7.1

a. True
b. False
c. False
d. True
e. True
f. True

7.2

a. False
b. True
c. True
d. True
e. False
f. False

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

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

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 using the algorithm from Theorem 7.14 is as follows:

	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. For any assignment of the boolean values for x and y, it always makes one of the four clauses false. Therefore, the formula, which is a conjunction of the four clauses, is always false for any assignment of x and y.

7.6

(a) Show that P is closed under union.

Answer: Suppose that language $L_1 \in \text{P}$ and language $L_2 \in \text{P}$. Thus, there are polynomial-time TMs M_1 and M_2 that decide L_1 and L_2, respectively. A Turing machine M_3 that decides $L_1 \cup L_2$ is the following:

M_3 = "On input w:
1. Run M_1 with input w. If M_1 accepts, *accept*.
2. Run M_2 with input w. If M_2 accepts, *accept*.
Otherwise, *reject*."

Thus, M_3 accepts w if and only if either M_1 or M_2 (or both) accepts w. Since M_1 and M_2 are both polynomial-time TMs, the overall running time of M_3 is also polynomial.

(b) Show that P is closed under concatenation.

Answer: Suppose that language $L_1 \in \text{P}$ and language $L_2 \in \text{P}$. Thus, there are polynomial-time TMs M_1 and M_2 that decide L_1 and L_2, respectively. A Turing machine M_3 that decides $L_1 \circ L_2$ is the following:

M_3 = "On input $w = a_1 a_2 \cdots a_n$, with each $a_i \in \Sigma$ a symbol:
1. For $i = 0, 1, 2, \ldots, n$, do
2. Run M_1 with input $w_1 = a_1 a_2 \cdots a_i$ and
run M_2 with input $w_2 = a_{i+1} a_{i+1} \cdots a_n$.
If both M_1 and M_2 accept, *accept*.
3. If none of the iterations in Stage 2 accept, *reject*."

The TM M_3 checks every possible way of splitting the input w into two parts w_1 and w_2, and checks if the first part w_1 is accepted by M_1 and the second part w_2 is accepted by M_2. Stage 2 is executed at most $n + 1$ times. Each time Stage 2 is executed, M_1 and M_2 are run on strings w_1 and w_2 with $|w_1| \leq |w|$ and $|w_2| \leq |w|$. Thus, Stage 2 runs in time that is polynomial in $|w|$. Hence, the overall running time of M_3 is also polynomial.

(c) Show that P is closed under complementation.

Answer: Suppose that language $L_1 \in \text{P}$, so there is a polynomial-time TM M_1 that decides L_1. A Turing machine M_2 that decides $\overline{L_1}$ is the following:

M_2 = "On input w:
1. Run M_1 with input w.
If M_1 accepts, *reject*; otherwise, *accept*."

The TM M_2 just does the opposite of what M_1 does, so M_2 decides $\overline{L_1}$. Since M_1 is a polynomial-time TM, so is M_2.

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 NTMsthat decide them in polynomial time. We construct a NTM M' that decides the union of L_1 and 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 the concatenation of L_1 and 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*."

7.8

The algorithm given in page 145 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 the last marks at least one additional node. Each execution of stage 3 uses at most $O(n^3)$ steps because G contains at most n to be checked and for each checked node, examining all adjacent nodes to see whether any have been marked uses at most $O(n^2)$ steps. 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 vertices v_1, v_2, v_3 in G:
2. If edges (v_1, v_2), (v_1, v_3), and (v_2, v_3), are all edges of G, *accept*.
3. No triangle has been found in G, so *reject*."

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

7.10

It is easy to see that a DFA accepts Σ^* if and only if all reachable states from the start state, q_0, are accepting. This can easily be decided in polynomial-time by performing a breadth- or depth-first search on the DFA from q_0. If at any time a non-accepting state is visited, *reject*, otherwise, if only accepting states are found, *accept*.

Note that this problem is much harder for NFAs; $\{\langle A \rangle \mid A \text{ is an NFA and } L(A) = \Sigma^*\}$ is NP-hard.

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. Stage 3 takes polynomial time. Therefore *ISO* $\in$ NP.

7.13

Show that MODEXP is in P by showing a polynomial algorithm to decide the language.

Let $|a|$, $|b|$, $|c|$, and $|p|$ represent the lengths of a, b, c, and p. Computing $(c \bmod p)$ takes $O(P(|c|,|p|))$, where $P(:)$ denotes a polynomial over the specified variables, while comparing two moduli takes $O(P(|p|))$. In order to compute $(a^b \bmod p)$, we will need to take advantage of the binary representation of b, and of the facts that

$$|k \bmod p| \leq |p|$$

and

$$k^{i2} \bmod p \equiv (k^i \bmod p)^2 \bmod p$$

First we compute $(a \bmod p)$ in $O(P(|p|, |a|))$, then

$$a^2 \bmod p \equiv (a \bmod p)^2 \bmod p$$

the second of which can be computed in $O(P(|a|,|p|))$. This process is repeated for higher powers of two, yielding $(a^4 \bmod p)$, $(a^8 \bmod p)$ up through $(a^q \bmod p)$, which ends when we have reached the first q such that

$$q \geq \log_2(b)$$

The final result is then computed by using

$$a^b \bmod p \equiv \left(\prod_i^{\log_2 b} b_i a^{2^i} \right) \bmod p$$

where each b_i correspond to the digits in the binary expansion of b, and after each multiplication we insert the $\bmod p$ operation to keep the size of the running multiplication less than the size of p. This process takes $O(P(|p|, |a|))$, and so MODEXP is in P.

7.14

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 ts binary representation. The total number of compositions is now at most the length of t, so the algorithm runs in polynomial time.

7.15

We show that P is closed under the star operation by dynamic programming. Let $A \in$ 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 such that $T[i,j] = 1$ if $w_i \cdots w_j \in A^*$ and $T[i,j] = 0$ otherwise, for all $1 \le i \le j \le n$.

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 $1 \le i \le j \le n$.
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 $T[k,j] = 1$, 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.

7.16 Let $A \in$ NP. Construct NTM M to decide A^* in nondeterministic polynomial time.

M = "On input w:

1. Nondeterministically divide w into pieces $w = x_1 x_2 \cdots x_k$.
2. For each x_i, nondeterministically guess the certificates that show $x_i \in A$.
3. Verify all certificates if possible, then *accept*. Otherwise, if verification fails, *reject*."

7.17

Unary representation uses a sequence of 1s to represent a positive integer (e.g., 5 is represented as '11111' in unary). The table shown in Fig. 7.57 is $O(n^2)$, so it is a polynomial time reduction. However, if the numbers in the same proof method are represented in unary, then the width of the table becomes roughly 10^{l+k},

which is exponential. That is why the proof fails to show that *UNARY-SSUM* is NP-complete.

To show that *UNARY-SSUM* is in P, you can try the dynamic programming method. Note that the dynamic programming method would be exponential if the numbers are not represented in unary, but is polynomial when the numbers are represented in unary.

7.18

Let A be any language in P except $A = \emptyset$ and $A = \Sigma^*$. 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 $x_{\text{out}} \notin A$ be two strings that are guaranteed to exist by virtue of our assumptions about 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.

7.21

a. Follow the marking algorithm for recognizing *PATH* while additionally keeping track of the length of the shortest paths discovered. Here is a more detailed description:

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

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

b. First, $LPATH \in$ NP because a nondeterministic machine can guess and verify a simple path of length at least k from a to b. Next, $UHAMPATH \leq_{\text{P}} LPATH$, because the following TM F computes the reduction f.

F = "On input $\langle G, a, b\rangle$, where G is an undirected graph, and a and b are nodes of G.

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

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

7.22

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 $SAT \leq_P DOUBLE\text{-}SAT$. 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 $\phi \wedge (x \vee \bar{x})$, where x is a new variable.**
2. **Output $\langle\phi'\rangle$"**

If $\phi \in SAT$, ϕ' has at least two satisfying assignments because we can obtain two assignments from the original assignment of ϕ by changing the value of x. If $\phi' \in DOUBLE\text{-}SAT$, ϕ is also satisfiable, because x does not appear in ϕ. Therefore $\phi \in SAT$ if and only if $f(\phi) \in DOUBLE\text{-}SAT$.

7.23 We give a polynomial time mapping reduction from *CLIQUE* to *HALF-CLIQUE*. The input to the reduction is a pair $\langle G, k\rangle$ and the reduction produces the graph $\langle H\rangle$ as output where H is as follows. If G has m nodes and $k = m/2$, then $H = G$. If $k < m/2$, then H is the graph obtained from G by adding j nodes, each connected to every one of the original nodes and to each other, where $j = m - 2k$. Thus, H has $m + j = 2m - 2k$ nodes. Observe that G has a k-clique iff H has a clique of size $k + j = m - k$, and so $\langle G, k\rangle \in CLIQUE$ iff $\langle H\rangle \in HALF\text{-}CLIQUE$. If $k > m/2$, then H is the graph obtained by adding j nodes to G without any additional edges, where $j = 2k - m$. Thus, H has $m + j = 2k$ nodes, and so G has a k-clique iff H has a clique of size k. Therefore, $\langle G, k\rangle \in CLIQUE$ iff $\langle H\rangle \in HALF\text{-}CLIQUE$. We also need to show $HALF\text{-}CLIQUE \in$ NP. The certificate is simply the clique.

7.24

a. We give a polynomial time decider M for CNF_2 as follows.

M = "On input $\langle\phi\rangle$ does the following:

1. Consider the first clause of ϕ. If it is of the form x, and there is a clause $\neg x$ in ϕ, reject.
2. Otherwise the first clause is of the form $x \vee A$. Scan through ϕ and if x or $\neg x$ occurs in more than 2 places, reject.
3. If x does not appear negated in other clauses, remove every clause where x occurs. If there remain no clauses in ϕ, accept,
4. If there are two clauses $x \vee A$ and $\neg x \vee B$ in ϕ, remove them from ϕ and add $A \vee B$ to ϕ, and go to stage 1."

It is clear that M is a decider of CNF_2. Observe that every time M processes one variable, either the result is reached (accept or reject) or the number of clauses in ϕ is decreased by 1 or 2. Hence the running time of M is polynomial in terms of

the number of variables.

b. First, CNF_3 is in NP because the following is a polynomial time verifier for CNF_3:

V = "On input $\langle\langle \phi \rangle, c\rangle$:

1. Test whether each variable in ϕ occurs in at most 3 places.
2. Test whether c is a satisfying assignment of ϕ.
3. If both pass, accept; otherwise, reject."

We show that $3SAT \leq_P CNF_3$. We define a polynomial time reduction f from $3SAT \leq_P CNF_3$ as follows. When given an input instance ϕ of $3SAT$, $f(\langle \phi \rangle)$ constructs an instance of CNF_3 according to the following procedure:

1. Pick the first propositional variable (reading from left to right) in ϕ that occurs more than 3 times in the formula, suppose it is p, and suppose it occurs at n places: $(x_1 \vee A_1)$, ..., $(x_n \vee A_n)$, where the x_i are p or $\neg p$. If no variable occurs more than 3 times, output ϕ.
2. Choose fresh variables p_1, ..., p_n, remove all the conjuncts $(x_i \vee A_i)$ from the formula and add as a conjunct the formula $(p_1 \vee A_1) \wedge (\neg p_1 \vee p_2) \wedge (p_2 \vee A_2) \wedge (\neg p_2 \vee p_3) \ldots (p_n \vee A_n) \wedge (\neg p_n \vee p_1)$.
3. Go to stage 1.

Clearly, $f(\langle \phi \rangle)$ is a formula in which every variable occurs at most three times. It is also clear that ϕ satisfiable iff $f(\langle \phi \rangle)$ is satisfiable. f is polynomial in terms of the number of variable in ϕ.

7.26

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.

7.27

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 multigraph, 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.)

7.28

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 correponding cards, True for up and False for down. Because every hole in column 1 is covered, every clause must contain at least one literal assigned True in this assignment. Hence it satisfies ϕ.

7.30

SET-SPLITTING in 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.

7.33 First, *SOLITAIRE* $\in$ NP because we can verify that a solution works, in polynomial time. Second, we show that *3SAT* $\leq_P$ *SOLITAIRE*. Given ϕ with m variables $x_1, \ldots, x_m$ and k clauses $c_1, \ldots, c_k$, construct the following $k \times m$ game G. We assume that ϕ has no clauses that contain both x_i and $\overline{x_i}$ because such clauses may be removed without affecting satisfiability.

If x_i is in clause c_j, put a blue stone in row c_j, column x_i. If $\overline{x_i}$ is in clause c_j, put a red stone in row c_j, column x_i. We can make the board square by repeating a row or adding a blank column as necessary without affecting solvability. We show that ϕ is satisfiable iff G has a solution.

($\rightarrow$) Take a satisfying assignment. If x_i is true (false), remove the red (blue) stones from the corresponding column. So stones corresponding to true literals remain. Because every clause has a true literal, every row has a stone.

($\leftarrow$) Take a game solution. If the red (blue) stones were removed from a column, set the corresponding variable true (false). Every row has a stone remaining, so every clause has a true literal. Therefore, ϕ is satisfied.

7.29

3COLOR is in NP because a coloring can be verified in polynomial time. We show *3SAT* $\leq_P$ *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.

SET-SPLITTING in 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.

7.31

Answer: SCHEDULE= $\{< F, S, T, h > | F$ is set of final exams, S is set of all students, $T = \{(s, f) | s \in S, f \in F$ and student s takes final exam $f\}$ and h is number of slots, there exist a conflict free schedule that uses at most h slots $\}$.

We first note that SCHEDULE is in NP. Given a schedule, we can check that it uses at most h slots and all the exams in any given slot are taken by at most one student.

Now we show that SCHEDULE is NP-Hard by reduction from k-coloring problem. Given a graph G, we construct an instance I as follows. We create an exam for every node and a student for every edge. If there is an edge between two pair of vertices, we make corresponding student take corresponding exams. Finally we set $h = k$. We claim that graph G is K-colorable iff $I \in$ SCHEDULE.

If G is k-colorable, then we construct solution to I as follows. We color the slots arbitrarily. If node is colored with color c, corresponding exam is assigned to slot colored c. Since every node is colored, every exam will be assigned to some slot. Also since nodes connected by an edge receive different colors, for corresponding student, both the exams he is taking will be assigned different slots.

If $I \in$ SCHEDULE, we can construct k-coloring for the graph as follows. Firstly we arbitrarily assign k colors to h slots (h = k, by construction). Now if some exams assigned to some slot, we color corresponding node with the color of that slot. Since for each student, two exams he takes will be assigned to different slots, nodes connected by corresponding edge will receive different colors.

7.34

To show that all problems in NP are polynomial time reducible to D we show that $3SAT \leq_P D$. Given a 3cnf formula ϕ we construct a polynomial p with the same variables. We represent the variable x in ϕ by x in p and its negation $\overline{x}$ by $1 - x$. Next we 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 $x1, \ldots, x_l$ take on Boolean values, and is positive otherwise. Hence polynomial $p = (1 - q)^2 + r$ has an integral root iff ϕ is satisfiable.

7.35

Answer: Note that dominating set has to include all isolated vertices. So we can alwyas construct equvivalent problem that does not allow graphs with isolated vertices and hence we assume our graph does not have isolated vertices.

We first note that DOMINATING-SET is in NP. Given a set of nodes, we can verify that it is subset of nodes of G and every other node in G is adjacent to some node in that set.

Next we show that DOMINATING-SET is NP-Hard by reduction from VERTEX-COVER. Given a graph G for which we want to find the vertex cover, we construct new graph G' as follows. G' contains all the vertices and edges of G. In addition for each edge (u,v) in G, we create a vertex and connect it to both u and v. Let's call first set of vertices (those came from G) as A and other set as B. We claim that $< G, K >\in$VERTEX-COVER iff $G' \in$DOMINATING-SET.

Given a vertex-cover of G, we argue that corresponding vertices of G' constitute dominating set for G'. To see this, note that since every edge in G is covered by vertex cover, for all edges (u,v) where both u and v are in A either u or v belongs to dominating set which covers v and also vertex in B corresponding to edge (u,v). So all vertices in A and B are covered.

Given a dominating set of G', we can change it so that it includes only vertices from set A as follows. If it contain any vertex from B, let (u,v) be the corresponding edge. Then we can safely replace it with either u or v. Now we claim that with modified dominating set of G', corresponding vertices of G constitute vertex cover. Since dominating set is entirely from A, every vertex not in dominating set is covered by adjacent vertex from set A. So for each edge (u,v) where both u and v belong to A, either u or v belong to dominating set of G' and hence corresponding vertex belong to vertex cover of G covering edge (u,v).

7.37

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 $3SAT \leq_P U$, 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 $\phi \in 3SAT$.

7.38

Show that if $P = NP$, there is a polynomial time algorithm that computes a satisfying assignment for a satisfiable Boolean formula.

If $P = NP$ then we have a polynomial time algorithm that indicates weather a Boolean formula is satisfiable or not, call this algorithm PB. Our algorithm, on input f will call PB on f. If PB returns *False*, the Boolean formula is not satisfiable, so our program returns with a negative result. Otherwise we run PB on f after substituting the first literal with *True*, which will let us know if the first literal should be assigned *True* or *False*. After making the correct assignment, we use the same procedure to determine the correct value for the second literal, and the third and so on until we have called PB once for each literal, plus once initially. Our program then returns the assignments made that satisfy the formula f. The total running time of our algorithm then is $(n+1)*O(PB)$, which is in polynomial time.

7.39

Show that if $P = NP$ then there is a polynomial time algorithm to factor integers.

If $P=NP$, then we will have a polynomial time algorithm, say it is called PF, to determine whether or not a number N has a non-trivial integer factor that is less than or equal to M, where $M<N$. On an input N our algorithm will call PF with $M = N/2$, and the original N. A returned value of *False* from PF means that N can't be factored into any smaller numbers, and so is itself a prime number, our program would then return the values *1* and N. If PF returns *True*, we call PF again on N and $M=N/4$. In this way we will essentially be running a Binary Search algorithm, where we will be narrowing down the search interval within which there is a factor by looking in half of the current interval, in order to find the smaller of the factors of N. This will eventually find a factor of N call it p, and we will divide N by p to get the second factor call it q. Our algorithm will then return (p,q). The running time of the algorithm is the running time of binary search times the running time of PF, which is $O(log(N)*Poly(N))$, which is also in $O(Poly(N))$.

7.40 If you assume that P = NP, then $CLIQUE \in$ P, and you can test whether G contains a clique of size k in polynomial time, for any value of k. By testing whether G contains a clique of each size, from 1 to the number of nodes in G, you can determine the size t of a maximum clique in G in polynomial time. Once you know t, you can find a clique with t nodes as follows. For each node x of G, remove x and calculate the resulting maximum clique size. If the resulting size decreases, replace x and continue with the next node. If the resulting size is still t, keep x permanently removed and continue with the next node. When you have considered all nodes in this way, the remaining nodes are a t-clique.

7.41

If we used 2×2 windows, we might not detect certain invalid computation histories where multiple heads occurred in the same row. For example, if the NTM had the nondeterministic choice of moving its head left or right when in a certain state reading some symbol, the next configuration must have one of these possibilities, but not both simultaneously. However, using a 2×2 window, a row that contained two state symbols corresponding to each possible next move would not appear incorrect, because each window would itself be legal.

Furthermore, once two heads appeared in a single row, they could collaborate to make the final row accepting, even though no actual accepting computation existed.

7.43

We build an NFA in polynomial time that accepts all non-satisfying assignments. The NFA operates by guessing a 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_j is represented by $m+1$ states $q_{j,1}, \ldots, q_{j,m+1}$. The start state q_0 branches nondeterministically to each state $q_{1,j}$ 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_i appears in clause c_j, and is 1 if $\overline{x_i}$ appears in c_j. If neither literals appear in c_j, 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.

7.44

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 the two implications above. Then 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 first show that if G contains an inconsistency cycle, no satisfying assignment can exist. The sequences of implications in the inconsistency cycle yields the logical equivalence $x_i \leftrightarrow \overline{x_i}$ for some i, and that is contradictory. Thus ϕ is unsatisfiable.

Next, we show that if G doesn't contain an inconsistency cycle, ϕ is satisfiable. Write $x \xrightarrow{*} y$ 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 $x \xrightarrow{*} y$ iff $\overline{y} \xrightarrow{*} \overline{x}$. Now we construct the satisfying assignment.

Pick any variable x_i. We cannot have both $x_i \xrightarrow{*} \overline{x_i}$ and $\overline{x_i} \xrightarrow{*} x_i$ because G doesn't contain an inconsistency cycle. Select literal x_i if $\overline{x_i} \xrightarrow{*} x_i$ 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 $x_i \xrightarrow{*} x_j$ and $x_i \xrightarrow{*} \overline{x_j}$ then $x_j \xrightarrow{*} \overline{x_i}$ and hence $x_i \xrightarrow{*} \overline{x_i}$ thus 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 ϕ because it satisfies every clause. As soon as one of the literals in a clause is assigned False, the other must be assigned True because G contains an arrow from the negation of the falsified literal to that other literal.

7.47

First we show that Z is in DP. Consider the following two languages:

$Z_1 = \{\langle G_1, k_1, G_2, k_2\rangle \mid G_1 \text{ has a } k_1 \text{ clique, } G_2 \text{ is a graph, and } k_2 \text{ is an integer} > 2\}$
$Z_2 = \{\langle G_1, k_1, G_2, k_2\rangle \mid G_2 \text{ has a } k_2 \text{ clique, } G_1 \text{ is a graph, and } k_1 \text{ is an integer} > 2\}$

Clearly Z_1 and Z_2 are in NP, and $Z = Z_1 \cap$ (complement of Z_2), so Z is in DP.

To show that Z is complete for DP we need to show that for all A in DP, A is polytime reducible to Z. Let $A = A_1 \cap$ (complement of A_2) for NP languages A_1 and A_2. By the NP completeness of CLIQUE, A_1 and A_2 are polytime reducible to CLIQUE. Let f_1 and f_2 denote the corresponding polytime reduction mappings. Then $f = (f_1, f_2)$ is a polytime reduction from A to Z.

7.48

CONNECT NECH NODE IN G1 TO EVERY NODE IN G2.

7.54

Hint: Each edge has two possible orientations in the result. So you can encode this decision with a propositional variable. Next, encode your requirements for nodes in C and outside C as clauses. How many clauses do you need for that?