The Theory of Computation
Bernard M.E. Moret
Addison-Wesley-Longman, 1998

Solution Manual
Chapter 7, Text Problems

Exercise 7.1

- We prove this problem NP-complete (membership in NP is obvious) by transforming *1in3SAT* to it. We have four cases to consider, according to how many complemented literals the *1in3SAT* clause has. If no complemented literals appear, then we leave the clause unchanged and place it in the set of clauses for *Positive 1in3SAT*. The other three cases require changes.
  1. One complemented literal: $\{\overline{a}, b, c\}$. Replace by the clauses $\{a, \alpha, \beta\}$, $\{a, \alpha, \gamma\}$, $\{\beta, \gamma, \delta\}$, and $\{b, c, \alpha\}$. The first three clauses force $\beta = \gamma =$ False and $\delta =$ True, which in turn forces $a = \overline{\alpha}$, so that the last clause becomes exactly the same as the replaced clause.
  2. Two complemented literals: $\{\overline{a}, \overline{b}, c\}$. Replace by the clauses $\{a, \alpha, \beta\}$, $\{b, \alpha, \gamma\}$, $\{c, \beta, \gamma\}$, $\{\alpha, \delta, \epsilon\}$, $\{\alpha, \delta, \zeta\}$, and $\{\delta, \epsilon, \zeta\}$. The last three clauses force $\delta =$ True and $\epsilon = \zeta =$ False, which in turn forces $\alpha =$ False, which, through the first two clauses, forces $a = \overline{\beta}$ and $b = \overline{\gamma}$. Thus the third clause becomes exactly the same as the replaced clause.
  3. Three complemented literals: $\{\overline{a}, \overline{b}, \overline{c}\}$. Replace by the clauses $\{a, \alpha, \beta\}$, $\{b, \alpha, \gamma\}$, $\{c, \alpha, \delta\}$, and $\{\beta, \gamma, \delta\}$. Taken with the first three clauses, the fourth clause forces $\alpha =$ False; through the first three clauses, this in turn forces $a = \overline{\beta}$, $b = \overline{\gamma}$, and $c = \overline{\delta}$, so that the last clause becomes exactly the same as the replaced clause.

  Each clause has its own set of new variables. It is easily verified by exhaustion that the new set of clauses admits a solution if and only if the original one does.

  Note that our strategy for clauses with two complemented literals suggests another transformation: since we can use three clauses to force some of its variables to True and some to False, we can then use one of the variables forced to False, call it ϵ, in clauses of the type $\{x, x', \epsilon\}$, which forces x and x' to behave as complements of each other, so that we can now replace every complemented occurrence of x by x'. In fact, we can simplify the whole thing quite a bit: simply set up variables x and x' for each original variable x and add a single new variable, d. For each original clause, write a corresponding clause where complemented literals have been replaced by the prime variables; for each original variable x, write a new clause $\{x, x', d\}$. Note that d cannot be set to True, since that would force all primed and unprimed variables to False; thus d must be set to False, which means that x and x' must have opposite truth values.

- We prove this problem NP-complete (membership in NP is obvious) by transforming *NAE3SAT* to it. We have four cases to consider, according to how many complemented literals the *NAE3SAT* clause has. If no complemented literals appear, then we leave the clause unchanged and place it in the set of clauses for *Positive NAE3SAT*. The other three cases require changes.

  1. One complemented literal: $\{\overline{a}, b, c\}$. Replace by the clauses $\{a, \alpha, \beta\}$, $\{a, \alpha, \gamma\}$, $\{a, \alpha, \delta\}$, $\{\beta, \gamma, \delta\}$, and $\{b, c, \alpha\}$. The first four clauses force $a = \overline{\alpha}$, so that the last clause becomes exactly the same as the replaced clause.
  2. Two complemented literals: $\{\overline{a}, \overline{b}, c\}$. Simply replace by the clause $\{a, b, \overline{c}\}$ with one complemented literal and apply the replacement given above.
  3. Three complemented literals: $\{\overline{a}, \overline{b}, \overline{c}\}$. Simply replace by the clause $\{a, b, c\}$.

  Each clause has its own set of new variables. It is easily verified by exhaustion that the new set of clauses admits a solution if and only if the original one does.

  We can also use systematically variables x' and x'' to represent literals x and $\overline{x}$; for each clause, say with variables x, y, and z (regardless of complementation), we introduce auxiliary clauses $\{x', x'', y'\}$, $\{x', x'', y''\}$, $\{y', y'', z'\}$, $\{y', y'', z''\}$, $\{z', z'', x'\}$, and $\{z', z'', x''\}$—and, needless to add, we rewrite the original clause itself of in terms of our new variables. Successive clauses require some care to avoid multiple copies of the same clause.

- We prove this problem NP-complete (membership in NP is obvious) by transforming *3SAT* to it.

  Our transformation produces instances with all the variables used in the original instance, plus additional variables as needed. Given a clause in *3SAT*, two possibilities arise: (i) the clause is already in monotone form, in which case it is written without change into the transformed instance; and (ii) it includes both complemented and uncomplemented literals, in which case we need to modify it. For each clause of this second type, we use two additional variables and produce a total of three clauses; we distinguish two cases. In the first case, the clause contains one complemented and two uncomplemented literals, say $c = \{\overline{x}, y, z\}$; we then produce two clauses with uncomplemented literals and one with complemented literals: $\{a_c, y, z\}$, $\{b_c, y, z\}$, and $\{\overline{a}_c, \overline{b}_c, \overline{x}\}$. (This is really just splitting the original clause into two pieces connected by one auxiliary variable, as in the reduction from *SAT* to *3SAT*; the second auxiliary variable is a redundant copy of the first, used solely to ensure that the split clauses all have three literals each.) In the second case, proportions are reversed; from a clause $c = \{\overline{x}, \overline{y}, z\}$ with two complemented literals, we produce two clauses with complemented literals, $\{\overline{x}, \overline{y}, \overline{a}_c\}$, and $\{\overline{x}, \overline{y}, \overline{b}_c\}$, and one with uncomplemented literals, $\{z, a_c, b_c\}$. The transformation is easily carried out in polynomial time. The reader can verify that the transformed instance admits a solution if and only if the original one does.

- We prove this problem NP-complete by transforming *NAE3SAT* to it.

  Given an instance of *NAE3SAT* with variables $V = \{x_1, \ldots, x_n\}$ and clauses $C = \{\{\hat{x}_i, \hat{y}_i, \hat{z}_i\}, 1 \leq i \leq r\}$ (where $\hat{x}$ is used to denote one of x or $\overline{x}$), we construct an instance of *2SAT* with the same variables and where each clause, $\{\hat{x}, \hat{y}, \hat{z}\}$, is replaced by 6 clauses as follows:
  $\{\hat{x}, \hat{y}\}$ $\{\hat{x}, \hat{z}\}$ $\{\hat{y}, \hat{z}\}$ $\{\overline{\hat{x}}, \overline{\hat{y}}\}$ $\{\overline{\hat{x}}, \overline{\hat{z}}\}$ $\{\overline{\hat{y}}, \overline{\hat{z}}\}$

  Observe that, whenever the original clause has a satisfying assignment (1 or 2 literals set to true), then exactly 5 of the 6 constructed clauses are satisfied, whereas only 3 of the 6 clauses are satisfied when the original clause has no satisfying assignment (all literals true or all literals false). Thus we should set K to $5r$. The only problem left is repetitions: our construction can easily produce multiple copies of the same 2-clause. We can handle this minor problem in a very simple way: whenever variable x is involved in copies, create variables x_i (as many as needed to handle all copies), use the x_i in the copy rather than the x, and add the clauses $\{x, \overline{x}_i\}$ and $\{\overline{x}, x_i\}$, both of which are forced to true by the bound (i.e., the bound is increased by two for each new x_i). The two clauses are true if and only if x and x_i are equivalent and the proof follows immediately.

The NAE version of this problem is also NP-complete: indeed, the transformation given above, but starting from *Positive NAE3SAT* and with a bound set to $4r$ (plus repeats) rather than $5r$ (plus repeats), will prove the assertion.

Exercise 7.2

Note that *VC* is clearly in NP, as it suffices to guess a cover of size obeying the bound and then to scan all edges of the graph, verifying that each is covered by some vertex in the cover.

The construction is along the same lines as for *MaxCut*, but starts from *3SAT*, connects vertices in the triangles to the corresponding vertices in the segments devoted to truth assignment (rather than to those corresponding to complements), and requires slightly different arguments. Note that, in order to cover all three edges of a triangle, at least 2 of the 3 vertices of the triangle must be selected as part of the cover. Thus we shall set the size the desired cover at $n + 2k$ (the number of variables plus twice the number of clauses). The whole transformation easily runs in low polynomial time.

Given a solution to *3SAT*, we shall place in the cover all vertices in the truth-setting component that correspond to true literals; these n vertices cover the truth-setting edges, as well as the edges connecting the truth-setting components to the true literals in each clause. For each triangle, select one of its vertices corresponding true literals (there may or may not be a choice) and place the other two in the cover. Note that, as observed, all edges leading to triangle vertices that correspond to true literals are already covered; selecting the other two vertices covers the triangle and any other edges connecting to the triangle. Thus we have a solution of size $n + 2k$ to the cover problem.

Given a vertex cover of size $n + 2k$, first note that, as mentioned earlier, at least $2k$ of these vertices are needed just to cover the triangles; since the truth-setting edges cannot be covered by these vertices, they use up all remaining n vertices, one for each edge. Thus the structure of the vertex cover is well-defined: one vertex chosen out of each pair used in truth-setting for the variables and two vertices chosen out of three in each triangle. Note that the edges leading from the truth-setting components to the third vertex of each triangle (the one not in the cover) must then be covered by the selected truth-setting vertex. To obtain a satisfying truth assignment, we set to True the literals that correspond to the selected truth-setting vertices; this ensures a legal truth assignment since exactly one vertex of each pair is selected and also ensures a satisfying truth assignment, since the one vertex not selected in the the cover of each triangle always corresponds to a true literal.

Exercise 7.3

There are many ways to show this. That the problem is in NP is obvious. One reduction is to take the text reduction for *HC* and produce, not a cycle, but a path, with the same construction (just open the loop at one of the vertices of degree 2). A direct reduction from *HC* is also possible: given an arbitrary instance of *HC*, add 3 vertices, call them u_1, u_2, and v; u_1 is a vertex of degree 1 connected only to u_2, so that it must be at one end of the Hamiltonian path; v will also be a vertex of degree 1 and thus the other end of the path. Now, let x be some arbitrarily chosen vertex of the instance of *HC*; we connect u_2 to all the neighbors of x and connect v to x. For there to be a Hamiltonian path, we must start at v, move to x (no choice), visit all vertices of the original graph, then take an edge to u_2 and end at u_1. This is possible if an only if the original graph has a Hamiltonian circuit: if it does, we use it from x all the way to the last vertex and, instead or returning to x, move to u_2; if we have a Hamiltonian path, it has to be as described, which implies that the piece of it that moves from x to the predecessor of u_2 visits every vertex of the original graph and we can then return to x by construction.

Exercise 7.4

That the problem is in NP requires a bit of thought: it is not enough to specify what the vertices of G that make up the isomorphic copy of H happen to be, since we do not know how to test for isomorphism in polynomial time. However, the certificate can give both the selected vertices of G and, for each such vertex, the vertex of H to which it corresponds; verifying the isomorphism is now a linear-time operation. We restrict the problem to *Clique* (with graph G and parameter k) by using G unchanged and restricting H to be the complete graph on k vertices—i.e., to be a clique of k vertices. Since a clique of $k' > k$ vertices contains a clique of k vertices and since two complete graphs on k vertices are automatically isomorphic, our conclusion follows.

Exercise 7.5

FP is the class of all functions computable in polynomial time; P-easy problems form the class of all problems that Turing reduce in polynomial time to some problem in P. We need to show containment in each direction.

If f is some function in FP, it is computable in polynomial time; that is, there exists a Turing machine M and a polynomial p such that, on any input x, M computes $f(x)$ and stops after at most $p(|x|)$ steps. In particular, note that the value computed by M on input x uses at most $p(|x|)$ symbols in the tape alphabet and thus cannot exceed $c^{p(|x|)}$ for some positive constant c. We claim that the decision version of f, namely, "does $f(x)$ exceed B," is in P and, moreover, that the computation of f can be Turing-reduced to its decision version. That the decision version is in P is easily seen: we just run machine M on x and compare the output to B, a task that takes at most $O(p(|x|))$ time since we can compare on a symbol by symbol basis. That we can reduce the computation of f to its decision version takes a little more work: we use binary search with B in the range from $-c^{p(|x|)}$ to $c^{p(|x|)}$ to find the value of $f(x)$; each probe in the binary search is a call to the decision version of the problem; at most $\log_2(2c^{p(|x|)})$ such calls will be made, which is $O(p(|x|))$ calls and thus polynomial. Thus any function in FP defines a P-easy problem.

Conversely, let Π be a P-easy problem. Then there exists a Turing reduction from Π to some problem in P; this Turing reduction, along with the algorithm that solves the problem in P, provides a polynomial-time solution for Π, so Π is in FP.

Exercise 7.6

The hint suggests recasting the problem before attempting to prove NP-easiness, because direct self-reduction in the original terms appears to fail. The goal of any test set is to separate every pair of classes that can be separated. (Indeed, we can assume that all classes in the given instance can be separated; otherwise, we can discard all but one of the classes in each inseparable group.) We can thus take the view that every pair of classes forms an element to be handled by at least one test and that each test can be characterized by the pairs that it can separate. This is the description of a set cover problem! It is also a description that lends itself to self-reduction: after selecting a test, the remaining instance has one fewer test and has "lost" all of the pairs covered by that test. With this step in mind, NP-easiness is easily proved in the "usual" manner: use binary search to derive the cost of the optimal solution, then attempt to include each test in turn as the first test in the set, asking for each attempt whether there exists a solution of optimal cost containing this test; once the first test has been selected, repeat the iteration to obtain the second, third, etc. tests.

Exercise 7.7

We solve the problem through direct self-reduction and also through reduction to the completion problem (given a partial TSP tour, find the optimal completion), which we prove to be NP-complete in its decision version.

The direct self-reduction is slightly tricky: once we have included a trip segment, say from city A to city B, we need to modify the problem (number of cities and distance measures) in order to enable us to use an optimal TSP for the modified problem to yield an optimal path through the remaining cities from B back to A. The obvious tool is merging all cities in the currently developed trip segment into a single city; the problem with this approach is how to define the cost of travelling from the merged city to the remaining cities—for instance, is the cost of travelling to C that of travelling from A to C or that of travelling from B to C? There are only two choices (we can only travel from the two ends of the developed trip segment), but they suffice to cause some trouble. The solution is to abandon symmetry (never required in the TSP problem in any case) and orient the currently developed trip segment, say from A to B, so that costs of travelling from the merged city to other cities are the costs of travelling from B to the other cities, while costs of travelling to the merged city are the costs of travelling to A. (Another, similar solution, is to remove all but the two endpoints of the developed trip segment and set up a cost of zero for travelling between the two endpoints of that segment.) Once this step is understood, the NP-easiness proof proceeds as usual: use binary search to obtain the optimal tour length, then start from some arbitrary city and develop the tour edge by edge by trying out in turn each possible edge from the last city reached, using the oracle to decide whether that edge can lead to an optimal solution.

Exercise 7.8

Assume that we have NP $\neq$ coNP—that is, that NP is not closed under complementation. We need to show that we must then have P $\neq$ NP. We use contradiction: assume that we have P $=$ NP; since P is closed under complementation, so is NP, which contradicts our assumption and proves our result.

Exercise 7.9

Let Π be some NP-complete problem and assume that Π also belongs to coNP. We want to show that NP must be closed under complementation. Let Π' be any other problem in NP; since Π is NP-complete, there exists a polynomial-time reduction f mapping Π' to Π. The same transformation f reduces in polynomial time the complement of Π' to the complement of Π. But Π is in coNP, so that its complement is in NP; hence we have a polynomial-time reduction from the complement of Π' to a problem in NP. Thus the complement of Π' is in NP (see Exercise 6.8), so that NP is closed under complementation, as was to be proved.

Exercise 7.10

- *Optimal Vertex Cover*. We prove this problem NP-hard by (Turing) reducing the standard *VC* to it. Given a graph, $G = (V, E)$ and a positive integer bound, $B < |V|$, we simply call the oracle for *Optimal Vertex Cover* repeatedly, starting with a bound of B and decreasing by 1 at each call, until we have either obtained an answer of "yes" (in which case we also answer "yes") or have reached a bound of 0 receiving only answers of "no" (in which case we answer "no").

  We prove this problem NP-easy by (Turing) reducing it to the standard *VC*. Given a graph $G = (V, E)$ and a positive integer bound, $B < |V|$, we simply call the oracle for *VC* twice, once with bound B and the second time with bound $B - 1$; if the first answer is "yes" and the second "no", we answer "yes", otherwise we answer "no". Thus the problem is NP-equivalent and also in D^p.

- *Minimal Unsatisfiability*. We prove this problem NP-hard with a many-one reduction from the coNP-complete problem *3UNSAT*. See Exercise 7.46 for details. This is a fairly elaborate reduction and one that does not make use of the power of Turing reduction, but it is surprisingly difficult to derive a simpler, Turing reduction.

  We prove this problem NP-easy by reducing it to the standard *SAT*. We begin with one call with the full instance, to establish that the instance is unsatisfiable. If the instance is indeed unsatisfiable, we proceed with a loop that will remove one clause from the instance and call the decision algorithm to establish that the resulting instance is now satisfiable; this loop proceeds as along as all answers remain positive and eventually ends up testing each clause through removal. If all of these tests are passed, then the instance is a "yes" instance of *Minimal Unsatisfiability*, otherwise it is a "no" instance. The total number of calls to the decision procedure is at most one more than the number of clauses in the instance—and thus is at most linear in the size of the instance.

- *Unique Satisfiability*. We prove this problem NP-hard by devising a many-one polynomial-time reduction from the known coNP-complete problem *UNSAT* to it—a rather tighter reduction than needed. The basic idea is to take the formula for *UNSAT*, call it $f(x_1, \dots, x_n)$, and turn it into the new formula, $g(x_0, x_1, \dots, x_n)$ given by

$$g(x_0, x_1, \dots, x_n) = (x_0 \wedge x_1 \wedge \cdots \wedge x_n) \vee (\overline{x_0} \wedge f(x_1, \dots, x_n))$$

It is easily verified that this new formula g, which is always satisfiable (in itself an interesting result, since it shows that deciding unique satisfiability is hard even when satisfiability is known), is uniquely satisfiable (by setting all $n + 1$ variables to true) if and only if the original formula f is unsatisfiable. Turning this formula, which is not in clause form, in a form suitable for *Unique Satisfiability* can be done in polynomial time, thereby completing our reduction and showing that *Unique Satisfiability* is coNP-hard and thus also NP-hard.

We prove this problem NP-easy by reducing it to the standard *SAT*. We begin with a single call with the same instance to establish satisfiability. If the answer is "no," we answer "no" and stop. Otherwise, we establish uniqueness as follows. We proceed one variable at a time. At stage i, we have determined the truth values of the first $(i - 1)$ variables and also determined that only this particular combination of truth values can lead to a satisfying truth assignment. Working now with variable x_i, we first test whether there exists a satisfying truth assignment with x_i set to true (by adjusting the clauses to reflect this assignment and calling the oracle on the resulting collection of clauses); if not, we set x_i to false and proceed to the next variable. If there exists a truth assignment, we now test whether there exists a satisfying truth assignment with x_i set to false; if not, we set x_i to true and proceed to the next variable; otherwise, we answer "no" and stop. On establishing that only one truth value works for the last variable, we can answer "yes." The total number of calls to the oracle is at most $2n + 1$ (two per variable plus the initial call) and thus polynomial in the size of the input.

Actually, two oracle calls to NP problems suffice. Note that the problem "does an instance of *SAT* have at least 2 satisfying assignments?" is NP-complete (it is in NP since the certificate need only give both assignments and it is complete because it is trivial to transform *SAT* in such a way that its number of satisfying assignments is exactly doubled, by introducing one redundant variable). Now we can solve *Unique Satisfiability* by one call to *SAT* and one to "does an instance of *SAT* have at least two satisfying assignments?", expecting a "yes" answer to the first question and a "no" to the second. Thus the problem is in fact in D^p.

- *TSP Factor*. We prove this problem NP-hard by reducing the standard *HC* to it. Given a graph, $G = (V, E)$, we create two instances of *TSP Factor*; both have $|V|$ cities and intercity distances $d(i, j)$ of 1 if $i, j \in E$ and of 2 otherwise. In the first call, we ask if $|V|$ is a factor; in the

second call, we ask if $2|V|$ is a factor. If the first answer is "yes" and the second "no", we answer "yes", otherwise we answer "no". (The TSP tour must have length at least $|V|$ and at most $2|V|$; it has length $|V|$ if and only if the *HC* instance admits a solution. Our second call eliminates tours of length $2|V|$ for graphs where $|E| = \emptyset$.)
We prove this problem NP-easy by reducing it to the standard *TSP*. We can find the value of the optimal tour, l_{opt}, by binary search with at most $\log l_{\text{opt}}$, which is polynomial (in fact, linear) in the input size. Once l_{opt} is known, we can check whether or not the given number is a factor of it in time polynomial in $\log l_{\text{opt}}$, using any standard division algorithm. There seems to be no way of avoiding the linear number of calls; indeed, the problem can be proved to be complete for Δ_2^p and thus unlikely to be in D^p like the first three.

Exercise 7.11

The version of *Minimal Unsatisfiability* defined for this problem is thus a version of *Satisfiability* where we ask whether or not the instance can be made satisfiable by removing any one of its clauses (not knowing whether or not the original instance is itself satisfiable).

The interesting part of the problem here is membership in NP. We need to exhibit a certificate for every "yes" instance. If an instance has n variables and m clauses, we can devise a certificate that will have m (possibly distinct) truth assignments, one for each clause that can be removed; such a certificate has length $\Theta(mn)$ and can be checked in polynomial time by checking each truth assignment in turn, ignoring for each its corresponding clause. Thus the problem has concise certificates that can be checked in polynomial time and is in NP.

To show that it is NP-complete, we use a very simple transformation from *Satisfiability*: we simply double the number of variables and write every clause "twice"—once with the first set of variables and once with the second set. If the original instance was satisfiable, the constructed instance is satisfiable as well—there is no need to remove any clause to satisfy it, but it never hurts to remove one. If the original instance is unsatisfiable, then no single clause can be removed to make the constructed instance satisfiable—by removing a single clause we might make one copy of the original instance satisfiable, but the other copy remains unchanged and thus unsatisfiable. That the transformation runs in polynomial time is obvious.

Exercise 7.12

In our notation for the polynomial hierarchy, we want candidates for the class $\Sigma_2^p - \Delta_2^p$.

One potential source of such problems is inequivalence of languages, since the simplest way to prove inequivalence is surely to guess a string that belongs to one language but not to the other. For example, we saw that *Star-Free Regular Expression Inequivalence* is NP-complete. We need versions where the membership problem is harder to solve—basically where we need the power of Δ_2^p to solve it. Note, however, that we must still operate with a certificate of polynomial length; thus the languages cannot be too hard to decide; for instance, we saw that *Regular Expression Inequivalence* (where Kleene closure is allowed) is PSPACE-complete, a result that, under our standard conjectures, puts *REI* beyond PH. One example of a candidate for the class $\Sigma_2^p - \Delta_2^p$ is *Integer Expression Inequivalence* (which is in fact complete for Σ_2^p, for which see Exercise 7.50): the guess appears to require the full power of Δ_2^p for verification. We can turn to formal languages and consider inequivalence of various types of languages; while the details are beyond the scope of this exercise, it turns out that *Inequivalence of Context-Free Grammars on a One-Symbol Alphabet* (see Additional Exercise 7.29) is a good candidate. (It is a curious problem, since a context-free grammar on a one-symbol alphabet must in fact generate a regular languages; what is happening here is that we have taken a slightly too easy problem, *Inequivalence of Regular Grammars*, and turned it into a slightly harder one by increasing the expressive power of the description—context-free grammars can define some regular languages more concisely than a regular grammar could—, thereby providing one more example of the notion of "succinctness" introduced on pp. 218–219.

Another source of such problems is the *Satisfiability* family. We have already seen how to turn the basic NP-complete problem into a PSPACE-complete problem through arbitrary quantification of the variables. We can provide an excellent candidate for $\Sigma_2^p - \Delta_2^p$ by using bounded quantification instead. This problem is described in some detail on p. 271 of the text, so we will not repeat the description here, except to note that the problem is constructed exactly to the design we gave above: it requires a guess and that guess, while concise, requires in turn the full power of Δ_2^p for verification.

Exercise 7.13

Any dynamic program that returns the optimal solution to a problem can be trivially made to count the number of optimal solutions: in addition to storing the optimal value at each step (in each table location), the program also stores the number of ways in which that value can be obtained. When computing the next optimal value, the program can combine the number of ways in which each of the pieces can be optimize to compute the number of ways in which the new, larger piece can be optimized. (Most of the time, this computation is a simple multiplication.)

So we need to write down the dynamic program recurrence for matrix chain products. The problem asks for the least expensive way to parenthesize (to associate) a product of n rectangular matrices. We can write an expression to find the best way to parenthesize the subexpression including operands i through j $(i \leq j)$:

$$min_cost(i, i) = 0$$

and, for $i < j$,

$$min_cost(i, j) = \min_{k=i,i+1,\ldots,j-1} (min_cost(i, k) + min_cost(k+1, j) + cost(i, k, k+1, j))$$

where the function $cost(i, k, k+1, j)$ returns the cost of multiplying the matrix formed by the product of matrices i through k with the matrix formed by the product of matrices $k+1$ through j. Since the cost of multiplying an $l \times m$ matrix by an $m \times n$ matrix is of order $O(lmn)$, the function $cost(i, k, k+1, j)$ just returns the product of the number of rows of matrix i, the number of columns of matrix k, and the number of columns of matrix j. Our dynamic program will also store the sum, over all values of k that achieve the minimum cost for $min_cost(i, j)$, of the products of the number of ways in which $min_cost(i, k)$ could be achieved and of the number of ways in which $min_cost(k+1, j)$ could be achieved; this sum is the number of ways in which $min_cost(i, j)$ can be achieved.

Thus the dynamic program can compute the number of distinct solutions at no additional (asymptotic) expense over its computation of the solution matrix.

Exercise 7.14

Exercise 7.15

If each edge is to be covered with exactly one vertex, we get a rather different problem. The text's transformation no longer suffices, since it produces instances where the optimal cover has edges covered at both ends (in the clause triangles). In fact, the problem is then in P and even solvable in linear time: basically, we are just asking if the graph is two-colorable in a certain way (with at most k vertices colored the same). Whenever one vertex is picked for the cover, the other endpoint must be discarded and any edge incident upon it must have its other endpoint in the cover. Thus the choice of a single vertex to place in the cover entirely determines what must be done with a connected component of the graph (in linear time); we can then count the number of vertices of each color and check whether either color has no more vertices than the given bound.

Exercise 7.16

The answer is yes, provided that things are done right. We can go through the same imaginary construction, but then merge multiple edges into a single edge of corresponding weight. Since we want to cut two "lightweight" edges per clause and one per variable, we set the total weight of real (weighted) edges to be cut to $n + 2k$. The transformation is a bit more expensive than that in the text, but still easily done in low polynomial time. That a "yes" instance of *NAE3SAT* transforms into a "yes" instance of *Weighted MaxCut* follows immediately from our reasoning about *MaxCut* itself. The hard part is to show that "yes" instances of *Weighted MaxCut* can only result from "yes" instances of *NAE3SAT*. In order to proceed, begin by noting that the total weight of all edges in the instance is $n + 3k$, the number of "lightweight" edges. Now note that the n edges connecting literal vertices to their complements all have unit weights (because a literal and its complement cannot occur together in the same clause), so that they can contribute at most n to the total cut weight. From our construction, we cannot do better than cut a total of $2k$ "lightweight" edges from the edges set up for clauses, so that the maximum cut weight from clauses edges is $2k$. Thus we can conclude that all n truth-setting edges must be cut, thereby ensuring a legal truth assignment. To obtain a total cut weight of $2k$ from the clause edges, each of the k triangles we deliberately set up (there may well be others that were created unintentionally) must be cut: failure to cut any of them reduces the number of "lightweight" edges cut and thus the total cut weight. Thus the truth assignment is also satisfying.

Exercise 7.17

First, what might make the transformation work? Since a bipartite graph is simply a 2-colorable graph, one can take a 3-colored graph, remove all vertices of the least common color (there can be at most $\lfloor |V|/3 \rfloor$ of these), and thus obtain a 2-colored, i.e., bipartite graph. So this transformation is guaranteed to transform yes instances of *G3C* into yes instances of *MVDBS*. The trouble is that it can also transform no instances of the first into yes instances of the second. One can think of the problem as arising from the fact that one could delete vertices connected by an edge, whereas vertices of the same color cannot be so connected; in other words, there is nothing in the transformation to *force* the deletion of vertices from one color class.

Another way to look at it is that the allowed bound is global, whereas bipartiteness is a very local property; hence one can take a graph with a lot of vertices, few edges, but a small, not 3-colorable subgraph (say K_4) and use most of the bound to remove the real problem vertices. For instance, K_4 plus two isolated vertices is not 3-colorable, yet gives rise to a bound of 2, which is sufficient to turn the graph into a bipartite graph by deleting 2 of the 4 vertices of the K_4 piece.

Exercise 7.18

We prove this problem NP-complete by transforming *Exact Cover By 3-Sets (X3C)* to it. The basic idea in the transformation is to add a fourth member to every triple of *X3C*, leaving enough choice that the necessary number of four-tuples can be selected—each with a different fourth member.

Given an instance S, $|S| = 3q$, $C \subseteq 2^S$, $(c \in C \Rightarrow |c| = 3)$ of *X3C*, we construct an instance T, $|T| = 4q$, $D \subseteq 2^T$, $(d \in D \Rightarrow |d| = 4)$ of $X4C$ as follows: $T = S \cup \{x_1, \ldots, x_q\}$, $D = \{\{a, b, c, x_i\},\ 1 \leq i \leq q,\ \forall \{a, b, c\} \in C\}$. This construction can obviously be done in polynomial time.

The proof is simple. If $X3C$ has a solution, then the q triples that form the solution determine a solution to $X4C$, where the i-th 4-tuple corresponds to the i-th triple augmented by x_i. The converse is equally simple: just drop the x_j from each 4-tuple in the solution to $X4C$ to obtain a solution to the original $X3C$.

Exercise 7.19

The problem is in NP: we can detect whether or not a graph has a cycle in linear time by breadth-first search. We prove that it is NP-complete by transforming *3SAT* to it. Our graph will have a vertex for each clause element, plus two vertices for each variable (one for the uncomplemented literal, one for the complemented literal), plus two distinguished vertices to represent *true* and *false*. The true and the false vertices are connected in a 2-cycle; the three elements of each clause are connected in a 4-cycle with the false vertex; and any two vertices corresponding to a complemented and an uncomplemented copy of a variable are connected in a 2-cycle. This transformation is obviously polynomial-time.

Now, in order to have a solution to our problem, each cycle must be broken. For the 2-cycles, this means that the two vertices involved must end up on different sides of the partition; for the 4-cycle, this means that at least one of the four vertices must end up on the opposite side from the others. Thus the true and false vertices end up on different sides, thereby labelling one side as that of the true literals and the other as that of the false literals.

The two vertices associated with each variables must end up on opposite sides, thereby giving us a truth assignment. At least one of the three vertices associated with each clause will end up on the true side (opposite the false vertex present in the 4-cycle) and thus ensure satisfaction of the clause. Finally, the 2-cycles connecting any two complementary literals guarantee the uniformity of the partition: all vertices representing the same literal must end up on the same side of the partition. Thus a solution to our problem yields a satisfying truth assignment for the *3SAT* problem. The converse is obvious.

A variation would omit the 2-cycles between complemented and uncomplemented copies of literals and represent each clause by a single vertex connected with the false vertex in a 2-cycle, but add 3 vertices for each clause, to serve as connection points; the clause vertex is connected to each of the 3 vertices by a 2-cycle, and each of the 3 vertices is in turn connected to the corresponding literal in the truth-setting component by a 2-cycle. Each 2-cycle must be broken, thereby ensuring that all connecting vertices will end up in the false side of the partition; the rest follows easily.

Both versions can be simplified if one transforms *NAE3SAT*, since there is then no need for the true and false vertices. There are probably as many proofs for this problem as there are people to write proofs.

Exercise 7.20

There is only one difference between *X3C* and *Three-Dimensional Matching (3DM)*: *3DM* introduces labeling and ordering—its elements come from three different sets of the same size and its 3-sets are really ordered triples (one element from each of the three sets). Membership in NP follows from the same considerations as for *X3C*. Completeness is proved by almost the same transformation. The change is in the ordering of the 3-sets and in the partition of the overall sets into three pieces.

Referring to pages 241 and 242, we will place all attaching points into one set, all even-indexed p_x^j (j even) into a second set, and all odd-indexed p_x^j (j odd) into the third set. The triples forming the variable pieces will be ordered, with the attaching point first, the even-indexed p_x^j second, and the odd-indexed p_x^j third. The 3-sets corresponding to a clause are also ordered, with the attaching point first, the literal second, and the truth indicator third; all literals are placed in the second set and all truth indicators in the third. For an instance of *Positive 1in3SAT* with n variables and m clauses, we get $6m$ attaching points and thus $6m$ elements in the first set; we then have $3m$ even-indexed and $3m$ odd-indexed p_x^j's, plus $3m$ literals and $3m$ truth indicators, so that each of the second and third sets also has $6m$ elements. The transformation then works without change.

Exercise 7.21

(Vertex-) Dominating set (DS)

We prove this problem NP-complete by transforming *VC* to it.

Given an instance $G = (V, E)$, K of *VC*, we construct an instance $G' = (V', E')$, K' of *DS* by letting $K' = K$, $V' = V \cup E - \{v | v \in V$ and degree$(v) = 0\}$, and the new set of edges include all of the old edges, plus two new edges for each old edge, so that, in effect, we replace each edge in G by a triangle in G', with the new vertex being uniquely associated with the old edge. Notice that the triangle can be dominated by any of its three vertices. is obviously polynomial-time.

Now if we have a solution to *VC*, then each edge must be covered; such a cover is trivially a dominating set in G, since we just added the one vertex and two edges for each old edge and this new construction is dominated by either end-vertex. Conversely, a solution to the dominating set problem must include at least one vertex from each triangle (since the 2 new edges are the only edges emanating from the new vertex). If all such vertices are drawn from V, we are done, since there is at least one for each edge and at most K of them; if some of them are new vertices, we may replace them without damage by one of the old vertices. Thus we are done.

Another simple proof can be constructed directly from *3SAT*. For each variable, we set a pair of vertices (the 2 literals), connected by an edge; we also add two extra vertices for each such edge, each connected to the two literals (those are enforcers). For each clause, we set a vertex and connect it to the corresponding 3 literals. The bound will be set equal to the number of variables; a solution must be just a collection of literals corresponding to a truth assignment. The enforcers ensure that we must pick one of the two "literals" to be in the dominating set (as otherwise we'd have to pick the two enforcers and thus be unable to meet the bound); since we must pick one of each pair, we cannot pick any clause vertex; the correctness of the construction follows immediately.

(Edge-) Dominating set (DS)

Same as above, but we want to dominate the edges of the graph. An edge is dominated by vertex x if either x is an endpoint or x is connected to each endpoint by an edge. The proof proceeds easily with the same construction. If we further restrict domination by excluding the cover (thus now a vertex dominates an edge if and only if it is connected by an edge to both endpoints of the edge), much the same proof works, but now we replace each edge of *VC* by two adjacent triangles (really, a K_4 minus an edge) rather than just one triangle.

Exercise 7.22

This problem is sometimes called *Perfect Dominating Set (PDS)* or also *1-Perfect Codes on Graphs*. We prove it NP-complete (it is obviously in NP) by transforming *Positive 1in3SAT* to it.

Given an instance of *Positive 1in3SAT* with variables $V = \{x_1, \ldots, x_n\}$ and clauses $C = \{\{x_i, y_i, z_i\}, 1 \leq i \leq r\}$, we construct an instance of *PDS* with $cn + r$ vertices and $n + 3r$ edges as follows. Each variable will be represented as 2 vertices connected by an edge; one of the vertices is designated as the attaching point. Each clause will be represented by one vertex. The composition of a clause is denoted by edges (three for each clause vertex) connecting the clause vertex to the attaching point of corresponding variable components.

Now, a clause vertex cannot be in V': putting it in V' excludes its three literals, but then the other vertex of each literal component won't be dominated by anything. Thus all r clause vertices must be in $V - V'$. In order to dominate them, exactly one variable attaching point per clause must be in V'; such variables are considered true and all others are considered false. The remaining vertex is in the dominating set if the variable is false, so that the variable's attaching point is dominated. It should now be obvious that a solution to *PDS* can easily be constructed from a satisfying truth assignment. This completes our proof.

From the standard *1in3SAT*, we use the same construction, where now the other end of each variable segment is the attaching point for complemented literals; we connect all clause vertices together in a complete subgraph. The latter prevents us from selecting any clause vertex for the dominating set, since it would allow only one and prevent domination of the variable vertices corresponding to the complements of the literals appearing in the selected clause.

Exercise 7.23

This problem is clearly in NP (we can guess not only the common subsequence, but also, for each string, the position of the subsequence's symbols within the string). We prove that the problem is NP-complete by transforming *Independent Set* to it. Given a graph, $G = (V, E)$, and a constant, $K \leq |V|$, we construct an instance of our problem as follows. There will be $|E| + 1$ strings, one for each edge to describe the graph and one to force a solution. The forcing string is just the list of the vertices, $v_1 v_2 \ldots v_n$. For each edge, $\{v_i, v_j\}$ (assume $i < j$), the corresponding string is

$$v_1 v_2 \ldots v_{i-1} v_{i+1} \ldots v_{j-1} v_j v_{j+1} \ldots v_n v_1 v_2 \ldots v_{i-1} v_i v_{i+1} \ldots v_{j-1} v_{j+1} \ldots v_n$$

K is the same for both problems.

If we have a solution to *Independent Set*, the vertices composing the solution form a solution subsequence: they clearly form a subsequence of the forcing string. They form a subsequence of each edge description: given the string describing edge $\{v_i, v_j\}$, one of three cases can occur: (i) neither v_i nor v_j is in the solution, in which case we can pick all solution vertices in the first part of the string; (ii) v_i is in the solution (and then v_j is obviously not in it), in which case we can pick all solution vertices up to v_i from the first part of the string and all remaining vertices from the second part (where all vertices except v_j are listed—and we do not need v_j); or (iii) v_j is in the solution (but not v_i), which is treated similarly to (ii).

The converse proof depends on the crucial observation that a common subsequence of length at least K cannot contain both vertices of an edge. This follows from the fact that the forcing string guarantees that vertices must appear in increasing subscript order and that the vertices of an edge, $\{v_i, v_j\}$ for $i < j$, can only be picked in reverse order in the string associated with that edge. Thus no two vertices in the subsequence are connected with edges: in other words, the vertices of the subsequence form an independent set of size at least K.

The transformation can also be based on vertices rather than edges: we keep the same forcing string and the same bound, but produce one string per vertex rather than one per edge. The string associated with vertex v_i has two parts: the first part lists in order all vertices except those connected to v_i by an edge; the second part lists all vertices except v_i. If there is an edge $\{v_i, v_j\}$, then the common substring cannot include both v_i and v_j: if $i < j$, then the string associated with v_j has v_i appearing *after* (and thus out of order) v_j while, conversely, $i > j$ has v_i appearing *before* (and thus out of order) v_j in the string associated with v_i. (In fact, the whole common substring will appear in the first part of the string if and only if the corresponding vertex is part of the independent set, in the second part of the string if and only if the corresponding vertex is not part of the independent set.)

Exercise 7.24

The decision problem is clearly in NP; we prove that it is NP-complete by transforming *X3C* to it. The identification problem will have an object for each element in the cover problem, call them $\{x_1, \ldots, x_n\}$, plus 3 extra elements, u, v, w; each subset of the cover problem will become a test in the identification problem; in addition, we'll have a test of the form $\{x_i\}$ for each x_i and tests $\{u\}$ and $\{v\}$. If the exact cover problem has a solution, we can construct an identification tree that successively uses all 3-sets of the solution to partition the objects into $n+1$ subsets of 3 objects each; the objects in each subset are then resolved by a simple subtree of 3 leaves (using, for instance, the $\{x_i\}$ tests). The subtree of 3 leaves has external path length of 5; the identification tree would have total external path length of $5+3\cdot(n-1)+\sum_{i=1}^{n} 5+3i = 5n+3\cdot(n^2+n-3)/2 = 3\cdot(n^2+4n-3)/2$. Now we let the bound be this value (which we will claim is optimal). Clearly, if *X3C* has a solution, so does our decision problem; conversely, we claim that, if our decision problem has a solution, its cost is precisely equal to the bound and the tree constructed is of the exact description given above (in which case we have a solution for *X3C*). That our claim is true is intuitively obvious: no test can do better than distinguish a 3-subset from the remaining objects, and that can be done to the end only if all 3-subsets are disjoint; a formal proof can also be given, for which see the original paper by Hyafil and Rivest (*Inf. Proc. Letters* 5 (1976), 15–16).

Exercise 7.25

This problem is clearly in NP (we can guess the subset of tests and check, for each pair of classes, that there exists one test in the subset that yields different answers for the two classes).

The problem is very closely related to set covering: we can think of the task of the test set as distinguishing between all pairs of classes and of each test as “covering” (distinguishing among) a collection of pairs; the problem is then to find the smallest collection of tests that, taken together, “cover” all pairs of classes. Thus we prove that the problem is NP-complete by transforming *X3C* to it.

The basic idea is to make every set element a class and every 3-set a test. However, a test collection corresponding to an exact cover would only manage to distinguish between groups of classes (3 classes per group); thus we need to add tests to distinguish among classes within a group—yet these additional tests should not help with any other breakdown into groups, because we want any solution to the test problem to include the equivalent of an exact cover by 3-sets. The solution is to create *a pair of* classes for each element of S and to use our 3-sets as tests that distinguish between the two classes in pair, but not between across pairs—for which purpose we shall have to add more tests.

Given a set $S=\{x_1,\ldots,x_{3n}\}$ and a collection of 3-sets, $\mathscr{C}=\{S_1,\ldots,S_k\}$, we construct an instance of our problem as follows. For each element x_i of S we create two classes y_i and z_i in our problem. For each 3-set $S_j=\{x_{j_1},x_{j_2},x_{j_3}\}$ in $\mathscr{C}$, we create a test $T_j=\{y_{j_1},y_{j_2},y_{j_3}\}$. We add $3n-1$ tests, one for each i, $1\le i<3n$, of the form $\{y_i,z_i\}$. Finally, we set the bound to $4n - 1$. Clearly, this transformation can be carried out in polynomial time.

Note that the tests that correspond to 3-sets cannot distinguish among any of the z_is; each such test distinguishes 3 of the y_is from their matching z_is, and also discriminates among the y_is themselves. Note also that the additional $3n$ tests cannot distinguish between a y_i and its matching z_i, but that taken together they can narrow the identification down to a single $\{y_i, z_i\}$ pair. Finally, note that we will need all $3n - 1$ of the additional tests in order to be sure to distinguish among all of the z_is—omitting even one will leave at least one pair of these indistinguishable, since no help can be provided by the tests corresponding to 3-sets. Thus any solution to the problem will include all $3n - 1$ additional tests and thus also can include at most n of the tests corresponding to 3-sets. Since the $3n - 1$ additional tests narrow the identification down to a single $\{y_i, z_i\}$ pair, we only need tests to distinguish between a y_i and its corresponding z_i. Each test corresponding to a 3-set can provide that information for exactly 3 pairs; we need that information for all $3n$ pairs, so we can get it if and only if the tests selected carry information about n disjoint 3-sets, that is, if and only if the instance of *X3C* has a solution.

Exercise 7.26

We prove this problem NP-complete by transforming *X3C* to it.

Given an instance S, $|S| = 3q$, $C \subset 2^S$, $(c \in C \Rightarrow |c| = 3)$ of *X3C*, we construct an instance $G = (V, E)$, $R \subset V$, $K \leq |V| - 1$ of *STG* as follows. We let $V = S \cup \{z\} \cup \{x_1, \ldots, x_{|C|}\}$ (where the x_is are intended to represent 3-sets), $R = S \cup \{z\}$, $E = \{\{a_i, x_i\}, \{b_i, x_i\}, \{c_i, x_i\} | \{a_i, b_i, c_i\} \in C\} \cup \{\{z, x_i\} | 1 \leq i \leq |C|\}$, and $K = 4q$. All edges have unit length. This transformation is clearly polynomial-time.

Now, given a solution to *X3C*, we obtain a solution to *STG* by including all corresponding x_is and their associated edges in our Steiner tree; the disjointness of the 3-sets in the solution guarantees that the result is a tree with exactly $4q$ edges. Conversely, observe that any solution to *STG* must have exactly $4q$ edges (at most that many are allowed, and at least that many are needed to produce a tree that includes all vertices from S as well as z). Thus in fact, exactly q edges emanating from z are included: those are the 3-sets in the solution for *X3C*; the tree structure guarantees disjointness.

An alternate construction omits the root and connects all 3-set vertices in a complete graph; $R = S$ and $K = 4q - 1$ and all edges still have unit length.

From *VC*, we can copy the graph, but complete it into the complete graph on $|V|$ vertices; now, for each original edge $e = \{u, v\}$, we set a new vertex e and associated edges $\{u, e\}$ and $\{v, e\}$. The bound is just $B + |E|$ (B is the bound of *VC*) and R consists of all the new vertices e. Since a tree of n vertices has $n - 1$ edges (a result we proved in Chapter 2) and since our edges all have the same length, we can simply look at the minimum number of vertices needed in the Steiner tree: it is clearly all of the new vertices (the set R of size $|E|$), plus enough additional vertices to turn the whole into a tree. Since the new vertices corresponds to edges of the old graph and are only connected to vertices of the original graph, any Steiner tree must include one or both of $\{u, e\}$ and $\{v, e\}$, i.e., a vertex cover of the original graph; these vertices can then be connected into a tree through a simple chain in the complete graph.

Exercise 7.27

1. The problem is clearly in NP. To prove that it is NP-complete, we transform *X3C* to it. Given an instance S, $|S| = 3q$, $C \subset 2^S$, $(c \in C \Rightarrow |c| = 3)$ of *X3C*, we construct an instance $G = (V, E)$, $K \leq |V| - 1$, $w\colon E \to \Re$ of *BDST* as follows. We set $V = S \cup \{z_0, z_1, z_2\} \cup \{x_1, \ldots, x_{|C|}\}$ (where the x_is are intended to represent 3-sets), and $E = \{\{a_i, x_i\}, \{b_i, x_i\}, \{c_i, x_i\} | \{a_i, b_i, c_i\} \in C\} \cup \{\{z_0, x_i\} | 1 \leq i \leq |C|\} \cup \{\{x_i, y_j\} | x_i, y_j \in C\} \cup \{\{z_0, z_1\}, \{z_1, z_2\}\}$. Then we set the maximum diameter to 4 and assign unit cost to all edges except the $\{z_0, x_i\}$ edges which get assigned a cost of 3; the bound on the total cost is then set to $5q + |C| + 2$. This transformation is clearly polynomial-time.
   The bound is easily reached with an exact cover by using a spanning tree with the root at z_0, with the solutions 3-sets as the first-level children, and with the set elements, as well as the discarded 3-sets, as grandchildren. Conversely, the diameter of 4 drastically limits the type of solution that can be obtained in our problem; in fact, since the eccentricity of z_0 is exactly 2 (due to the "tail" of z_is and the fact that it is also 2 edges away from the set elements), we must have z_0 as center ("root") of the solution tree. The required cost allows at most q edges of cost 3, which then designate the solution sets, since all set elements must be reachable from these q vertices.
   Another reduction starts from *1in3SAT*. The construction is exactly as in the text's reduction from *NAE3SAT* to *G3C*, except that an extra "tail" of 3 vertices is attached to the "root" of the construction. All edges are given length 1 or 2: the edges in the tail, those forming the clause triangles, and those connecting the complemented and uncomplemented versions of each variable are assigned unit lengths; the others are assigned length 2. The diameter bound is then set to 6 and the length bound to $4k + 3n + 3$ (for n variables and k clauses).

The tail ensures that the eccentricity of the "root" is 3, so that the vertices of the clause triangles must be reachable from said root in no more than 3 edges—which prevents any possibility of reaching the vertices of one triangle from a vertex of another. Spanning the tail will contribute 3 to the total length, leaving $4k + 3n$. Now reaching all three vertices of a clause triangle can be done in one of three ways: (i) through 3 edges from the variables triangles, for a length contribution of $3 \cdot 2 = 6$; (ii) through 2 such edges plus a triangle edge, for a length contribution of $2 \cdot 2 + 1 = 5$; or (iii) through one such edge plus two triangle edges, for a length contribution of $2 + 2 \cdot 1 = 4$. If all clauses are spanned in this last manner (corresponding to one literal per clause being true—that touched by the edge from the variable triangles), the contribution to the total length reaches a minimal value of $4k$. Finally, the variable triangles must be spanned by one edge from the "root" (of length 2)—indicating the true literal—and one from the true literal to the false (of length 1); any other possibility contributes more than $3n$ to the total length. Notice that each path from the "root" to the clause vertices must go through the true literals, since otherwise there would be paths of 4 edges ("root" to true literal, true literal to false, false literal to first clause vertex, first clause vertex to other clause vertex) and such a path would result in a tree diameter of 7. Thus we have a direct correspondence between the true literal of each clause and a truth assignment.

Yet another construction is from *3SAT*; similar to above, but each clause is simply a single vertex. The diameter is set to 4 and the tail has only two edges. The edge connecting literal and complement for each variable has length 1; the edges connecting these pairs to the "root" have length 2; the edges connecting to the clause vertices have length 3. The bound is set to 3*(#vars+#clauses). The proof follows easily.

2. We restrict this problem to *Hamiltonian Path* by restricting its instances to those where D is set to 2. Since every node in the tree must have degree 2 or less, the tree is in fact just a simple path; since it is a spanning tree, it is a Hamiltonian path.

3. This is the other variant of the spanning tree problem that is not just a Hamiltonian path in disguise. Fortunately, we can reuse the construction used in Exercise 7.26 (Steiner tree).

   One construction is exactly the same as in our solution for Exercise 7.26; the bound is set to $K = 2q + p$, where $n = 3q$ is the number of elements and p the number of 3-sets in the instance of *X3C*.

We claim that the "root" of the constructed graph must be the "root" of the spanning tree. If it were not, all 3-set vertices would have to be reached through other 3-set vertices and their elements; but then we cannot have enough leaves in all. (The "best" case—although one in which there is no solution in either instance—has one element belonging to every 3-set; reaching the other $3q - 1$ elements requires $\lceil (3q-1)/2 \rceil$ 3-set elements; the resulting maximal number of leaves is $(3q-1) + (k - \lceil (3q-1)/2 \rceil) + 1$, which simplifies to $k + 3q - \lceil (3q-1)/2 \rceil$, which is smaller than $k + 2q$ for all $q > 1$. For $q = 1$, the values are equal, but that number is only reachable with a no-instance, since all 3-sets overlap.) (This whole reasoning could easily be avoided by endowing the "root" with a "tail," as in many similar constructions.) This being established, only one other observation need be made: any edge from an element to a 3-set is advantageously replaced by an edge from the "root" to the 3-set: such a change does not impair the spanning property and can only increase the number of leaves. The proof follows easily.

An alternate construction starts from *3SAT*. For each variable x, set up 3 vertices, say x, x_{mid}, and $\overline{x}$, connected in that order by two edges. Set up an additional vertex (root R) and connect it to all vertices corresponding to complemented and uncomplemented literals ($2n$ edges). Finally, set up one vertex for each clause and connect it to the 3 corresponding literal vertices. Set K to the number of clauses plus twice the number of variables. Since the x_{mid} vertices have degree 2, one of the x and $\overline{x}$ vertices cannot be a leaf; the proof follows easily.

4. We restrict this problem to *Hamiltonian Path* by restricting its instances to those where the bound is set to 2. A tree that has only two leaves (two nodes of degree 1) is a simple path; since it is a spanning tree, it is a Hamiltonian path.

5. We restrict this problem to *Hamiltonian Path* by restricting its instances to those where the subset of leaves has exactly two leaves. A tree with two leaves, as just remarked, is a simple path; thus any solution tree is a Hamiltonian path between the two specified nodes.

6. We restrict this problem to *Hamiltonian Path* by restricting its instances to those where the tree is in fact a simple path of $|V| - 1$ edges. Such a tree has exactly the number of edges needed for a spanning tree, but it is a path and thus the solution must be a Hamiltonian path.

Exercise 7.28

Membership in NP is trivial: bipartiteness can be tested in linear time by depth-first search. Notice that a graph is bipartite if and only if there exists a partition of its vertices into two subsets such that each edge of the graph is cut (has its endpoints on opposite sides of the partition). This suggests a very simple transformation from *MxC*. Given an instance of *MxC*, $G = (V, E)$ and K, we produce an instance of our problem by using the same graph but setting the bound to $|E| - K$. If the resulting graph is bipartite, all its edges can be cut, and its has $|E| - (|E| - K) = K$ of them left, thereby satisfying the cut requirement. Conversely, if the graph can be partitioned such that K edges are cut, then removing the remaining uncut edges ($|E| - K$ of them) will make the graph bipartite.

- *Minimum vertex-deletion bipartite subgraph*

Instance: as above.
Question: as above, but vertices get deleted instead of edges.

Membership in NP is obvious. We transform *VC* to our problem as follows. Given an instance $G = (V, E)$ and K of *VC*, we produce an instance $G' = (V', E')$ and K' of our problem where $V' = V \cup E$, $E' = E \cup \{\{e, v_i\}, \{e, v_j\} | e = \{v_i, v_j\} \in E\}$, and $K' = K$. That is, we produce a triangle in G' for each edge in G (a construction used in the *Dominating set* example). The construction is easily carried out in polynomial time. Now, notice that each triangle thus formed will have to be broken in order to obtain a bipartite graph; that is, one vertex in each triangle must be removed. Thus, if we have a solution to *VC*, we obtain a solution to our problem by removing the covering vertices from the transformed graph: since the vertices form a cover, they include one endpoint of each edge and thus one point in each triangle. The graph left after removal of these K vertices includes only edges of the form $\{e, v_i\}$, for $e = \{v_i, v_j\} \in E$, and is thus bipartite (just put the vertices in $V' - E$ on one side and the others on the other side). Conversely, a solution to our problem includes one point in each triangle; this point can be taken as a member of $V' - E$ without loss of generality (if the solution includes a degree 2 vertex from E, we can replace it by another vertex from the triangle, this time from $V' - E$, at no loss). Thus a set of K vertices from V includes one endpoint of each edge: it is a vertex cover. (The remark about vertices of degree 2 can be avoided by setting two extra vertices for each edge, with each vertex connected to both endpoints of the original edge; now both triangles must be destroyed, so that only original vertices can be used.)

Another transformation starts from *NAE3SAT* and just produces the usual triangle per clause, segment per variable, and connections between complements, with K set to the number of clauses. The "don't care" literal in each clause is removed and the rest split: the proof is trivial.

Exercise 7.29

We prove this problem NP-complete (it is obviously in NP) by transforming *NAE3SAT* to it. The proof is a local replacement—with simple pieces for all but the truth-setting component. For each clause, we set up a simple triangle, where vertices correspond to literals. Cutting each triangle corresponds to setting at least one literal to true and one to false in each clause. For each variable, we set up a truth-setting component with two connecting vertices (corresponding to the uncomplemented and the complemented literals). The problem here is to ensure that these two vertices be placed on opposite sides of the partition; this is not as easy as it seems: see below for a suitable graph piece. Finally, we connect each clause literal to the corresponding attaching point in the truth-setting component; the connection piece ensures that its two extremities be placed on the same side of the partition. A suitable piece consists of a triangle and the two extremities, where each extremity is connected to every vertex of the triangle. (Notice that this is just a K_5 with one missing edge—between the extremities; notice also that K_5 is the smallest graph that is not partitionable.) The proof follows immediately. (Notice that the connection piece could be used for truth setting rather than the truth setting piece itself, following the lines of the next proof.)

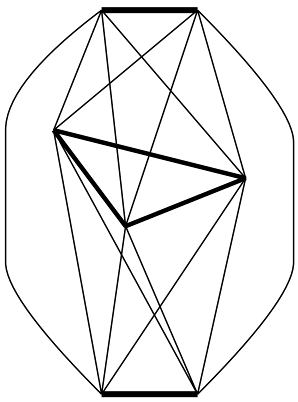
A larger, but also simpler piece builds 3 pairs of triangles with the two vertices as bases: u, v, t_{i1} and u, v, t_{i2}, with additional edge $\{t_{i1}, t_{i2}\}$, for $i = 1, 2, 3$; then it adds a new triangle, say t_1, t_2, t_3, and connects t_i to both t_{i1} and t_{i2}. If vertices u and v have color 1, they force all t_{ij} to have color 2, which in turn forces all vertices t_i to have color 1, which forms a monochromatic triangle.

From *Positive NAE3SAT*, things are easier; we only need use our connector, setting up a triangle for each clause and a single vertex for each variable. That this construction works is obvious. Alternately, we can set up a triangle for each clause, one for each variable, and connect each clause "variable" to all three vertices of the corresponding variable triangle. (This is in fact the same thing, with a little simplification arising from dispensing with the variable vertex and re-using the connecting piece's triangle.)

Exercise 7.30

The problem is clearly in NP: we can verify in polynomial time (cubic time at worst) that no triangle exists on either side of the partition induced by the edge coloring. (For instance, simply create two graphs, each on the same set of vertices, one with the blue edges, the other with the red edges; then check each graph for the presence of a triangle.)

We transform *Positive NAE3SAT* to *Monochromatic Edge Triangle*. Our old idea of using a triangle for each clause works perfectly here, since each triangle will have to be partitioned in order to avoid forming a monochromatic one. The only problem left, therefore, is consistency. Consistency here is a bit more difficult than in the vertex version: we shall need a gadget that connects two edges in such a way that the two edges must always be colored the same way in any solution. Fortunately, our experience with K_5 in the previous problem helps. Note that the smallest graph that cannot be edge-partitioned without leaving a monochromatic triangle is not K_5: if we draw K_5 as a pentagon with all edges internal to the pentagon, we can color the pentagon edges one color and the others another and avoid any monochromatic triangle. K_6 cannot be partitioned so as to avoid a monochromatic triangle (easily verified), but K_6 minus an edge is not the smallest critical graph in terms of edges (it has 20 edges and the piece described below has only 19)—nor is it colorable in such a way that two designated edges always get assigned the same color. A suitable gadget can be derived much the same way we derived the gadget for the vertex version: we shall place a new triangle between the two edges and connect each endpoint of each edge to all three summits of the triangle; in addition, we also match the endpoints of the two edges. The result is shown below. As in the vertex version, we can now simply connect all clause



triangle edges corresponding to the same variable into a simple chain by using our gadget for each link.

Exercise 7.31

We prove this problem NP-complete (membership in NP is obvious) by transforming *Hamiltonian Path* (HP) to it. We simply set up a matrix with $|E|$ columns and $|V|$ rows; each column corresponds to an edge and each row to a vertex; the entries are zeros and ones, where a_{ij} is one if and only if the ith vertex is an endpoint of the jth edge. Thus each column has exactly two one entries and $|V| - 2$ zero entries. We then set K to $|V| - 1$: the selected columns will correspond to the edges of the Hamiltonian path.

If the graph has a Hamiltonian path, we select those columns of the matrix that correspond to the edges of the Hamiltonian path. There are $|V| - 1$ such edges, so we exactly meet the bound K.

Since each column has exactly two ones, making them consecutive by a permutation of the rows simply indicates that the edge corresponding to the column is between two vertices adjacent in the new row ordering. Since all edges are distinct and since we have $|V| - 1$ of them, all adjacencies $(i, i + 1)$ for $1 \le i < |V|$ must be covered. But a Hamiltonian Path is exactly a permutation of the vertices (read rows) such that there is an edge between any two consecutive vertices. Thus our transformed instance has a solution if and only if the original problem does.

Exercise 7.32

Membership in NP is obvious. We transform *Vertex Cover* to our problem as follows. S will contain V and an additional $|V| - K$ elements, call them $a_1, \ldots, a_{|V|-K}$. We set up C to have two types of elements: the first type is such that only a set X corresponding to a vertex cover will not be contained in any of them; the second provides an enforcement of K by ensuring that any reasonable X will be contained in all of them. The first type has an element for each edge $\{v_i, v_j\}$; the element is the set $V - \{v_i, v_j\}$; the second type has elements $s_1, \ldots, s_{|V|-K}$, where $s_i = V \cup \{a_i\}$. (The only reason for the a_i is to make it possible to have $|V| - K$ "copies" of V in a set.) Now B is such that any set of n elements is contained in $|V| - n$ its elements: B has $|V|$ elements, with the ith element being $V - \{v_i\}$. (Note that B is set up this way to offer a threshold equal to that of C; note also that the threshold depends on K, which is the whole reason for this part: otherwise we might as well have let $B = \emptyset$ and C be composed only of the first type of element.) The whole construction is easily carried out in polynomial time.

Now, a solution to vertex cover easily yields one to our problem: let X be composed of the elements of the cover. By design, X is contained in each of the s_i in C and not in any of the other elements of C; thus it is contained in $|V| - K$ elements of C; since it contains $|X|$ elements, it is contained in $|V| - |X|$ elements of B; but $|X| \le K$ by hypothesis, hence we have a solution. Conversely, note first that any solution X never contains any of the a_i elements; thus a solution X is in effect a subset of V. Now, X is automatically contained at least $|V| - K$ times in C (the s_i elements) and at least $|V| - |X|$ times in B. It could be contained more often in C, but we must have as a minimum condition $|V| - |X| \geq |V| - K$, hence $|X| \leq K$. Thus X must meet the size requirements for a cover. We claim that it does correspond to a vertex cover for the original graph. The problem is that we could very well get a solution set X with $|X| < K$ such that X is contained more often than $|V| - K$ times in C and thus does not correspond to a cover; but this is easily corrected: we'll create X' such that $X \subseteq X'$, $|X'| \leq K$, and X' corresponds to a cover. We simply take any element of C in which X is contained that indicates an edge not covered and add one of the vertices to X; this operation decreases the number of containments in C by at least 1 (possibly more) and the number of containments in B by exactly 1; thus our larger X is still a solution. We repeat until the set obtained, X' is contained exactly $|V| - K$ times in C; by our earlier discussion, X' is then a solution. Since it is contained exactly $|V| - K$ times in C, it also corresponds to a cover.

Exercise 7.33

Membership in NP is obvious. We transform *Positive NAE3SAT* to our problem as follows. For each variable, x, in the original instance, we place one element in A; for each clause, c, we place one element also; in addition, A also includes an enforcer element, call it u; thus A has $m + n + 1$ elements. For each clause, $c = \{x, y, z\}$, we set up two triples, (x, c, y) and (c, u, z) in our transformed instance. This transformation is easily accomplished in polynomial time. Now, given a solution to the *Positive NAE3SAT* instance, we obtain a solution to our transformed instance by placing (in any order) the true variables on one "side" of u and the false variables on the other "side." If the first two variables of a clause have the same truth value, we place the clause element between these two variables in the order; if not, then we can place the clause element anywhere in the order. Conversely, a solution to our transformed instance yields a solution to *Positive NAE3SAT*: since c must be on the side opposite to z and yet between x and y, it follows that at least one of x or y must be on the side opposite to z, so that the clause has a not-all-equal assignment.

Exercise 7.34

There are really two versions of this problem, depending on whether the pair $(M, p())$ is fixed for the problem or is part of each instance. If the pair is fixed for the problem, then each pair $(M, p())$ defines a different problem; and different problems have different complexity. For instance, a checker for *SAT* that expects a truth assignment specified by a binary string defines a problem where every possible string is a valid certificate for some "yes" instance of *SAT*. In this case, the complexity of the problem is constant—the solution algorithm is simply `print "yes"`. The relevant question for this version of the problem is simply "how difficult can any of these problems be?" Checking that the string is in the proper syntactic form (that it follows the encoding expected by the checker) is easily done in polynomial time—indeed, any reasonable encoding scheme must be such that the decoding can be done in time polynomial in the length of the code. So, given a syntactically correct string, must this string be a valid certificate? and if not, can we decide whether it is valid or not in polynomial time? The answer to the first question remains unknown, but the answer to the second is yes, as can be deduced from our reasoning about the more general problem below.

The other version, in which the checker and its bound are part of the instance, defines a single problem; it was the intended problem and is more interesting. Since we do not know for what particular NP problem the checker is intended, we cannot easily reconstruct an instance for what we may believe to be a valid certificate, nor can we determine even as much as the syntactic correctness of a string—at least not without analyzing the checker program and determining its properties, something that is in general undecidable. We appear, therefore, to be reduced to using non-determinism to guess an instance x and to feeding the pair (x, y) to the checker and waiting for the answer. Even this step is not obvious: the checker takes at most $p(|x|)$ time to carry out its computation, but our input is y, not x: can we translate the polynomial bound in $|x|$ into some bound in $|y|$? Unfortunately, while we know that $|y|$ is bounded by a polynomial function of $|x|$, we have no such guarantee in the reverse direction.

However, we can make some useful observations. First, if there exist several instances x for which the same y is a valid certificate, then we can always consider the shortest of these instances (For instance, given any graph with vertex cover of size K, we can always add any number of independent vertices to the graph without changing the solution and thus without changing the certificate, at least in some encodings; given the certificate, though, we can reconstruct a graph for it with at most K edges and thus concise in the size of the instance. Similarly, given a satisfiable Boolean formula, we can always add any number of redundant variables by padding the clauses as seen in the transformation for *3SAT*; while a concise certificate need not include the truth values of these redundant variables and thus can appear increasingly small, we can always reconstruct from such a certificate of length n an instance of *Satisfiability* with n 1-literal clauses.) Secondly, given an NP-complete problem, non-trivial instances in which all "fluff" (such as unconnected vertices in most graph problems) has been eliminated cannot really have very short certificates; if their certificates had length $O(\log n)$, for instance, we could try them all in deterministic polynomial time and thus show $\mathrm{P} = \mathrm{NP}$. We conclude that we must have $|y| = \omega(\log |x|)$, or $|x| = o(c^{|y|})$, so that our problem is at least in NExp—not a particularly encouraging result.

To tighten our conclusion, we must revisit the examination of certificates. We want to show that, for each valid certificate, there exists at least one instance that is concise in the length of the certificate. For that, we can appeal to Exercise 6.19 about honest polynomial functions, in which we used a well-behaved pairing function to produce the instances of an NP problem—along with their certificates. In that pairing, the certificate which serves to validate the instance is of size comparable to the instance, at least when the output is produced honestly. Thus we can conclude that, for any problem in NP, valid certificates can be used to certify "yes" instances of polynomially related sizes. This observation allows us to place our problem in NP: guess an instance of polynomially related size and feed it along with the certificate to the checker, which will then run in time polynomial in both arguments. It does not appear that we can do better, even if all syntactically correct strings are valid certificates, since we cannot check for syntactic correctness without running the checker and since the checker must have both an instance and a putative certificate in order to run.

Exercise 7.35

The problem is in P: we can grow S systematically by first finding the result of each operation on $S \times S$, then the result on each pair containing one or two of the newly created elements, and so on until no new element is created; then we verify whether w was among those created. The number of iterations is at most $|A| - |S|$ (since we must add at least one new element at each iteration) and the work at each iteration is no larger than $O(|A|^2)$, so that our algorithm is at worst $O(|A|^3)$.

To prove that the problem is complete, we transform *PSA* to it. Let the *PSA* problem be given by X, $S \subseteq X$, $T \subseteq X$, and $R \subseteq X \times X \times X$. We let the set of elements of our generation problem be $A = X \cup R \cup X \times R \cup \{\omega, \alpha\}$, where $\alpha, \omega \notin X$, with initial subset S. The element to be derived will be α (ω is an absorbing element) and the operation is defined by

$a \odot \omega = \omega \odot a = \omega$, for any $a \in A$
$a \odot \alpha = \alpha \odot a = a$, for any $a \in A$

$$a \odot a = \begin{cases} \alpha & a \in T \\ a & \text{otherwise} \end{cases}$$
$x \odot r = r \odot x = (x, r)$ for all $x \in X$ and $r \in R$
$(x, r) \odot (y, r) = z$ if $r = (x, y, z)$
all other compositions yield ω

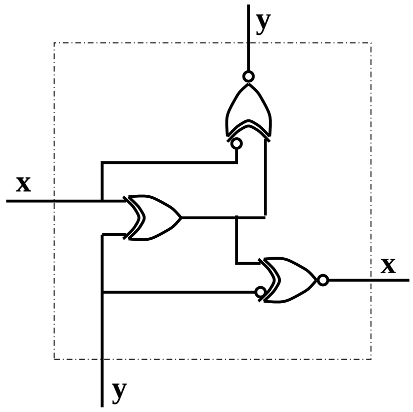
(That operation is not generally associative.) The multiple "copies" of each element of X are there to allow unique definition of each operation even in the presence of several triples in R with the same first two entries. The entries from R serve as labels, with the operation doing the labelling $\odot\colon X \times R \to X \times R$; note however that the result of an operation which corresponds to a relation is put back in unlabelled form. It is easily seen that f can be derived from the initial subset if and only if some element of T is accessible.

Exercise 7.36

The problem is in P: *CV* itself is in P, so we need only verify planarity of the circuit's graph, which can be done in linear time as mentioned in Chapter 2.

To prove that the problem is complete, we transform the version of *CV* produced by our reduction from *PSA* to it. In a case like this, the standard approach is to develop a *crossing* gadget (see Chapter 8), that is, a planar circuit piece of polynomial size that can replace each crossing in an embedding of the (presumably) nonplanar original instance. The only crossings that can occur in our instance are crossings between a left-to-right propagation line and a vertical propagation line. So we need a planar circuit piece that will propagate these values properly; from the statement of the problem, it appears that negation will be required to realize such a circuit.

We use the fact that the XOR of two values can be decoded back into these values if we know one or the other value. We produce the following simple planar circuit.



The reader can easily verify that logical values are properly propagated from left to right and from bottom to top; inverting top and bottom yields a circuit that properly propagates logical values from left to right and from top to bottom. Thus we have a suitable crossing component.

Our circuit uses negation, as expected; unfortunately, it also uses XOR gates, whose direct implementation with AND and OR gates is not planar. The question was not entirely precise, since the *CV* problem could obviously be defined with any family of Boolean connectives. Our construction shows that *Planar CV* with connectives XOR, NOT, AND, and OR is P-complete. It remains to decide whether we can strengthen this statement by eliminating XOR gates from the choice of Boolean connectives.

Exercise 7.37

That *Digraph Reachability* is in NL follows from the following nondeterministic solution algorithm: start at vertex 1, with counter k set to 0; at each step, guess the next vertex on the solution path, increment the counter k, and verify that there exists an arc from the current vertex to the next vertex. Stop and accept as soon as the destination vertex is reached; stop and reject as soon as the counter reaches n or no arc is found between the current vertex and the guessed vertex. The storage requires is the current vertex, the guess vertex, and the counter; each takes logarithmic space.

To show that *Digraph Reachability* is NL-complete, we reduce a generic problem in NL to it, using much the same style of reduction as of *PSA*. We shall represent the state of the NL machine by full IDs consisting, as in the proof of Theorem 6.8, of a k-tuple. (But this time we can afford full IDs, because the machine we are encoding uses only logarithmic space.) Since an NL machine has three tapes, we need to indicate the head position on both tapes and the contents of the scratch tape. (The output tape is irrelevant in a language acceptor, since the output is either 0 or 1; and the contents of the input tape is just x, unchanging, so we do not encode it—our simulation can always recover the value of a particular tape square by reading the input from tape square 1 to the desired tape square.) Thus an ID will contain the current control state, the current step number, the current head position on each tape, and the current contents of each square on the scratch tape. The current control state and the current contents of one tape square require $O(1)$ space only; the current head position on the input tape requires $O(\log n)$ space and that on the scratch tape $O(\log \log n)$ space; the step counter may have to count a polynomial number of steps and thus also takes logarithmic space; and finally, we need to encode a logarithmic number of tape squares on the scratch tape since we are dealing with an NL machine. Thus a full ID of the NL machine takes logarithmic space. We can thus generate all full IDs (the nodes of the graph) using just logarithmic space (we just need suitably nested loops) and then generate any arcs as needed on the fly by testing the two IDs for the possibility of a single transition (at which time we need to consult the input tape to extract the tape symbol under the head—a rather slow process, since we can only do it by reading from the start of the input to the desired symbol, but one that does not use storage beyond what is needed for a counter). If the NL machine has multiple accepting configurations, we add a single target node to the graph and add an (implicit) arc from each accepting configuration to the target node.

Exercise 7.38

The problem is in NL: since the operation is associative, w can be derived from S, if at all, as a chain of operations on the elements of S. Thus we can guess the chain and verify it operation by operation; all we need memorize is the result of the last operation verified (i.e., one element of A).

To show that the problem is complete, we transform *Digraph Reachability* to it. Let the digraph be given by $D = (V, A)$; we let the set of elements of our generation problem be $(V \times V) \cup \{(0, 0)\}$. The operation on our set is defined by $(i, j) \odot (k, l) \to (i, k)$ if $j = k$ and $(0, 0)$ otherwise; this is clearly associative. Finally, we let $w = (1, n)$ and $S = \{(i, j) | (i, j) \in A\}$. It is obvious that w can be derived in our new problem if and only if n is reachable from 1 in the reachability problem. (Many other ways to define the operation; it always needs an absorbing element, however.)

Exercise 7.39

We want to identify a variable, x, such that the clauses imply $(x \to \overline{x})$ and $(\overline{x} \to x)$, a contradiction; in fact, we will prove that a set of 2-CNF clauses is a contradiction if and only if such a variable exists. Since $\rightarrow$ is transitive, the implications may be through a chain, $(x \rightarrow y_1)(\overline{y}_1 \rightarrow y_2) \ldots (\overline{y}_k \rightarrow \overline{x})$ and $(\overline{x} \rightarrow z_1)(\overline{z}_1 \rightarrow z_2) \ldots (\overline{z}_l \rightarrow x)$. Recalling that $(a \rightarrow b)$ is equivalent to $(\overline{a} + b)$, we then claim:

> A Boolean formula in 2-CNF is not satisfiable if and only if there exist two subsets of clauses $\{(x + x_1) \cdot (\overline{x}_1 + x_2) \ldots (\overline{x}_k + x)\}$ and $\{(\overline{x} + x_{k+1}) \cdot (\overline{x}_{k+1} + x_{k+2}) \ldots (\overline{x}_n + \overline{x})\}$.

The if part is trivial. If we can prove the only if part, then it is obvious that the problem is in NL, since we can guess the series of clauses and then verify their existence in the original expression.

To prove the only if part, we proceed by induction on the number of clauses. The basis is trivial. Suppose that the theorem holds true for all instances of n and fewer clauses. The formula is not satisfiable if and only if neither of its restrictions to $x = true$ and $x = false$ is satisfiable either. But each restriction can be simplified down to a 2-CNF formula with fewer clauses: each clause in which x appears either has a literal set true—in which case it can be dropped—or a literal set to false, in which case we set the other one to true, continuing until we get a contradiction (a clause wit both literals set to true) or until we are left only with clauses of 2 as-yet-undefined literals. In the latter case, the resulting 2-CNF must also be unsatisfiable, so that the inductive hypothesis applies and we get the desired collection of clauses. In the earlier case, another induction proof (this time trivial) on the number of clauses involved in forcing other literals shows that we have the desired pattern of clauses. Hence our proof is complete.

Now, to show that the problem is complete, we transform *Digraph Reachability* to it. Each vertex, i, will become a variable, x_i; for each arc (i, j), we set up two clauses, $(x_i + \overline{x}_j)$ and $(\overline{x}_i + x_j)$. Finally, we add the two clauses $(x_n + x_1)$ and $(\overline{x}_n + \overline{x}_1)$. Since a collection of clauses of the type described above has one clause where both literals are complemented and one where both literals are uncomplemented, and since only two such clauses exist in our transformed problem (those involving 1 and n), it is easy to see that our set of clauses is a falsehood if and only if vertex n is reachable from vertex 1.

Exercise 7.40

We know (see Exercise 7.24) that the associated decision problem is NP-complete; thus we need only show that the optimization version is NP-easy.

We show how to use an oracle for the decision problem to solve the optimization problem. Notice that the external path length is an integer quantity and cannot exceed a maximum (using only type $\{x_i\}$ tests) of

$n + 2 + \Sigma_i i,\ 1 \leq i \leq n + 2 = (n^2 + 7n + 10)/2$; thus we can use the oracle in a binary search to determine the minimum cost for the problem with at most $O(\log n)$ queries. Then we can undertake the process of finding the root: pick an arbitrary test and separately find the minimum cost for each of the 2 subproblems (using the same binary search procedure), all of which can be done in time $O(n)$; then put these costs together to get the cost of the optimal tree with that root choice (just the sum of the 2 costs and of the number of objects in the problem at hand) and compare it with the known optimal; if equal, we have found the root, if not, try another test. The process is then repeated recursively on each subproblem. The total time is at worst $O(n^2)$ for a root, or $O(n^3)$ for the whole tree, with at most $O(n^2 \cdot \log n)$ queries to the oracle. Thus the optimization problem is NP-easy; since it is obviously NP-hard, we are done.

Exercise 7.41

We shall prove the circular chain of implications:

$$\mathrm{NP} = \mathrm{coNP} \Rightarrow \Delta_2^p = \mathrm{NP} \cup \mathrm{coNP} \Rightarrow D^p = \mathrm{NP} \cup \mathrm{coNP} \Rightarrow \mathrm{NP} = \mathrm{coNP}$$

Since $\mathrm{NP} \cup \mathrm{coNP} \subseteq D^p \subseteq \Delta_2^p$, the middle implication is immediate (D^p is "sandwiched" in the middle).

The first implication is fairly easy to prove, for instance as follows. Since we have NP = coNP, any problem in coNP must have a concise certificate—or, equivalently, both yes and no instances of any problem in NP must have concise certificates. Thus we can replace any call to an oracle for NP or for coNP by a certificate for the answer; in other words, a nondeterministic polynomial-time machine can do all the work by guessing the correct oracle answers, verifying them with the correct certificates, and otherwise doing the same work as the deterministic polynomial-time oracle machine. Thus the problem is in NP, i.e., $\Delta_2^p \subseteq \mathrm{NP}$; since the containment holds the other way, the two classes are equal, which proves our first implication.

The last implication is the most difficult to prove. Perhaps the simplest way to go about it is to consider the known D^p-complete problem *SAT-UNSAT*; this problem consists of two collections of clauses and asks whether it is the case that the first collection is satisfiable while the second is not. If $D^p = \mathrm{NP} \cup \mathrm{coNP}$, then this problem is in NP or in coNP (or in both). If it is in both, we are done, since the problem is complete for D^p. If the problem is in NP (but not known to be in coNP), then it has a concise certificate; in particular, this means that the *UNSAT* part of the problem has a concise certificate and thus is in NP. But *UNSAT* is coNP-complete, so that we have NP = coNP. The symmetric reasoning holds if the problem is in coNP.

Exercise 7.42

Obviously, what we need is a proof by induction. In order to examine what the induction step involves, let us simply look at going from NP to Σ_2^p. If we can work it out, we can then repeat the idea to go from Σ_k^p to Σ_{k+1}^p and thus prove the result (since it is clearly sufficient to show it for one type of class). In order to establish that problems in $\Sigma_2^p = \mathrm{NP}^{\mathrm{NP}}$ can be solved in $O(n^{\log^k n})$ time for some constant k, we need to "push" a problem from that class down to NP, use the known solution algorithm for NP problems, then recover a running time for the original problem. What normally makes $\mathrm{NP}^{\mathrm{NP}}$ different from NP is that the oracle can handle both yes and no answers, not just yes answers, preventing us from guessing a polynomial number of certificates (one for each oracle call). However, we know now that the oracle, which simply answers a question in NP, can be replaced by a certificate that uses $O(n^{\log n})$ bits—because the certificate can simply be the entire sequence of steps taken by the solution algorithm known to exist for NP problems. Thus we can solve a problem in $\mathrm{NP}^{\mathrm{NP}}$ non-deterministically by guessing the polynomial number of oracles answers in the form of a polynomial number of certificates, each of size $O(n^{\log n})$, and by otherwise doing whatever the original nondeterministic machine did. In other words, our problem of size n in $\mathrm{NP}^{\mathrm{NP}}$ corresponds to a problem of size $O(p(n) \cdot n^{\log n})$ in NP, where $p(n)$ is some polynomial. But we have a deterministic solution algorithm for problems in NP; on a problem of size $S = O(p(n) \cdot n^{\log n})$, this algorithm will take $O(S^{\log S}) = O\left((p(n) \cdot n^{\log n})^{(\log(p(n) \cdot n^{\log n}))}\right)$ time. Simplifying this last expression takes some work; note that $\log(p(n) \cdot n^{\log n})$ is $O(\log^2 n)$ and that $p(n) \cdot n^{\log n}$ is $O(n^{k+\log n})$ for some constant $k > 0$, which in turn is $O(n^{\log^2 n})$, so that the entire expression is $O(n^{\log^4 n})$, proving this first step. The very same line of reasoning proves the general step, so we are done.

Exercise 7.43

In reviewing our proof of Cook's theorem (Theorem 6.6, pp. 203–207), we note that each of the variables we defined must take on a unique value in any assignment that reflects a particular ID of the simulated Turing machine, with the single exception of the variables that define the contents of the tape to the left of the head at step 0. Thus the number of distinct truth assignments that satisfy the entire collection of clauses is exactly the number of distinct certificates that lead to acceptance. In other words, our construction is (strictly) parsimonious. Thus counting the number of distinct satisfying truth assignments is indeed #P-complete.

Exercise 7.44

- at least 3 This is in NP: the certificate simply consists of 3 distinct truth assignments; it is easy to check that the three are distinct and that they are valid, both in polynomial time. It is in fact NP-complete: given an instance of *3SAT*, just add to it new variables and clauses that can be satisfied in exactly 3 ways (for instance, variables x, y, z and clauses $x + y + z$, $x + y + \overline{z}$, $x + \overline{y} + \overline{z}$, $\overline{x} + y + \overline{z}$, and $\overline{x} + \overline{y} + \overline{z}$); then the transformed instances has 3 times as many satisfying truth assignments as the original. Note that this can be done for any fixed value of k: i.e., if k is a constant and not a parameter, then "at least k" is NP-complete.
- at most 3 This is coNP-complete, as it is the complement of "at least 4."
- exactly 3 This is the combination of "at least 3" and "at most 3;" in other words, it is the combination of an NP-complete question and a coNP-complete question. As a result, it is certainly in Δ_2 (i.e., NP-easy); but we can do a bit better. Note that it is not necessarily in NP $\cup$ coNP, since it needs the power of both at once. We mentioned the class D_p, a superset of NP $\cup$ coNP and a subset of Δ_2, composed of languages that are intersections of an NP-complete and a coNP-complete language. This is exactly what our problem is and thus it is in D_p; it also turns out to be complete for D_p, but that is a bit harder to show.

Exercise 7.45

Interestingly, the obvious symmetric counterpart of this statement is not known to hold: that is, we do not know whether *Unique Satisfiability* can be in coNP without forcing NP = coNP.

Assume that *Unique Satisfiability* is in NP. Then it has a concise certificate, which attests not only to the satisfiability of the instance, but also to the fact that it cannot be satisfied in more than one way. We want to show that *Unsatisfiability* is then in NP. Given an instance of *Unsatisfiability*, we transform it into an instance of *Unique Satisfiability* as follows. Denote the instance of *Unsatisfiability*, a Boolean expression over the variables $\{x_1, \ldots, x_n\}$, by E; set all variables to true and evaluate E, then evaluate E. If the result is true, we have a "no" instance of *Unsatisfiability* and map it to a fixed "no" instance of *Unique Satisfiability*, say the instance composed of one clause over two variables, $\{x_1, x_2\}$. Otherwise, set up $E' = E \cup \{\alpha, \overline{\alpha}\}$, where α does not appear in E. Then, if E has a satisfying truth assignment, E' has two, since the value of α is indifferent.

Otherwise, consider the new expression

$$(x_1 \Rightarrow E') \wedge (x_2 \Rightarrow E') \wedge \ldots \wedge (x_n \Rightarrow E') \wedge (\alpha \Rightarrow E')$$

Suppose that E cannot be satisfied; then this new expression can only be satisfied by setting every x_i and α to false (since then the implications are all of the type "false implies false," which is true). Thus the satisfying truth assignment for the new instance is unique. If E can be satisfied, then there are several ways in which we can satisfy the new expression: we can still assign "false" to all variables, but we can also assign a satisfying truth assignment for E—in which case every implication is either "false implies true" or "true implies true," both of which are true. Thus we have the desired result: the new expression is uniquely satisfiable if and only if the original is unsatisfiable. Note that, interestingly, the new expression is *always* satisfiable: the tricky part is the uniqueness.

The new expression can be constructed in polynomial time: each term in it is of the form $x \Rightarrow E$, or $\overline{x} \vee E$; the logical OR distributes over the clauses simply by adding the literal $\overline{x}$ inside each clause of E. Thus we effectively get $n + 1$ copies of the collection E, where each clause has been augmented by one literal. This is easily carried out in polynomial (at most quadratic) time.

Thus we have a many-one polynomial-time reduction from the coNP-complete problem *Unsatisfiability* to *Unique Satisfiability*. If the latter is in NP, it follows immediately from (the converse of) Exercise 7.9 that we must have NP = coNP.

Exercise 7.46

That this problem is in D^p is easy to see. The problem "is an instance of *SAT* satisfiable with any of its clauses removed?" is in NP, since a certificate need only list, for each clause of the instance, the truth assignment that satisfies the instance with that clause removed. We only need to intersect that problem with the problem "is an instance of *SAT* unsatisfiable?", a problem which is of course our most basic coNP-complete problem.

We begin with a polynomial-time many-one reduction from *3UNSAT* to our problem. Let f be the formula in the *3UNSAT* problem with n variables and k clauses. Consider the set S of all possible truth assignments to the n variables of f; each clause "covers" some part of that set through the truth assignments in which all three of its literals are set to false. f is satisfiable if the union of the regions covered by its clauses does not equal S; it is minimally unsatisfiable if the union does equal S, but each clause covers a part not covered by any other clause. We create a new formula, g, in $n+k$ variables, using all variables of f, call them x_i, plus one new variable for each clause of f, call these new variables y_j. We shall set up, for each clause in f, a clause in g which acts like the clause of f when all y_js are set to false. Specifically, if $c_j = \{\tilde{x}, \tilde{y}, \tilde{z}\}$ is a clause of f, we place a clause in g which is simply $\{\tilde{x}, \tilde{y}, \tilde{z}, y_1, \ldots, y_{j-1}, y_{j+1}, \ldots, y_k\}$. Call these clauses of type (a). Each such new clause covers some unique territory, since it can be made false while setting variable y_j to true. Now we need only cover the rest of the uncovered territory in such a manner that each new clause has its own exclusive "patch." We begin by covering the rest of the region where y_j is true and all other new variables are false: for each clause $c_j = \{\tilde{x}, \tilde{y}, \tilde{z}\}$, we place three new clauses $\{\overline{\tilde{x}}, y_1, \ldots, y_{j-1}, \overline{y_j}, y_{j+1}, \ldots, y_k\}$, $\{\overline{\tilde{y}}, y_1, \ldots, y_{j-1}, \overline{y_j}, y_{j+1}, \ldots, y_k\}$, and $\{\overline{\tilde{z}}, y_1, \ldots, y_{j-1}, \overline{y_j}, y_{j+1}, \ldots, y_k\}$. Call these clauses of type (b). Finally, we cover regions where two or more new variables are set to true: for every pair $1 \leq l < m \leq k$, we add a clause $\{\overline{y_l}, \overline{y_m}\}$. Call these clauses of type (c). (Note that any truth assignment that sets at most one new variable to true satisfies all clauses of this type.)

Note that, if f is satisfiable, so is g (simply set all new variables to false, which reduces g to f). We claim that, if f is unsatisfiable, g is minimally so. First note that, if f is unsatisfiable, so is g: we can set at most one of the new variables to true, thereby satisfying all but one of the clauses of type (a) and all but three of the clauses of type (b); but these four clauses now read $\{\tilde{x}, \tilde{y}, \tilde{z}\}$, $\{\overline{\tilde{x}}\}$, $\{\overline{\tilde{y}}\}$, and $\{\overline{\tilde{z}}\}$, which is a contradiction. Let us now verify minimality. Suppose we remove a single clause of type (c), say $\{\overline{y_{i_1}}, \overline{y_{i_1}}\}$; we can then set both of these variables to true, automatically satisfying all clauses of type (a) and all clauses of type (b). Suppose we remove a single clause of type (b), say clause $\{\overline{\tilde{y}}, y_1, \ldots, y_{j-1}, \overline{y_j}, y_{j+1}, \ldots, y_k\}$. We then set variable y_j to true and all other new variables to false, thereby satisfying all clauses of type (c), all but one clause of type (a)—the jth clause—and all but two clauses of type (b)—the two remaining clauses derived from the original jth clause. These three unsatisfied clauses now reduce to $\{\tilde{x}, \tilde{y}, \tilde{z}\}$, $\{\overline{\tilde{x}}\}$, and $\{\overline{\tilde{z}}\}$, which we can easily satisfy by setting $\tilde{x} = \text{false}$, $\tilde{y} = \text{true}$, and $\tilde{z} = \text{false}$. Finally, suppose we remove a single clause of type (a), say the jth such; we then set y_j to true and all other new variables to false. This satisfies all remaining clauses of type (a) (they all include y_j), all clauses of type (c), and all but three clauses of type (b), which reduce to $\{\overline{\tilde{x}}\}$, $\{\overline{\tilde{y}}\}$, and $\{\overline{\tilde{z}}\}$, which we can satisfy by setting $\tilde{x} = \text{false}$, $\tilde{y} = \text{false}$, and $\tilde{z} = \text{false}$.

We now augment this reduction to make it a valid many-one, polynomial-time reduction from *3SAT* to *Minimal Unsatisfiability* and thus provide the two pieces needed in reducing *SAT/UNSAT* to *Minimal Unsatisfiability* and proving the latter to be D^p-complete. The augmentation is really quite simple—we have done all the hard work above. We simply add a single clause $\{y_1, y_2, \ldots, y_k\}$ that includes all our new variables. Now our new g' is unsatisfiable: we must set at least one new variable to true (because of our new clause) and may set at most one new variable to true (because of clauses of type (c)); setting one new variable to true will satisfy all but one of the clauses of type (a) and all but three of the clauses of type (b), but these four clauses then read $\{\tilde{x}, \tilde{y}, \tilde{z}\}$, $\{\overline{\tilde{x}}\}$, $\{\overline{\tilde{y}}\}$, and $\{\overline{\tilde{z}}\}$, which is a contradiction. If f is not satisfiable, then g' is not minimally unsatisfiable: we can remove the new clause, thereby reducing g' to g, which is still unsatisfiable as shown earlier. On the other hand, if f is satisfiable, then g' is minimally unsatisfiable. Certainly, removing the new clause turns g' into g, which is satisfiable when f is, as noted earlier. Removing a clause of any other type was covered earlier; note that in all three cases we set at least one of the additional variables (the y_is) to true, which also satisfies our new clause.

Finally, we need to combine our two reductions into a single reduction; this will give us the desired many-one reduction from the D^p-complete problem *SAT-UNSAT* to *Minimal Unsatisfiability*. Given the two instances of *Minimal Unsatisfiability* produced by our two reductions, we combine them into a single instance of *Minimal Unsatisfiability* so as to preserve minimality as follows: if the clauses of of the first instances are $c_1, \ldots, c_m$ and those of the second instance are $d_1, \ldots, d_n$—on an entirely disjoint set of variables, then the clauses of our single new instances are all clauses of the form $c_i + d_j$, for all pairs $1 \leq i \leq m$, $1 \leq j \leq n$. Thus removing a single clause from the new instance is equivalent to removing one clause from each of our two intermediate instances and the new instance remains unsatisfiable, since we are using every possible pair.

Exercise 7.47

This is a very hard problem; as with most of the double-starred problems in this text, the intent is not to assign these problems for the students to solve, but perhaps to assign them as reading material. The pertinent reference here is the 1984 paper of C.H. Papadimitriou and M. Yannakakis entitled "The complexity of facets and some facets of complexity," which appeared in final form in *J. Comput. Syst. Sci.* **28** (1988), pp. 244–259.

Exercise 7.48

This is a very hard problem; as with most of the double-starred problems in this text, the intent is not to assign these problems for the students to solve, but perhaps to assign them as reading material. The pertinent reference here is the 1984 paper of C.H. Papadimitriou entitled "On the complexity of unique solutions," which appeared in *J. ACM* **31** (1984), pp. 392–400.

Exercise 7.49

We begin by noting that *Boolean Expression Inequivalence* is in NP (guess a truth assignment that causes the two expressions to evaluate to different values) and thus that *Boolean Expression Equivalence* is in coNP. Now consider the negated version of our problem: *Boolean Expression Nonminimality*. This problem is in Σ_2^p: we guess a shorter equivalent Boolean expression and use one call to the oracle for *Boolean Expression Equivalence* to verify that the guessed expression is indeed equivalent.o Thus our given problem is in coΣ_2^p, i.e., in Π_2^p.

Exercise 7.50

This is a very hard problem; as with most of the double-starred problems in this text, the intent is not to assign these problems for the students to solve, but perhaps to assign them as reading material. The pertinent reference here is the 1981 paper of E.W. Leggett, Jr., and D.J Moore, entitled "Optimization problems and the polynomial hierarchy," which appeared in *Theoret. Comput. Sci.* **15** (1981), pp. 279–289.

Exercise 7.51

Exercise 7.52

The answer is deceptively simple. Indeed, the problem as stated appears tricky only because of the deliberately vague language used. The key to the answer is to realize that classes and machines are not the same thing—we can typically attach a variety of different machines to the same complexity class. Indeed, we have done so repeatedly (in particular in Chapter 4) while convincing ourselves that the choice of machine ultimately did not matter.

The problems described in Exercises 6.25 and 7.51 are not problems about the classes NP or NP ∩ coNP; they are problems about classes of machines—more precisely, classes of pairs formed by a machine and a string. We could have P = NP = NP ∩ coNP and still have a completely different notion of a machine for P vs. a machine for NP ∩ coNP: even though both sets of machines would then have equivalent power, they would have different structure and descriptions—indeed, this is why we called one type semantic and the other syntactic: machines for syntactic classes can be described naturally with simple rules for their transitions (and can accept simply by entering an accepting state), while machines for semantic classes rely on complex descriptions of acceptance conditions.

Thus the problem is NP-complete for NP because we have a natural description of the NP machines, one that can be used in a simulation; but the problem is undecidable for NP ∩ coNP, even though we do not know whether the two classes are distinct, because the description of a machine for that class does not allow its direct simulation.

Exercise 7.53

This result is often attributed to Kirchoff (who formalized the famous laws bearing his name that form the basis for circuit analysis). We proceed in two steps: first we show that we can compute the number of directed spanning trees of a directed Eulerian graph, then we use that result to compute the number of directed Eulerian circuits. The number of Eulerian circuits of an undirected graph is just half of the number of Eulerian circuits of the corresponding directed graph (where each edges has been replaced by a pair of arcs).

Define the Kirchoff matrix of a digraph $G = (V, A)$ to be a $|V| \times |V|$ matrix K where the (i, j)th entry is

$$K(i, j) = \begin{cases} \text{in-degree of vertex } i & i = j \\ 0 & i \neq j \text{ and } (i, j) \notin A \\ -1 & i \neq j \text{ and } (i, j) \in A \end{cases}$$

Note that the sum of the entries in a column of K is zero. We claim that, if every vertex of the graph has in-degree at most 1, then the graph contains a directed spanning rooted at vertex i if and only if the determinant of K_i (the matrix obtained by removing from K its ith row and column) equals 1. The "only if" part is simple. Assume that the graph has a spanning tree rooted at vertex i. Then the spanning tree accounts for all arcs of the graph, except possibly an arc into the root, since all vertices have in-degree at most 1. Thus the diagonal entry for vertex i is either 0 or 1 and all other diagonal entries equal 1; moreover, if we reorder the matrix according to a breadth-first search traversal of the spanning tree, the the (i, j)th entry must be 0 for $i > j$ and $i > 1$, so that the matrix K_i reordered in this fashion is upper-triangular with all diagonal elements equal to 1 and thus has determinant equal to 1. Now assume that no such spanning tree exists. If any vertex other than the ith has zero in-degree, its corresponding diagonal element is

0 and thus the determinant will be 0, as desired. Assume then that all other vertices have in-degree 1 and thus that all diagonal elements other than the ith equal 1. Because every vertex (other than the ith) thus has a unique predecessor, we can trace back from an arbitrary vertex to its predecessor, its predecessor's predecessor, etc. Such tracing either reaches the i vertex or it does not. If it does, it is a branch of a partial spanning tree of G rooted at the ith vertex; thus there must be at least one case that it does not, but that can only happen if there is a cycle (that avoids the ith vertex). The sum of the columns corresponding to the vertices in that cycle is zero—that is, these columns are not linearly independent and thus the determinant is zero, as desired.

Now we can compute the number of distinct directed spanning trees of a digraph. The number of distinct spanning trees rooted at the ith vertex of an arbitrary digraph is simply the determinant of K_i. (This result is Kirchoff's main result, although he proved it in the case of directed multigraphs.) Since we know that determinants can be computed in time proportional to the cube of the dimension of the matrix, we can compute the total number of distinct directed spanning trees of a digraph on n vertices in $O(n^4)$ time.

Now we need to relate spanning trees and Eulerian circuits. The relationship was hinted at in the proof of Euler's theorem (Exercise 2.17). If we have a spanning tree rooted at some vertex v, we can build an Eulerian circuit as follows: start with any arc incident upon u and trace it back to its tail vertex; then continue tracing back, subject to not using the same arc twice and not using an arc from the spanning tree as long as another arc is available; stop when no unused arc remains. Since every vertex has equal in- and out-degrees, this process must terminate back at vertex v; we need only verify that it has used every arc of the digraph in the process. This is a simple proof by contradiction: if there was an unused arc (i, j), then there would have to be another unused arc (k, i), since the in-degree of i equals its out-degree; moreover that arc must be a tree arc, since such arcs are the last one used. Now we can repeat the construction starting from unused arc (k, i); eventually, we obtain an arc (v, w) that must be unused and thus an arc (t, v) that must be unused. But termination occurs at v when there is no unused arc incident upon v, a contradiction. Thus the circuit constructed is Eulerian.

From this procedure, we can count the number of Eulerian circuits in a digraph. For each directed spanning tree, we can construct one or more Eulerian circuits in this fashion. The number of choices has to do with the in-degree of each vertex: a vertex of in-degree k gives us $(k-1)!$ choices (not $k!$, because of our requirement that the tree arc be used last): $k-1$ ways out of it the first time, then $k-2$ the second time, and so on. Thus the number of choices for any fixed spanning tree is $\Pi_{v \in V}(\text{in-degree}(v) - 1)!$; note (the crucial observation) that this value is independent of the particular choice of spanning tree! Thus the total number of Eulerian circuits of a digraph is simply

$$ST(G) \cdot \Pi_{v \in V}(\text{in-degree}(v) - 1)!$$

where $ST(G)$ is the total number of directed spanning trees of G.

We have seen that we can compute $T(G)$ in $O(n^4)$ time; ignoring issues of number size, we can naively compute the product in time proportional to n^2; thus the complete computation takes $O(n^4)$ time and is polynomial.

Exercise 7.54

Exercise 7.55

This problem offers a small glimpse into a large area of structure theory: the relationship between the complexity of problems and their density (or sparseness), that is, the number of distinct instances of the same size in the set. By its nature, a tally set is at one extreme of the density spectrum: it has at most one string of each length, since each element of a tally set is entirely determined by its length. Thus the exercise asks to show that very sparse sets cannot be NP-complete (unless, of course, P equals NP). We prove that, if a tally set is NP-complete, then we must have P = NP. The proof is somewhat tricky, although its basic engine is the self-reducibility of an NP-complete problem, here *SAT*: we shall consider partial truth assignments and use the induced versions of *SAT*, storing some partial results in order to run the entire algorithm in polynomial time.

Our algorithm will be the following self-reduction, where t_i denotes a partial truth assignment wherein the first i variables are set (the ordering of the variables is fixed):

```
if i=n
   then if the induced instance is empty
           then return "yes", else return "no"
look up in the table at index h(t_i)
if a value is found for h(t_i), return it
otherwise assign "true" to the (i+1)st variable and recurse;
if the returned valued was "no"
   then assign "false" to the (i+1)st variable and recurse;
if either recursive call returned "yes"
   then return "yes", else return "no"
in either case, store the returned value under h(t_i)
```

The function h acts much as a hashing function; its goal is to avoid unnecessary evaluation of subfunctions that have already been evaluated. Thus, whenever $h(t_i) = h(t'_j)$, it must be the case that the two induced instances are either both satisfiable or both unsatisifiable. It is also desirable that h be a good hash function in that its range is (very) small compared to its domain. This is where the reduction to a tally set comes in: the mapping used in the reduction will provide just the h we need.

Let f be a mapping from *SAT* to our tally set T. We can assume that f maps any instance of *SAT* to a string in $\{a\}^*$ (otherwise, we can remap any string that uses other symbols into a fixed string in $\{a\}^* - T$). Because f runs in polynomial time, the strings produced by f are at most polynomial in length; for an original instance with n variables, any string produced has length $O(n^{O(1)})$, which is very small compared to the number of possible instances of *SAT* on n variables. Finally, because f is a valid reduction, if must be the case that, whenever it maps two instances of *SAT* to the same string, then the two instances are both "yes" instances or are both "no" instances. Thus f meets the requirements placed upon h in the self-reduction procedure described above.

It remains to show that the self-reduction procedure described above, with the mapping f used as hash function, runs in polynomial time. Each recursive call takes polynomial time: its most expensive component is the computation of $h(t_i)$, the polynomial-time mapping from *SAT* to T. Thus we need to bound the number of recursive calls. Given an instance with n variables, our hash table will store at most $p(n)$ values, for some polynomial p. Note that the recursive calls form a binary tree of calls of depth at most n; denote the number of nodes in this tree by $S(n)$. We want to show that $S(n)$ is $O(n^{O(1)})$. We do this rather indirectly by showing that we can pick a subset of internal nodes (not leaves) of the tree of size at least $S(n)/2n$ and such that no two nodes are on the same path in the tree (or, equivalently, such that any two nodes give rise to disjoint sets of possible truth assignments). We can construct such a set by first removing all leaves of the tree (thus leaving $S(n)/2$ nodes), then iteratively applying the following step: select any new leaf node in the tree, add it to the set, and remove it and all of its ancestors from consideration. Since we remove at most n nodes in each iteration and start with $S(n)/2$ nodes, it will take at least $S(n)/2n$ iterations to delete every node in the tree and thus our set will have at least $S(n)/2n$ elements. By construction, none of these elements is a leaf of the original tree; also by construction, no prefix of a selected partial truth assignment can be in the set. We claim that each element of the set must be mapped to a different table location by our function h. Assume that such was not the case and let t_i and t_j be two such elements. Since t_i and t_j are on different paths on the tree, one was generated after the recursive call to the other had completed and returned; but then, since the two have the same h value, the second one would have succeeded in the table look-up and thus would have been a leaf in the original tree—a contradiction. Thus our table has size at least $S(n)/2n$; but we already noted that it has size at most $p(n)$; it follows that we have $S(N)/2n \leq p(n)$ or $S(n) \in O(n^{O(1)})$, as desired.

Exercise 7.56

This exercise does not really propose a problem—instead, it solves one. Its goal is to incite the reader to reflect on the construction described. Thus there is no additional proof to give here, but perhaps we can make a few comments on the construction.

It is perhaps surprising that the construction is so simple given how long we had to wait for a proof of NL = coNL. Note however that the construction is actually fairly involved: it uses recursion to build the set and nondeterminism in a somewhat devious way to count the number of solutions. It also checks its guesses through a clever mechanism. Finally, the proof of the theorem depends on yet one more realization, namely that counting can be parlayed into a negative answer by checking the number of distinct IDs. Thus, while the final result is very clear and elegant, there are good reasons for its having been rather slow in coming.

It is also immediately clear from the construction that the space bound is absolutely crucial; a time bound would not help here—there is no chance that we can somehow extend this technique to show something like NP = coNP, alas.

Solution Manual
Chapter 7, Additional Problems

Additional Problem 1

The problem is clearly in NP: we need only guess a truth assignment and count the number of clauses it satisfies in NAE fashion, then compare the result with the bound. We prove the problem NP-complete by reducing *NAE3SAT* to it. Given an instance of *NAE3SAT* with variables $V = \{x_1, \ldots, x_n\}$ and clauses $C = \{\{\hat{x}_i, \hat{y}_i, \hat{z}_i\}, 1 \le i \le r\}$ (where $\hat{x}$ is used to denote one of x or $\overline{x}$), we construct an instance of *NAEMax2SAT* with the same variables and where each clause, $\{\hat{x}, \hat{y}, \hat{z}\}$, is replaced by 6 clauses as follows: $\{\hat{x}, \hat{y}\}$ $\{\hat{x}, \hat{z}\}$ $\{\hat{y}, \hat{z}\}$
Observe that, whenever the original clause has a satisfying assignment (1 or 2 literals set to true), then exactly 2 of the 3 constructed clauses are satisfied in NAE fashion (i.e., have exactly one literal set to true), whereas none of the 3 clauses is satisfied in that fashion when the original clause has no satisfying assignment (all literals true or all literals false). Thus we should set K to $2m$, where m is the number of clauses in the *NAE3SAT* instance. The only problem left is repetitions: our construction can easily produce multiple copies of the same 2-clause (not within a single group of 3, but between groups). We can handle this minor problem in a very simple way: whenever variable x is involved in copies, create variables x_i (one for each copy) use the x_i in the copy rather than x; for each x_i thus created, add the clause $\{x, \overline{x}_i\}$, and increase the bound by 1. By increasing the bound by 1, we force the additional clause to be satisfied in NAE fashion (because there is no way to satisfy more than 2 out of the 3 clauses in each original group, no matter whether the variable used there is called x or x_i), i.e., we force x and x_i to have the same truth value, so that every appearance of x_i is equivalent to an appearance of x. The proof follows immediately.

A brute-force approach to the same problem simply creates m new variables for each original variable of *NAE3SAT*; given the original clause $c = \{\hat{x}, \hat{y}, \hat{z}\}$, we replace it by 3 clauses as follows: $\{\hat{x}_c, \hat{y}_c\}$ $\{\hat{x}_c, \hat{z}_c\}$ $\{\hat{y}_c, \hat{z}_c\}$
We then add clauses of the type $\{x, \overline{x}_i\}$, $1 \le i \le m$, and increase the bound by m for each original variable, or by mn overall. This transformation creates mn variables and $m(n+3)$ clauses, and sets the bound to $K = m(n+2)$; it is clearly doable in polynomial time, in spite of the typically large number of useless variables (a lot of them never appear in any of our groups of 3) created.

Somewhat harder would be to reduce something like *Max2SAT* to our problem: this would entail introducing symmetry and thus much the same trick as used in reducing *3SAT* to *NAE3SAT* in the notes.

Additional Problem 2

That the problem is in NP is immediate: *Satisfiability* is in NP and we need only make one additional check to verify that each clause either is a Horn clause or has two literals. We prove that the problem is NP-complete by reducing *Satisfiability* to it.

If a clause of *Satisfiability* has one or two literals, we pass it unchanged; *ditto* if it is a Horn clause. We provide an "alter ego" for (the complement of) each of our variables as follows: for variable x, we create the new variable X and add the two clauses $\{\overline{x}, \overline{X}\}$ and $\{x, X\}$. Together, these two clauses enforce $X \equiv \overline{x}$. Now, given some clause of *Satisfiability* with $m > 2$ literals and $k > 1$ of them complemented, we replace every complemented literal in the clause by its alter ego and place the result in the new collection of clauses.

The resulting collection of clauses is logically equivalent to the original collection and all of the new clauses are Horn clauses or have length two. Given an instance of *Satisfiability* with n variables and m clauses, we produce a new instance with $2n$ variables and $m + 2n$ clauses, in a process that obviously runs in polynomial time.

Additional Problem 3

Membership in NP is clear. We prove this problem NP-complete by transforming *3SAT* to it.

Given an instance of *3SAT* with variables $V = \{x_1, \ldots, x_n\}$ and clauses $C = \{\{\hat{x}_i, \hat{y}_i, \hat{z}_i\}, 1 \leq i \leq r\}$ (where $\hat{x}$ is used to denote one of x or $\overline{x}$), we construct an instance S, $C \subseteq 2^S$ of *SS* by letting $S = \{x, \overline{x} | x \in V\} \cup \{u\}$ and $C = \{\{x, \overline{x}\} | x \in V\} \cup \{\{\hat{x}, \hat{y}, \hat{z}, u\} | \{\hat{x} + \hat{y} + \hat{z}\}$ is a clause$\}$. The extra variable u is to allow the splitting of sets that correspond to clauses for which an assignment sets all literals true; it also determines which side of the solution (partition) to SS corresponds to the assignment true in *3SAT*. Our transformation is obviously polynomial-time.

Given a solution to *3SAT*, we get a solution to *SS* by putting in one set all literals assigned false plus the extra variable, and on the other side all the literals assigned true. Since each clause is true, at least one literal in it is assigned true and thus goes in the 'true' partition side; the extra variable u always goes on the other side; this guarantees that all subsets derived from the clauses get split. The other sets (the truth setting sets) are obviously split by any truth assignment. Conversely, a solution to *SS* must split those last sets, thereby guaranteeing that no variable can be on the same side as its complement. We shall assign false to these literals that are on the same side as u, true to those on the other side. Since all clause-derived subsets are split, at least one literal in them is not on u's side and thus is assigned true, thereby satisfying the clause. Hence we are done.

From *NAE3SAT*, there is no need for the extra u and a much simpler proof results: just make the literals the set elements (thus $2n$ set elements for n variables), make each clause into a 3-set, and a a 2-set $\{x, \overline{x}\}$ for each variable x.

From *Positive NAE3SAT*, we do not even need a transformation: we can restrict *SS* to *Positive NAE3SAT* by allowing only instances where the subsets have three elements each. Splitting a 3-set is exactly equivalent to satisfying a 3-clause in NAE manner.

Additional Problem 4

Membership in NP is simple, but must be checked negatively, since there could be an exponential number of cycles; thus we remove the guessed set of vertices from the set of vertices and verify (by breadth- or depth-first search) that the induced graph is acyclic. We prove this problem NP-complete by transforming *VC* to it.

Given an instance $G = (V, E)$ and K of *VC*, we construct an instance $G' = (V', A)$ and K' of *FVS* by letting $V' = V$, $K' = K$, and $A = \{(u, v), (v, u) \mid \{u, v\} \in E\}$. Thus the resulting directed graph is identical to the original undirected one, except that each edge of the old graph is replaced by 2 arcs in the new graph, one in each direction. Notice that this establishes a cycle, which is basic to the proof. This construction can obviously be done in polynomial time.

The proof is now easy. By definition of a vertex cover, each edge of E is covered, hence each arc of A is covered; since any cycle must use at least one arc, each cycle passes through one vertex in of the cover. Conversely, since each edge of E gives rise to a cycle in A, a solution to *FVS* must include one of the two vertices in that cycle and thus cover each edge.

Additional Problem 5

We reduce *FVS* (see Additional Problem 4) to this problem. The simplest way is to replace each vertex of *FVS* by a pair of vertices with an arc between them, with the vertex at the tail being used for all in-arcs and the vertex at the head for all out-arcs. This clearly does not add new cycles; removing a vertex in *FVS* has exactly the same effect as removing the corresponding arc in *FAS*.

Additional Problem 6

This is *exactly* the dominating set problem (Exercise 7.21), where the vertices are labeled 1 for the dominating set.

Additional Problem 7

The problem is clearly in NP. We prove it NP-complete by transforming *HP between specified points*. First we show a somewhat more general version to be NP-complete, where edge weights can have any positive value as well as -1. In that case, we simply give a weight of $|V|-3$ to all arcs out of s, and a weight of -1 to all other arcs. For an s–t path to have negative weight, it must have at least $|V|-2$ arcs of weight -1 to compensate for the arc of weight $|V|-3$, and thus a total of $|V|-1$ arcs, which makes it a directed Hamiltonian path between s and t. Now, with only weights of unit value allowed, we simply replace the arcs of weight $|V|-3$ by chains of $|V|-3$ arcs, each of weight 1. A simpler construction sets up a chain from s of just enough arcs to get the positive component, then has the branching. To allow even graphs of 3 or fewer vertices to be transformed, add matched arcs of weight +1 and -1, so that we need not worry about a negative number of arcs.

Additional Problem 8

The problem is clearly in NP. We prove it NP-complete by restricting it to *Partition*. Let the graph have 3 vertices and two edges, with the middle vertex being the designated vertex, and let the sum of all $w(f_i)$ be even. The fire trucks must be assigned either to the left-end vertex or to the right-end vertex; the response time will simply be the larger of the sums of the $w(f_i)$ on each side, since the trucks will have to move strictly sequentially on each side.

Additional Problem 9

We prove this problem NP-complete with several different constructions, by transformation from *3-Dimensional Matching* or *X3C*.

1. Our first transformation (from *3DM*) creates a piece for each element, but with one connecting point for each occurrence of the element within a triple. Specifically, an element that appears in only one triple is transformed into a single vertex with that label; for each additional occurrence within a triple, we add three more vertices, one of which is labelled by the element name also (an attaching point); the other two are connected by an edge and, in addition,

one of the two is connected to all attaching points for the element (meaning all such points eventually defined, not just those existing so far in the construction). (Actually, it is sufficient to connect in series: attachment–edge–attachment–edge–attachment, as long as needed.) Edges connect the pieces in $w - x - y$ pattern, with the proviso that each attaching point is used exactly once (for pieces from W or Y) or twice (for pieces from X) for such edges. This transformation is clearly polynomial-time (quadratic). (Note that the same transformation works from *X3C*, as it does not rely on ordering: the 3 sets can be connected as stated or in a triangle.)

Since one of the two dummy vertices is connected to just one other vertex (the second dummy, that is connected to all attaching points for that piece), both dummy vertices must be included in an element of the partition, together with one of the attaching points (the only candidates for the third vertex). Thus only one attaching point of each piece is available for forming an element of the partition with vertices from other pieces; since the only edges left are the $w - x - y$ edges, such elements of the partition must be composed of a vertex from W, one from X, and one from Y. Thus we have a solution to *3DM* if and only if we have one from *PPL2*.

2. A second, simpler construction (also from *3DM*) sets up an edge for each element (with one endpoint designated for connections) and a 3-vertex, 2-edge piece for each triple, with the obvious connections (3 edges) between the triples' pieces and the elements' vertices. Since the other endpoint of each element's edge is of degree 1, any partition can use only one edge out of each element's connection point. The triple can be partitioned in only two ways, which finishes our proof.
3. Using one vertex per element of the set, and adding one connection piece for each triple, we can devise a number of transformations. One just uses an "E" (with 6 vertices and 5 edges, plus 3 connecting edges) and ends up looking quite similar to the previous construction; another uses triangles.
4. From *X3C*, a variety of transformations is also possible. Easiest is to set up one triangle for each 3-set, one segment for each element, and use one segment endpoint as connecting point, joined to all relevant triangle vertices by edges; in the solution, the chosen 3-sets correspond to triangles completely broken apart, while 3-sets discarded correspond to triangles forming a *PPL2* of their own. The E-shaped piece also works (with 9 vertices total, having vertices of degree 1 hanging from each branch of the E). A subtler trick, similar to the 1st transformation from *3DM*, uses 1 vertex for each element,

and add extra pieces for extra connection points as needed; each addition consists of 3 vertices, say a_i, b_i, and v_i, with a_i connected to b_i, v_i, and the original v, and with v_i serving as new connection point.

Additional Problem 10

Membership in NP is obvious, since multiplication can be done in time polynomial in the length of the numbers to be multiplied.

We prove the problem NP-complete by transforming *Positive NAE3SAT* to it. The idea is to use prime encoding for truth values, using one prime for each variable. Thus, if the variables of *Positive NAE3SAT* are $x_1, \ldots, x_n$, we use prime numbers $p_1, p_2, \ldots, p_n$, where p_i corresponds to $x_i = \text{true}$. Since $p_n \in O(n^2)$ and since we can test a number k for primality in $O(\sqrt{k} \log k)$ time, we can obtain these n primes in polynomial time. Denote the product of all n primes by P; i.e., $P = \prod_{i=1}^{n} p_i$. Let the clauses of *Positive NAE3SAT* be $c_1, \ldots, c_m$; the $\{a_i\}$ sequence consists of $3m$ numbers and the $\{b_j\}$ sequence of $4m - 1$ numbers. For clause $c_k = \{x_{k_1}, x_{k_2}, x_{k_3}\}$, let product P_k be defined by $P_k = P/(p_{k_1} p_{k_2} p_{k_3})$. For each clause c_k, we place numbers $p_{k_1} P_k$, $p_{k_2} P_k$, and $p_{k_3} P_k$ in the $\{b_j\}$ sequence and numbers $p_{k_1} p_{k_2} P_k$, $p_{k_1} p_{k_3} P_k$, and $p_{k_2} p_{k_3} P_k$ in the $\{a_i\}$ sequence. Verify that, if c has none of the factors p_{k_1}, p_{k_2}, or p_{k_3}, or has all three of them, then it divides the same proportion on both sides (all 3 in absence, none in presence of the 3 primes), whereas it divides one more a_i value than b_j value in cases where it has one or two of the factors. Finally, we place $m - 1$ copies of P in the b_j sequence. Any certificate c always divides P. Hence, for a certificate to divide more numbers in the a_i series than in the b_j series, it must make up that difference of $m - 1$, plus one: hence it must gain m over the first $3m$ elements. But we have observed that it can gain at most one per clause, and that only when the clause is satisfied in *NAE3SAT* mode; hence all clauses must be satisfied. (There are as many ways of solving this as there are people to write proofs; one can start from any of the *SAT* variations and derive a simple proof.)

Additional Problem 11

We prove this problem NP-complete (it is obviously in NP) by transforming *3SAT* to it. Our transformation is based on very simple graph pieces: while Grundy numberings appear fairly complex, we appeal only to the simplest results.

Specifically, note that a 2-cycle has only two possible Grundy numberings, each of which labels one vertex with a 0 and the other with a 1; thus 2-cycles can serve as truth-setting components. Now, note that 3-cycles (or

k-cycles, $k > 2$, for that matter) do not have a Grundy numbering; however, a 3-cycle, one vertex of which is connected (by an arc issuing from the vertex) to a fourth vertex, has a Grundy numbering if and only if the fourth vertex is labelled with a 0. This gives us the solution: each clause will be represented by a 3-cycle, one vertex of which will be connected to the three corresponding "literals" in the truth-setting components; a vertex labelled 0 in the truth-setting component corresponds to a true literal. The rest follows easily.

From *NAE3SAT*, a similar transformation works, but now we connect each vertex in the 3-cycle to the corresponding literal vertex. A 3-cycle, each vertex of which is connected to one other vertex labelled 0 or 1, can be labelled if and only if the 3 endpoints are not all labelled 0 and not all labelled 1.

Additional Problem 12

That *Simple Rural Postman* is in NP is somewhat non-trivial as a circuit could include the same edge many times and thus be very long. We need to show that a yes-instance always has a short certificate, that is, a short circuit—that the circuit is bounded in length by K does not help, since K could be exponential in the size of the input. (Although our proof produces a graph where any solution circuit is simple and although a similar proof can designed which produces a graph where any solution circuit is in fact simple, this only shows NP-completeness of the special cases and NP-hardness of the general case.) Assume that the graph is biconnected (if not, run a biconnected analysis (in polynomial time) and consider each component at a time); at worst, $E' = E$ and we must traverse each edge at least once. From the articulation point (or from an arbitrary point), we need to return to it at most as many times as there are edges to be visited in the component; hence each edge can never be visited more than $|E|$ times, putting an upper limit on the size of the circuit of $|E|^2$. This proves membership in NP.

To prove that it is NP-complete, we transform *Hamiltonian circuit* to it. Given the graph for *HC*, say $G = (V, E)$, we augment it as follows. For each vertex in $v \in V$, we add two new vertices, v' and v'', and associated edges, $\{v, v'\}$, $\{v', v''\}$, and $\{v'', v\}$. These new edges form the subset of designated edges; the maximum number of edges is set to $4|V|$. Such a transformation is clearly polynomial-time. (In effect, what we have done is to add an "ear" to each vertex; we now will require the postman tour to visit the ears by travelling along a Hamiltonian circuit at the base of the ears.)

Now, if *HC* has a solution, then a solution to *RP* can be formed by using the Hamiltonian circuit and, at each vertex, traversing the corresponding ear. The result is a non-simple cycle which traverses $|V| + 3 \cdot |V| = 4|V|$

edges. Conversely, a solution to *RP* must yield a solution to *HC*, as follows. Since the *RP* cycle has no more than $4|V|$, it cannot traverse more than $|V|$ edges in E (because it must traverse the $3|V|$ designated edges anyway); since it must visit each vertex in V (in order to traverse the associated designated edges) as part of the cycle, it traverses exactly $|V|$ edges in E; hence the *RP* cycle restricted to the edges of E is a Hamiltonian circuit for the original problem.

Additional Problem 13

That *Kernel* is in NP is clear. To show that it is NP-complete, we transform *3SAT* to it. As usual, the transformation involves a truth-setting component, a satisfaction-testing component, and appropriate connections between the two. For each variable in *3SAT*, we put 2 vertices in the graph, connected as a 2-cycle; those vertices correspond to the uncomplemented and the complemented literals for the variable. Those are the truth setting components: at most one vertex from a 2-cycle can be in a kernel and we shall make sure that at least one vertex must be selected by not having any arc terminate at a vertex from the 2-cycle. Placing a vertex in the kernel will correspond to setting its associated literal to true. For each clause in *3SAT*, we place 3 vertices in the graph, connected in a 3-cycle; the vertices correspond to the three literals of the clause. This is the satisfaction testing component; it is a sink, just like the 2-cycle is a source; it will also have at most one vertex in a kernel, yet one vertex alone does not cover the 3-cycle, so that there must be at least one arc coming from a kernel vertex to one of the other vertices of the cycle; this arc will "point" to the literal that makes the clause true. Finally, we add arcs from the literals in the truth-setting components to their matching part in the satisfaction-testing components. It should now be obvious that we have a solution to *3SAT* if and only if we have a kernel for the graph.

Additional Problem 14

Membership in NP is clear. We transform *Positive NAE3SAT* to our problem as follows. The number of partitions will be 2. For each clause in *NAE3SAT*, we place a triangle in our problem. For each variable, we set up a vertex in our problem. Finally, we connect the variable vertex to each of the triangle vertices corresponding to the occurrences of the variable in clauses by a connection piece which ensures that both ends of the connection must reside on the same side of any partition; that piece must also prevent any path from existing between its two ends on either side of the partition. Note that the connection piece that consists simply of a triangle, each vertex of which is connected to each of the two vertices (overall, a K_5 missing one edge), does not work, as it allows a path from one end to the other and thus may lead to cycles through two or more variable vertices and two or more clause triangles. A suitable piece builds upon this by connecting both endpoints to each end of a K_5 missing one edge (the edge between these two ends). The proof follows immediately.

From regular *NAE3SAT* we need an XOR piece rather than an equivalence piece; we can then use this XOR piece in a construction similar to our *MxC* construction. A suitable piece uses a K_4 in the middle, with the two end vertices connected to each other and each connected to the two vertices of the K_4 on its side. This piece forces the two endpoints on opposite sides, thereby also preventing any path through. (We can also get a piece from the failed triangle idea mentioned above: use two of these from one endpoint, leading to two new vertices; connect these two new vertices to the other endpoint; the other endpoint cannot be on the same side, or a cycle would exist through the equivalence components.)

An elegant construction from *G3C* simply adds a new triangle to the graph and connects every vertex of the original graph to each of the three vertices of the new triangle. Since the coloring induces no edge at all, we can also partition the triangle into 3 pieces and be done. In reverse, being by noting that the triangle must be cut in 3: otherwise, the one monochrome edge would close a cycle with any vertex of the graph that has the same "color". Thus the triangle provides one vertex in each of the 3 pieces; since each such vertex is connected to all vertices of the original graph, it follows that any edge in the original graph must broken lest a cycle occur.

Additional Problem 15

Membership in NP is clear (a solution can be verified edge by edge). We transform *NAE3SAT* to our problem, relying as usual on triangles, as follows. Each clause will be represented by a triangle. Since at most one edge per triangle can be part of the matching, each triangle must have at least one node labelled *null*; if we can also ensure that at least one edge from each triangle must be present in the matching, then the labelling of each triangle is fixed, with each vertex having a different label. Each variable will be represented by a simple path; if variable x_i occurs n_i times in all, its component includes $4n_i$ vertices, call them $x_{ij}, \overline{x}_{ij}, a_{ij}$, and b_{ij}, for $1 \le j \le n_i$, and $4n_i - 1$ edges, connecting x_{ij} to $\overline{x}_{ij}$, $\overline{x}_{ij}$ to a_{ij}, and a_{ij} to b_{ij}, as well as b_{ij} to $x_{i,\ j+1}$. The intent is to have the x_{ij} and b_{ij} vertices labelled by either *source* or *sink* ("true" and "false") and the $\overline{x}_{ij}$ and a_{ij} vertices labelled the other way, with edges $\{x_{ij}, \overline{x}_{ij}\}$ and $\{a_{ij}, b_{ij}\}$ in the matching. (We could also simply do without the a_{ij} and b_{ij} by connecting $\overline{x}_{ij}$ to $\overline{x}_{i,\ j+1}$, and flip every time.) Finally, we connect the triangle vertices to the matching connecting points; an uncomplemented literal gets connected to the appropriate x_{ij}; a complemented literal to the appropriate $\overline{x}_{ij}$. The matching size is set to $7m$, where m is the number of clauses. The construction produces $15m$ vertices and $17m$ edges and is easily carried out in polynomial time.

Given a solution to *NAE3SAT*, a solution to the transformed instance is easily constructed. The components corresponding to the variables account for $6m$ edges in the matching with a labelling as described above. In each triangle, one true literal is labelled "source," one false literal is labelled "sink," and the remaining literal is labelled "null." This contributes one matching edge for each triangle, for a total of $7m$. That the labeling conditions are respected is easily verified.

Conversely, note that the required number of edges in the matching forces $14m$ of the $15m$ vertices to be involved in the matching; in other words, $7m$ vertices must be labelled *sink*, $7m$ others must be labelled *source*, and the remaining m must be labelled *null*. A triangle and its connecting edges can only yield from one to three matching edges; the one matching edge can be a triangle edge, but in order to obtain two or more matching edges, the triangle edges cannot be included. Moreover, in order to obtain three matching edges, all triangle vertices must be labelled with the same label (either all are labelled *source* or all are labelled *sink*)—and thus the other endpoints of the matching edges are also labelled with the same label. Now, note that using a connecting edge in the matching reduces by one the number of matching edges that a variable's component can provide. Thus, using one connecting edge per triangle gains nothing (over using just one triangle edge) and loses one; using two connecting edges gains one and loses two; using three connecting edges gains two and loses three. Since we have provided separate attachment points for these edges, the losses, as well as the gains, are independent of each other. In sum, using *any* connecting edge always results in a loss of one; since the required matching size is just met by using triangle edges, it cannot be met if any connecting edge is used. Hence any solution must be composed of one edge per triangle, no connecting edge, and $6m$ edges from the variables' components. The proof follows easily.

A variation on this theme keeps the triangles for the clauses, but uses only one edge per variable, connecting the endpoints (uncomplemented and complemented literals) to the triangle vertices with a suitable graph component. The component ensures that the two vertices it connects cannot be labelled one *source* and the other *sink*. A suitable component connects the two vertices, say u and v, with an edge and adds 8 more vertices, say u_{11}, u_{12}, u_{21}, and u_{22}, with similar four for v, and 10 more edges, $\{u, u_{11}\}$,

$\{u, u_{21}\}$, $\{u_{11}, u12\}$,$\{u_{21}, u_{22}\}$, with similar versions for v, and $\{u_{11}, v_{11}\}$ and $\{u_{21}, v_{21}\}$. With u and v given identical labels, or with one of them labelled *null*, four edges of the components can be matched, by labelling the x_{-1} vertices *source* and the x_{-2} vertices *sink* (or vice-versa, depending on the non-*null* label given to u or v). With the u, v vertices such that $\{u, v\}$ is in the matching, at most three edges of the component can be matched. The bound is then set to $13m + n$ ($12m$ for the $3m$ components, m for the clause triangles, and n for the variable segments). The proof follows immediately. A simpler component just uses two extra vertices, say e_1 and e_2, with connections $\{u, v\}$, $\{u, e_1\}$, $\{e_1, v\}$, and $\{e_1, e_2\}$; this last edge can be matched if the vertices u and v are not matched.

A third style of proof keeps the clause triangles and the simple connections between the clauses and the variables, but replaces the variables single edge by a simple lozenge of four edges, with the two extreme points representing the two literals. Since using connecting edges at most yields $2n$ matches (for n variables) and since we now require $2n + m$ matches, compensation cannot occur through connecting edges. The proof follows immediately.

Finally, starting from *NAE Max2SAT*, we can devise a transformation where no *null* label is ever needed. For each variable use a lozenge as above; for each clause use a single edge; connect literal vertex in clause to literal vertex in lozenge through two copies of a lozenge with one tail (of one edge) at each end; set $K_{TR} \leftarrow K_{2SAT} + 2n + 8m$. The proof follows the lines above.

Additional Problem 16

Membership in NP is clear. We transform *Directed HC* to our problem. The transformation is almost simplistic; it relies on the fact that a graph that has a directed Hamiltonian circuit is a strongly connected graph (although the reverse may not be true). Given the graph $G = (V, A)$, our transformation produces the graph $G' = (V, \emptyset)$, the weight function $w\colon (v_i, v_j) \to 1$ if $(v_i, v_j) \in A$ and $w\colon (v_i, v_j) \to 2$ otherwise, and the bound $B = |V|$. The transformation can easily be carried out in polynomial time. Given a solution to *DHC*, we obtain a solution to our transformed instance by adding exactly those arcs used by the circuit. The total cost is B, since each such arc exists in A and thus has unit cost; the resulting graph is strongly connected since it has a directed Hamiltonian circuit. For the converse, observe that, in any strongly connected graph, each vertex must have degree at least 2; given the bound, each vertex will have degree 2 exactly, since at most $|V|$ arcs may be added. The only connected graph where every vertex has degree 2 is a circuit.

Additional Problem 17

Membership in NP is clear. NP-completeness is easily shown by restriction to *Partition*, by considering only instances where length and weight functions are equal, where the sum of all weights is even where all tasks have the same deadline, and where said deadline is equal to the bound K and equal to half the sum of all weights.

Additional Problem 18

Actually, this problem is just a generalization of *X3C*, so we can prove it to be NP-complete by direct restriction. A direct reduction from *3SAT* builds one schedule for each literal ($2n$ in all), with bit entries for presence in clause and correspondence to a variable, using a requirement of all 1s—plus padding as needed.

Additional Problem 29

The problem is clearly in NP. We prove it NP-complete by transforming *Positive 1in3SAT* to it. Let the *Positive 1in3SAT* instance have m clauses and n variables; we produce $n + 1$ vectors of m components each; one vector is the enforcer and all its entries are 1s; each remaining vector corresponds to a variable and has a component of 1 for each clause in which the variable appears and a component of 0 otherwise. Finally, we set the bound B to 4.

If the *Positive 1in3SAT* instance has a solution, then we obtain a solution to our transformed instance by placing the enforcer alone in C_3, the vectors corresponding to false variables in C_2, and the remaining vectors in C_1. Since each clause has 1 true and 2 false variables, C_1 contributes 1 and C_2 contributes 2 to the sum; C_3, of course, contributes 1 and we have the desired bound. For the converse, note that each partition piece contributes at least one and that, since the component sum is uniformly 4 across the whole vector set, the three pieces must contribute at least 4. Thus a solution contributes exactly 4, as 1+1+2. If the enforcer vector is by itself, then the other two pieces are the true variables (the piece that contributes 1) and the false (the piece that contributes 2); otherwise, any of the three pieces can be selected for the true literals (!).

Another, very concise and elegant transformation starts from *G3C*. Start by eliminating the vertices of degree zero. Now, for each vertex we set up a vector of $|E|$ components, with a 1 in each component corresponding to an edge incident upon the vertex. We set the bound to 3. Note that the bound can only be equalled, since each partition piece must contribute at least one. Thus each partition piece in a solution must contribute exactly one. This means that the two vectors corresponding to the two endpoints of an edge can never be in the same partition piece, thereby corresponding to a legal 3-coloring.

Additional Problem 20

Note that the problem is in P if all a_{ii} are at most 2.

Membership in NP is non-trivial, because the entries can be arbitrarily large and yet encoded succinctly, so that we cannot simply guess the sets in the collection. The key to membership in NP is also the key to part of the transformation: since at most $n(n-1)/2$ different intersections exist, it is sufficient to consider at most that many different elements; any larger intersections must arise from simple copying. Thus we need only establish "elements" x_{ij} and guess which of these are contained in each C_i and in how many copies. Verifying that the intersections have the proper size is then trivial.

We transform *G3C* to our problem as follows. The basic tool we have in our problem is setting certain entries to 0 to force disjointness and setting certain others to non-zero values to force overlap. Both are necessary (although they can be scaled up by adding the same quantity to all entries). We shall use one C_i, say C_0, to denote the set of available colors; since there are 3 colors, a_{00} will be set to 3. The other C_i will be used to denote the coloring of each vertex; thus there will be n of them, one per vertex. We force each to intersect with C_0 in exactly one element (the color); thus $a_{i0} = a_{0i} = 1$ for all $1 \leq i \leq n$. Whenever two vertices are connected by an edge, they cannot share a color; thus we set $a_{ij} = a_{ji} = 0$ whenever there is an edge in the original graph between vertices i and j. When two vertices are not connected, they could share a color; thus we set $a_{ij} = a_{ji} = 1$ whenever there is no edge between vertices i and j. However, setting a_{ij} to 1 *forces* overlap between the sets C_i and C_j, whereas the correct coloring may or may not assign the same color to vertices i and j: thus other means of intersection must be provided. This shows us how to assign values to the $a_i i$ for $i > 0$: such values must be large enough to allow the corresponding C_i to have a large number of elements for possible intersections. More specifically, we could have up to half the vertices in the graph colored in the same color, yet not connected to each other nor connected to some fixed vertex colored in another color. The corresponding sets must all intersect; those with the same color intersect by their color, but each must otherwise be disjoint from the others while at the same time intersecting with a common set. This indicates that a minimum value is $\lfloor n/2 \rfloor$; picking n will put all doubt to rest. (To reduce the value, one needs to start with a graph of restricted degree; we know that *GC* remains NP-complete for graphs of degree 3 and thus could use this special case for the transformation, leading to diagonal values of just 3.) The transformation is easily accomplished in polynomial time.

The construction ensures that a solution to our problem implies the existence of a solution to the coloring problem: each vertex gets assigned a single color chosen from a set of three and two vertices connected by an edge have different colors since their sets do not intersect. Conversely, given a legal coloring we can construct the sets C_i piece piece as follows. C_0 has three elements, call them r, g, and b (the colors). Each C_i includes exactly one of these three elements, as indicated by the coloring. Now remove from the matrix the $i0$ and $0i$ elements, subtract one from each ij entry whose corresponding intersection will include one of the color elements and consider the resulting matrix. It is an $n \times n$ matrix; its diagonal entries are all $n - 1$ and its off-diagonal entries are 0 or 1. We need $C_1, \ldots , C_n$, each of size $n - 1$, to provide the intersection pattern. Think of each as a vector; we use the jth entry of C_i to provide an intersection with C_j if necessary (by giving both the same jth element); otherwise, we assign all distinct elements. This provides a solution. Hence we have a solution to the original instance if and only if we have one for the transformed instance.

An alternative solution starts with *Positive 1in3SAT* and creates a matrix of size $m + n + 1$ from an instance with m clauses on n variables. The entries for the first $m + n$ rows and columns are the obvious ones: an entry a_{ij} where i and j denote two variables is set to 0 (the variables cannot share their own truth assignment); one that denotes two clauses is set to the number of literals the two clauses share; and one that denotes a variable and a clause is set to corresponding to presence or absence of a variable from a clause or to the intersection of two clauses. The diagonal elements for the variables are set to 2 (not 1), in order to allow the two truth values; the diagonal elements for the clauses are set to 3. The $m + n + 1$st row and column serve as enforcer; all of their elements are set to 1, except for the bottommost, rightmost element, which is set to n; this row and column correspond to the truth assignment.

The intent is to have as sets collections built from a total of $2n$ elements, namely the elements true(i) and false(i), for each $1 \leq i \leq n$. The set for variable i is {true(i), false(i)}; the set for a clause contains 3 elements, the true version of each variable in the clause; and the set for the enforcer is the truth assignment, i.e., one of each pair for each variable. The proof follows easily.

Additional Problem 21

Membership in NP is non-trivial, because we cannot verify a complete derivation (as the number of steps could well exceed a polynomial function of $\log_2 K$). The fact that the language must in fact be regular (any language generated by a context-free grammar on an alphabet of one symbol is easily seen as regular: the nesting property is irrelevant, since all symbols can be arbitrarily permuted) does not directly help either. However, note that the only point of interest is whether a string of the correct length can be generated, using productions chosen from a collection of constant size. Thus, if a derivation is very long, it can only reach that length by using the same productions over and over again; thus we can ask that the certificate be a runlength-encoded description of the sequence of productions used—grouping subsequences and adding to each group a repeating count. Such a certificate is now concise (because of elementary properties of context-free languages) and easily verified with some simple arithmetic.

Now we prove that this problem is NP-hard by transforming *Subset Sum* to it. Given an instance of subset sum with set $S = \{e_1, \ldots, e_n\}$ and size function $s\colon S \to \Re$, we construct an instance of our problem as follows. The idea is to set up a grammar where the start symbol will derive all "subsets" of S; to that effect, there will be one non-terminal, E_i, per element, e_i, of S and the start symbol will derive all strings of length $1, 2, \ldots, n$, composed of these non-terminals, where said non-terminals appear in increasing order of subscript. Each non-terminal, E_i, will then derive a string of *a*s of length $s(e_i)$; however, we cannot write down such productions directly, as they would involve lengths of order $s(e_i)$, which is an exponential function of the input size of $\log_2 s(e_i)$. Thus, we turn to binary representations and write productions that generate 2^j *a*s, for $0 \le j \le \lceil \log_2 smax \rceil$ (i.e., binary "digits"), where *smax* is the maximum size of any element of S; then each E_i will generate a string of nonterminals corresponding to the non-zero digits in its binary representation.

The complete transformation thus produces $2n + 1 + \lceil \log_2 smax \rceil$ non-terminals and the following $3n + 1 + \lceil \log_2 smax \rceil$ productions:

$S \to F_1 F_2 \ldots F_n$
$F_i \to E_i | \epsilon$, for all i, $1 \le i \le n$
$E_i \to$corresponding non-zero digits[1]
$< 0 > \to a$
$< 1 > \to < 0 >< 0 >$
$< 2 > \to < 1 >< 1 >$
$\ldots$
$< \lceil \log_2 smax \rceil > \to < \lceil \log_2 smax \rceil - 1 >< \lceil \log_2 smax \rceil - 1 >$

[1]For instance, if $s(e_i)$ is $21 = 2^4 + 2^2 + 2^0$, then $E_i \to < 4 >< 2 >< 0 >$.

Now each E_i derives exactly one string, composed of $s(e_i)$ as. The start symbol S derives all 2^n strings corresponding to the various subsets of the set S; thus the start symbol derives a terminal string of length K if and only if there is a subset of the set S, the sizes of whose elements sum up to K.

Additional Problem 22

We prove this problem NP-complete (it is clearly in NP) by transforming *HP* to it. Given an instance of *HP*, $G = (V, E)$, we produce an instance of *Consecutive Sets* where $\Sigma = E \cup V$, $\Sigma_i = \{\{v_i, v_j\} | \{v_i, v_j\} \in E\} \cup \{v_i\}$, for $1 \le i \le |V|$, and $K = 2 \cdot |E| + 1$. Such a construction is clearly doable in polynomial time.

Note that, for $i \neq j$, we have $\Sigma_i \not\subseteq \Sigma_j$; moreover, we have $\Sigma_i \cap \Sigma_j = \{\{v_i, v_j\}\}$ if and only if $\{v_i, v_j\} \in E$. Finally we note that $K = 2 \cdot |E| + 1$ is exactly $\sum_{i=1}^{|V|} |\Sigma_i| - (|V| - 1)$.

Now, if there is a Hamiltonian path in G, say $v_{\pi(1)}, \ldots, v_{\pi(|V|)}$ for some permutation π, we can construct a solution string w of length K as follows. The ith block will contain the symbols of $\Sigma_{\pi(i)}$; the last symbol written in block i is also the first symbol in block $i + 1$ and corresponds to the edge $\{v_{\pi(i)}, v_{\pi(i+1)}\}$. Thus the string has length $\sum_{i=1}^{|V|} |\Sigma_i| - (|V| - 1)$ (the last term to account for shared "edges") and hence has length exactly K. Conversely, if there is a string w of length K in which the elements of Σ_i, for $i = 1, \ldots, |V|$, appear consecutively, then there is a corresponding Hamiltonian path in G. Note that such a string has a block of length $|\Sigma_i|$ for each Σ_i, yet has overall length shorter by $|V| - 1$ than the sum of the blocks' lengths; thus any two consecutive blocks share one symbol. As observed above, this symbol can only correspond to an edge between the corresponding vertices. Hence the sequence of vertices in G which corresponds to the sequence of blocks in w is such that any two consecutive vertices are connected by an edge: it is a Hamiltonian path.

Additional Problem 23

Membership in NP is non-trivial. The number of s–t paths in a digraph of n vertices can be exponential in n, so we cannot verify path by path that a given collection of arcs has the desired property. Note, however, that the collection as a whole has certain properties that allow verification. First note that we can eliminate any vertices and arcs that are not present on any path from s to t in linear time, thereby cleaning up the input. Now, if we were to remove every arc of the selected collection from the cleaned-up digraph, then we should observe that the resulting digraph has at most one path from any vertex to any other vertex—something that we can easily check in linear time. This property is clearly sufficient, so that the problem is in NP.

To prove the problem NP-complete, we reduce *Vertex Cover* to it. We assume that instances of *VC* do not include isolated vertices (or, equivalently, we discard such vertices if they exist). Given an instance of *VC*, i.e., a graph, $G = (V, E)$, and a bound, K, we create the following instance of our problem. The set of vertices of the digraph will be $V \cup E \cup \{s, t\}$, i.e., there will be a vertex for each vertex of the original graph, a vertex for each edge, and additional vertices s and t. We connect s to every vertex in V by an arc and connect every vertex in E to t by an arc; for each edge $e = \{u, v\}$ in the original graph, we place arcs (u, e) and (v, e) in the new digraph; and we also add the arc (s, t). The bound for our new problem is set to $|E| + K$.

Note that every path from s to t in our digraph (except for the single arc (s, t) itself) has length 3 and is of the form $s \to v \to e \to t$, where v is an endpoint of edge e in the instance of *VC*. Every vertex e in the digraph has in-degree two (from the two vertices that formed the endpoints of the corresponding edge in *VC*). Note that the number of s–t paths in our digraph is exactly $2|E| + 1$.

Given a solution V' to *VC*, we select as a solution in our new problem all arcs (s, v) where $v \in V'$, plus all arcs (e, t), for a total of at most $K + |E|$ arcs. This set of arcs distinguishes any pair of paths, as can be seen from the following two observations: (i) it distinguishes between the $|E| + 1$ arcs that reach t; and (ii) for each vertex e, it distinguishes between the two ways of reaching it (one from each vertex corresponding to an endpoint of the edge e) since it corresponds to a cover.

Conversely, note that: (i) any solution to our problem must distinguish between the two ways of reaching any vertex e corresponding to an edge, which will require that, for each such vertex e, at least one of the edges leading to it be included; (ii) it must distinguish between the $|E| + 1$ paths leading to t from its immediate predecessors; and (iii) the two are independent. To satisfy (ii), we need to select at least $|E|$ arcs. To satisfy (i), then, we are left with just K arcs to select; since the K arcs must distinguish, for each vertex e corresponding to an edge $e = \{u, v\}$, whether the path passes through u or through v, at least one arc on one of these paths must be included—which corresponds to selecting at least one of the two vertices u and v, which corresponds to a vertex cover.

Additional Problem 24

Additional Problem 25

Additional Problem 26

Additional Problem 27

Recall that a context-free grammar is given by a terminal alphabet, Σ, a set of non-terminal symbols, $\mathcal{N}$, a starting symbol $S \in \mathcal{N}$, and a set of productions, each of the form $A \to w$, where $A \in \mathcal{N}$ and $w \in \{\Sigma \cup \mathcal{N}\}^*$, i.e., w is a string made of terminal and non-terminal symbols or is the empty string, ε (in which case $A \to w$ is an ε production). A derivation in the grammar starts from the start symbol, S; applies to it any production whose left-hand side is S, replacing S by the right-hand side of the production, and continues by replacing any non-terminal symbol in the resulting string by any right-hand side of a production whose left-hand side is that non-terminal, until the string is composed only of terminal symbols. The strings of terminals (i.e., strings in Σ^*) that can thus be derived form the language defined by the grammar. A useless non-terminal is one which can never appear in any intermediate string derived from the start symbol (such a non-terminal is effectively "unreachable").

The problem is easily shown to be NL-hard by transforming *Digraph Reachability* to it. We make each vertex a non-terminal in the grammar; for each arc, (i, j), we add a production $i \to j$; in addition, there is a starting symbol, S, with productions $S \to i$ for each i (so that no nonterminal is inaccessible), productions $i \to a$ for some terminal a and each i (so that each nonterminal derives a string) and production $n \to a1$. This last production ensures that the language is infinite if and only if there is a path (a series of productions) from 1 to n. Note that the resulting grammar is in fact regular; together with the second half of the proof below, this show that the problem "is the language generated by a regular grammar infinite" is itself NL-complete.

To show that the problem is in NL, we need only show that there exists some nonterminal, A, such that there is a derivation $A \Rightarrow \alpha A \beta$, for some strings of terminals and nonterminals α and β (not both empty). Since no nonterminal is useless and there are no ε productions, nonemptiness of one of the two strings guarantees that $\alpha A \beta$ derives a terminal string and that that string is longer than that derived by using the same productions on A. The derivation from A to $\alpha A \beta$ need not be longer than n, where n is the number of nonterminals of the grammar. Since we can count to n in logarithmic space, the solution is then as follows: the machine guesses the nonterminal, then the derivation, production by production; we can verify in logarithmic space if the original nonterminal appears on the right-hand side, if at least one of the productions used increased the length of the sentential form, and if each production can be applied to the current sentential form. (We need not store the sentential form: the machine can also guess which nonterminal need be stored, since we look only for a sequence of productions.)

Additional Problem 28

A graph is strongly connected if and only if there is a path between any 2 vertices. Equivalently, a graph is strongly connected if and only if there is a path from vertex 1 to vertex 2, from vertex 2 to vertex 3, . . . , from vertex $n-1$ to vertex n, and from vertex n to vertex 1. Since we can count to n in logarithmic space, the problem is clearly in NL: our machine guesses each path arc by arc; no path between the two vertices i and $i+1$ can be longer than $n-2$; we can verify the existence of each arc and their sequence in logarithmic space.

To show that the problem is complete, we transform *Digraph Reachability* to it. The new graph will have the same vertices as the old, all of the arcs of the old, plus an arc (n, i) for each $1 \leq i < n$ and an arc $(i, 1)$ for each $1 < i \leq n$. Thus, if n is reachable from 1, then there is a path from any i to any j of the form $i \to 1 \ldots n \to j$; conversely, there is a path from 1 to n only if n was reachable from 1 in the original problem, since our new arcs did not introduce any new path from 1 to n (something we can prove formally by showing that, if there exists a path from 1 to n in the transformed graph, then there exists one such path that does not use any of the new arcs). Hence n is reachable from 1 if and only if the new graph is strongly connected.

Additional Problem 29

Additional Problem 30

We prove that all four versions are NP-hard by using them in a Turing reduction to solve *Hamiltonian Circuit*. We simply ask for the smallest disjoint cycle cover and count the number of cycles in it. Clearly, a Hamiltonian cycle covers the entire graph, so that the graph is Hamiltonian if and only if the smallest disjoint cycle cover has exactly one cycle in it. (The directed version of *HC* is known to be equivalent to the undirected version.)

To show that they are NP-easy, we use the standard search technique to discover the number of cycles in the optimal cover with a logarithmic number of calls, then proceed to build a cover of that size. Our query technique will simply be to merge two vertices joined by an edge and verify that the resulting graph still has a DCC of the same size. If the answer is yes, then the two vertices form part of a cycle in an optimal DCC, so we keep them merged and proceed; otherwise, they are in different cycles, so we remove their common edge and proceed. In either case, each query eliminates an edge of the original graph, so we obtain our answer with a linear number of queries. Details change somewhat among the four versions—as just outlined, the procedure applies directly to the undirected, vertex-disjoint version—, but the changes are rather minor.

Additional Problem 31

We consider the following cases: (i) fixed number of moves, unlimited number of pebbles (in which case re-pebbling obviously gains nothing on no re-pebbling); (ii) fixed number of pebbles, no re-pebbling; and (iii) fixed number of pebbles, re-pebbling.

1. With an unlimited number of pebbles, the problem always has answer yes (assuming no cycles in the graph and connectivity). With the number of moves limited by some constant, it is only necessary to check whether or not all of the designated vertex' ancestors can be pebbled in time. Thus we need only count the number of ancestors of the designated vertex and compare this number to the given bound. This is easily done in polynomial time. It appears rather harder to do it in logarithmic space, however, since we need to make sure that we are not counting the same ancestor twice. This may lead us to hypothesize that the problem in not in PolyL, which we might support by attempting to prove that the problem is P-complete.
2. Given a fixed number of pebbles, and with no re-pebbling allowed, the problem is clearly in NP(just list the moves and verify them; since there cannot be cycles, the total number of moves cannot exceed twice the number of vertices of the graph). The problem is in fact NP-complete (it is the register sufficiency problem of Garey & Johnson); the completeness reduction is quite tricky—see the paper "Complete register allocation problems" by R. Sethi, in *SIAM J. Comput.* **4** (1975), pp. 226–248.
3. When re-pebbling is allowed, the problem becomes PSpace-complete, a perhaps surprising fact since the game involves only one player. However, re-pebbling allows cycling, which allows an exponential number of moves. The state of the board can be represented a binary string of n (number of vertices) digits; the problem can be solved in non-deterministic polynomial space (and hence in deterministic polynomial space) by guessing successive states and verifying that each move is legal and that the designated vertex gets pebbled after a number of moves not exceeding 2^n. We need only store two states (linear space) and one counter (also linear space). The completeness

proof is quite tricky: see the article "The pebbling problem is complete in polynomial space" by J.R. Gilbert, T. Lengauer, and R.E. Tarjan, in *SIAM J. Comput.* **9** (1980), pp. 513–524.

Additional Problem 32

If we have PH = PSpace, then PH has complete problems. But then, as we have noted, PH must collapse down to some level. Thus, somewhat paradoxically, making PH as big as possible by equating it to PSpace actually collapses it!

Of course, collapsing PH has other consequences; among other things, it means that nondeterminism stops being useful at that level, so that we will have equality of deterministic, nondeterministic, and co-nondeterministic time classes above that level. The collapse also has additional consequences, which we shall not discuss, in terms of other related classes either not mentioned in this text or not yet introduced (such as classes of randomized complexity).

Additional Problem 33

Using the hint, we know of at least one such class: PSpace. Thus we let A be a PSpace-complete set, such as *QSAT*. Now P^A is the set of all problems decidable in polynomial time with the help of an oracle for PSpace, so that we certainly have $\mathrm{PSpace} \subseteq \mathrm{P}^A$, and NP^A is the set of all problems solvable nondeterministically in polynomial time with the help of an oracle for PSpace, which is surely contained within NPSpace, the set of all problems solvable nondeterministically in polynomial space. Finally, we clearly have $\mathrm{P}^A \subseteq \mathrm{NP}^A$ and we know from Savitch's theorem that NPSpace equals PSpace. Combining all these relations, we can write

$$\mathrm{PSpace} \subseteq \mathrm{P}^A \subseteq \mathrm{NP}^A \subseteq \mathrm{NPSpace} = \mathrm{PSpace}$$

so that all of these sets are in fact equal.