is in NP because a coloring of the elements can be guesses and verified to have the desired properties in polynomial time. We give a polynomial time reduction f from *3SAT* to *SET-SPLITTING* as follows.

Given a 3cnf ϕ, we set $S = \{x_1, \overline{x_1}, \ldots, x_m, \overline{x_m}, y\}$. In other words, S contains an element for each literal (two for each variable) and a special element y. For every clause c_i in ϕ, let C_i be a subset of S containing the elements corresponding to the literals in c_i and the special element $y \in S$. We also add subsets C_{x_i} for each literal, containing elements x_i and $\overline{x_i}$. Then $C = \{C_1, \ldots, C_l, C_{x_1}, \ldots, C_{x_l}\}$.

If ϕ is satisfiable, consider a satisfying assignment. If we color all the true literals red, all the false ones blue, and y blue, then every subset C_i of S has at least one red element (because c_i of ϕ is "turned on" by some literal) and it also contains one blue element (y). In addition, every subset C_{x_i} has exactly one element red and one blue, so that the system $\langle S, C\rangle \in$ *SET-SPLITTING*.

If $\langle S, C\rangle \in$ *SET-SPLITTING*, then look at the coloring for S. If we set the literals to true which are colored differently from y, and those to false which are the same color as y we obtain a satisfying assignment to ϕ.

7.30 *SCHEDULE* $= \{\langle F, S, h\rangle |$ F is a list of finals, S is a list of subsets of finals each student is taking, and finals can be scheduled into h slots so that no student is taking two exams in the same slot$\}$.

SCHEDULE is in NP because we can guess a schedule with h slots and verify in polynomial time that no student is taking two exams in the same slot. We reduce *3COLOR* to *SCHEDULE* in polynomial time. Given a graph (V, E), transform it into $\langle F, S, h\rangle$ where $F = V$, $S = E$, $h = 3$.

Given a 3-coloring for the original graph, we can get a schedule for $\langle V, E, 3\rangle$ by assigning finals that correspond to nodes colored with color i to slot i for $(i = 1, 2, 3)$. Adjacent nodes are colored by different colors, so corresponding exams taken by the same student are assigned to different slots.

Given a schedule for $\langle V, E, 3\rangle$, we can get a 3-coloring for the original graph by assigning colors according to the slots of corresponding nodes. Since exams taken by the same student are scheduled in different slots, adjacent nodes are assigned different colors.

7.31 To show that all problems in NP are poly-time reducible to D we show that *3SAT* $\leq_{\text{P}}$ D. Given a 3cnf formula ϕ construct a polynomial p with the same variables. Represent the variable x in ϕ by x in p, and its negation $\overline{x}$ by $1 - x$. Represent the Boolean operations $\wedge$ and $\vee$ in ϕ by arithmetic operations that simulate them, as follows. The Boolean expression $y_1 \wedge y_2$ becomes $y_1 y_2$ and $y_1 \vee y_2$ becomes $(1 - (1 - y_1)(1 - y_2))$. Hence, the clause $(y_1 \vee y_2 \vee y_3)$ becomes $(1-(1-y_1)(1-y_2)(1-y_3))$. The conjunction of the clauses of ϕ becomes a product of their individual representations. The result of this transformation is a polynomial q which simulates ϕ in the sense they both have the

same value when the variables have Boolean assignments. Thus the polynomial $1 - q$ has an integral (in fact Boolean) root if ϕ is satisfiable. However, $1 - q$ does not meet the requirements of p, because $1 - q$ may have an integral (but non-Boolean) root even when ϕ is unsatisfiable. To prevent that situation, we observe that the polynomial $r = (x_1(1-x_1))^2(x_2(1-x_2))^2 \cdots (x_m(1-x_m))^2$ is 0 only when the variables $x_1, \ldots, x_l$ take on Boolean values, and is positive otherwise. Hence polynomial $p = (1-q)^2 + r$ has an integral root iff ϕ is satisfiable.

7.33 The following solution to this difficult problem is due to Victor Kuncak. We give a polynomial time reduction from *3SAT* to K, the language describing the set of inputs for which a DFA exists with the specified properties. Clearly K is in NP. Next we describe the reduction.

Given a formula ϕ with m variables, output a collection of states Q and pairs $\{(s_1, r_1), \ldots, (s_k, r_k)\}$. The pairs are designed to restrict the form of the DFA. We write the pair (s_i, r_i) as $s_i \to r_i$. We give convenient names for the states, $Q = \{q_0, \ldots, q_m, h, v_t, v_f, l_t, l_f, s, r\}$. ($v_t$ stands for variable true, v_f for variable false, l_t and l_f for literal true and false, s for satisfied) Observe we can force a transition arrow with label 0 from state t to state u by including pairs $w \to t$ and $w0 \to u$ for some string w. Here are the pairs along with the transitions they force.

i) $1^k \to q_k$, for $0 \le k < m$ forces $\delta(q_k, 1) = q_{k+1}$
ii) $1^{n+2} \to q_0$ forces $\delta(m, 1) = q_0$
iii) $0 \to h$ forces $\delta(q_0, 0) = h$
iv) $00 \to v_t$ forces $\delta(q_0, 0) = v_t$
v) $01 \to v_f$ forces $\delta(q_0, 0) = v_f$
vi) $000 \to l_t$ forces $\delta(q_0, 0) = l_t$
vii) $001 \to l_f$ forces $\delta(q_0, 0) = l_f$
viii) $010 \to l_f$ forces $\delta(q_0, 0) = l_f$
ix) $011 \to l_t$ forces $\delta(q_0, 0) = l_t$
x) $0000 \to s$ forces $\delta(q_0, 0) = s$
xi) $0001 \to r$ forces $\delta(q_0, 0) = r$
xii) $0010 \to q_0$ forces $\delta(q_0, 0) = q_0$
xiii) $0011 \to r$ forces $\delta(q_0, 0) = r$
xiv) $00000 \to s$ forces $\delta(q_0, 0) = s$
xv) $00001 \to s$ forces $\delta(q_0, 0) = s$
xvi) $00010 \to q_0$ forces $\delta(q_0, 0) = q_0$
xvii) $00011 \to q_0$ forces $\delta(q_0, 0) = q_0$

The 0-transition from the q_i states (for $i > 0$) encode the assignment, going to v_t indicates TRUE and v_f indicates FALSE. We force one or the other to occur by adding the pairs $1^k011 \to r$, because v_t and v_f are the only states that can reach r by reading two 1s. Consider a clause $(l_1 \vee l_2 \vee l_3)$ in ϕ where each l_i is a literal x_j or $\overline{x_j}$. We force that clause to be satisfied by adding the pair $t_1t_2t_3 \to s$ where each $t_i = 1^j0b0$ and $b = 0$ if $l_i = x_j$ and $b = 1$ if $l_i = \overline{x_j}$.

7.34 U is in NP because on input $\langle d, x, 1^t \rangle$ a nondeterministic algorithm can simulate M_d on x (making nondeterministic branches when M_d does) for t steps and accept if M_d accepts. Doing so takes time polynomial in the length of the input because t appears in unary. To show *3SAT* $\le_{\mathrm{P}} U$, let M be a NTM that decides *3SAT* in time cn^k for some constants c and k. Given a formula ϕ, transform it into $w = \langle \langle M \rangle, \phi, 1^{c|\phi|^k} \rangle$. By the definition of M, $w \in U$ if and only if $\phi \in$ *3SAT*.

7.35 Assuming P = NP, we have a polynomial time algorithm deciding *SAT*. To produce a satisfying assignment for a satisfiable formula ϕ, substitute $x_1 = 0$ and $x_1 = 1$ in ϕ and test the satisfiability of the two resulting formulas ϕ_0 and ϕ_1. At least one of these must be satisfiable because ϕ is satisfiable. If ϕ_0 is satisfiable, pick $x_1 = 0$; otherwise ϕ_1 must be satisfiable and pick $x_1 = 1$. That gives the value of x_1 in a satisfying assignment. Make that substitution permanent and determine a value for x_2 in a satisfying assignment in the same way. Continue until all variables been substituted.

7.36 Let $F = \{\langle a, b, c\rangle |\ a, b, c$ are binary integers and $a = pq$ for $b \leq p \leq c\}$. We see that $F \in$ NP because a nondeterministic algorithm can guess p and check that p divides a and that $b \leq p \leq c$ in polynomial time. If P = NP then $F \in$ P. In that case, we can locate a factor of a by successively running F on smaller and smaller intervals b, c, in effect performing a binary search on the interval 1 to a. The running time of that procedure is the log of the size of the interval, and thus is polynomial in the length of a.

7.38 *3COLOR* is in NP because a coloring can be verified in polynomial time. We show *3SAT* $\leq_{\mathrm{P}}$ *3COLOR*. Let $\phi = c_1 \wedge c_2 \wedge \cdots \wedge c_l$ be a 3cnf formula over variables $x_1, x_2, \cdots, x_m$, where the c_i's are the clauses. We build a graph G_ϕ containing $2m + 6l + 3$ nodes: 2 nodes for each variable; 6 nodes for each clause; and 3 extra nodes. We describe G_ϕ in terms of the subgraph gadgets given in the hint.

G_ϕ contains a variable gadget for each variable x_i, two OR-gadgets for each clause, and one palette gadget. The four bottom nodes of the OR-gadgets will be merged with other nodes in the graph, as follows. Label the nodes of the palette gadget T, F, and R. Label the nodes in each variable gadget $+$ and $-$ and connect each to the R node in the palette gadget. In each clause, connect the top of one of the OR-gadgets to the F node in the palette. Merge the bottom node of that OR-gadget with the top node of the other OR-gadget. Merge the three remaining bottom nodes of the two OR-gadgets with corresponding nodes in the variable gadgets so that if a clause contains the literal x_i, one of its bottom nodes is merged with the $+$ node of x_i whereas if the clause contains the literal $\overline{x_i}$, one of its bottom nodes is merged with the $-$ node of x_i.

To show the construction is correct, we first demonstrate that if ϕ is satisfiable, the graph is 3-colorable. The three colors are called T, F, and R. Color the palette with its labels. For each variable, color the $+$ node T and the $-$ node F if the variable is True in a satisfying assignment; otherwise reverse the colors. Because each clause has one True literal in the assignment, we can color the nodes of the OR-gadgets of that clause so that the node connected to the F node in the palette is not colored F. Hence we have a proper 3-coloring.

If we start out with a 3-coloring, we can obtain a satisfying assignment by taking the colors assigned to the $+$ nodes of each variable. Observe that neither node of the variable gadget can be colored R, because all variable nodes are connected to the R node in the palette. Furthermore, if both bottom nodes of an OR-gadget are colored F, the top node must be colored F, and hence, each clause contain a true literal. Otherwise, the three bottom nodes that were merged with variable nodes would be colored F, and then both top nodes would be colored F, but one of the top nodes is connected to the F node in the palette.

7.39 *PUZZLE* is in NP because we can guess a solution and verify it in polynomial time. We reduce *3SAT* to *PUZZLE* in polynomial time. Given a Boolean formula ϕ, convert it to a set of cards as follows.

Let $x_1, x_2, \ldots, x_m$ be the variables of ϕ and let $c_1, c_2, \ldots, c_l$ be the clauses. Say that $z \in c_j$ if z is one of the literals in c_j. Let each card have column 1 and column 2 with l possible holes in each column numbered 1 through l. When placing a card in the box, say that a card is *face up* if column 1 is on the left and column 2 is on the right; otherwise *face down*. The following algorithm F computes the reduction.

$F =$ "On input ϕ:

1. Create one card for each x_i as follows: in column 1 punch out all holes except any hole j such that $x_i \in c_j$; in column 2 punch out all holes except any hole j' such that $\overline{x_i} \in c_{j'}$.
2. Create an *extra* card with all holes in column 1 punched out and no hole in column 2 punched out.
3. Output the description of the cards created."

Given a satisfying assignment for ϕ, a solution to the *PUZZLE* can be constructed as follows. For each x_i assigned True (False), put its card face up (down), and put the extra card face up. Then, every hole in column 1 is covered because the associated clause has a literal assigned True, and every hole in column 2 is covered by the extra card.

Given a solution to the *PUZZLE*, a satisfying assignment for ϕ can be constructed as follows. Flip the deck so that the extra card covers column 2. Assign the variables according to the corresponding cards, True for up and False for down. Because every hole in column 1 is covered, every clause contains at least one literal assigned True in this assignment. Hence it satisfies ϕ.

7.40 The most obvious algorithm multiplies a by itself b times then compares the result to c, modulo p. That uses $b-1$ multiplications and so is exponential in the length of b. Hence this algorithm doesn't run in polynomial time. Here is one that does:

"On input $\langle a, b, c, p \rangle$, four binary integers:

1. Let $r \leftarrow 1$.
2. Let $b_1, \ldots, b_k$ be the bits in the binary representation of b.
3. For $i = 1, \ldots, k$:
4. If $b_i = 0$, let $r \leftarrow r^2 \mod p$.
5. If $b_i = 1$, let $r \leftarrow ar^2 \mod p$.
6. If $(c \mod p) = (r \mod p)$, *accept*; otherwise *reject*."

The algorithm is called *repeated squaring*. Each of its multiplications and modular operations can be done in polynomial time in the standard way, because the r is never larger than p. The total number of multiplications is proportional to the number of bits in b. Hence the total running time is polynomial.

7.41 First, note that we can compose two permutations in polynomial time. Composing q with itself t times requires t compositions if done directly, thereby giving a running time that is exponential in the length of t when t is represented in binary. We can fix this problem using a common technique for fast exponentiation. Calculate $q^1, q^2, q^4, q^8, \ldots, q^k$, where k is the largest power of 2 that is smaller than t. Each term is the square of the previous term. To find q^t we compose the previously computed permutations corresponding to the 1s in t's binary representation. The total number of compositions is now at most the length of t, so the algorithm runs in polynomial time.

7.42 We show that P is closed under the star operation by dynamic programming. Let $A \in \mathrm{P}$, and M be the TM deciding A in polynomial time. On input $w = w_1 \cdots w_n$, use the following procedure *ST* to construct a table T where $T[i, j] = 1$ if $w_i \cdots w_j \in A^*$ and

$T[i,j] = 0$ otherwise, for all $1 \leq i \leq j \leq n$.
ST = "On input $w = w_1 \cdots w_n$:

1. If $w = \varepsilon$, *accept*. [[handle $w = \varepsilon$ case]]
2. Initialize $T[i,j] = 0$ for $1 \leq i \leq j \leq n$.
3. For $i = 1$ to n, [[test each substring of length 1]]
4. Set $T[i,i] = 1$ if $w_i \in A$.
5. For $l = 2$ to n, [[l is the length of the substring]]
6. For $i = 1$ to $n - l + 1$, [[i is the substring start position]]
7. Let $j = i + l - 1$, [[j is the substring end position]]
8. If $w_i \cdots w_j \in A$, set $T[i,j] = 1$.
9. For $k = i$ to $j - 1$, [[k is the split position]]
10. If $T[i,k] = 1$ and $T[k+1,j] = 1$, set $T[i,j] = 1$.
11. *Accept* if $T[1,n] = 1$; otherwise *reject*."

Each stage of the algorithm takes polynomial time, and *ST* runs for $O(n^3)$ stages, so the algorithm runs in polynomial time.

7.44 The reduction fails because it would result in an exponential unary instance, and therefore would not operate in polynomial time.

We give a polynomial time algorithm for this problem using dynamic programming. The input to the algorithm is $\langle S, t \rangle$ where S is a set of numbers $\{x_1, \ldots, x_k\}$. The algorithm operates by constructing lists L_i giving the possible sums of subsets of $\{x_1, \ldots, x_i\}$. Initially $L_0 = \{0\}$. For each i the algorithm constructs L_i from L_{i-1} by assigning $L_i = L_{i-1} \cup \{a + x_i | \ a \in L_{i-1}\}$. Finally, the algorithm accepts if $t \in L_k$, and rejects otherwise.

7.45 Let A be any language in P except $\emptyset$ and Σ^*. To show that A is NP-complete, we show that $A \in$ NP, and that every $B \in$ NP is polynomial time reducible to A. The first condition holds because $A \in$ P so $A \in$ NP. To demonstrate that the second condition is true, let $x_{\text{in}} \in A$ and $x_{\text{out}} \notin A$. The assumption that P = NP implies that B is recognized by a polynomial time TM M. A polynomial time reduction from B to A simulates M to determine whether its input w is a member of B, then outputs x_{in} or x_{out} accordingly.

7.46 We make use of the following number theoretic fact: an integer p is prime if and only if there exists a generator g of the multiplicative group of Z_p such that $g^{p-1} \equiv 1 \bmod p$ and for all prime divisors d of $p - 1$, $g^{p-1/d} \not\equiv 1 \bmod p$. This fact follows from two observations: (1) the multiplicative group Z_p is cyclic and has a generator and (2) $\phi(p)$ is at most $p - 1$ and is equal to $p - 1$ if and only if p is prime where $\phi()$ is the Euler totient function.

We now prove that there exists a polynomial length proof that an integer m is prime. Our proof begins with a generator g of the multiplicative group Z_m and the prime factorization of $m - 1$. Then we can verify that g is a generator by checking that $g^{m-1} \equiv 1 \bmod n$ and that $g^{m-1/d} \not\equiv 1 \bmod n$ for all prime divisors of $m - 1$. The prime factorization of $m - 1$ (which has at most $O(\log m)$ factors) is included in our proof. We prove that the factors of $m - 1$ are prime recursively, by including these subproofs within our proof. We must argue that the length of the entire proof resulting from this recursive construction is polynomial in $\log m$.

The depth of the recursion is no more than $\log m$. The recursive calls starting from an integer k are for integers of size at most $k/2$ because they are factors of k. Furthermore,

the width of our recursive process is at most $\log m$, because the product of the leaves of the recursion tree is at most m. Hence the total size of the recursion tree is $\log^2 m$. The length of the entire proof, which consists of the transcript of the generators and prime factorizations made at each step in the recursion is then $O(\log^3 m)$.

7.47 We believe that *PATH* isn't NP-complete because *PATH* $\in$ P and hence if *PATH* were NP-complete then P $=$ NP. But *proving* that *PATH* is not NP-complete is another matter. Problem 7.44 shows that if P $=$ NP then *PATH* would be NP-complete. Hence if we could prove that *PATH* isn't NP-complete, we would have shown that P $\neq$ NP.

7.48 **a.** The marking algorithm for recognizing *PATH* can be modified to keep track of the length of the shortest paths discovered. Here is a detailed description of the algorithm.

"On input $\langle G, a, b, k\rangle$ where m-node graph G has nodes a and b:

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

b. First, *LPATH* $\in$ NP because we can guess a simple path of length at least k from s to t and verify it in polynomial time. Next, *UHAMPATH* $\leq_{\mathrm{P}}$ *LPATH*, because the following TM F computes the reduction f.

$F =$ "On input $\langle G, a, b\rangle$, where a and b are nodes of graph G:

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

If $\langle G, a, b\rangle \in$ *UHAMPATH*, then G contains a Hamiltonian path of length k from a to b, so $\langle G, a, b, k\rangle \in$ *LPATH*. If $\langle G, a, b, k\rangle \in$ *LPATH*, then G contains a simple path of length k from a to b. But G has only k nodes, so the path is Hamiltonian. Thus $\langle G, a, b\rangle \in$ *UHAMPATH*.

7.49 On input ϕ, a nondeterministic polynomial time machine can guess two assignments and accept if both assignments satisfy ϕ. Thus *DOUBLE-SAT* is in NP. We show *SAT* $\leq_{\mathrm{P}}$ *DOUBLE-SAT*. The following TM F computes the polynomial time reduction f.

$F =$ "On input $\langle \phi\rangle$, 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 $\phi \in$ *SAT*, then ϕ' has at least two satisfying assignments that can be obtained from the original satisfying assignment of ϕ by changing the value of x. If $\phi' \in$ *DOUBLE-SAT*, then ϕ is also satisfiable, because x does not appear in ϕ. Therefore $\phi \in$ *SAT* iff $f(\phi) \in$ *DOUBLE-SAT*.

7.51 **a.** To test whether ϕ is satisfiable, repeat the following with each of its variables x.

i) If x appears only once, or twice in the same clause, remove the clause it which it appears and continue.

ii) If x appears only positively or only negatively in two clauses, remove these two clauses and continue.

iii) If x appears positively in one clause $(x \vee y)$ and negatively in one clause $(\overline{x} \vee z)$, replace these two clauses with $(y \vee z)$. If the positive appearance occur in a clause with one literal (x), replace both clauses with the clause (z). Similarly if the negative appearance occur in a clause with one literal $(\overline{x})$, replace both clauses with the clause

(y). If both positive and negative appearances appear in clauses with one literal, *reject*.

iv) If all clauses are removed, *accept*.

b. We reduce *3SAT* to *CNF*$_3$. Let ϕ be an arbitrary 3cnf-formula. For each variable x that appears $k > 3$ times in ϕ, we replace x with k new variables, say $x_1, x_2, \ldots, x_k$, and add the clauses

$$(\overline{x}_1 \vee x_2) \wedge (\overline{x}_2 \vee x_3) \wedge \cdots \wedge (\overline{x}_k \vee x_1),$$

which are equivalent to

$$(x_1 \rightarrow x_2) \wedge (x_2 \rightarrow x_3) \wedge \cdots \wedge (x_k \rightarrow x_1).$$

Doing so enforces the restriction that the new variables all take the same truth value. It is easy to see that this transformation does not affect satisfiability. Moreover, in the new formula x does not appear at all, while each of $x_1, x_2, \ldots, x_k$ appears exactly 3 times: twice in the "cycle" above and once as a replacement of x. The new formula we obtain in this way is satisfiable iff ϕ is. The transformation clearly can be done in polynomial time.

7.52 The Boolean expression $(\overline{x} \vee (\ldots))$ is equivalent to $(x \rightarrow (\ldots))$. The following algorithm shows *CNF*$_{\mathrm{H}} \in \mathrm{P}$. It proceeds by simplifying a formula without changing whether it is satisfiable, until the result is obviously satisfiable or unsatisfiable. An *empty clause* is a clause $()$ with no literals. An empty clause is not satisfiable. A *singleton clause* is a clause (x) or $(\overline{x})$ that has only one literal.

"On input $\langle\phi\rangle$:

1. If ϕ contains an empty clause, *reject*.
2. If ϕ contains no singleton clauses, *accept*.
3. If ϕ contains a singleton clause (x), remove it and all other clauses that contain x.
4. If ϕ contains a singleton clause $(\overline{x})$, remove it and remove x from all other clauses.
5. Repeat the above stages."

The algorithm operates in polynomial time because every stage either terminates or makes ϕ shorter. It operates correctly because in Stage 2 we can satisfy ϕ by setting all of its (remaining) variables positively. Observe that Stage 3 preserves satisfiability by setting x positively and Stage 4 preserves satisfiability by setting x negatively.

7.53 **a.** In an $\neq$-assignment each clause has at least one literal assigned 1 and at least one literal assigned 0. The negation of an $\neq$-assignment preserves this property, so it too is an $\neq$-assignment.

b. To prove that the given reduction works, we need to show that if the formula ϕ is mapped to ϕ', then ϕ is satisfiable (in the ordinary sense) iff ϕ' has an $\neq$-assignment. First, if ϕ is satisfiable, we can obtain an $\neq$-assignment for ϕ' by extending the assignment to ϕ so that we assign z_i to 1 if both literals y_1 and y_2 in clause c_i of ϕ are assigned 0. Otherwise we assign z_i to 0. Finally, we assign b to 0. Second, if ϕ' has an $\neq$-assignment we can find a satisfying assignment to ϕ as follows. By part (a) we may assume the $\neq$-assignment assigns b to 0 (otherwise, negate the assignment). This assignment cannot assign all of y_1, y_2, and y_3 to 0, because doing so would force one of the clauses in ϕ' to have all 0s. Hence, restricting this assignment to the variables of ϕ gives a satisfying assignment.

c. Clearly, $\neq$*SAT* is in NP. Hence, it is NP-complete because *3SAT* reduces to it.

7.54 First, *MAX-CUT* $\in$ NP because a nondeterministic algorithm can guess the cut and check that it has size at least k in polynomial time.

Second, we show $\neq$*3SAT* $\leq_{\mathrm{P}}$ *MAX-CUT*. Use the reduction suggested in the hint. For clarity, though, use c instead of k for the number of clauses. Hence, given an instance ϕ of $\neq$*3SAT* with v variables and c clauses. We build a graph G with $6cv$ nodes. Each variable x_i corresponds to $6c$ nodes; $3c$ labeled x_i and $3c$ labeled $\overline{x_i}$. Connect each x_i node with each $\overline{x_i}$ node for a total of $9c^2$ edges. For each clause in ϕ select three nodes labeled by the literal in that clause and connect them by edges. Avoid selecting the same node more than once (each label has $3c$ copies, so we cannot run out of nodes). Let k be $9c^2v + 2c$. Output $\langle G, k\rangle$. Next we show this reduction works.

We show that ϕ has a $\neq$-assignment iff G has a cut of size at least k. Take an $\neq$-assignment for ϕ. It yields a cut of size k by placing all nodes corresponding to true variables on one side and all other nodes on the other side. Then the cut contains all $9c^2v$ edges between literals and their negations and two edges for each clause, because it contains at least one true and one false literal, yielding a total of $9c^2v + 2c = k$ edges.

For the converse, take a cut of size k. Observe that a cut can contain at most two edges for each clause, because at least two of the corresponding literals must be on the same side. Thus, the clause edges can contribute at most $2c$ to the cut. The graph G has only $9c^2v$ other edges, so if G has a cut of size k, all of those other edges appear in it and each clause contributes its maximum. Hence, all nodes labeled by the same literal occur on the same side of the cut. By selecting either side of the cut and assigning its literals true, we obtain an assignment. Because each clause contributes two edges to the cut, it must have nodes appearing on both sides and so it has at least one true and one false literal. Hence ϕ has a $\neq$-assignment.

A minor glitch arises in the above reduction if ϕ has a clause that contains both x and $\overline{x}$ for some variable x. In that case G would need to be a multi-graph, because it would contain two edges joining certain nodes. We can avoid this situation by observing that such clauses can be removed from ϕ without changing its $\neq$-satisfiability.

Chapter 8

8.1 We can convert a standard deterministic, one-tape TM M_1 to an equivalent deterministic, two-tape, read only input TM M_2 as follows. TM M_2 first copies its input on tape 1 to tape 2, then simulates M_1 on tape 2. These two TMs use exactly the space because we consider only the cells that are used on tape 2 when measuring M_2's space complexity.

Conversely, we can convert a deterministic, two-tape, read only input TM M_2 to an equivalent standard deterministic, one-tape TM M_1 using the standard simulation of two-tape TMs by one-tape TMs in the proof of Theorem 3.13. Then if M_2 uses $O(f(n))$ space, M_1 uses $n + O(f(n))$ space because on M_2 we don't count the input tape, but on M_1 we do count the n cells that are used to simulate that tape. We are assuming that $f(n) \geq n$, so $n + O(fn))$ is $O(f(n))$ and both TMs have the same asymptotic space complexity.

8.2 Let's number the locations in the game 1 thru 9, going across the columns from left to right, then down the rows. In the given board position, $\times$ occupies locations 1 and 9, and $\bigcirc$ occupies locations 5 and 7. In a winning strategy for the $\times$-player, she begins at location 3. If the $\bigcirc$-player responds at any location other than location 2, then the $\times$-player moves at 2 and wins. If the $\bigcirc$-player moves at 2, then the $\times$-player moves at 6 and wins.

8.3 Player I is forced to start at 1, then Player II is forced to move at 2. After that Player I has a choice to move at either 3 or 4. Choosing 3 would lead to a forced win for Player II, so Player I moves at 4. Then Player II is forced to move at 5 and Player I is forced to move at 6 (note that 2 is unavailable at this point, having already been used). Player II has move and thus loses at that point. So Player I has a winning strategy, but Player II does not.

8.4 Let A_1 and A_2 be languages that are decided by PSPACE machines M_1 and M_1. 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 receives input w.

Machine $N_\cup$ simulates M_1 and M_2. It accepts if either of these machines accepts; otherwise it rejects.

Machine $N_\circ$ iteratively selects each position on the input to divide it into two substrings. For each division, $N_\circ$ simulates M_1 on the first substring and simulates M_2 on the second substring, and accepts if both machines accept. If no division is found for which both M_1 and M_2 accept, $N_\circ$ rejects.

Machine N_* accepts immediately if $w = \varepsilon$. Otherwise, N_* iteratively selects each of the 2^{n-1} possible ways of dividing w into substrings. Note that we can represent a

division of w into substrings by a binary string b of length $n-1$ where $b_i = 1$ means a division between w_i and w_{i+1}. For each way of dividing w, machine N_* simulates M_1 on each of the pieces of w and accepts if M_1 accepts each piece. If no division is found for which M_1 accepts each piece, N_* rejects.

8.6 If A is PSPACE-hard then every language in PSPACE is polynomial-time reducible to A. Moreover, PSPACE contains NP so every language in NP is polynomial-time reducible to A, and hence A is NP-hard.

8.9 The standard multiplication algorithm runs in log space if implemented carefully. Given $a = a_n \dots a_0$ and $b = b_m \dots b_0$ we want to be able to find the bits of $c = a \times b$ using only log space. Consider the table (between the horizontal lines):

$$
\begin{array}{cccccccccc}
 & & & & a_n & & \dots & & a_0 & \\
 & & & b_m & b_{m-1} & & \dots & & b_0 & \\
\hline
 & & b_m & b_{m-1} & & \dots & & b_0 & & \\
 & b_m & b_{m-1} & & \dots & & b_0 & & & \\
 & & & & \vdots & & & & & \\
b_m & b_{m-1} & & \dots & & b_0 & & & & \\
\hline
c_k & c_{k-1} & & \dots & & & & & c_0 &
\end{array}
$$

We find the bits of c by adding down the columns, keeping track of the carry. We can't write down the table within log space, but we can find individual entries using pointers because the entries are either b_i or 0 (if the corresponding digit of a is 0). We can store the carry using $\log |a| + \log |b|$ bits, because the total number of 1s in the table is at most $|a||b|$ so the carry value never grows to more than $|a||b|$.

8.10 **a.** We don't have enough space to write down $x^{\mathcal{R}}$ or store the sum $x + x^{\mathcal{R}}$, so we need to employ the technique we introduced in Theorem 8.23 which showed that if $A \leq_{\mathrm{L}} B$ and $B \in \mathrm{L}$, then $A \in \mathrm{L}$. The technique allows us to save space by recomputing the bits of the reduction when each one is needed. A similar proof shows that log space reducibility is transitive.

The function which takes two binary numbers and outputs their sum is computable in log space by implementing the usual method of addition that starts with the low order bits and maintains the carry bit. Furthermore, the function that output the reverse of a string is computable in log space. Hence we can compose these functions and show that the function $\mathcal{R}^+(x)$ is log space computable. Thus we conclude that that $A_2 \in \mathrm{L}$.

b. Following the same reasoning as in part (a), we conclude that the composition of $\mathcal{R}^+(x)$ with itself is likewise log space computable.

8.11 **a.** We use the addition standard algorithm, summing x and y digit by digit, from right to left, carrying a 1 when necessary, and checking at each step that the digit in the sum is the corresponding digit of z. We can do so with three pointers, one for each of the current digits of x, y and z, so this algorithm works in logarithmic space. If a pointer reaches the left end of a number before another, it is treated as if there were 0s to the left of it.

b. We use the standard addition algorithm (as described in part (a)) to build a log space transducer that takes $\langle x, y \rangle$ and outputs $\langle x + y \rangle$. This shows that *PAL-ADD* $\leq_{\mathrm{L}}$ *PALINDROME*. But *PALINDROME* is clearly in L because we can check that the input is a palindrome by using two pointers. Therefore, *PAL-ADD* is also in L.

8.12 Let G be an undirected graph. For each node in G we assign a cyclic ordering of the adjacent edges in an arbitrary way. (A *cyclic ordering* gives a way to place objects on a circle, so that every object has a "next" one, by going clock-wise around the circle. We may select a cyclic ordering of the edges adjacent to a node by considering the labels of the other nodes to which the edges connect. For any adjacent edge, the "next" adjacent edge is the one connected to the node with the next higher label, or to the lowest label if no higher label exists.)

Given a start node x and and adjacent edge e, define a path starting at (x, e) as follows. Begin at x and exit along e. Every time a node is entered along some edge, exit along the next edge in the ordering. Stop when x is first reentered.

Note that this procedure is reversible, so it cannot get caught in a loop without reentering x, and so it must eventually terminate. Say the path is *good* if x is reentered along edge e, and *bad* if x is reentered along some other edge.

The log space algorithm for *UCYCLE* operates as follows. For every node x and adjacent edge e, it follows the defined path starting at (x, e). If every such path is good, it *rejects*, otherwise it *accepts*. Observe that this procedure obviously runs in log space. Next we show that it works correctly. Below we refer to *paths* as the paths defined by the procedure.

Claim: Graph G is a forest (i.e., contains no cycles) if and only if all of the paths defined above are good.

The forward direction of this claim is obvious. We prove the reverse direction by contradiction. Suppose all paths are good, but G contains a cycle $x_1, x_2, \ldots, x_l, x_{l+1}$, where $x_1 = x_{l+1}$. Consider the path p starting at x_1 going along edge $e = (x_1, x_2)$. Consider the least i where the edge (x_i, x_{i+1}) doesn't appear in p. Note that $i < l$, otherwise x_1 was reentered along an edge different from e. At some point, p enters x_i from x_{i-1}. Every time p leaves x_i along some edge, p reenters x_i along the same edge (because all paths are good). Hence p must be traversing the edges of x_i in cyclic order, and must therefore get to the edge (x_i, x_{i+1}) before returning along edge (x_i, x_{i-1}). That contradicts the choice of i.

8.13 Let $\Sigma = \{\mathtt{0}, \mathtt{1}, \mathtt{\#}\}$. Let $R_1 = \Sigma^*$. Design R_2 to generate all strings except for the following string

$$z = \mathtt{000\cdots000\#000\cdots001\#000\cdots010\#\cdots\#111\cdots111}$$

Each binary block in z is of length n. The string contains 2^n blocks corresponding to the binary representations of the numbers 0 through $2^n - 1$. Observe that z is the unique string satisfying the following properties.

1. Begins with n 0s then a #.
2. Ends with n 1s.
3. Each adjacent pair ab of symbols is followed $n - 1$ symbols later by a pair cd from a certain legal list. Legal a, b, c, d are those where:
   i. $a, b, c, d \in \{\mathtt{0},\mathtt{1}\}$ and either $ab = cd$ or $ab = cd + 1 \pmod 4$; or
   ii. $a = c = \mathtt{\#}$ and $b \leq d$; or
   iii. $b = d = \mathtt{\#}$ and $a = \overline{c}$.

Clearly z satisfies these properties. No other string does, because the property 3 forces adjacent #'s to be separated by exactly n symbols so the binary blocks have length n; adjacent binary blocks must represent numbers of different odd/even parity; and each block is at most 1 larger than its predecessor.

We construct R_2 to be the set of strings which fail to have one of these properties and hence it describes all strings except z. (For any regular expression R, let R^l be

shorthand for writing R concatenated with itself l times.)

$$R_2 = R_{\text{start}} \cup R_{\text{end}} \cup R_{\text{increment}}$$

$R_{\text{start}} = \left((\Sigma \cup \varepsilon)^{n-1}(1 \cup \#) \cup \Sigma^n(0 \cup 1)\right) \Sigma^* \cup \varepsilon.$

$R_{\text{end}} = \Sigma^* (0 \cup \#)(\Sigma \cup \varepsilon)^{n-1}.$

$R_{\text{increment}} = \bigcup_{\text{illegal}(a,b,c,d)} \Sigma^* ab\Sigma^{n-1}cd\,\Sigma^*.$

8.14 We first prove that a graph is bipartite if and only if it has no odd cycles. Assume to the contrary that G is bipartite, but $C = (x_0, x_1, \ldots, x_{2k})$ is an odd cycle in G on $2k+1$ vertices for some $k \geq 1$. Then the vertices $\{x_1, x_3, \ldots, x_{2k-1}\}$ must lie on one side of the partition and the vertices $\{x_0, x_2, \ldots, x_{2k}\}$ on the other side of the partition. But (x_0, x_{2k}) is an edge in G between two vertices in the same side of the partition, a contradiction.

Conversely, assume G has no odd cycles. Work with each connected component independently. Let v be an arbitrary vertex of G. Then, because G has no odd cycles, placing all vertices of G which are an even distance from v on one side and those which are an odd distance from v on the other side gives a bipartition of G.

Here is an NL algorithm for the language $\overline{\textit{BIPARTITE}} = \{\langle G\rangle |\ G \text{ has an odd cycle}\}$. The algorithm is similar to the NL algorithm for *PATH*. It nondeterministically guesses a start node s and then nondeterministically guesses the steps of a path from s to s (i.e. a cycle containing s). It uses a counter to keep track of the number of steps it has taken and rejects if this counter exceeds the number of nodes in G. If the algorithm reaches s after an odd number of steps then it accepts.

The algorithm uses $O(\log n)$ space. It needs to store only the counter, the start node s, and the current node of the path, each of which requires $\log n$ bits of storage. Thus $\overline{\textit{BIPARTITE}} \in$ NL, and so *BIPARTITE* $\in$ coNL. But NL $=$ coNL and hence *BIPARTITE* $\in$ NL.

8.15 Given $G = (V, E)$ the n nodes, the reduction produces a graph $H = (V', E')$ where

i) $V' = \{(i, v, b) |\ 1 \leq i \leq n, v \in V, \text{ and } b \in \{0, 1\}\} \cup \{s, t\}$,

ii) $E' = \{((i_1, v_1, b_1), (i_2, v_2, b_2)) |\ (v_1, v_2) \in E \text{ and } b_1 = 1 - b_2\}$
$\cup\ \{(s, (i, v_1, 0)), (t, (i, v_i, 1)) |\ i \leq n\}$.

This reduction runs in log space using a few pointers and counters.

8.16 To see that *STRONGLY-CONNECTED* is $\in$ NL is we loop over all pairs of vertices and nondeterministically guess a connecting path for each pair of vertices. We accept if we find a path for each pair.

To show that *STRONGLY-CONNECTED* is NL-complete, we reduce from *PATH*. Given $\langle G, a, b\rangle$ (assume directed graph G has more than 2 nodes) build graph G' as follows:

Begin with G. For every node x not equal to a, add the edge (x, a). For every node y not equal to b, add the edge (b, y). Now we prove this is indeed a logspace mapping reduction.

If G has a directed $a - b$ path then G' is strongly connected since for each pair of nodes (x, y), we can get from x to y by the edge (x, a), then an $a - b$ path and finally the edge (b, y).

Conversely, if there is no $a - b$ path in G then there is no $a - b$ path in G' (we cannot use the new edges (b, y) because we cannot get to b and the edges (x, a) are useless because we can already reach a), and so G' will not be strongly connected.

Notice that our reduction loops over all pairs of nodes and so uses only logspace to determine what pair of nodes we are currently working with.

8.17 First, to show that $BOTH_{\mathsf{NFA}}$ is in NL, we use nondeterminism to guess a string that is accepted by both M_1 and M_2. In order to do this in logspace, we guess the string symbol by symbol (with the choice of guessing the end of string at each step) and we simulate (using nondeterminism) both NFAs in parallel on each guessed symbol. Once we guessed the end of string we accept if and only if both machines are in an accept state.

To show completeness we show a logspace reduction from *PATH*. Given an instance $\langle G, s, t\rangle$ of *PATH* we map it to the following instance of $BOTH_{\mathsf{NFA}}$:

M_1 is an NFA over the alphabet $\Sigma = \{0\}$ with a state q_i for each node i of G, q_s being the start state and q_t being the only accept state. For each edge (i, j) in G we define a transition from state q_i to state q_j of M_1 on symbol 0: $\delta_{M_1}(q_i, 0) = \{q_j | (i, j)$ is an edge in $G\}$.

We define M_2 to be an NFA that accepts Σ^*. Clearly $\langle M_1, M_2\rangle \in BOTH_{\mathsf{NFA}}$ if and only if $L(M_1) \neq \emptyset$ which is equivalent to the fact that there is a path from s to t in G, i.e. $\langle G, s, t\rangle \in$ *PATH*. Thus the reduction is proved and it is clearly done in logspace.

8.18 First we show that $A_{\mathsf{NFA}} \in$ NL by sketching an NL-algorithm. Given input $\langle B, w\rangle$ where B is an NFA and w is a string, we simulate B on w by maintaining a pointer to B's current state. If B has several possible next moves, we use nondeterminism to follow all possibilities. We accept if B is in an accept state after reading all of W.

Next we show that *PATH* $\leq_{\mathrm{L}}$ A_{NFA}. We map $\langle G, s, t\rangle$ to $\langle B, \varepsilon\rangle$, where B is an NFA that has a state for each node of G. The start state corresponds to the node s. We designate the state corresponding to node t to be an accept state. The transitions of B correspond to the edges of G, where each transition is an ε-transition. We assign that no other transitions in G. Clearly G has a path from s to t iff B accepts ε. The reduction can be carried out in log space because it is a very simple transformation from the input.

8.19 To prove that E_{DFA} is NL-complete, we can show instead that E_{DFA} is coNL-complete, because NL = coNL. Showing that E_{DFA} is coNL-complete is equivalent to showing that $\overline{E_{\mathsf{DFA}}}$ is NL-complete. First we show that $\overline{E_{\mathsf{DFA}}} \in$ NL. The NL-algorithm guesses a path along transitions from the start state to an accept state and accepts if it finds one.

Second we give a log space reduction from *PATH* to $\overline{E_{\mathsf{DFA}}}$. The reduction maps $\langle G, s, t\rangle$ to a DFA B which has a state q_i for every node v_i in G and one additional state d. DFA B has input alphabet $\{a_1, \ldots, a_k\}$ where k is the number of nodes of G. If G has an edge from v_i to v_j we put a transition in B from state q_i has a transition to q_j on input a_j. If G has no transition from v_i to v_j we put a transition from q_i to d on input a_j. In addition we put transitions from d to itself on all input symbols. The state corresponding to node s is the start state and the state corresponding to node t is the only accept state. The role of d in this construction is to make sure that every state has an exiting transition for every input symbol. We argue that this construction works by showing that G has a path from s to t iff $L(B) \neq \emptyset$. If G has a path $s = v_{i_1}, \ldots, v_{i_l} = t$ from s to t, the B accepts the input $a_{i_2} \cdots a_{i_l}$. Similarly, if B accepts some input $a_{i_1} \cdots a_{i_l}$, then it must correspond in G to the path $s, a_{i_1}, a_{i_2}, \ldots, a_{i_l} = t$. The reduction can be carried out in log space because it is a very simple transformation from the input.

8.20 First, show that *2SAT* $\in$ NL by showing $\overline{\textit{2SAT}} \in$ NL and using NL = coNL. Given a 2cnf formula ϕ, map it to a directed graph G as in the solution to Problem 7.25. Formula ϕ is unsatisfiable iff G contains an inconsistency cycle. Testing whether G contains an

inconsistency cycle can be done in NL, by going through the nodes sequentially, and for each node, nondeterministically guessing the inconsistency cycle. The NL-algorithm accepts if it finds one. The mapping from ϕ to G can be done in log space. Therefore testing the unsatisfiability of ϕ is in NL.

Second, show that *PATH* $\leq_{\mathrm{L}}$ $\overline{\textit{2SAT}}$ and hence that *PATH* $\leq_{\mathrm{L}}$ *2SAT*, again using NL = coNL. We map the graph G to a 2cnf ϕ that has a variable x_v for every node v of G. If (i, j) is an edge of G, add the clause $(\overline{x_i} \vee x_j)$ to ϕ. That clause is logically equivalent to $(x_i \rightarrow x_j)$. In addition, add the clause $(\overline{x_t} \vee \overline{x_s})$ to ϕ. That clause is logically equivalent to $(x_t \rightarrow \overline{x_s})$. Finally, add the clause $(x_s \vee x_s)$.

Show ϕ is unsatisfiable iff G has a path from s to t. Any satisfying assignment of ϕ must assign x_s to be True because of the $(x_s \vee x_s)$ clause. Considering the clauses using $\rightarrow$ instead of $\vee$ shows that any satisfying assignment must assign True to all variables x_i where i is a node that is reachable from s in G. If t is reachable, then x_t must be assigned True, but then the $(x_t \rightarrow \overline{x_s})$ clause requires that x_s be assigned False, thereby showing that ϕ is unsatisfiable. However, if t isn't reachable from s, then we can assign x_t and all remaining variables False, and obtain a satisfying assignment.

8.21 First show *CNF*$_{\mathrm{H1}} \in$ NL by representing it as a graph problem. Let ϕ be of the specified form. Create a graph G with a node a_x for each variable x. The clause $(\overline{x} \vee y_1 \vee \cdots \vee y_k)$ where $k \geq 1$ is logically equivalent to $(x \rightarrow (y_1 \vee \cdots \vee y_k))$. Represent that clause in G by adding an edge from node a_x to each node a_{y_i} for $1 \leq i \leq k$. Furthermore, if (x) is a clause, then mark a_x with T (for True) and if $(\overline{x})$ is a clause, then mark a_x with F (for False). Say a path in G is *positive* if none of its nodes are marked F. Check that a positive path exists starting from every node marked T to any node with no outgoing edges. It is straightforward to see that ϕ is satisfiable iff that condition holds and that this process can be carried out in NL.

Next, show that *CNF*$_{\mathrm{H1}}$ is NL-hard by reducing *LEVELED-PATH* to it (see the solution to Problem 8.22 for the definition of *LEVELED-PATH* and a proof that it is NL-complete. Given a *LEVELED-PATH* instance $\langle G, s, t\rangle$, construct an instance ϕ of *CNF*$_{\mathrm{H1}}$ as follows. Each node a of G corresponds to a variable x_a of ϕ. Start by putting the clause (x_s) in ϕ. For each node a which is connected to nodes $b_1, \ldots, b_k$, add the clause $(\overline{x_a} \vee x_{b_1} \vee \cdots \vee x_{b_k})$. Additionally, for each node c in G which has no outgoing edges, except for $c = t$, add the clause $(\overline{c})$. It is straightforward to show that ϕ is satisfiable iff G has a path from s to t.

8.22 The following is an example of an NL-complete context-free language.

$$\begin{aligned} C = \{&(x_1^1, y_1^1), \ldots, (x_{l_1}^1, y_{l_1}^1); \\ &(x_1^2, y_1^2), \ldots, (x_{l_2}^2, y_{l_2}^2); \\ &\ldots; \\ &(x_1^k, y_1^k), \ldots, (x_{l_k}^k, y_{l_k}^k) \mid \exists i_1, i_2, \ldots, i_k \, \forall j < k, \, \mathrm{Rev}(y_{i_j}^j) = x_{i_{j+1}}^{j+1}\}, \end{aligned}$$

where Rev(w) is the reverse of w. Construction of a PDA for this language is routine, so it is a CFL. It is equally easy to see that C is in NL. We now show that it is NL-hard.

A *leveled* directed graph is one where the nodes are divided into groups, $A_1, A_2, \ldots, A_k$, called *levels*, and the edges are only allowed to go from one level to the next, i.e., from A_i to A_{i+1} for each i. Let *LEVELED-PATH* $= \{\langle G, s, t\rangle \mid G$ is a leveled graph, s is a node in the first level, t is a node in the last level, and there is a directed path from s to $t\}$.

First we reduce *PATH* to *LEVELED-PATH*. Given a directed graph G, we construct the graph G' consisting of m copies of the m nodes of G. In other words, for each node i in G we have the nodes $[i,1],[i,2],\ldots,[i,m]$. If (i,j) is an edge in G we have the edges $([i,k],[j,k+1])$ for all $k<m$. We also include the edges $[(t,k),(t,k+1)]$ for all $k<m$. This is clearly a leveled graph. All the nodes with the same second component form a level and there are m such levels. The instance of *LEVELED-PATH* we output is $\langle G',[s,1],[t,m]\rangle$.

Now consider *POINTED-LPATH* = $\{\langle G\rangle \mid G$ is a leveled graph, the first level consists of s alone, the last level consists of t alone, and there is a directed path from s to $t\}$. It is easy to see that *LEVELED-PATH* $\leq_L$ *POINTED-LPATH*. Moreover, observe that *POINTED-LPATH* $\leq_L C$ by mapping edges between consecutive levels to the appropriate pairs with the second vertex name reversed. In total, *PATH* $\leq_L \mathcal{L}$, which proves $\mathcal{L}$ is NL-hard.

8.24 The procedure that is given in Theorem 1.54 shows how to convert a regular expression into a NFA that is linear in the size of the regular expression, in polynomial time. Therefore we can reduce EQ_{REX} to EQ_{NFA}. The following nondeterministic algorithm for $\overline{EQ_{\mathsf{NFA}}}$ is based on the algorithm for $\overline{ALL_{\mathsf{NFA}}}$ that appears in Example 8.4.

$M =$ "On input $\langle A, B\rangle$ where A and B are NFAs.

1. Place a marker on the start states of the NFAs.
2. Repeat 2^p times, where p is the sum of the numbers of states in the two NFAs.
3. Nondeterministically select an input symbol and change the positions of each machine's markers to simulate reading that symbol.
4. *Accept* if Stages 2 and 3 reveal some string that one NFA accepts and the other rejects. Otherwise, *reject*."

Hence the algorithm runs in nondeterministic space $O(n)$. By Savitch's theorem, we can conclude that $\overline{EQ_{\mathsf{NFA}}}$ and hence EQ_{NFA} and EQ_{REX} are in PSPACE.

8.25 We construct a polynomial space NTM N so that $L(N) = LADDER_{\mathsf{DFA}}$. Machine N on input $\langle M, s, t\rangle$ initially writes s on it's work tape and checks that $s \in L(M)$. It then repeats the following stage $|\Sigma^{|s|}|$ times (it maintains a counter which it increments each time and which can count to $|\Sigma^{|s|}|$, requiring polynomial space): Nondeterministically guess a position p on the work tape and a symbol σ of Σ and replace the contents of position p with σ and verify that this new string is in $L(M)$. If it is not, reject. If the string equals t then a chain from s to t has been found, so accept. Otherwise just continue. When the counter passes $|\Sigma^{|s|}|$, the machine rejects.

This shows that $LADDER_{\mathsf{DFA}} \in$ NPSPACE. But NPSPACE = PSPACE by Savitch's Theorem, so $LADDER_{\mathsf{DFA}} \in$ NPSPACE.

8.26 The following algorithm decides whether player "X" has a winning strategy in instances of go-moku; in other words, it decides GM. We then show that this algorithm runs in polynomial space. Assume that the position P indicates which player is the next to move.

$M =$ "On input $\langle P\rangle$, where P is a position in generalized go-moku:

1. If "X" is next to move and can win in this turn, *accept*.
2. If "O" is next to move and can win in this turn, *reject*.
3. If "X" is next to move, but cannot win in this turn, then for each free grid position p, recursively call M on $\langle P'\rangle$ where P' is the updated position P with player "X"'s marker on position p and "O" is next to move. If one

or more of these calls accepts, then $accept$. If none of these calls accept, then $reject$.

4. If "O" is next to move, but cannot win in this turn, then for each free grid position p, recursively call M on $\langle P' \rangle$ where P' is the updated position P with player "O"'s marker on position p and "X" is next to move. If all of these calls accept, then $accept$. If one or more of these calls rejects, then $reject$."

The only space required by the algorithm is for storing the recursion stack. Each level of the recursion adds a single position to the stack using at most $O(n)$ space and there are at most n^2 levels. Hence, the algorithm runs in space $O(n^3)$.

8.27 If every NP-hard language is PSPACE-hard, then *SAT* is PSPACE-hard, and consequently every PSPACE language is polynomial-time reducible to *SAT*. Then PSPACE $\subseteq$ NP because *SAT* $\in$ NP and therefore PSPACE $=$ NP because we know NP $\subseteq$ PSPACE.

8.29 To show that A_{LBA} is in PSPACE, recall the book's algorithm for A_{LBA}. It simulates the input LBA on the input string for qng^n steps where q is the number of states, g is the alphabet size and n is the length of the input string, by maintaining the current configuration of the LBA and a counter of the number of steps performed. Both can be stored by using polynomial space.

To show that A_{LBA} is PSPACE-hard, we give a reduction from *TQBF*. Note that *TQBF* is decidable with an LBA. The book's algorithm for *TQBF* uses linear space, and so it can run on an LBA B. Given a formula ϕ, the reduction outputs $\langle B, \phi \rangle$. Clearly, $\phi \in$ *TQBF* iff B accepts ϕ.

8.31 Let a graph G with n vertices V and hole location h be given. We call $c \in V \times V \times \{\text{cturn}, \text{mturn}\}$ a *position*. Each position corresponds to a location of the cat, a location of the mouse, and an indicator specifying which player moves next. We give a polynomial time algorithm which finds all positions that guarantee a win for the cat. The algorithm maintains a variable called CAT-WINS describes all winning positions so far found by the algorithm.

1. Initialize CAT-WINS to $\{(a, a, t) |\ a \in V,\ t \in \{\text{cturn}, \text{mturn}\}\}$.

2. For $i = 1$ to $2n^2$:

3. For every position (c, m, cturn), check whether the cat can move to a position (c', m, mturn) in CAT-WINS. If so, add (c, m, cturn) to CAT-WINS.

4. For every position (c, m, mturn) check whether all of the mouse's moves lead to a position (c, m', cturn) in CAT-WINS. If so, add (c, m, mturn) to CAT-WINS.

5. Output CAT-WINS."

An inductive argument shows that at the ith iteration of the loop in (2) we have found all positions where the cat can win in at most i moves. Since a game has $2n^2$ possible positions and the game is a draw if a position repeats, we find all winning positions for the cat in $O(n^2)$ steps. Each step requires at most $O(n^2)$ time. Hence, the algorithm runs in polynomial time.

8.32 **a.** "On input ϕ:

1. For each formula ψ shorter than ϕ:

2. For each assignment to the variables of ϕ:
3. If ϕ and ψ differ on that assignment, continue with the next ψ, else continue with the next assignment.
4. *Reject*: ϕ and ψ are equivalent.
5. *Accept*: no shorter equivalent formula was found."

The space used by this algorithm is for storing one formula and one assignment, and is therefore linear in the size of ϕ.

b. If $\phi \in \overline{\textit{MIN-FORMULA}}$, some smaller formula ψ is equivalent to ϕ. However, this formula ψ may not be an NP-witness to the fact "$\phi \in \overline{\textit{MIN-FORMULA}}$" because we may not be able to check it within polynomial time. Checking whether two formulas are equivalent may require exponential time.

8.33 "On input w:
1. Initialize a counter $c = 0$.
2. For each input symbol w_i:
3. If $w_i =$ (, increment c.
4. If $w_i =$), decrement c.
5. If c becomes negative, *reject*.
6. If $c = 0$, *accept*; otherwise *reject*."

Storing i and c uses logarithmic space, so this algorithm runs in log space.

8.34 Let the *type* of a symbol be whether it is a parenthesis or a bracket (so (and) are of one type, and [and] are of another type). The algorithm checks whether each symbol has a matching symbol of the same type.

"On input w of length n:
1. Check that w is properly nested ignoring symbol types, using the algorithm in the solution to Problem 8.33. *Reject* if the check fails.
2. For each $i = 1, \ldots, n$
3. If w_i is) or], continue the loop with the next i.
4. Set $c = 0$.
5. For each $j = i, \ldots, n$:
6. If w_j is (or [, increment c.
7. If w_j is) or], decrement c.
8. If $c = 0$, check whether w_i and w_j are of different types.
If so, *reject*; if not, continue the Stage 2 loop with the next i.
9. No symbol matching w_i was found, *reject*.
10. No mismatch was discovered, *accept*."

Storing i, j, and c uses logarithmic space, so this algorithm runs in log space.

Chapter 9

9.7 Solutions to some parts are provided in the text.

9.10 **a.** We show that $A_{\text{LBA}} \notin \text{NL}$ with a proof by contradiction. By the space hierarchy theorem, some language B is in $\text{SPACE}(n)$ but not in $\text{SPACE}(\log^2 n)$. Therefore, by Savitch's theorem, B is not in NL. But linear space TMs can be converted to equivalent LBAs, so B is recognized by some LBAM. The reduction $w \longrightarrow \langle M, w\rangle$ is a logspace reduction from B to A_{LBA}. So if $A_{\text{LBA}} \in \text{NL}$, then $B \in \text{NL}$, a contradiction.

b. We do not know whether $A_{\text{LBA}} \in \text{P}$, because that question is equivalent to whether $\text{P} = \text{PSPACE}$. If $\text{P} = \text{PSPACE}$ then $A_{\text{LBA}} \in \text{P}$ because $A_{\text{LBA}} \in \text{PSPACE}$. Conversely, if $A_{\text{LBA}} \in \text{P}$ then $\text{P} = \text{PSPACE}$ because A_{LBA} is PSPACE-complete.

9.15 Let ϕ be a fully quantified Boolean formula and let x be its first variable. Then a correct oracle C for TQBF has to satisfy the following condition:

$$C(\phi) = C(\phi|_{x=0}) \vee C(\phi|_{x=1}) \quad \text{if } x \text{ is a "}\exists\text{" variable;}$$
$$C(\phi) = C(\phi|_{x=0}) \wedge C(\phi|_{x=1}) \quad \text{if } x \text{ is a "}\forall\text{" variable.}$$

If an oracle doesn't satisfy this condition, we say that it "errs" on variable x.

If both oracles A and B agree on the input formula ϕ, our algorithm outputs their answer. If they differ, we query both oracles on $\phi|_{x=0}$ and $\phi|_{x=1}$. If one of the oracles "errs" on x, we output the answer of the other. If both oracles do not "err" on x then they differ on at least one of $\phi|_{x=0}$ and $\phi|_{x=1}$. We recursively substitute values for the next variables in the formula until one of the oracles "errs" or until we find an expression with all variables substituted on which the oracles differ. We can verify which oracle is correct on such an expression in linear time.

9.16 For any oracle A define the language $L(A) = \{x|$ for all $y, |y| = |x|, y \notin A\}$. A coNP oracle machine with access to A can decide this language since an NP oracle machine with A can decide the complement of $L(A)$ by guessing a y of length $|x|$ and checking that y is in A.

To show $L(A)$ is not in NP, we construct A iteratively as in the proof that there exists an oracle separating P from NP. To fool a particular NP machine, run the machine on an input of size n such that no queries of that size were asked in earlier steps of A's construction and such that 2^n is greater than the running time of A. Every time the machine asks if a certain string s of size n is in A, we say that it is not. If the machine rejects, include no words of size n in the oracle. If the machine has an accepting path, then there is some string of size n that was not asked on the path because the machine can only make a polynomial number of queries on any branch of its computation. We

include this string in the oracle A. This path must still be an accepting path, even though the machine should now reject. Thus, we have "fooled" every NP machine.

9.19 Regular expressions are created using four operations: concatenation, union, star, and exponentiation. The atomic expressions are $\emptyset$, ε, and $\mathtt{a} \in \Sigma$. We use the following recursive algorithm A to test whether $L(R) = \emptyset$:

1. If R is atomic, check if $R = \emptyset$. If so, *accept*, otherwise *reject*.
2. If $R = R_1 R_2$, recursively run A on R_1 and R_2.
   Accept if either R_1 or R_2 are accepted; else *reject*.
3. If $R = R_1 \bigcup R_2$, recursively run A on R_1 and R_2.
   Accept if both R_1 and R_2 are accepted; else *reject*.
4. If $R = R_1^*$ then *accept*. (Note, $\varepsilon \in L(R)$)
5. If $R = R_1^p$, for some exponent p, recursively run A on R_1.
   Accept if R_1 is accepted; else *reject*."

This algorithm runs in polynomial time because it looks at each symbol and/or operator at most once.

9.20 The error is in the inference: "Because every language in NP is poly-time reducible to *SAT* (and *SAT* $\in \mathrm{TIME}(n^k)$) then $\mathrm{NP} \in \mathrm{TIME}(n^k)$."

The problem is that we are not accounting for the time it takes to perform the reduction; all we know is the reduction is polynomial, but it may take more than $\mathrm{TIME}(n^k)$. Indeed there are languages for which the poly-time reduction to SAT would take strictly more than $\mathrm{TIME}(n^k)$ (e.g., a language in $\mathrm{TIME}(n^{k+1}) - \mathrm{TIME}(n^k)$).

9.21 First we prove $A \in \mathrm{TIME}(n^6)$ implies *pad*$(A, n^2) \in \mathrm{TIME}(n^3)$. Suppose A is decided by a TM M that runs in time $O(n^6)$. Consider the machine
M' = "On input w:

1. Check that w is of the form $s\#^*$, for s without #'s. If not, *reject*.
2. Check that $|s|^2 = |w|$. If not, *reject*.
3. Run M on s. If M accepts, *accept*. Otherwise, *reject*."

Clearly, M' decides *pad*(A, n^2). Stage 1 needs only linear time and Stage 2 can be done in $O(n^2)$ time (actually $O(n \log n)$ is enough). Stage 3 is performed only when $|s| = \sqrt{n}$; it then takes time $O\big((\sqrt{n})^6\big) = O(n^3)$. In total, M' computes in time $O(n^3)$.

9.22 We prove that NEXPTIME $\neq$ EXPTIME implies NP $\neq$ P. Suppose some language $A \notin$ EXPTIME is decided by a nondeterministic TM N running in time $O(2^{n^k})$, for some k. We prove that $A' =$ *pad*$(A, 2^{n^k})$ is in NP, but not in P; hence NP $\neq$ P.

Construct the machine
N' = "On input w:

1. Check that w is of the form $s\#^*$, for s without #'s. If not, *reject*.
2. Check that $2^{|s|^k} = |w|$. If not, *reject*.
3. Run N on s."

Clearly, N' decides A'. Stage 3 is executed only when $2^{|s|^k} = n$; and then every thread of the nondeterministic computation uses time $O(2^{|s|^k}) = O(n)$. Hence, N' runs in nondeterministic polynomial time.

We prove $A' \notin P$ by contradiction. Assume some TM M' decides A' in time $O(n^c)$, for some $c \geq 1$. Consider the machine
M = "On input w:

1. Append $2^{|w|^k} - |w|$ new # symbols to w. Let w' be the new, padded string.

2. Run M' on w'. If M' accepts, $accept$. Otherwise, $reject$.**"**

Clearly, M accepts a string w iff $pad(w, 2^{n^k}) \in A'$; which is true iff $w \in A$. Hence, M decides A. Moreover, M runs in time $O\big((2^{n^k})^c\big) = O(2^{cn^k}) = O(2^{n^{k+1}})$. Hence, $A \in$ EXPTIME, a contradiction.

9.25 We prove that every language that is decided by a 2DFA can also be decided by a Turing machine running in $\mathrm{TIME}(n^3)$. We then use the time hierarchy theorem to find a language in, say, $\mathrm{TIME}(n^4)$ but not in $\mathrm{TIME}(n^3)$.

We define a configuration of a 2DFA to be a string consisting of the position of the tape heads and the current state.

Chapter 10

10.2 Letting $a = 2$ and $p = 12$ we get $a^{p-1} \mod p = 2^{11} \mod 12 = 2048 \mod 12 = 8 \neq 1$. Hence 12 is not prime.