Chapter 34

Michelle Bodnar, Andrew Lohr

April 12, 2016

Exercise 34.1-1

Showing that LONGEST-PATH-LENGTH being polynomial implies that LONGEST-PATH is polynomial is trivial, because we can just compute the length of the longest path and reject the instance of LONGEST-PATH if and only if k is larger than the number we computed as the length of the longest path.

Since we know that the number of edges in the longest path length is between 0 and $|E|$, we can perform a binary search for it's length. That is, we construct an instance of LONGEST-PATH with the given parameters along with $k = \frac{|E|}{2}$. If we hear yes, we know that the length of the longest path is somewhere above the halfway point. If we hear no, we know it is somewhere below. Since each time we are halving the possible range, we have that the procedure can require $O(\lg(|E|))$ many steps. However, running a polynomial time subroutine $\lg(n)$ many times still gets us a polynomial time procedure, since we know that with this procedure we will never be feeding output of one call of LONGEST-PATH into the next.

Exercise 34.1-2

The problem LONGST-SIMPLE-CYCLE is the relation that associates each instance of a graph with the longest simple cycle contained in that graph. The decision problem is, given k, to determine whether or not the instance graph has a simple cycle of length at least k. If yes, output 1. Otherwise output 0. The language corresponding to the decision problem is the set of all $\langle G, k\rangle$ such that $G = (V, E)$ is an undirected graph, $k \geq 0$ is an integer, and there exists a simple cycle in G consisting of at least k edges.

Exercise 34.1-3

A formal encoding of the adjacency matrix representation is to first encode an integer n in the usual binary encoding representing the number of vertices. Then, there will be n^2 bits following. The value of bit m will be 1 if there is an edge from vertex $\lfloor m/n \rfloor$ to vertex $(m\%n)$, and zero if there is not such an edge.

An encoding of the adjacency list representation is a bit more finessed. We'll be using a different encoding of integers, call it $g(n)$. In particular, we will place a 0 immediately after every bit in the usual representation. Since this only doubles the length of the encoding, it is still polynomially related. Also, the reason we will be using this encoding is because any sequence of integers encoded in this way cannot contain the string 11 and must contain at least one zero. Suppose that we have a vertex with edges going to the vertices indexed by $i_1, i_2, i_3, \ldots, i_k$. Then, the encoding corresponding to that vertex is $g(i_1)11g(i_2)11 \cdots 11g(i_k)1111$. Then, the encoding of the entire graph will be the concatenation of all the encodings of the vertices. As we are reading through, since we used this encoding of the indices of the vertices, we won't ever be confused about where each of the vertex indices end, or when we are moving on to the next vertex's list.

To go from the list to matrix representation, we can read off all the adjacent vertices, store them, sort them, and then output a row of the adjacency matrix. Since there is some small constant amount of space for the adjacency list representation for each vertex in the graph, the size of the encoding blows up by at most a factor of $O(n)$, which means that the size of the encoding overall is at most squared.

To go in the other direction, it is just a matter of keeping track of the positions in a given row that have ones, encoding those numerical values in the way described, and doing this for each row. Since we are only increasing the size of the encoding by a factor of at most $O(\lg(n))$(which happens in the dense graph case), we have that both of them are polynomially related.

Exercise 34.1-4

This isn't a polynomial-time algorithm. Recall that the algorithm from Exercise 16.2-2 had running time $\Theta(nW)$ where W was the maximum weight supported by the knapsack. Consider an encoding of the problem. There is a polynomial encoding of each item by giving the binary representation of its index, worth, and weight, represented as some binary string of length $a = \Omega(n)$. We then encode W, in polynomial time. This will have length $\Theta(\lg W) = b$. The solution to this problem of length $a+b$ is found in time $\Theta(nW) = \Theta(a * 2^b)$. Thus, the algorithm is actually exponential.

Exercise 34.1-5

We show the first half of this exercise by induction on the number of times that we call the polynomial time subroutine. If we only call it zero times, all we are doing is the polynomial amount of extra work, and therefore we have that the whole procedure only takes polynomial time.

Now, suppose we want to show that if we only make $n + 1$ calls to the polynomial time subroutine. Consider the execution of the program up until just before the last call. At this point, by the inductive hypothesis, we have only taken a polynomial amount of time. This means that all of the data that

we have constructed so far fits in a polynomial amount of space. This means that whatever argument we pass into the last polynomial time subroutine will have size bounded by some polynomial. The time that the last call takes is then the composition of two polynomials, and is therefore a polynomial itself. So, since the time before the last call was polynomial and the time of the last call was polynomial, the total time taken is polynomial in the input. This proves the claim of the first half of the input.

To see that it could take exponential time if we were to allow polynomially many calls to the subroutine, it suffices to provide a single example. In particular, let our polynomial time subroutine be the function that squares its input. Then our algorithm will take an integer x as input and then square it $\lg(x)$ many times. Since the size of the input is $\lg(x)$, this is only linearly many calls to the subroutine. However, the value of the end result will be $x^{2^{\lg(x)}} = x^x = 2^{x \lg(x)} = 2^{\lg(x) 2^{\lg(x)}} \in \omega(2^{2^{\lg(x)}})$. So, the output of the function will require exponentially many bits to represent, and so the whole program could not of taken polynomial time.

Exercise 34.1-6

Let $L_1, L_2 \in P$. Then there exist algorithms A_1 and A_2 which decide L_1 and L_2 in polynomial time. We will use these to determine membership in the given languages. An input x is in $L_1 \cup L_2$ if and only if either A_1 or A_2 returns 1 when run on input x. We can check this by running each algorithm separately, each in polynomial time. To check if x is in $L_1 \cap L_2$, again run A_1 and A_2, and return 1 only if A_1 and A_2 each return 1. Now let $n = |x|$. For $i = 1$ to n, check if the first i bits of x are in L_1 and the last $n - i$ bits of x are in L_2. If this is ever true, then $x \in L_1 L_2$ so we return 1. Otherwise return 0. Each check is performed in time $O(n^k)$ for some k, so the total runtime is $O(n(n^k + n^k)) = O(n^{k+1})$ which is still polynomial. To check if $x \in \overline{L_1}$, run A_1 and return 1 if and only if A_1 returns 0. Finally, we need to determine if $x \in L_1^*$. To do this, for $i = 1$ to n, check if the first i bits of x are in L_1, and the last $n - i$ bits are in L_1^*. Let $T(n)$ denote the running time for input of size n, and let cn^k be an upper bound on the time to check if something of length n is in L_1. Then $T(n) \leq \sum_{i=1}^{n} cn^k T(n-i)$. Observe that $T(1) \leq c$ since a single bit is in L_1^* if and only if it is in L_1. Now suppose $T(m) \leq c' m^{k'}$. Then we have:

$$T(n) \leq cn^{k+1} + \sum_{i=0}^{n-1} c' i^{k'} \leq cn^{k+1} + c' n^{k'+1} = O(n^{\max k, k'}).$$

Thus, by induction, the runtime is polynomial for all n. Since we have exhibited polynomial time procedures to decide membership in each of the languages, they are all in P, so P is closed under union, intersection, concatenation, complement, and Kleene star.

Exercise 34.2-1

To verify the language, we should let the certificate be the mapping f from the vertices of G_1 to the vertices of G_2 that is an isomorphism between the graphs. Then all the verifier needs to do is verify that for all pairs of vertices u and v, they are adjacent in G_1 if and only if $f(u)$ is adjacent to $f(v)$ in G_2. Clearly it is possible to produce an isomorphism if and only if the graphs are isomorphic, as this is how we defined what it means for graphs to be isomorphic.

Exercise 34.2-2

Since G is bipartite we can write its vertex set as the disjoint union $S \sqcup T$, where neither S nor T is empty. Since G has an odd number of vertices, exactly one of S and T has an odd number of vertices. Without loss of generality, suppose it is S. Let $v_1, v_2, \ldots, v_n$ be a simple cycle in G, with $v_1 \in S$. Since n is odd, we have $v_1 \in S$, $v_2 \in T$, $\ldots$, $v_{n-1} \in T$, $v_n \in S$. There can be no edge between v_1 and v_n since they're both in S, so the cycle can't be Hamiltonian.

Exercise 34.2-3

Suppose that G is hamiltonian. This means that there is a hamiltonian cycle. Pick any one vertex v in the graph, and consider all the possibilities of deleting all but two of the edges passing through that vertex. For some pair of edges to save, the resulting graph must still be hamiltonian because the hamiltonian cycle that existed originally only used two edges. Since the degree of a vertex is bounded by the number of vertices minus one, we are only less than squaring that number by looking at all pairs ($\binom{n-1}{2} \in O(n^2)$). This means that we are only running the polynomial tester polynomially many independent times, so the runtime is polynomial. Once we have some pair of vertices where deleting all the others coming off of v still results in a hamiltonian graph, we will remember those as special, and ones that we will never again try to delete. We repeat the process with both of the vertices that are now adjacent to v, testing hamiltonicity of each way of picking a new vertex to save. We continue in this process until we are left with only $|V|$ edge, and so, we have just constructed a hamiltonian cycle.

Exercise 34.2-4

This is much like Exercise 34.1-6. Let L_1 and L_2 be languages in NP, with verification algorithms A_1 and A_2. Given input x and certificate y for a language in $L_1 \cup L_2$, we define A_3 to be the algorithm which returns 1 if either $A_1(x, y) = 1$ or $A_2(x, y) = 1$. This is a polynomial verification algorithm for $L_1 \cup L_2$, so NP is closed under unions. For intersection, define A_3 to return 1 if and only if $A_1(x, y)$ and $A_2(x, y)$ return 1. For concatenation, define A_3 to loop through $i = 1$ to n, checking each time if $A_1(x[1..i], y[1..i]) = 1$ and $A_1(x[i+1..n], y[i+1..n]) = 1$. If so, terminate and return 1. If the loop ends, return 0. This still takes polynomial time, so NP is closed under concatenation. Finally, we need to check Kleene star. Define A_3 to loop through $i = 1$ to n, each time checking if $A_1(x[1..i], y[1..i]) = 1$,

and $y[i+1..n]$ is a certificate for $x[i+1..n]$ being in L_1^*. Let $T(n)$ denote the running time for input of size n, and let cn^k be an upper bound on the time to verify that y is a certificate for x. Then $T(n) \leq \sum_{i=1}^{n} cn^k T(n-i)$. Observe that $T(1) \leq c$ since we can verify a certificate for a problem of length 1 in constant time. Now suppose $T(m) \leq c'm^{k'}$. Then we have:

$$T(n) \leq cn^{k+1} + \sum_{i=0}^{n-1} c'i^{k'} \leq cn^{k+1} + c'n^{k'+1} = O(n^{\max k,k'}).$$

Thus, by induction, the runtime of A_3 is polynomial. Note that we only needed to deal with the runtime recursion with respect to the length of x, since it is assumed $|y| = O(|x|^c)$ for some constant c. Therefore NP is closed under Kleene star.

A proof for closure under complement breaks down, however. If a certificate y is given for input x and $A_1(x, y)$ returns false, this doesn't tell us that y is a certificate for x being in the complement of L_1. It merely tells us that y didn't prove $x \in L_1$.

Exercise 34.2-5

Suppose that we know that the length of the certificates to the verifier are bounded by n^{k-1}, we know it has to be bounded by some polynomial because the verifier can only look at polynomially many bits because it runs in polynomial time. Then, we try to run the verifier on every possible assignment to each bit of the certificates of length up to that much. Then, the runtime of this will be a polynomial times $2^{n^{k-1}}$ which is little oh of 2^{n^k}.

Exercise 34.2-6

The certificate in this case would be a list of vertices $v_1, v_2, \ldots, v_n$, starting with u and ending with v, such that each vertex of G is listed exactly once and $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq n-1$. Since we can check this in polynomial time, HAM-PATH belongs to NP.

Exercise 34.2-7

For a directed acyclic graph, we can compute the topological sort of the graph by the method of section 22.4. Then, looking at this sorting of the vertices, we say that there is a Hamiltonian path if as we read off the vertices, each is adjacent to the next, and they are not if there is any pair of vertices so that one is not adjacent to the next.

If we are in the case that each is adjacent to the next, then the topological sort itself gives us the Hamiltonian path. However, if there is any pair of vertices so that one is not adjacent to the next, this means that that pair of vertices do not have any paths going from one to the other. This would clearly imply that

there was no Hamiltonian path, because the Hamiltonian path would be going from one of them to the other.

To see the claim that a pair of vertices u and v that are adjacent in a topological sort but are not adjacent have no paths going from one to the other, suppose to a contradiction there were such a path from u to v. If there were any vertices along this path they would have to be after u since they are descendants of u and they would have to be before v because they are ancestors of v. This would contradict the fact that we said u and v were adjacent in a topological sort. Then the path would have to be a single edge from u to v, but we said that they weren't adjacent, and so, we have that there is no such path.

Exercise 34.2-8

Let L' be the complement of TAUTOLOGY. Then L' consists of all boolean formulas ϕ such that there exists an assignment $y_1, y_2, \ldots, y_k$ of 0 and 1 to the input variables which causes ϕ to evaluate to 0. A certificate would be such an assignment. Since we can check what an assignment evaluates to in polynomial time in the length of the input, we can verify a certificate in polynomial time. Thus, $L' \in$ NP which implies TAUTOLOGY $\in$ co-NP.

Exercise 34.2-9

A language is in $coNP$ if there is a procedure that can verify that an input is not in the language in polynomial time given some certificate. Suppose that for that language we a procedure that could compute whether an input was in the language in polynomial time receiving no certificate. This is exactly what the case is if we have that the language is in P. Then, we can pick our procedure to verify that an element is not in the set to be running the polynomial time procedure and just looking at the result of that, disregarding the certificate that is given. This then shows that any language that is in P is in $coNP$, giving us the inclusion that we wanted.

Exercise 34.2-10

Suppose NP $\neq$ co-NP. Let $L \in$ NP \co-NP. Since $P \subset$ NP $\cap$ co-NP, and $L \notin$ NP $\cap$ co-NP, we have $L \notin P$. Thus, P $\neq$ NP.

Exercise 34.2-11

As the hint suggests, we will perform a proof by induction. For the base case, we will have 3 vertices, and then, by enumeration, we can see that the only Hamiltonian graph on three vertices is K_3. For any connected graph on three vertices, the longest the path connecting them can be is 2 edges, and so we will have $G^3 = K_3$, meaning that the graph G was Hamiltonian.

Now, suppose that we want to show that the graph G on $n+1$ vertices has the property that G^3 is Hamiltonian. Since the graph G is connected we know

that there is some spanning tree by Chapter 23. Then, let v be any internal vertex of that tree. Suppose that if we were to remove the vertex v, we would be splitting up the original graph in the connected components $V_1, V_2, \ldots, V_k$, sorted in increasing order of size. Suppose that the first ℓ_1 of these components have a single vertex. Suppose that the first ℓ_2 of these components have fewer than 3 vertices. Then, let v_i be the vertex of V_i that is adjacent to v in the tree. For $i > \ell_1$, let x_i be any vertex of V_i that is distance two from the vertex v. By induction, we have Hamiltonian cycles for each of the components $V_{\ell_2+1}, \ldots, V_k$. In particular, there is a Hamiltonian path from v_i to x_i. Then, for each i and j there is an edge from x_j to v_i, because there is a path of length three between them passing through v. This means that we can string together the Hamiltonian paths from each of the components with $i > \ell_1$. Lastly, since $V_1, \ldots, V_\ell$ all consist of single vertices that are only distance one from v, they are all adjacent in G^3. So, after stringing together the Hamiltonian paths for $i > \ell_1$, we just visit all of the single vertices in $v_1, v_2, \ldots, v_{\ell_1}$ in order, then, go to v and then to the vertex that we started this path at, since it was selected to be adjacent to v, this is possible. Since we have constructed a Hamiltonian cycle, we have completed the proof.

Exercise 34.3-1

The formula in figure 34.8b is

$$((x_1 \vee x_2) \wedge (\neg(\neg x_3))) \wedge (\neg(\neg x_3) \vee ((x_1) \wedge (\neg x_3) \wedge (x_2))) \wedge ((x_1) \wedge (\neg x_3) \wedge (x_2))$$

We can cancel out the double negation to get that this is the same expression as

$$((x_1 \vee x_2) \wedge (x_3)) \wedge ((x_3) \vee ((x_1) \wedge (\neg x_3) \wedge (x_2))) \wedge ((x_1) \wedge (\neg x_3) \wedge (x_2))$$

Then, the first clause can only be true if x_3 is true. But the last clause can only be true if $\neg x_3$ is true. This would be a contradiction, so we cannot have both the first and last clauses be true, and so the boolean circuit is not satisfiable since we would be taking the and of these two quantities which cannot both be true.

Exercise 34.3-2

Suppose $L_1 \leq_P L_2$ and let f_1 be the polynomial time reduction function such that $x \in L_1$ if and only if $f_1(x) \in L_2$. Similarly, suppose $L_2 \leq_P L_3$ and let f_2 be the polynomial time reduction function such that $x \in L_2$ if and only if $f_2(x) \in L_3$. Then we can compute $f_2 \circ f_1$ in polynomial time, and $x \in L_1$ if and only if $f_2(f_1(x)) \in L_3$. Therefore $L_1 \leq_P L_3$, so the $\leq_P$ relation is transitive.

Exercise 34.3-3

Suppose first that we had some polynomial time reduction from L to $\bar{L}$. This means that for every x there is some $f(x)$ so that $x \in L$ iff $f(x) \in \bar{L}$. This means that $x \in \bar{L}$ iff $x \notin L$ iff $f(x) \notin \bar{L}$ iff $f(x) \in L$. So, our poly-time computable function for the reduction is the same one that we had from $L \leq_P \bar{L}$. We can do an identical thing for the other direction.

Exercise 34.3-4

We could have instead used as a certificate a satisfying assignment to the input variables in the proof of Lemma 34.5. We construct the two-input, polynomial time algorithm A to verify CIRCUIT-SAT as follows. The first input is a standard encoding of a boolean combinatiorial circuit C, and the second is a satisfying assignment of the input variables. We need to compute the output of each logic gate until the final one, and then check whether or not the output of the final gate is 1. This is more complicated than the approach taken in the text, because we can only evaluate the output of a logic gate once we have successfully determined all input values, so the order in which we examine the gates matters. However, this can still be computed in polynomial time by essentially performing a breath-first search on the circuit. Each time we reach a gate via a wire we check whether or not all of its inputs have been computed. If yes, evaluate that gate. Otherwise, continue the search to find other gates, all of whose inputs have been computed.

Exercise 34.3-5
We do not have any loss of generality by this assumption. This is because since we bounded the amount of time that the program has to run to be polynomial, there is no way that the program can access more than an polynomial amount of space. That is, there is no way of moving the head of the turning machine further than polynomially far in only polynomial time because it can move only a single cell at a time.

Exercise 34.3-6

Suppose that $\emptyset$ is complete for P. Let $L = \{0,1\}^*$. Then L is clearly in P, and there exists a polynomial time reduction function f such that $x \in \emptyset$ if and only if $f(x) \in L$. However, it's never true that $x \in \emptyset$, so this means it's never true that $f(x) \in L$, a contradiction since every input is in L. Now suppose $\{0,1\}^*$ is complete for P, let $L' = \emptyset$. Then L' is in P and there exists a polynomial time reduction function f'. Then $x \in \{0,1\}^*$ if and only if $f'(x) \in L'$. However x is always in $\{0,1\}^*$, so this implies $f'(x) \in L'$ is always true, a contradiction because no binary input is in L'.

Finally, let L be some language in P which is not $\emptyset$ or $\{0,1\}^*$, and let L' be any other language in P. Let $y_1 \notin L'$ and $y_2 \in L'$. Since $L \in P$, there

exists a polynomial time algorithm A which returns 0 if $x \notin L$ and 1 if $x \in L$. Define $f(x) = y_1$ if $A(x)$ returns 0 and $f(x) = y_2$ if $A(x)$ returns 1. Then f is computable in polynomial time and $x \in L$ if and only if $f(x) \in L'$. Thus, $L' \leq_P L$.

Exercise 34.3-7

Since L is in NP, we have that $\bar{L} \in coNP$ because we could just run our verification algorithm to verify that a given x is not in the complement of L, this is the same as verifying that x is in L. Since every coNP language has its complement in NP, suppose that we let S be any language in $coNP$ and let $\bar{S}$ be its compliment. Suppose that we have some polynomial time reduction f from $\bar{S} \in NP$ to L. Then, consider using the same reduction function. We will have that $x \in S$ iff $x \notin \bar{S}$ iff $f(x) \notin L$ iff $f(x) \in \bar{L}$. This shows that this choice of reduction function does work. So, we have shown that the compliment of any NP complete problem is also NP complete. To see the other direction, we just negate everything, and the proof goes through identically.

Exercise 34.3-8

To prove that a language L is NP-hard, one need not actually construct the polynomial time reduction algorithm F to compute the reduction function f for every $L' \in NP$. Rather, it is only necessary to prove that such an algorithm exists. Thus, it doesn't matter that F doesn't know A. If L' is in NP, we know that A exists. If A exists, dependent on k and that big-oh constant, we know that F exists and runs in polynomial time, and this is sufficient to prove CIRCUIT-SAT is NP-hard.

Exercise 34.4-1

Suppose that it is a circuit on two inputs, and then, we have n rounds of two and gates each, both of which take both of the two wires from the two gates from the previous round. Since the formulas for each round will consist of two copies of the formulas from the previous round, it will have an exponential size formula.

Exercise 34.4-2

To make this more readable, we'll just find the 3-CNF formula for each term listed in the AND of clauses for ϕ' on page 1083, including the auxiliary variables p and q as necessary.

$$
\begin{aligned}
y &= (y \vee p \vee q) \wedge (y \vee p \vee \neg q) \wedge (y \vee \neg p \vee q) \wedge (y \vee \neg p \vee \neg q) \\
(y_1 \leftrightarrow (y_2 \wedge \neg x_2)) &= (\neg y_1 \vee \neg y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee x_2) \wedge (y_1 \vee \neg y_2 \vee x_2) \\
(y_2 \leftrightarrow (y_3 \vee y_4)) &= (\neg y_2 \vee y_3 \vee y_4) \wedge (y_2 \vee \neg y_3 \vee \neg y_4) \wedge (y_2 \vee \neg y_3 \vee y_4) \wedge (y_2 \vee y_3 \vee \neg y_4) \\
(y_3 \leftrightarrow (x_1 \rightarrow x_2)) &= (\neg y_3 \vee \neg x_2 \vee x_2) \wedge (y_3 \vee \neg x_1 \vee \neg x_2) \wedge (y_1 \vee x_1 \vee \neg x_2) \wedge (y_3 \vee x_1 \vee x_2) \\
(y_4 \leftrightarrow \neg y_5) &= (\neg x_4 \vee \neg y_5 \vee q) \wedge (\neg x_4 \vee \neg y_5 \vee \neg p) \wedge (x_4 \vee y_5 \vee p) \wedge (x_4 \vee y_5 \vee \neg p) \\
(y_5 \leftrightarrow (y_6 \vee x_4)) &= (\neg y_5 \vee y_6 \vee x_4) \wedge (y_5 \vee \neg y_6 \vee \neg x_4) \wedge (y_5 \vee \neg y_6 \vee x_4) \wedge (y_5 \vee y_6 \vee \neg x_4) \\
(y_6 \leftrightarrow (\neg x_1 \leftrightarrow x_3)) &= (\neg y_6 \vee \neg x_1 \vee \neg x_3) \wedge (\neg y_6 \vee x_1 \vee x_3) \wedge (y_6 \vee \neg x_1 \vee x_3) \wedge (y_6 \vee x_1 \vee \neg x_3).
\end{aligned}
$$

Exercise 34.4-3

The formula could have $\Omega(n)$ free variables, then, the truth table corresponding to this formula would a number of rows that is $\Omega(2^n)$ since it needs to consider every possible assignment to all the variables. This then means that the reduction as described is going to incrase the size of the problem exponentially.

Exercise 34.4-4

To show that the language $L =$ TAUTOLOGY is complete for co-NP, it will suffice to show that $\overline{L}$ is NP-complete, where $\overline{L}$ is the set of all boolean formulas for which there exists an assignment of input variables which makes it false. We showed in Exercise 34.2-8 that TAUTOLOGY $\in$ co-NP, which implies $\overline{L} \in$ NP. Thus, we need only show that $\overline{L}$ is NP-hard. We'll give a polynomial time reduction from the problem of determining satisfiability of boolean formulas to determining whether or not a boolean formula fails to be a tautology. In particular, given a boolean formula ϕ, the negation of ϕ has a satisfying assignment if and only if ϕ has an assignment which causes it to evaluate to 0. Thus, our function will simply negate the input formula. Since this can be done in polynomial time in the length of the input, boolean satisfiability is polynomial time reducible to $\overline{L}$. Therefore $\overline{L}$ is NP-complete. By Exercise 34.3-7, this implies TAUTOLOGY is complete for co-NP.

Exercise 34.4-5

Since the problem is in disjunctive normal form, we can write it as $\vee_i \phi_i$ where each ϕ_i looks like the and of a bunch of variables and their negations. Then, we know that the formula is satisfiable if and only if any one if the ϕ_i are satisfiable. If a ϕ_i contains both a variable and its negation, then it is clearly not satisfiable, as one of the two must be false. However, if each variable showing up doesn't have its negation showing up, then we can just pick the appropriate value to assign to each variable. This is a property that can be checked in linear time, by just keeping two bit vectors of length equal to the number of variables, one representing if the variable has shown up negated and one for if the variable

has shown up without having been negated.

Exercise 34.4-6

Let A denote the polynomial time algorithm which returns 1 if input x is a satisfiable formula, and 0 otherwise. We'll define an algorithm A' to give a satisfying assignment. Let $x_1, x_2, \ldots, x_m$ be the input variables. In polynomial time, A' computes the boolean formula x' which results from replacing x_1 with true. Then, A' runs A on this formula. If it is satisfiable, then we have reduced the problem to finding a satisfying assignment for x' with input variables $x_2, \ldots, x_m$, so A' recursively calls itself. If x' is not satisfiable, then we set x_1 to false, and need to find a satisfying assignment for the formula x' obtained by replacing x_1 in x by false. Again, A' recursively calls itself to do this. If $m = 1$, A' takes a polynomial-time brute force approach by evaluating the formula when x_m is true, and when x_m is false. Let $T(n)$ denote the runtime of A' on a formula whose encoding is of length n. Note that we must have $m \leq n$. Then we have $T(n) = O(n^k) + T(n')$ for some k and $n' = |x'|$, and $T(1) = O(1)$. Since we make a recursive call once for each input variable there are m recursive calls, and the input size strictly decreases each time, so the overall runtime is still polynomial.

Exercise 34.4-7

Suppose that the original formula was $\wedge_i(x_i \vee y_i)$, and the set of variables were $\{a_i\}$. Then, consider the directed graph which has a vertex corresponding both to each variable, and each negation of a variable. Then, for each of the clauses $x \vee y$, we will place an edge going from $\neg x$ to y, and an edge from $\neg y$ to x. Then, anytime that there is an edge in the directed graph, that means if the vertex the edge is coming from is true, the vertex the edge is going to has to be true. Then, what we would need to see in order to say that the formula is satisfiable is a path from a vertex to the negation of that vertex, or vice versa. The naive way of doing this would be to run all pairs shortest path, and see if there is a path from a vertex to its negation. This however takes time $O(n^2 \lg(n))$, and we were charged with making the algorithm as efficient as possible. First, run the procedure for detecting strongly connected components, which takes linear time. For every pair of variable and negation, make sure that they are not in the same strongly connected component. Since our construction was symmetric with respect to taking negations, if there were a path from a variable to its negation, there would be a path going from its negation to itself as well. This means that we would detect any path from a variable to its negation, just by checking to see if they are contained in the same connected component or not.

Exercise 34.5-1

To do this, first, notice that it is in NP, where the certificate is just the injection from G_1 into G_2 so that G_1 is isomorphic to its image.

Now, to see that it is NP complete, we will do a reduction to clique. That

is, to detect if a graph has a clique of size k, just let G_1 be a complete graph on k vertices and let G_2 be the original graph. If we could solve the subgraph isomorphism problem quickly, this would allow us to solve the clique problem quickly.

Exercise 34.5-2

A certificate would be the n-vector x, and we can verify in polynomial time that $Ax \leq b$, so 0-1 integer linear programming (01LP) is in NP. To prove that 01LP is NP-hard, we show that 3-CNF-SAT $\leq_P$ 01LP. Let ϕ be 3-CNF formula with n input variables and k clauses. We construct an instance of 01LP as follows. Let A be a $k + 2n$ by $2n$ matrix. For $1 \leq i \leq k$, set entry $A(i, j)$ to -1 if $1 \leq j \leq n$ and clause C_i contains the literal x_j. Otherwise set it to 0. For $n+1 \leq j \leq 2n$, set entry $A(i, j)$ to -1 if clause C_i contains the literal $\neg x_{j-n}$, and 0 otherwise. When $k+1 \leq i \leq k+n$, set $A(i, j) = 1$ if $i-k = j$ or $i-k = j-n$, and 0 otherwise. When $k+n+1 \leq i \leq k+2n$, set $A(i, j) = -1$ if $i-k-n = j$ or $i-k-n = j-n$, and 0 otherwise. Let b be a $k + 2n$-vector. Set the first k entries to -1, the next n entries to 1, and the last n entries to -1. It is clear that we can construct A and b in polynomial time.

We now show that ϕ has a satisfying assignment if and only if there exists a 0-1 vector x such that $Ax \leq b$. First, suppose ϕ has a satisfying assignment. For $1 \leq i \leq n$, f x_i is true, make $x[i] = 1$ and $x[n + i] = 0$. If x_i is false, set $x[i] = 0$ and $x[n + i] = 1$. Since clause C_i is satisfied, there must exist some literal in it which makes it true. If it is x_j, then $x[j] = 1$ and $A(i, j) = -1$, so we get a contribution of -1 to the i^{th} row of b. Since every entry in the upper k by $2n$ submatrix of A is nonpositive and every entry of x is nonnegative, there can be no positive contributions to the i^{th} row of b. Thus, we are guaranteed that the i^{th} row of Ax is at most -1. The same argument applies if the literal $\neg x_j$ makes clause i true. For $1 \leq m \leq n$, at most one of x_m and $\neg x_m$ can be true, so at most one of $x[m]$ and $x[m + n]$ can be true. When we multiply row $k + m$ by x, we get the number of 1's among $x[m]$ and $x[m + n]$. Thus, the $(k + m)^{th}$ row of b is at most 1, as required. Finally, when we multiply row $k + n + m$ of A by x, we get negative 1 times the number of 1's among $x[m]$ and $x[m + n]$. Since this is at least 1, the $(k + n + m)^{th}$ row of b is at most -1. Therefore all inequalities are satisfied, so x is a 0-1 solution to $Ax = b$.

Next we must show that any 0-1 solution to $Ax = b$ provides a satisfying assignment. Let x be such a 0-1 solution. The inequalities ensured by the last $2n$ rows of b guarantee that exactly one of $x[m]$ and $x[n + m]$ is set equal to 1 for $1 \leq m \leq n$. In particular, this means that each x_i is either true or false, but not both. Let this be our candidate satisfying assignment. By the construction of A, we get a contribution of -1 to row i of Ax every time a literal in C_i is true, based on our candidate assignment, and a 0 contribution every time a literal is false. Since row i of b is -1, this guarantees at least one literal which is true per our assignment in each clause. Since this holds for each of the k clauses, the candidate assignment is in fact a satisfying assignment. Therefore 01LP is NP-complete.

Exercise 34.5-3

We will try to show a reduction to the 0-1 integer programming problem. To see this, we will take our A from the 0-1 integer programming problem, and tack on a copy of the $n \times n$ identity matrix to its bottom, and tack on n ones to the end of b from teh 0-1 integer programming problem. This has the effect of adding the restrictions that every entry of x must be at most 1. However, since, for every i, we needed x_i to be an integer anyways, this only leaves the option that $x_i = 0$ or $x_i = 1$. This means that by adding these restrictions, we have that any solution to this system will be a solution to the 0-1 integer programming problem given by A and b.

Exercise 34.5-4

We can solve the problem using dynamic programming. Suppose there are n integers in S. Create a t by n table to solve the problem just as in the solution to the 0-1 knapsack problem described in Exercise 16.2-2. This has runtime $O(tn \lg t)$, since without loss of generality we may assume that every integer in S is less than or equal to t, otherwise we know it won't be included in the solution, and we can check for this in polynomial time. The extra $\lg t$ term comes from the fact that each addition takes $O(\lg t)$ time. Moreover, we can assume that S contains at most t^2 integers. If t is expressed in unary then the length of the problem is at most $O(t + t^2 \lg t) = O(t^3)$, since we express the integers in S in binary. The time to solve it is $O(t^4)$. Thus, the time to compute the solution is polynomial in the length of the input.

Exercise 34.5-5

We will be performing a reduction from the subset sum problem. Suppose that S and t are our set and target from our subset sum problem. Let x be equal to $\sum_{s \in S} s$. Then, we will add the elements $x + t, 2x - t$. Once we have added the elements, note that the sum of all of the elements in the new set S' will be $4x$. We also know that we cannot have both of the new elements that we added be on same side of the partition, because they add up to $3x$ which is three times all the other elements combined. Now, this set of elements will be what we pass into our set partition solver. Note that since the total is $4x$, each side will add up to $2x$. This means that if we look at the elements that on the same side as, but not equal to $2x - t$, they must add up to t. Since they were also members of the original set S, this means that they are a subset with the desired sum, solving the original instance of subset sum. Since it was proved in the section that subset sum is NP-complete, this proves that the set-partition problem is NP hard.

To see that it is in NP, just let the certificate be the set of elements of S that forms one side of the partition. It is linear time to add them up and make sure that they are exactly half the sum of all the elements in S.

Exercise 34.5-6

We'll show that the hamiltonian-path problem HAM-PATH is NP-complete. First, a certificate would be a list $\{v_1, v_2, \ldots, v_n\}$ of the vertices of the path, in order. We can check in polynomial time whether or not $\{v_i, v_{i+1}\}$ is an edge for $1 \leq i \leq n-1$. Thus, HAM-PATH is in NP.

Next, we'll show that HAM-PATH is NP-complete by showing that HAM-CYCLE $\leq_P$ HAM-PATH. Let $G = (V, E)$ be any graph. We'll construct a graph G' as follows. For each edge $e_i \in E$, let G_{e_i} denote the graph with vertex set V and edge set $E - e_i$. Let e_i have the form $\{u_i, v_i\}$. Now, G' will contain one copy of G_{e_i} for each $e_i \in E$. Additionally, G' will contain a vertex x connected to u_1, an edge from v_i to u_{i+1} for $1 \leq i \leq |E| - 1$, and a vertex y and edge from $v_{|E|}$ to y. It is clear that we can construct G' from G in time polyomial in the size of G.

If G has a Hamiltonian cycle, then G_{e_i} has a Hamiltonian path starting at u_i and ending at v_i for each i. Thus, G' has a Hamiltonian cycle from x to y, obtained by taking each of these paths one after another. On the other hand, suppose G fails to have a Hamiltonian cycle. Since x and y have degree 1, the only way G' can have a Hamiltonian path is if it starts at x and ends at y. Moreover, since $\{v_1, u_2\}$ is a cut edge, it must be in the Hamiltonian path if it exists. Since we can not traverse this edge a second time, any Hamiltonian path must start with a Hamiltonian path from x to v_1. However, this means there is a Hamiltonian path from u_1 to v_1. Since $\{u_1, v_1\}$ is an edge in G, this implies there is a Hamiltonian cycle in G, a contradiction. Thus, G has a Hamiltonian cycle if and only if G' has a Hamiltonian path. Therefore HAM-PATH is NP-hard, so HAM-PATH is in fact NP-complete.

Exercise 34.5-7

The related decision problem is to, given a graph G and integer k decide if there is a simple cycle of length at least k in the graph G. To see that this problem is in NP, just let the certificate be the cycle itself. It is really easy just to walk along this cycle, keeping track of what vertices you've already seen, and making sure they don't get repeated.

To see that it is NP-hard, we will be doing a reduction to Hamilton cycle. Suppose we have a graph G and want to know if it is Hamilton. We then create an instance of the decision problem asking if the graph has a simply cycle of length at least $|V|$ vertices. If it does then there is a Hamiltonian cycle. If there is not, then there cannot be any Hamiltonian cycle.

Exercise 34.5-8

A certificate would be an assignment to input variables which causes exactly half the clauses to evaluate to 1, and the other half to evaluate to 0. Since we can check this in polynomial time, half 3-CNF is in NP. To prove that it's

NP-hard, we'll show that 3-CNF-SAT $\leq_p$ HALF-3-CNF. Let ϕ be any 3-CNF formula with m clauses and input variables $x_1, x_2, \ldots, x_n$. Let T be the formula $(y \vee y \vee \neg y)$, and let F be the formula $(y \vee y \vee y)$. Let $\phi' = \phi \wedge T \wedge \ldots \wedge T \wedge F \wedge \ldots \wedge F$ where there are m copies of T and $2m$ copies of F. Then ϕ' has $4m$ clauses and can be constructed from ϕ in polynomial time. Suppose that ϕ has a satisfying assignment. Then by setting $y = 0$ and the x_i's to the satisfying assignment, we satisfy the m clauses of ϕ and the m T clauses, but none of the F clauses. Thus, ϕ' has an assignment which satisfies exactly half of its clauses. On the other hand, suppose there is no satisfying assignment to ϕ. The m T clauses are always satisfied. If we set $y = 0$ then the total number of clauses satisfies in ϕ' is strictly less than $2m$, since each of the $2m$ F clauses is false, and at least one of the ϕ clauses is false. If we set $y = 1$, then strictly more than half the clauses of ϕ' are satisfied, since the $3m$ T and F clauses are all satisfied. Thus, ϕ has a satisfying assignment if and only if ϕ' has an assignment which satisfies exactly half of its clauses. We conclude that HALF-3-CNF is NP-hard, and hence NP-complete.

Problem 34-1

a) The related decision problem should be to, given a graph and a number k decide whether or not there is some independent set of size at least k. If we take the compliment of the given graph, then it will have a clique of size at least k if and only if the original graph has an independent set of size at least k. This is because if we take any set of vertices in the original graph, then it will be an independent set if and only if there are no edges between those vertices. However, in the compliment graph, this means that between every one of those vertices, there is an edge, which means they form a clique. So, to decide independent set, just decide clique in the compliment.

b) We know that since all independent sets are subsets of the set of vertices, then the size of the largest independent set will be an integer in the range $1..|V|$. Then, we will perform a binary search on this space of valid sizes of the largest independent set. That is, we pick the middle element, ask if there is an independent set of that size, if there is, we know we are in the upper half of this range of values for the size of the largest independent set, if not, then we are in the lower half. The total runtime of this procedure to find the size of the largest independent set will only be a factor of $\lg(|V||)$ higher than the solution to the decision problem. Call the size of the largest independent set k.

   Now, for every pair of vertices, try adding an edge, and check if the procedure from before determines that the size of the largest independent set has decreased. If it hasn't that means that that pair of vertices doesn't prevent us from attaining an independent set of the given size. That is, we aren't in the case that there is only one maximal set of the given size and that pair of vertices belongs to it. So, add that edge to the graph, and continue in this

fashion for every pair of vertices. Once we are done, the size of the largest independent set will be the same, and we will have that every edge is filled in except for those going between an independent set of the given size. So, we just list off all the vertices whose degree is less than $|V| - 1$ as being members of our independent set.

c) Since every vertex has degree 2, and so self edges are allowed, the graph must look like a number of disjoint cycles. We can then consider the independent set problem separately for each of the cycles. If we have an even cycle, the largest independent set possible is half the vertices, by selecting them to be alternating. If it is an odd cycle, then we can do half rounded down, since when we get back to the start, we are in the awkward place where there are two unselected vertices between two selected vertices. It's easy to see that these are tight, because there is so little freedom in selecting an independent set in a cycle. So, to calculate the size of the smallest independent set, look at the sizes of each cycle c_i, then, the largest independent set will have size $\lfloor \frac{c_i}{2} \rfloor$.

d) First, find a maximal matching. This can be done in time $O(VE)$ by the methods of section 26.3. let $f(x)$ be defined for all vertices that were matched, and let it evaluate to the point that is paired with x in the maximal matching. Then, we do the following procedure. Let S_1 be the unpaired points, Let $S_2 = f(N(S_1)) \setminus S_1$, where we extend f to sets of vertices by just letting it be the set containing all the pairs of the given points. Similarly, define $S_{i+1} = f(N(S_i)) \setminus \left(\cup_{j=1}^{i} S_j\right)$. First, we need to show that this is well defined. That means that we want to make sure that we always have that every neighbor of S_i is paired with something. Since we could get from an unpaired point to something in S_i by taking a path that is alternating from being an edge in the matching and an edge not in the matching, starting with one that was not, if we could get to an unpaired point from S_i, that last edge could be tacked onto this path, and it would become an augmenting path, contradicting maximality of the original matching. Next, we can note that we never have an element in some S_i adjacent to an element in some S_j. Suppose there were, then we could take the path from an unpaired vertex to a vertex in S_i, add the edge to the element in S_j and then take the path from there to an unpaired vertex. This forms an augmenting path, which would again contradict maximality. The process of computing the $\{S_i\}$ must eventually terminate by becoming $\emptyset$ because they are selected to be disjoint and there are only finitely many vertices. Any vertices that are neither in an S_i or adjacent to one consist entirely of a perfect matching, that has no edges going to picked vertices. This means that the best we can do is to just pick everything from one side of the remaining vertices. This whole procedure of picking vertices takes time at most $O(E)$, since we consider going along each edge only twice. This brings the total runtime to $O(VE)$.

Thanks to John Chiarelli, a fellow graduate student at Rutgers, for helpful discussion of this part of the problem.

Problem 34-2

a. We can solve this problem in polynomial time as follows. Let a denote the number of coins of denomination x and b denote the number of coins of denomination y. Then we must have $a + b = n$. In order to divide the money exactly evenly, we need to know if there is a way to make $(ax + by)/2$ out of the coins. In other words, we need to determine whether there exist nonnegative integers c and d less than or equal to a and b respectively such that $cx + dy = (ax + by)/2$. There are $(a + 1)(b + 1) \leq (n + 1)^2$ many such linear combinations. We can compute each one in time polynomial in the length of the input numbers, and there are polynomially many combinations to compute, so we can just check all combinations to see if we find one that works.

b. We can solve this problem in polynomial time as follows. Start by arranging the coins from largest to smallest. If there are an even number of the current largest coin, distribute them evenly to Bonnie and Clyde. Otherwise, give the extra one to Bonnie and then only give coins to Clyde until the difference has been resolved. This clearly runs in polynomial time, so we just need to show that this will always yield an even division if such a division is possible. Suppose that for some input of coins which can be divided evenly, the algorithm fails. Then there must exist a last time at which there were an odd number of a denomination 2^i, so that Bonnie got ahead and had 2^i more dollars than Clyde. At this point, we start giving coins only to Clyde. Since every denomination decrease cuts the amount in half, it will never be the case that Clyde had strictly less than Bonnie, was given an additional coin, and then had an amount strictly greater than Bonnie. Thus, the sum of all coins of size less than 2^i must not exceed 2^i. Since we assumed the coins can be divided evenly, there exists $b_0, b_1, \ldots$ and $c_0, c_1, \ldots$ such that we assign Bonnie b_i coins of value 2^i and Clyde c_i coins of value 2^i, and both receive an equal amount. Now remove all coins of value smaller than 2^i. Bonnie now has $\sum_{k=i}^{\infty} b_k 2^k$ dollars and Clyde has $\sum_{k=i}^{\infty} c_k 2^k$ dollars. Moreover, we know that there is an uneven distribution of wealth at this point, and since every coin has value at least 2^i, the difference is at least 2^i. Since the sum of the smaller coins is strictly less than 2^i, there is no way to distribute the smaller coins to fix the difference, a contradiction since we started with an even split! Thus, the proposed algorithm is correct.

c. This problem is NP-complete. First, an assignment of each check to either Bonnie or Clyde represents a certificate which can be checked in polynomial time by adding up the amounts on each of Bonnies checks, and ensuring that it is equal to the sum of the amounts on each of Clyde's checks. Next we'll show this problem, SPLIT-CHECKS is NP-hard by showing that SET-PARTITION $\leq_P$ SPLIT-CHECKS. Let S be a set of numbers. We can think of each one as giving the value of a check. If there exists a set $A \subset S$ such that $\sum_{x \in A} x = \sum_{x \in (S-A)} x$, then we can assign each check in A to Bonnie

and each check in $S - A$ to Clyde to get an equal division. On the other hand, if there is a way to evenly assign checks then we may just take A to be the set of checks given to Bonnie, so by contrapositive, if we can't find a set partition which evenly splits the set then we can't evenly divide the checks. Thus, the problem is NP-hard, so it is NP-complete.

d. An assignment of each check to either Bonnie or Clyde represents a certificate, and we can check in polynomial time whether or not the total amounts given to Bonnie and Clyde differ by at most 100. Thus, the problem is in NP. Next, we'll show it's NP-hard by showing SET-PARTITION $\leq_P$ SPLIT-CHECKS-100. Let $S = \{s_1, s_2, \ldots, s_n\}$ be a set, and let $xS = \{xs_1, xs_2, \ldots, xs_n\}$. Choose $x = \frac{101}{\min_{i,j} s_i - s_j}$. Then the difference between any two elements in xS is more than 100. If there exists $A \subset S$ such that $\sum_{s \in A} s = \sum_{s \in S-A} s$, then we give Bonnie all the checks in xA and Clyde all the checks in $x(S - A)$, for a perfectly even split of the money, which means the difference is less than 100. On the other hand, suppose there exists no such A. Then for any way of splitting the elements of S into two sets, their sums will differ by at least the minimum difference of elements in S. Thus, any way of splitting the checks in xS will result in a difference of at least 101, so there is no way to split them so that the difference is at most 100. Therefore SET-PARTITION $\leq_P$ SPLIT-CHECKS-100, so the problem is NP-hard. Since we showed it is in NP, it is also NP-complete.

Problem 34-3

a) To two color a graph, we will do it a connected component at a time, so, suppose that the graph is a single component. Pick a vertex and color arbitrarily, and color that vertex that color. Then, we repeatedly find a vertex that has a colored neighbor and color it the other color. If we are ever in the case that a vertex has neighbors of both colors, then the graph is not 2-colorable. This procedure is able to 2-color if the graph is 2-colorable, since the only point where our hand isn't forced is at the beginning when we pick a vertex and a color, but this choice is only a false one because of the symmetry of the two colors. If it finds it is 2-colorable, it also outputs a valid 2-coloring.

b) The equivalent decision problem is to, given a graph G and an integer k say if there is a coloring that uses at most k colors. The easy direction is showing that if the original problem is solvable in poly thime, then the decision problem is solvable in poly time. To do this, just compute the minimum number of colors needed and output true if this is $\leq k$.

The other direction is a bit harder. Suppose that we can solve the decision problem in polynomial time, then, we will try to show how we can actually compute the minimum number of colors needed. A trivial bound on the number of colors needed is the number of vertices, because if each vertex has it's own color, then the coloring has to be valid. So, we perform a binary

search on the number of colors, starting with the range $1..|V|$, halving it each time until we are down to a single possible number of colors needed in order to color the graph. This will only add a log factor to the runtime of the decision problem, and so will run in polynomial time.

c) For this problem, we need to show that if we can solve the decision problem quickly, then we can decide the language 3-COLOR quickly. This is just a matter of running the decision procedure with the same graph and with $k = 3$. This gets us the reduction we need to show that 3-COLOR being NP-complete implies the decision problem is NP-hard. The decision problem is in NP because we can just have the certificate explicitly be the coloring of the vertices of the graph.

d) When we restrict the graph to the vertices $x_i, \neg x_i, RED$, then we will obtain a K_3 because of the literal edges. This means that all three colors must show up in it. Since there is already $c(RED)$, then the other two must be $c(TRUE)$ and $c(FALSE)$. No matter whether we choose x_i or $\neg x_i$ to be $c(TRUE)$, we can just select the other one to be $c(FALSE)$, this gets us that, if we only care about the literal edges, we always have a 3 coloring regardless of whether we want each x_i to be true or false.

e) For convenience, we will call the vertices a, b, c, d, e from the figure, where we are reading from top to bottom and left to right for vertices that are horizontal from one another. Since we are trying to check that it is 3 colorable if and only if at least one of x, y, z are $c(TRUE)$, we can negate the only if direction. That is, we suppose they are all colored $c(FALSE)$ and show that the graph is not 3 colorable.

Suppose $c(x) = c(y) = c(z) = c(FALSE)$. Then, we have that the only possibility for vertex e is to be $c(RED)$. This means the only possibility for c is to be $c(FALSE)$. However, this means that $c(a) \neq c(x) = c(FALSE)$, $c(d) \neq c(y) = c(FALSE)$, and $c(b) \neq c(c) = c(FALSE)$. So, we have a contradiction because any K_3 must have one of each color, and none of the vertices in this K_3 can be $c(FALSE)$. This shows that the graph is not 3-colorable.

For the other direction, we do not negate. So, we assume there is a vertex colored $c(TRUE)$ and we show that the graph is 3-colorable. We will split into the following cases. Note that because x and y play a symmetric role, we can reduce the number of cases from 7 to 5

x	y	z	a	b	c	d	e
$c(TRUE)$	$c(TRUE)$	$c(TRUE)$	$c(FALSE)$	$c(TRUE)$	$c(FALSE)$	$c(RED)$	$c(RED)$
$c(FALSE)$	$c(TRUE)$	$c(TRUE)$	$c(RED)$	$c(TRUE)$	$c(FALSE)$	$c(FALSE)$	$c(RED)$
$c(FALSE)$	$c(FALSE)$	$c(TRUE)$	$c(RED)$	$c(FALSE)$	$c(RED)$	$c(TRUE)$	$c(FALSE)$
$c(TRUE)$	$c(TRUE)$	$c(FALSE)$	$c(FALSE)$	$c(TRUE)$	$c(FALSE)$	$c(RED)$	$c(RED)$
$c(FALSE)$	$c(TRUE)$	$c(FALSE)$	$c(TRUE)$	$c(RED)$	$c(FALSE)$	$c(FALSE)$	$c(RED)$

Then, in every case where at least one of the inputs is true, there is an assignment of colors to the other vertices that produces a valid 3-coloring.

f) Suppose we are given any instance of 3-CNF-SAT, we will be using exactly the same construction as described in the problem for the reduction. First, we note that each of the vertices corresponding to a variable and its negation must only have the colors $c(TRUE)$ and $c(FALSE)$, and exactly one can be $c(TRUE)$ because of the literal edges. This means that each of the clause vertices will only be colorable if we assign a color of true to one of the variable vertices that are in the clause. This means that if we set each variable that has $c(x_i) = c(TRUE)$ true and each variable that has $c(\neg x_i) = c(TRUE)$ to be false, we will of obtained an assignment that makes at least one of the entries in each clause true, and so, is a satisfying assignment of the formula. Since 3-CNF-SAT is NP-complete, this means that 3-COLOR is NP-hard.

To see that it is in NP, just let the certificate be the coloring. Checking the coloring can be done in linear time.

Problem 34-4

a. For fixed k, does there exist a permutation $\sigma \in S_n$ such that if we run the tasks in the order $a_{\sigma(1)}, \ldots, a_{\sigma(n)}$, the total profit is at least k?

b. It is clear that the decision problem is in NP because we can view a certificate as a permutation of the tasks. We then check the tasks in that order to see if they have finished by their deadlines and what profit we incur, finally comparing this to k.

c. Suppose that a particular task a_i is the first to be performed. Then we solve the subproblem of deciding whether there exists a solution to the problem with tasks $a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n$, each with their respective profits and times, deadlines such that $d_j = d_j - t_i$, and with total profit at least $k - p_i$. To find this out, we make a lookup table which keeps track of the optimal schedule computed bottom-up.

d. There are 2^n possible profits that could be made, based on whether or not we finish each task by its deadline. We can binary search these for the one with maximum profit which satisfies the decision problem, which we can determine in polynomial time by part c. Since binary search takes $\lg(2^n) = n$, we need only run the polynomial time algorithm of part *c.* n times before we find the maximal k which solves the decision problem, and thus solves the optimization problem.