8.1 First, we can simulate a SPACE($f(n)$), for $f(n) \geq n$, single-tape TM on a SPACE($f(n)$) two tape read-only TM as follows. Step one: scan the read-only tape, copying its contents to the work tape. Step two: simulate the remainder of the computation, treating the read/write work-tape as the input tape of a "single-tape" TM. We can copy the contents of the counter, using only $\log n$ space to keep track of our position in the input. We can write all of the input on our work-tape, since $f(n) \geq n$.

Secondly, we can simulate a SPACE($f(n)$), for $f(n) \geq n$, two tape read-only TM on a SPACE($f(n)$) single-tape TM by refraining from writing over the first n input symbols, using the remainder of the tape for our work area. Since this only adds n amount of space used, and we were already using at least n space, this increases our space by at most a constant.

8.6 Suppose language A is PSPACE-hard. Then every language in PSPACE is polynomial time reducible to A. Since NP $\subseteq$ PSPACE, every language in NP is also polynomial time reducible to A, i.e., A is NP-hard.

8.8 We show that $EQ_{\mathsf{REX}} \in$ PSPACE by proving $\overline{EQ_{\mathsf{REX}}} \in$ NPSPACE. The following NTM M decides $\overline{EQ_{\mathsf{REX}}}$ in linear space.

M = "On input $\langle R, S \rangle$, where $R, S \in \mathrm{RE}_\Sigma$:

1. Construct NFAs $N_X = (Q_X, \Sigma, \delta_X, q_X, A_X)$ such that $L(N_X) = L(X)$, for $X \in \{R, S\}$.
2. Let $m_X = \{q_X\}$ (e.g., modelled by marking states in Q_X), for $X \in \{R, S\}$.
3. Repeat $2^{\max_{X \in \{R,S\}} |Q_X|}$ times:
4. If $m_R \cap A_S = \emptyset \Leftrightarrow m_S \cap A_S \neq \emptyset$, *accept*.
5. Pick any $a \in \Sigma$ and change m_X to $\bigcup_{q \in m_X} \delta_X(q, a)$, for $X \in \{R, S\}$.
6. If haven't yet accepted, *reject*."

8.9 It suffices to show that *LADDER* $\in$ NPSPACE. The idea is as follows: given $\langle M, s, t \rangle$, reject if $|s| \neq |t|$. Otherwise, consider a graph G of exponential size whose vertices are indexed by strings in $\Sigma^{|s|}$, and there is a directed edge from w_1 to w_2 iff w_1 and w_2 differ in exactly one character, and $w_1, w_2 \in L(M)$. Then, $\langle M, s, t \rangle \in$ *LADDER* iff there is a path from s to t in G. This we can check in NPSPACE by guessing the path (akin to the NL algorithm for *PATH*), and at each step, storing only the name of current vertex (which is a string in $\Sigma^{|s|}$). To guess the path, at vertex w_1, we will nondeterministically select a new vertex w_2 that differs from w_1 in exactly one character, and verify that M accepts

w_2. (We can ensure that the machine always halts by keeping a counter and incrementing it with each guess, and rejecting when the counter hits $|\Sigma|^{|s|}$. This is because if there exists a chain from s to t, then there exists one of length at most $|\Sigma|^{|s|}$ by removing loops.)

8.10 First let's assume that P is written as a grid of X, O and empty, so the length of the input is $O(n^2)$. Let's define a recursive algorithm to solve $GM(P)$, which accepts if there is a winning strategy for player X starting at position P:

(1) For all spaces i in position P without markers on them, (potential X moves)

1. Put an X marker on space i, thus changing the position to P'. If there are now 5 X's in a row, *accept*, this is obviously a good move. If the board is now full, and no one has won, *reject*.
2. Otherwise, for all spaces j in position P' without markers on them, (potential O moves)
   (a) Put an O on space j, thus changing the position to P''. If there are now 5 O's in a row, or the board is full and no one has won, loop to the next i (go to step **(1)**); putting an X on i is obviously a bad move.
   (b) Otherwise, run $GM(P'')$. If it accepts, loop to the next j (go to step 2). If it rejects, loop to the next i (go to step **(1)**).
3. If all j cause $GM(P'')$ to accept, i is a good X move, since it covers all possible O moves, so *accept*.

(2) If no i in step **(1)** cause an accept, reject, there are no good moves from this position, so *reject*.

We can safely loop through all moves i, j at each step, since we can just reuse the space. Our recursion is only $O(n^2)$ deep, since there are at most n^2 moves during a game, and we just need to store configurations P', P'' on the stack at each level, which takes $O(n^2)$ space. So, the total space needed is $O(n^4)$, which is polynomial in the input length, since we assumed the input had length $O(n^2)$.

Why not just a 19×19 board? Well, that would mean there were 19^2 spaces on the board, and 3^{19^2} possible configurations. This is a big big number, but asymtotically, it's tiny. So, we just search through all possible moves sequences and see if X can win. In the world of complexity theory, any expression without a variable is just a small constant, even if it's bigger than the number of atoms in the universe, which that number

most certainly is. This is why we like to generalize things to variable lengths, so we can fit them into our complexity classes.

If we write the configuration down as n, followed by a list of marker locations, we run the risk of having the input be too small; for example, let's say that no markers have been placed, so we want to know who has a winning strategy from the beginning. Our input has length $\log n$, since it takes that much space to write down the number n. Our algorithm might still take $O(n^4)$ space, but now that's exponential in our input length. So, we need to impose an appropriate encoding to ensure that we stay within PSPACE in all cases.

8.11 We know that NP $\subseteq$ PSPACE. We must show the opposite inclusion, namely that PSPACE $\subseteq$ NP. To see this, note that SAT $\in$ PSPACE and SAT is NP-complete. By the definition of NP-complete, SAT is also NP-hard, and by our assumption, SAT is PSPACE-hard. Thus for any $L \in$ PSPACE, we have $L \leq_P$ SAT. But since SAT $\in$ NP, we have that $L \in$ NP. Therefore PSPACE $\subseteq$ NP.

8.12

Solution 1: (borrowed from Spring 2001) To show that TQCNF is PSPACE-complete, we need to show that TQCNF $\in$ PSPACE and that TQBF $\leq_P$ TQCNF.

TQCNF $\in$ PSPACE because TQCNF is a special case of TQBF, so we can use the same decider for TQCNF as for TQBF.

To show that TQBF $\leq_P$ TQCNF, we need to construct a polynomial time computable translation function, f, such that $\langle\phi\rangle \in$ TQBF $\iff f(\langle\phi\rangle) \in$ TQCNF. The following procedure F indicates, with one bug, how to compute f:

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

1. Let Q be the part of $\langle\phi\rangle$ containing all the quantifiers.
2. Let ϕ be the part of $\langle\phi\rangle$ containing no quantifiers.
3. Construct ϕ' that is the equivalent to ϕ in conjunctive normal form:
   - Using DeMorgan's laws ensure that all NOTs in ϕ are only on individual variables and not on expressions.
   - Apply distributivity
     $$(A \wedge B) \vee C = (A \vee C) \wedge (B \vee C)$$
     where possible.
   - Repeat until no more conversions of the above type are possible.
4. Output $Q[\phi']$."

Since ϕ' is equivalent ϕ, we have that $\langle\phi\rangle \in$ TQBF $\iff f(\langle\phi\rangle) \in$ TQCNF.

Now we consider the time complexity of the reduction, and we discover the bug. The conversion from ϕ to ϕ' may blow up exponentially: the problem is in the applications of distributivity, because each application of the law converts one copy of C into two copies, potentially doubling the size of the formula at each application.

The fix is to construct ϕ' equivalent to ϕ such that ϕ' is a *quantified* CNF Boolean formula but $|\phi'|$ remains bounded by a polynomial in $|\phi|$. The trick for doing this is to show a slightly more general construction: construct from any Boolean formula, ϕ, a quantified Boolean formula, $E_{\phi,x}$, with one extra free variable, x, such that (i) $E_{\phi,x}$ is equivalent to the formula $x \equiv \phi$, (ii) $E_{\phi,x}$ would be in CNF if we ignored the quantifiers, and (iii) $|E_{\phi,x}|$ is bounded by a polynomial in $|\phi|$. Namely,

$$\begin{aligned}
E_{y,x} &\stackrel{\text{def}}{=} (x \vee \neg y) \wedge (y \vee \neg x), \\
E_{\neg A,x} &\stackrel{\text{def}}{=} (\exists x_1)[E_{A,x_1} \wedge (\neg x \vee x_1) \wedge (x \vee \neg x_1)], \\
E_{A_1 \wedge A_2,x} &\stackrel{\text{def}}{=} (\exists x_1, x_2)[E_{A_1,x_1} \wedge E_{A_2,x_2} \wedge (x \vee \neg x_1 \vee \neg x_2) \wedge (\neg x \vee x_1) \wedge (\neg x \vee x_2)], \\
E_{A_1 \vee A_2,x} &\stackrel{\text{def}}{=} (\exists x_1, x_2)[E_{A_1,x_1} \wedge E_{A_2,x_2} \wedge (x \vee \neg x_1) \wedge (x \vee \neg x_2) \wedge (\neg x \vee x_1 \vee x_2)].
\end{aligned}$$

We leave it to the reader to verify that this construction satisfies (i–iii) above.

Now note that we can always rename quantified variables so no two quantifiers bind identical variables. Also notice that none of the quantifiers in $E_{\phi,x}$ are within a negated subformula. So if we pull all quantifiers to the beginning of $E_{\phi,x}$, we obtain an equivalent formula $E'_{\phi,x}$ of the same size as $E_{\phi,x}$ that is a quantified (in fact existentially quantified) CNF formula.

Now define $\phi' \stackrel{\text{def}}{=} (\exists x)[x \wedge E'_{\phi,x}]$. With this revised construction of ϕ', the QCNF $f(\langle\phi\rangle)$ is of size polynomial in $|\langle\phi\rangle|$ and can also be computed in time polynomial in $|\langle\phi\rangle|$. Hence this revised mapping f is the required polytime reduction from TQBF to TQCNF.

8.13

Proof:

- Containment in PSPACE: "On input $\langle M, w\rangle$: simulate M on w". This takes only linear space, so it belongs to PSPACE.
- PSPACE-hardness: Let $L \in$ PSPACE. We show $L \leq_P A_{LBA}$.
  - Because $L \in$ PSPACE then there is a decider M running in space n^k such that $L(M) = L$.
  - Poly-time reduction: On input w: output $\langle M, w(\sqcup)^{|w|^k}\rangle$.
  - Clearly, $w \in L$ iff M accepts $w(\sqcup)^{|w|^k}$.
  - M runs in space n^k, so M on input $w(\sqcup)^{|w|^k}$ acts as LBA. □

8.14

Answer: We first note that the game can have only $2n^2$ configuration, defined by position of the cat, position of the mouse and if it is cat's turn. So we can construct a directed graph consisting of $2n^2$ nodes where each node corresponds to a game configuration and there is an edge from node u to node v, if we can go from configuration corresponding to u to configuration corresponding to v in one move. Now following algorithm solves HAPPY-CAT.

1. Mark all nodes (a,a,x) where a is a node in G, and $x \in \{true, false\}$.
2. If for a node u = (a,b,true), there is a node v = (c,b,false) which is marked and (u,v) is an edge then mark u.
3. If for a node u = (a,b,false), all nodes v = (a,c,true) are marked and (u,v) is an edge then mark u.
4. Repeat steps 2 and 3 until no new nodes are marked.
5. Accept if start node s = (c,m,true) is marked.

The algorithm takes $O(n^2)$ time to perform step 1, $O(n^2)$ time per iteration of the loop and loop is executed $O(n^2)$ times. Hence runs in polynomial time.

8.15

We will reduce *TQBF* to *PUZZLE*. Consider any *TQBF* instance in its *CNF* form. Let m denote the number of clauses and n denote the number of variables. We will consider an instance of *PUZZLE* with n cards, each of which corresponds to one of the variables and in the same order as the quantifiers in the *TQBF* instance. The box have *m* holes all on the right column. We will define each card as follows: For each row *i*, the card would have exactly one hole on the right for row *i* if letting the variable to be true would satisfies the *i*-th clause; it would have exactly one hole on the left if letting the variable to be false would satisfies the clause; let there be two holes otherwise. Each step, the player chooses one side of the cards and hence choose the truth assignment for the corresponding variable (front for true, and back for false). The *TQBF* instance is satisfied if-and-only-if for each row at least one of the cards blocks the hole, that is, at least one of the literals in the clause has value true.

8.16

Answer:
Part a: Consider following algorithm
On input F:
1. For each string s such that $|s| < |F|$, if s is a valid representation of a formula (this can be easily checked) which is equivalent to F (this can be checked in polynomial space by evaluating both F and s over all possible truth assignments and comparing the results) then reject.
2. If all string have been tried without rejecting, accept.

Correctness of the algorithm should be evident. Only space used by the algorithm is for storing forumula F, string s and current assignment of literals which amounts to polynomial space. Hence MIN-FORUMLA $\in$ PSPACE.
Part b: It fails because it is not known if we can verify equivalance of two boolean formulae is polynomial time.

8.17

Proof: Our **L** machine that decides A works as follows.

"On input $w \in \{(,)\}^*$:

1. initialise counter c to 0 on the work tape
2. read the next letter ℓ of the input word
3. case ℓ of

   '(' : increment c

   ')' : if $c = 0$ then *reject*, otherwise decrement c

   '$\sqcup$' : if $c = 0$ then *accept*, else *reject*
4. go to 2"

8.23

Hint: We can try to search the tree by always traversing the edges incident on a vertex in lexicographic order i.e. if we come in through the ith edge, we go out through the $(i+1)$th edge or the first edge if the degree is i. How does this algorithm behave on a tree? How about a graph with a cycle?

SOLUTION: Note that the above process performs a DFS on a tree and we always come back to a vertex through the edge we went out on. However, if the graph contains a cycle, there must exist at least one vertex u and at least one starting edge (u, v) such that if we start the traversal through (u, v), we will come back to u through an edge different that (u, v). Hence, we enumerate all the vertices and all the edges incident on them, and start a traversal through each one of them. If we come back to the starting vertex through an edge different than the one we started on, we declare that the graph contains a cycle. Since we can enumerate all vertices and edges in logspace and also remember the starting vertex and edge using logarithmic space, this algorithm shows $UCYCLE \in \mathbf{L}$.

8.24

Hint: The expression

$$(\underbrace{11\cdots1}_{p\text{ times}})^*11\underbrace{?1?\cdots1?}_{p-2\text{ times}}$$

recognizes 1^k for every k that is *not* a multiple of p.

Can you construct a polynomial-length regular expression that recognizes 1^k for every k except ones that are multiples of all of the first n primes?

(Useful knowledge: The nth prime is less than $n(\ln n + \ln\ln n)$ for $n \geq 6$, so the sum of the first n primes is $O(n^2 \log n)$).

8.25

SOLUTION: Dividing the graph into two sets is the same as 2-coloring the graph such that all the edges are between vertices of different color. If the graph has an odd cycle, then the odd cycle in particular cannot be 2-colored and hence the graph cannot be bipartite. If the graph does not contain an odd cycle, we consider the DFS tree (or forest, if it is not connected) of the graph and color the alternate levels (say) red and blue. By construction, all the tree edges are between vertices of different color. Suppose an edge not in the tree connects two vertices of the same color. But these vertices must be connected by a path of even length in the tree and this extra edge will create an odd cycle, which is not possible. Hence, the above 2-coloring is a valid one which proves that the graph is bipartite.

We show that $BIPARTITE \in \mathbf{coNL}$ or $\overline{BIPARTITE} \in \mathbf{NL}$, which suffices since $\mathbf{NL} = \mathbf{coNL}$. However, $\overline{BIPARTITE}$ is precisely the set of all graphs which contain an odd cycle. We guess a starting vertex v, guess an (odd) cycle length l and go for l steps from v, guessing the next vertex in the cycle at each step. If we come back to v (we can remember the starting vertex in logspace), we found a tour of odd length. Since any odd tour must contain an odd (simple) cycle, we accept and declare that the graph is non-bipartite.

8.26

Consider the following two decision problems:

$$UPATH = \{(G, s, t) : G \text{ is an undirected graph with a path from } s \text{ to } t\}$$
$$BIPART = \{G : G \text{ is a undirected graph without cycles of odd length}\}.$$

Show that $UPATH \in \mathrm{L} \implies BIPART \in \mathrm{L}$ and $BIPART \in \mathrm{L} \implies UPATH \in \mathrm{L}$.

Solution

First let's assume $UPATH \in \mathrm{L}$ and show that $BIPART \in \mathrm{L}$. Recall that the graph G^2 is obtained by taking all paths of length 2 in G. Notice that G is bipartite if and only if for all edges (u, v) of G, u and v are not connected by a path in G^2. To see this, notice that if G has a cycle of odd length, and (u, v) is any edge on this cycle, then (u, v) are connected by a path of even length, which is a path of G^2. Conversely, an edge in G plus a path in G^2 between (u, v) yield a cycle of odd length in G.

8.27

Solution 2: First, we will show that SC is in NL. We give a nondeterministic log space TM M that decides SC.

M="On input $\langle G \rangle$,

1. Count the nodes of G; let that total be n.
2. For s=1 to n,
   For t=1 to n,
   If $s \neq t$, simulate the NL PATH decider on $\langle G, s, t \rangle$.
3. If all simulations accepted, accept; otherwise, reject."

This algorithm works by checking that there is a directed path between every two nodes in each graph – the definition of a strongly connected graph. M runs in nondeterministic log space: two $\log n$ counters are needed for s and t, the PATH algorithm takes nondeterministic log space – which we can reuse for each simulation. Thus, we conclude that SC is in NL.

Next, we will show that PATH $\leq_L$ SC.

F="On input $\langle G, s, t \rangle$,

1. Count the nodes of G; let that total be n.
2. Copy G to the output tape.
3. For u=1 to n,
   If $u \neq s$, add directed edge (u, s) to output tape.
   If $u \neq t$, add directed edge (t, u) to output tape."

This algorithm only needs $\log n$ space to store the counter for u.

If there is no path from s to t in G, then there is also no path from s to t in G'. To see this, observe that we only added incoming edges to s and outgoing edges to t. Thus, if $\langle G, s, t \rangle \notin$ PATH, then $\langle G' \rangle \notin$ SC.

If there is a path from s to t in G, then, for any nodes u, v in G', there is a directed path that starts with (u, s), takes the directed path from s to t, and ends with (t, v). Thus, if $\langle G, s, t \rangle \in$ PATH, then $\langle G' \rangle \in$ SC.

8.28

[**Hint:** For NL-hardness, you may assume that A_{NFA} is NL-complete.]

8.29

First, A_{NFA} is in NL, as we can, on input NFA M and x, nondeterministically choose one next state transition. The variables recording "how much of x has been read" and "what state M is in" are only logarithmically long.

To show A_{NFA} is NL-hard, we need only show that *PATH* log-space reduces to A_{NFA}: Given as input a directed graph $G = (V, E)$ and two of its nodes, x and y, we construct an NFA N with its states corresponding to G's nodes, its transitions under alphabet 0 corresponding to G's directed edges. Then, for N's accepting state (the state representing y), we let it go back to itself on all input alphabets, and, for other states, upon reading a 1, we go into a "trap" state which is deemed to reject.

It can now be shown that x goes to y in G iff on input 00...000 ($|V|$ of 0's), N accepts. This completes the proof.

8.30

Answer: We show that $N = \overline{E_{DFA}}$ is NL-complete and then use the fact that NL = co-NL to conclude that E_{DFA} is NL-complete.
Part I: N is in NL.
We can non-deterministically guess the string accepted by N and then verify that string is indeed accepted in log-space as all we need to keep track of is current state of N and current position in the string.

Part II: N is NL-complete.
We reduce PATH to N as follows. Given instance $< G(V, E), s, t >$ of PATH, Let k denote maximum outdegree of any node in G. So we construct a DFA $D = \{Q, \Sigma, \delta, q_0, F\}$ as follows. $Q = V \cup \{d\}$, where d is new dead state. $\Sigma = \{a_1, a_2, .., a_k\}$ be arbitrary set of input alphabet. Each outgoing edge from every state of D is labeled with arbitrary symbol from Σ. Note that we always have enough symbols to unquiely label outgoing edges from given state. If any state has less than k outgoing edges, we add transition from that state to dead state for all symbols not used in any transition from that state. Finally we set $q_0 = s$ and $F = \{t\}$. It is easy to see that graph G has a path from s to t if and only if DFA D accepts at least one string.

Another answer?

Answer: E_{DFA} is in co-NL (a sufficient certificate is an accepted string – one always exists of length less than the number of states), thus it is in NL. We prove NL-hardness by a reduction from $\overline{PATH}$ (which is co-NL-complete, and thus NL-complete).

The idea of the reduction from $\overline{PATH}$ is simple. Given $G = (V, E)$ and vertices s, t, we will construct a DFA whose state graph is G and s is the initial state and t is the final state. Then the language of this DFA is empty if and only if t is not reachable from s (that is, $\langle G, s, t\rangle \notin \overline{PATH}$).

So it suffices to show this is doable with a log-space transducer.

First, for each vertex, we count the maximum out-degree, d. Our alphabet will have $|\Sigma| = d$ (in particular, we will take $\Sigma = \{1, \ldots, d\}$, where we can write each number in $\log d = O(\log m)$ bits). This computation is easily done in log-space, by counting at each vertex and only maintaining a maximum value.

Each vertex u will be translated (in the order the vertices appear on the tape) into a state, and each edge (u, v_i) (in the order the edges out of u appear on the tape) into a transition rule $\delta(u, i) = v_i$. If we reach the end of the list of vertices without giving transitions for all d letters, we add copies of the last edge we visited so that u has transitions for all letters. This step is done in log-space, since we remember at most one vertex and two edges at a time.

It is then easy to set s to be the start state, and t to be the final state. So we can do this reduction in log-space, and so E_{DFA} is NL-complete.

8.31

Since NL $=$ coNL, it suffices to show that $\overline{\text{2SAT}}$ in NL-complete. In order to prove $\overline{\text{2SAT}} \in$ NL, construct the graph G as in Exercise 2.2.3. Convince yourself that this construction can be done in (deterministic) log-space. Now nondeterministically choose a vertex x in G. Use an NL algorithm for PATH to check if G contains both $x, \bar{x}$ and $\bar{x}, x$ paths. If both paths exist, accept, else reject. The correctness of this algorithm is established by the claim in Exercise 2.2.3.

In order to prove the NL-hardness of $\overline{\text{2SAT}}$, I show that PATH $\leqslant_L \overline{\text{2SAT}}$. Given G, s, t we have to construct a 2-cnf formula ϕ such that G has an s,t-path if and only if ϕ is unsatisfiable. Let $V(G) = \{s, t, y_1, \ldots, y_m\}$. The resulting formula ϕ consists of $m+1$ variables $x, y_1, \ldots, y_m$. The vertex s is identified (relabeled) with x and the vertex t with $\bar{x}$. The clauses of ϕ will be $x \vee x$ and $\bar{u} \vee v$ for every edge (u, v) of G. That completes the construction of ϕ.

In order to show that this construction works, first assume that G has an s,t-path. Let this path be $s, u_1, \ldots, u_k, t$. Then ϕ contains the clauses $(\bar{x} \vee u_1), (\bar{u}_1 \vee u_2), \ldots, (\bar{u}_k, \bar{x})$. If we assign $x = 1$, at least one of these clauses is not satisfied (easy check). On the other hand, if $x = 0$, then the clause $(x \vee x)$ of ϕ is not satisfied. So ϕ is zero for any assignment of $x, y_1, \ldots, y_m$.

Conversely, suppose that G contains no s,t-path. Let U be the set of vertices in G reachable from s, V the set of vertices from which t is reachable and $W := V(G) \setminus (U \cup V)$. By hypothesis and construction, U, V, W are pairwise disjoint, and there are no edges from U to $V \cup W$ or from $U \cup W$ to V. Let me assign the true value to every variable in $U \cup W$ (including x) and the false value to every variable in V (including the *literal* $\bar{x}$). It is now simple to check that ϕ is satisfied for this truth assignment.