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

Solution Manual
Chapter 6, Text Problems

Exercise 6.1

We shall map the elements of the set in *X2C* to vertices of the graph in *Matching* and 2-sets of *X2C* to edges of the graph in *Matching*. Specifically, given an instance of *X2C* as the set S (where $|S|$ is even) and the collection $\mathscr{C}$ of 2-subsets of S, we create an instance $G = (V, E)$ of *Matching* by setting $V = S$ and $E = \mathscr{C}$. We claim that the instance of *X2C* has a solution (an exact cover) if and only if the instance of *Matching* has a matching of size $|S|/2$.

Assume then that we have an exact cover for S, that is, a collection of $|S|/2$ of 2-subsets of S, the union of which is S. Then these subsets, viewed as edges of G, share no endpoint (because they are pairwise disjoint) and thus form a matching of the required size. Conversely, assume that we have a matching for G of size $|S|/2$, that is, we have $|S|/2$ edges with no shared endpoint. Each edge correspond to a 2-subset of *X2C*; since these edges share no endpoint, their corresponding subsets are all disjoint; but then they form a cover, since together they cover $2 \cdot |S|/2 = |S|$ elements, i.e., the entire set.

The reduction takes linear time, since it basically does no more than copy the input.

Exercise 6.2

Let an instance of *Partition* be given by a set S and a size assignment $s\colon S \to \mathbb{N}$, such that the sum of all sizes is even, $\sum_{x \in S} s(x) = 2n$ for some $n \in \mathbb{N}$. We shall map the elements of *Partition* into elements of *Kth Largest Subset*, with identical size assignment. We also set $B = n$. We then start a binary search process to determine the exact number of subsets with total size less than or equal to n; since there are up to $2^{|S|}$ subsets, a binary search will take $\log_2 2^{|S|} = |S|$ calls to the routine for *Kth Largest Subset*, each time with a different value of K. What we need to know, however, is the number of distinct subsets with total size *equal* to n: if that number is nonzero, then we can answer "yes" for our original instance of *Partition*. So we now set $B = n - 1$ and conduct the same binary search again; with another $|S|$ calls to the routine for *Kth Largest Subset*, we determine the number of subsets with total size less than or equal to $n - 1$. The difference between our first number and our second one is the desired number of subsets with total size equal to n. If that number is 0, we answer "no," otherwise we answer "yes." The reduction itself does very little work beyond the calls to the subroutine for *Kth Largest Subset*: the mapping of elements and sizes is mere copying of the input and the rest involves only keeping track of two counters of $|S|$ bits each and making one subtraction. The main work is in calling the routine for *Kth Largest Subset* a total of $2|S|$ times.

Exercise 6.3

If A many-one reduces to B, then it also Turing reduces to B: we can view the many-one reduction as a very specialized Turing reduction in which we make a single call to the decision procedure for B and do no postprocessing whatsoever, returning the answer given by the decision procedure for B.

That many-one reductions are transitive is easy to see: if A many-one reduce to B through mapping f and B many-one reduces to C through mapping g, then the mapping $g \cdot f$ will many-one reduce A to C. Formally, given $x \in A$, applying the mapping yields $(g \cdot f)(x) = g(f(x)) = g(y)$. We have $y \in B$ by hypothesis, since f is a many-one reduction from A to B; now, given $y \in B$ and since g is a many-one reduction from B to C, we must have $g(y) \in C$ as desired. Conversely, given $x' \notin A$, then applying the mapping yields $(g \cdot f)(x') = g(f(x')) = g(y')$. We have $y' \notin B$ by hypothesis, since f is a many-one reduction from A to B; now, given $y' \notin B$ and since g is a many-one reduction from B to C, we must have $g(y') \notin C$ as desired.

That Turing reductions are transitive is equally easy to see. If we have a procedure $\mathcal{P}_1$ to reduce A to B and another procedure $\mathcal{P}_2$ to reduce B to C, then we can reduce A directly to C by replacing within $\mathcal{P}_1$ each call to the decision procedure for B by a call to $\mathcal{P}_2$. The resulting algorithm only makes calls to the decision procedure for C and, by hypothesis, correctly decides membership in A.

Exercise 6.4

The reason is simply that $\mathcal{C}$-completeness requires membership in the class $\mathcal{C}$, whereas $\mathcal{C}$-equivalence does not.

Exercise 6.5

The proof is deceptively simple: the machine can simply replace its input string by the one string of the same length that induces the same behavior. (Typically, that string is thought of as a string of 1s.) Since we use off-line Turing machines for space bounds, however, the machine cannot actually alter the input tape, which is read-only; thus what it does is to write onto the scratch tape the desired string, then proceed as the original machine did, except that the input is always read from the scratch tape instead of the input tape. Since we thus immediately use at least n tape squares on the scratch tape for an input of size n, this technique can only be used for constructing space functions that are at least as large as n everywhere.

Exercise 6.6

What we need to prove is that, if some problem A is in P and some problem B many-one reduces to A in polynomial time, then B is also in P; the we should repeat for Exp; finally we need to show that such need not be the case for E.

We begin with P. Since A is in P, there is a Turing machine M and a polynomial p such that M can decide an instance x of A in time $p(|x|)$. Let f be the polynomial-time transformation from B to A and let q be its polynomial bound. We build a decision machine M' for B by using the machine M for A and the the transformation f. Specifically, M' will decide an instance x of B by transforming it in time $q(|x|)$ to an instance $y = f(x)$ of A and running the decision procedure M on y in $p(|y|)$ time. Since the transformation takes time $q(|x|)$, what it produces, the string y, cannot exceed $q(|x|)$ in length; thus the time taken by solver M, which is $p(|y|)$, is at most $p(q(|x|))$, which is a polynomial in $|x|$. Hence our solver M' runs in time polynomial in $|x|$ and thus B is in P.

The same reasoning applies if we replace the polynomial bound p by an exponential bound of the type 2^{n^k} for some $k \in \mathbb{N}$, since we can bound any quantity of the form $2^{q^k(n)}$ by another exponential of the form 2^{n^m} for a suitable choice of m.

However, if the bound is a bound for E, that is a bound of the form 2^{kn}, then the running time after the transformation is of the form $2^{kq(n)}$, which, if q is a polynomial of degree higher than 1, cannot be bounded by any function of the form 2^{mn} for any constant m. Thus E need not be closed under polynomial-time transformations. To prove that it is actually not closed, we need a specific counterexample; that is much harder, since it requires knowledge of specific problems with specific lower bounds—we need to know of at least one problem requiring more time than is available in E and that happens to reduce to some problem in E in polynomial time.

Exercise 6.7

What we need to prove is that, if some problem A is in PolyL and some problem B many-one reduces to A in logarithmic space, then B is also in PolyL. Since A is in PolyL, it is in L^i for some $i \in \mathbb{N}$. We need to visualize the reduction process on our off-line Turing machines. What happens is that the reduction takes an instance x of B and, using logarithmic space—that is, $O(\log |x|)$—on the scratch tape, produces an instance $f(x)$ of A on the output tape, which is then used as input tape by the machine for A. That machine, in turn, uses L^i space—that is $O(\log^i |f(x)|)$—on its scratch tape to decide the instance. If we ignore the space used to communicate $f(x)$, what we need to show is that $\log^i |f(x)|$ is polylogarithmic in $|x|$, the size of the input. Note that the reduction process through f take $O(\log |x|)$ space and thus cannot take more than $O(c^{\log |x|})$ time; in particular, the output it produces, $f(x)$, cannot be longer than the number of steps used by the reduction and thus has length $O(c^{\log |x|})$. Thus the space used by the decision procedure for A on instance $f(x)$ is $O(\log^i |f(x)|) = O(\log^i c^{\log |x|}) = O(\log^i |x|)$, as desired.

The issue of the space taken by the output of the reduction, before that is used up by the decision procedure for the target set, is taken up in Exercise 6.11.

Exercise 6.8

What we need to prove is that, if some problem A is in NP and some problem B many-one reduces to A in polynomial time, then B is also in NP. Since A is in NP, there is a Turing machine M and a polynomial p such that a certificate for a "yes" instance x of A can be verified by M in time $p(|x|)$. Let f be the polynomial-time transformation from B to A and let q be its polynomial bound. We build a checker M' for B by using the checker M for A and the the transformation f. Specifically, M' will expect certificates for instances of A, not B, because it will verify such a certificate for an instance x of B by transforming x into an instance $y = f(x)$ of A in time $q(|x|)$, running M on y in time $p(|y|)$, and returning the answer given by M. Since the transformation takes time $q(|x|)$, what it produces, the string y, cannot exceed $q(|x|)$ in length; thus the time taken by checker M, which is $p(|y|)$, is at most $p(q(|x|))$, which is a polynomial in $|x|$. Hence our checker M' runs in time polynomial in $|x|$ and thus B is in NP.

Exercise 6.9

Let the problem be Π_0 and denote the larger class by C_2, the smaller class by C_1. Proving Π_0 to be C_2-complete proves that Π_0 belongs to C_2; thus it remains only to show that Π cannot belong to C_1, assuming that the completeness was shown under a reduction that uses resources no larger than those available in C_1. We show that, if Π_0 is in C_1, then we must have $C_1 = C_2$, thereby contradicting the hypothesis of proper containment. Let Π be some arbitrary problem in C_2; since Π_0 is C_2-complete, Π reduces to Π_0, using no more resources than available in C_1. If Π_0 is in C_1, then we can solve with the resources available in C_1; thus, by combining the algorithm for the reduction from Π to Π_0 and the algorithm for the solution of Π_0, both of which run with the resources available in C_1, we get a solution algorithm for Π which runs with the resources available in C_1, leading us to conclude that Π is in C_1. Since Π was an arbitrary problem of C_2, we conclude that C_2 is a subset of C_1, a contradiction.

Note that the only somewhat vague point in this argument is whether or not the reduction used is transitive; but note that it must be transitive in order for the notion of completeness to have any use.

Exercise 6.10

While it is true that the expression constructed in the generic reduction to *QBF* contains only existential quantifiers, the quantifiers are not all at the front of the expression: many are embedded within the expression, including some embedded within implications. In transforming the Boolean expression to conjunctive form for *QSAT*, implications are replaced by their disjunctive equivalent: for $a \Rightarrow b$, we substitute the equivalent $a \vee \overline{b}$. This change introduces negation in front of some existential quantifiers; moving the negation through the quantifier changes it from existential to universal, replacing $\neg\exists x P(x)$ by the equivalent $\forall x \neg P(x)$. Thus a mix of quantifiers appears.

Exercise 6.11

Transitivity is trivial for polynomial-time reductions, but more difficult for logspace reductions, simply because there is no easy way to store the intermediate instance. If A reduces to B and B reduces to C, the output of the first reduction serves as input to the second in the compound reduction from A to C—but we cannot afford to store more than a few bits of that output, since now that is only an intermediate result and so must find place on our scratch tape.

On the other hand, we have no direct bound on time, so we can trade time for space as follows. We run the second reduction normally, except that it does not have an input tape; instead, whenever it needs to read a bit of its input, it calls upon (a slightly modified version of) the first reduction to generate that bit of its output. In order to identify the needed bit, the second reduction keeps a counter which identifies the position of the read-only head it would have on its input tape and passes that counter to the (modified) first reduction. The (modified) first reduction generates the bit at that position on its (non-existent) output tape by restarting from scratch every time, using its input tape, but not writing any output—instead simply counting how many bits it would have produced had it been writing, stopping when the proper count is reached, and delivering that last bit to the second reduction. The first reduction runs in logarithmic space by hypothesis and so the modified version never exceeds that bound whenever called upon to produce a bit of its output. Moreover, the size of its output is at most polynomial, since it cannot run for more than polynomial time within its logspace bound. The second reduction also runs in logarithmic space on any input; however, it now runs on an intermediate result—the input to the first reduction is an instance of A of size n, but the input to the second reduction is an instance of B is size $O(p(n))$ for some polynomial $p()$. Fortunately, $\log p(n)$ is $O(\log n)$, so that our second reduction still runs in logarithmic space as measured against the instance of A. Thus the total process never uses more than logarithmic space and we are done.

Exercise 6.12

Why is it that both P and PSPACE have complete problems, while POLYL does not, particularly when the argument for the latter is one based on viewing the class as an infinite union of increasingly larger classes? We can view P as the infinite union of $\text{TIME}(n^k)$ and PSPACE as the infinite union of $\text{SPACE}(n^k)$ and try to apply the same argument. (There is a technical problem with viewing P in that way, since the component classes, $\text{TIME}(n^k)$, are not model-independent; this problem does not arise with the component classes for PSPACE and can safely be ignored.) The key difference is in the effect of the chosen reduction on the component classes. The component classes of POLYL are closed under logarithmic-space reductions, as shown in Exercise 6.11. But the component classes of P are *not* closed under logarithmic-space reductions, since a logarithmic-space reduction can take any polynomial amount of time. Thus having a P-complete problem in some class $\text{TIME}(n^k)$ does not imply that any problem that reduces to it will also be in that class; it could well be only in $\text{TIME}(n^j)$ for $j > k$. The same flaw appears for PSPACE: its component classes, $\text{SPACE}(n^k)$, are not closed under polynomial-time reductions, which can use any polynomial amount of space. (Interestingly, however, these component classes *are closed* under logarithmic-space reductions, so that the argument used for POLYL applies to PSPACE under these circumstances and shows that PSPACE cannot have complete problems under logarithmic-space reductions. But P and NP do have such problems: so should we conclude that PSPACE is strictly larger than P or NP?)

Exercise 6.13

If problem A is $\mathcal{C}$-hard, then there exists some problem B complete for $\mathcal{C}$ under the appropriate reduction t that t-reduces to A, $B \leq_t A$. But every problem in $\mathcal{C}$ t-reduces to B, since B is complete; thus, if $\mathcal{C}$ contains any intractable problem, B must be one of them. But then A is intractable—otherwise we could solve B by t-reducing it to A, solving A, and thus obtaining a solution for B.

Exercise 6.14

The class EXP is the class of all problems solvable in exponential time, that is in $O(2^{n^{O(1)}})$ time. The hierarchy theorem for time tells us that there exist problems solvable in $O(2^n)$ time that are not solvable in $O(n^{O(1)})$ time. Thus EXP contains problems not solvable in polynomial time and hence intractable. The same comment can be made for EXPSPACE compared to PSPACE; since anything solvable in polynomial time is solvable in polynomial space, it follows that EXPSPACE contains problems not solvable in polynomial time, i.e., intractable problems.

Exercise 6.15

That constant functions are both time- and space-constructible is obvious: for each fixed constant c, it is easy to construct a Turing machine that ignores its input and makes exactly c steps (or writes exactly c tape squares) before halting.

For polynomial and exponential functions, we assume that all coefficients are positive, since we cannot in any case represent negative values.

That polynomial and exponential functions are space-constructible is almost as easy: we can easily construct a Turing machine that uses exactly as much space as its input (by copying its input tape onto its scratch tape), any fixed multiple of its input (by encoding in the finite state the number of copies), the square of its input (by keeping on the scratch tape a counter into the input tape, effectively marking each square for each copy made—the counter itself is no longer than the first copy and can be kept in the space used by the first copy), any power of its input (repeat the squaring or multiplication), any integer constant larger than 1 raised to the power of its input (similar tricks), and thus any polynomial or exponential function. For time construction, we can certainly do any simple linear functions (just spend as many steps per input symbol as needed) and any first-degree polynomial (just add the constant extra number of steps through the finite-state control). For higher powers and exponentials, we need to count, which will by itself consume some steps; for instance, the simplest way to construct a machine for large-valued functions is to evaluate the function at the given argument, set up a counter with that value, and start decrementing the counter. For each specific function we can easily compute in advance the amount of overhead incurred in the various maintenance operations and thus compute an appropriately decreased function.

Exercise 6.16

The only trick in padding is making sure that the padding need not be handled at all by the existing algorithm—e.g., by adding a distinct separator between the "real" input and the padding. The other consideration is the amount of padding needed, but that is fairly simple: we want to pad an input of size n to size n' so that a running space of $g(f(n))$ appears to be just $g(n')$. In other words, we need to pad enough that the input of size n becomes a new input of size at least $f(n)$; perhaps the simplest way to do this is the make the new input of size $f(n)$ by tacking onto the original input a separator and $f(n) - n$ useless symbols. The algorithm that reads the input need not read beyond the separator, even though the input is much longer; the running space is gauged in terms of the total input size, however, so that the original input of size n that took $g_1(f(n)$ space to process, now appears to take only $g_1(n')$ space to process—even though nothing has changed in the processing. Since the padded problem is solvable within $g_1(n')$ space, it is also solvable within $g_2(n')$ space. The algorithm that runs in g_2 space may not "know" enough to stop reading at the separator, but we can simulate it to ensure that nothing gets read past the separator (a very simple simulation, where we simply check at every move whether the separator has been read and, if so, simulate the transition made by the original machine on end of input tape). This simulation takes the same amount of space as the machine it simulates plus $\log n'$ space to keep track of the original head position on the input tape and thus runs in $g_2(n') + \log n'$ space; in terms of n, this is a running space of $g_2(f(n)) + \log f(n)$ space, which is at most $2 \cdot g_2(f(n))$ space (because we have assumed that the g_is are at least logarithmic). By the compression theorem, we can reduce this quantity to just $g_2(f(n))$ space by a change in alphabet.

Note that this argument requires f to be at least as large as n everywhere, so that $f(n) - n$ is non-negative, and the g_is to be at least logarithmic, so that their contribution is at least as large as that of the counter used to keep track of the input head position.

Exercise 6.17

That's a very open-ended question. Rather than try to resolve it, we simply give a few examples of what can be done. The basic question asked here is how much of a nondeterministic hierarchy for space can be built without having a hierarchy theorem for nondeterministic space? Proving a nondeterministic version of our hierarchy theorems is quite challenging, so a simpler alternative, at least for space, might be to use existing tools to build the hierarchy indirectly. Our tools are

1. the hierarchy theorem for deterministic space
2. the basic relationship $\text{DS}\textsc{pace}(f) \subseteq \text{NS}\textsc{pace}(f)$
3. Savitch's relationship $\text{NS}\textsc{pace}(f) \subseteq \text{DS}\textsc{pace}(f^2)$
4. translational lemmata, including Lemma 6.3: $\text{S}\textsc{pace}(g_1) \subseteq \text{S}\textsc{pace}(g_2) \Rightarrow \text{S}\textsc{pace}(g_1 \cdot f) \subseteq \text{S}\textsc{pace}(g_2 \cdot f)$, for f at least as large as n. but also what we called out downward separation results: if nondeterminism no longer pays at some level in the hierarchy, it cannot pay any higher up in the hierarchy.

The key is Savitch's theorem, since it gives us a rather narrow bracket (a tight "upper bound") for the nondeterministic class to sit in—the same program of construction for nondeterministic time would fail completely because the upper bound is much too high.

For instance, the hierarchy theorem for space tells us that quadratic space is a proper subset of cubic space; it follows that nondeterministic linear space is a proper subset of nondeterministic cubic space, because we can write

$$NSPACE(n) \subseteq \text{DS}\textsc{pace}(n^2) \subset \text{DS}\textsc{pace}(n^3) \subseteq \text{NS}\textsc{pace}(n^3)$$

(Of course, we can tighten this relationship to state that nondeterministic linear space is a proper subset of any super-quadratic nondeterministic space class, since that is all we need to invoke the hierarchy theorem.)

The problem with this approach is that the gap imposed by Savitch's theorem seems unbridgeable; for instance, if we want to consider the class $\text{NS}\textsc{pace}(n^3)$, the best this strategy will get is is that it is a proper subset of any class $\text{NS}\textsc{pace}(n^{6+\varepsilon})$. Using translational lemmata helps us overcome this part of the problem. We can prove, for instance, that $\text{NS}\textsc{pace}(n^3)$ is a proper subset of $\text{NS}\textsc{pace}(n^4)$ by contradiction: if we have $\text{NS}\textsc{pace}(n^3) = \text{NS}\textsc{pace}(n^4)$, then we can translate this equality by applying $g(x) = x^{4/3}$ to both resource bounds, obtaining $\text{NS}\textsc{pace}(n^4) = \text{NS}\textsc{pace}(n^{16/3})$, which allows us to conclude $\text{NS}\textsc{pace}(n^3) = \text{NS}\textsc{pace}(n^{16/3})$; then we can again translate with the same g to obtain $\text{NS}\textsc{pace}(n^{16/3}) = \text{NS}\textsc{pace}(n^{64/9})$, and thus conclude $\text{NS}\textsc{pace}(n^3) = \text{NS}\textsc{pace}(n^{64/9})$. But this last equality contradicts our earlier result that NSPACE(n^3) is a proper subset of of any class NSPACE($n^{6+\varepsilon}$), so that we must in fact have had NSPACE(n^3) $\subset$ NSPACE(n^4), as desired.

The technique just demonstrated can be applied over and over to amplify difference in resource functions for space until we contradict a firm gap obtained by a combination of the hierarchy theorem for deterministic space and Savitch's theorem. The result is a very detailed hierarchy for nondeterministic space.

Exercise 6.18

It is clear that, if we can prove the implication in the hint, we get the desired contradiction, since we would obtain PSPACE $\subseteq$ DSPACE(n), whereas the hierarchy theorem for space implies DSPACE(n) $\subset$ PSPACE, with strict containment. To prove the desired implication, we take a string of an arbitrary problem in PSPACE, pad it to make it look like it belongs to a problem in DSPACE(n), which places it also in P, then show that the padding does not affect the solvability in polynomial time.

So let Π be a problem in PSPACE; if it is in PSPACE, it is because it is in DSPACE(k) for some fixed k. Let $x \in \Pi$ be an arbitrary instance of Π; consider the new string $y = x\#0^{|x|^{k-1}}$, where the character # is one never used by the known DSPACE(k) algorithm that solves Π. That algorithm solves instance x in $O(|x|^k)$ space. But it (well, a very slight modification thereof) also solves y in $O(|y|)$ space—the character # acts as a separator beyond which the modified algorithm never reads; thus the input y has length $|y| = |x|^k + 1$, but the algorithm only "sees" the first $|x|$ bits of it. So, the set of all ys derived in this fashion from all instances x of Π forms a problem of its own, call it Π'. We have just shown that Π' is in DSPACE(n); thus, by our hypothesis, it is also in P. Thus it remains to prove that, if we can solve Π' in polynomial time, we can also solve Π in polynomial time. If there exists an algorithm that solves an instance y of Π' in $p(|y|)$ time for some polynomial p, then an instance x of Π can be solved as follows: first pad the instance as described above to form $y = x\#0^{|x|^k}$, then solve the y with the known polynomial-time algorithm. The first step (padding) takes $O(|x|^k)$ time, while the second (solving) takes time $O(p(|y|)) = O(p(|x|^k))$; thus both steps run in time polynomial in $|x|$ and thus x can be solved in polynomial time. Our conclusion follows.

Exercise 6.19

We begin by proving the easy half: if a set is the range of an honest polynomially computable function, then it is in NP. Let S be such a set;

that is, there exists some honest polynomially computable function f such that $f(\Sigma^*) = S$. Let the polynomial bound on the length of f be $p_1()$ and let the polynomial bound on the running time of f be $p_2()$, and pick some new polynomial $p()$ that is everywhere larger than both $p_1()$ and $p_2()$. Now, given some element $x \in S$, we know that there exists some string y such that: (i) $f(y) = x$; (ii) $|y| \leq p(|x|)$; and (iii) $f(y)$ is computable in $p(|y|)$ time—which is at most $p(p(|x|))$ time by (ii). There is clear parallel between y and a certificate: like an NP certificate, y depends on its matching x, is concise (bounded by some polynomial in the length of x), and leads to a polynomial-time deterministic computation (here that of $f(y)$). Given an element x of S, the certificate is the string y described above and the checker simply computes $f(y)$ and verifies that $f(y) = x$; thus every element of x has a concise certificate checkable in polynomial time, so that S is in NP.

We now prove the harder half: every set S in NP is the range of an honest polynomially computable function. If a set is in NP, there exists a TM M and a polynomial $p()$ such that, for each $x \in S$, there is a certificate c_x such that $M(x, c_x)$ accepts after at most $p(|x|)$ steps. Using M, $p()$, and the existence of concise certificates, we must show that S is the range of an honest, polynomially computable function f; in other words, $f()$ must produce every element of S and nothing else, and must do so honestly and in polynomial time. In attempting to reverse the reasoning used above, we could define $f(y) = x$ whenever $M(x, y)$ accepts in at most $p(|x|)$ steps and $|y| \leq p(|x|)$, i.e., if y is a concise certificate for x. This approach fails for at least three reasons: (i) it leaves $f()$ ill-defined, because there could be many different $x \in S$ for which the same y is a valid concise certificate (for instance, a truth assignment to n variables is a certificate for a very large collection of possible Boolean formulae); (ii) even if this problem were somehow resolved, it is not clear that $f()$ would generate all elements of S; and (iii) there is no obvious way to compute this function, short of searching through all possible strings x until one is found for which $M(x, y)$ accepts—an exponential-time search. These three observations lead us to a working solution: since all we have time to do is little more than running the checker machine M on one pair of values, we should encode both values into the argument of $f()$—or rather, since the argument is given, we should regard it as the encoding of a pair of values, just as in a dovetailing argument. Let the enumeration of the two indices be given by the dovetailing function $\langle -, - \rangle$; given an argument y for $f()$, we regard it as being produced by the dovetailing function, i.e., we write $y = \langle y_1, y_2 \rangle$. Note that this will automatically enumerate every element of Σ^* in each coordinate, so that we shall not have to worry about missing elements of S.

Let z_0 be some arbitrary element of S (we can always compute one off-line if needed). Given $y = \langle y_1, y_2 \rangle$, we begin by decoding y to recover y_1 and y_2. Next, we check whether $|y_2| \le p(|y_1|)$; if so, then we run $M(y_1, y_2)$ for at most $p(|y_1|)$ steps. If M accepts, then we set $f(y) = f(\langle y_1, y_2 \rangle) = y_1$—because we have established that y_2 is a concise certificate for y_1 and thus that y_1 is an element of S. If either test fails, we set $f(y) = z_0$.

By construction, it is clear that the range of f is contained in S and that f can be computed in polynomial time (assuming that we can decode the dovetailing pair in polynomial time, which is easy). We are left to verify that every element of S is in the range of $f(\,)$ and that f is honest. But both follow immediately from the hypotheses: since every element x to be produced by f is in $S \in \mathrm{NP}$, it has a concise certificate c_x; let $y = \langle x, c_x \rangle$, then it follows that $f(y) = x$, an honest production of x. (Note that our function is not honest for every single argument: many of the arguments y that lead to $f(y) = z_0$ are such that $|y| > p(|z_0|)$; but that does not matter, since at least one of the arguments produces z_0 honestly.)

Exercise 6.20

We already know $\mathrm{P} \subseteq \mathrm{NP}$ and $\mathrm{Exp} \subseteq \mathrm{NExp}$. So our additional hypothesis is $\mathrm{NP} \subseteq \mathrm{P}$ and what we need to prove is $\mathrm{NExp} \subseteq \mathrm{Exp}$. In order to prove it, we start with some set $S \in \mathrm{NExp}$ and proceed to pad its elements so that the set of padded instances, call it S_p, now belongs to NP; we then use the hypothesis $\mathrm{NP} \subseteq \mathrm{P}$ to conclude that there exists some deterministic algorithm A that decides S_p in polynomial time; finally, we show that A (or a trivial modification of it) also decides the original S, albeit in exponential time, since the elements of S are much more concise than those of S_p. This sequence of implications proves that any $S \in \mathrm{NExp}$ is also part of Exp, so that we have, as desired, $\mathrm{NExp} \subseteq \mathrm{Exp}$.

The critical step is obviously the padding. Without loss of generality, let the alphabet be binary. Given $x \in S$, we generate $x_p \in S_p$ by setting $x_p = x\#0^{2^{|x|}}$, where # is a new symbol—we can keep the binary alphabet by the simple device of recoding 0 as 00, 1 as 01, and coding the new # as 11. The new string is exponentially longer than the old one. The deterministic checking algorithm for S can be trivially modified to read its input only up to the code for # and decode the 2-digit code back into its "native" alphabet and thus will serve as a checker for S_p; since the checker ran in time $O(2^{p(|x|)})$ on input x, it runs in the same amount of time on the new input (at worst, there is some small—linear—additive term due to the check for # and the decoding), but now that time is polynomial in $|x_p|$. It follows that S_p is in NP and thus, by hypothesis, in P. Hence there exists some deterministic polynomial-time algorithm A that solves S_p. Now, note that, whatever A looks like, there is no need for it to scan its input past the marker #; that is, there exists some similar algorithm A', that acts just like A except that it never reads its input tape past the marker. A trivial modification to A' allows it to run on elements of S as well—just replace the check for the marker # by a check for a blank. Hence this new algorithm, call it A'', solves S deterministically; its running time is some polynomial function of $|x_p|$, say $q(|x_p|)$; however, running on $x \in S$, the running time becomes $q(2^{p(|x|)})$ and thus exponential in $|x|$. Hence $S \in$ EXP, as desired.

Exercise 6.21

No alteration is needed in the construction used in the proof of Cook's theorem, but we may want to review the machinery just a bit. Basically, all of our clauses are generated in groups, each group indexed by 1, 2, or 3 indices. This means that we can produce all of the variables and all of the clauses through a small constant number of blocks, each block consisted of 1, 2, or 3 nested loops running through simple indices; the body of the innermost loop simply prints out the clause for that block, indexed through the loops indices. Thus all that need be kept in memory are the values of the loop indices, which takes logarithmic space.

Exercise 6.22

The main thing that it would show is that logarithmic space and polynomial time are not the same resources when it comes to many-one reductions within NP. This would translate into a simple proof of $L \neq P$, thereby solving an important open problem. Since the answer is negative, we cannot translate it into answers for the equivalent problems higher in the hierarchy (such as PSPACE *vs.* EXP). (In contrast, a proof of $L = P$ would yield, through padding, a proof of PSPACE = EXP and similar results at various levels of the hierarchy. But we can have $L \neq P$ and still have PSPACE = EXP.) Since NC^1 is (more or less) built from L, it would also show $NC^1 \neq P$. It would not resolve the question of NL vs. P, however.

Exercise 6.23

Savitch's theorem gives us a solution: take the nondeterministic algorithm and make it deterministic through the construction of the theorem. Since that is a rather massive undertaking, we can instead describe its simplified version for the specific problem of *Digraph Reachability*.

If vertex n can be reached from vertex 1, it can be reached through a path with at most $n - 1$ arcs. Our algorithm will use divide-and-conquer, recomputing everything as needed from one part of the problem to the other. In general, we will be looking for a way to reach vertex j from vertex i in at most k steps; we will do so by trying every possible vertex (including i or j) as an intermediate vertex that we can reach from i in at most $k/2$ steps and from which we can reach j in at most $k/2$ steps, solving each subproblem recursively, until one choice of intermediate vertex works (exit current call with success) or no intermediate vertex can be found (exit current call with failure). The depth of recursion is $\log n$ since we halve the path length at each step; the local storage is three vertices (source, destination, current choice of intermediate vertex), which takes logarithmic space. Hence the total storage used is $O(\log^2 n)$, as desired.

Exercise 6.24

It is easily seen that any polynomial function is subexponential; thus SubExp is closed under polynomial-time transformations, so that it makes sense to use such transformations for reductions inside SubExp. It follows immediately that, if some NP-complete problem is in SubExp, then NP is a subset of SubExp. From the definition of the class, we can immediately write $\mathrm{P} \subseteq \textsc{SubExp} \subseteq \textsc{Exp}$. Are these containments proper? Consider the family of functions $2^{(logn)^k}$, for $k > 1$; these functions are not polynomial, yet they are subexponential; since

$$\lim_{n\to\infty} \frac{m \cdot \log n \cdot n^m}{2^{(\log n)^k}} = 0,$$

for any constants $k, m > 1$, the hierarchy theorem for time tells us that the two classes P and SubExp are distinct. Since we do not have a general form for the fastest growing functions in SubExp, separating it from Exp is harder, but can also be done on a similar basis: any problem that belongs to SubExp can be solved in time $O(f(n))$, where f is a subexponential function; every subexponential function grows strictly more slowly than any exponential, so that the hierarchy theorem for time tells us that SubExp is a proper subset of Exp. Now $\textsc{SubExp} \subseteq \mathrm{NP}$ (or $\textsc{SubExp} \subseteq \mathrm{PS}\textsc{pace}$) would imply $\mathrm{P} \neq \mathrm{NP}$ ($\mathrm{P} \neq \mathrm{PS}\textsc{pace}$), which we do not know. Conversely, the straightforward simulation of NP (and, *a fortiori*, of PSpace), can require true exponential time, so that we do not know whether containment holds the other way either. (All these remarks carry over, of course, to any class in the polynomial hierarchy.) This leaves the relationship between SubExp and L^k and NL^k, for $k > 1$. If a language is in L^k, then it can use at most $c^{((\log n)^k}$ time for some constant c; thus it is in SubExp, by a remark made above; hence $\mathrm{L}^k \subseteq \textsc{SubExp}$, for any k; by Savitch's theorem, it follows that $\mathrm{NL}^k \subseteq \textsc{SubExp}$ as well. Thus $\textsc{PolyL} \subseteq \textsc{SubExp}$. The containments are presumably proper, but our characterization of NL^k does not allow an answer.

Exercise 6.25

First we verify that any problem with class $\mathcal{C}$ many-one reduces (within the appropriate resource bound) to the bounded halting problem for $\mathcal{C}$. Our set is given as $\{(M, x) \mid M \in \mathcal{C}$ and M accepts $x\}$. If some problem Π belongs to the class $\mathcal{C}$, we must have a machine M_Π for it—we have defined all four classes NL, P, NP, and PSPACE, through machines, so that proof of membership in these classes is the demonstration of a suitable machine. Thus, given some instance x of Π, we simply map it into the pair (M_Π, x); this transformation is just an injective embedding that takes linear time and at the very most logarithmic space and clearly is a valid many-one reduction to our set.

Now we need to show that our set belongs to the class $\mathcal{C}$ itself. But that is totally obvious in view of its definition: given the pair (M, x), we simply run M on x, which takes exactly the appropriate amount of resources for the class, and obtain the answer. One subtle point of major importance that we need to point out here is that the definition of the set includes the somewhat cavalier statement $M \in \mathcal{C}$. How do we verify that a machine belongs to the class? Granted that, if the machine does belong to the class, membership of our set to the class is trivial; but what of the machine? Fortunately, we saw, in our development of the hierarchy theorem, that we can use a variety of techniques to bound the running space (by prior marking of the tape) or running time (by keeping a counter on the side) of a machine and that other issues (such as space-bounded machines that never halt) can also be handled. Thus we can take some machine M and make sure, when we run it on x, that it respects the resource bounds for the class, at least for those classes mentioned here.

The key to these syntactic classes is that the classes themselves are defined through easily characterizable machines and so the bounded halting problem is the most natural characterization of the class—although perhaps not a very useful problem for further reductions.

Exercise 6.26

- Using a non-constructive proof is not a problem; after all, the classes P and NP themselves are defined non-constructively. That we would presumably be unable to construct the DTM is thus no obstacle. However, the big flaw is that the argument only applies for a *specific* yes instance: the NDTM, arriving at some decision point, will presumably take different branches depending on the instance, so that we would have, not one, but many DTMs corresponding to that one NDTM. So, the logic is correct for a single instance—but we knew that a single instance is easy to solve anyway; and the logic completely fails for an infinite collection of instances (we would have to guess which DTM to use at each point, bringing us right back to non-determinism).

- It is really quite simple to add unique labels to each transformed instance: a trivial example is just to repeat the original instance as a "postscript" to the transformed instance—this certainly ensures a one-to-one transformation! Indeed, for many encoding schemes, each code is a valid instance, which means that each instance is its own index. Hence we can derive, using the Schroeder-Bernstein theorem, a bijection between the two problems.
  The problem is that this bijection may not be a polynomial-time isomorphism, i.e., it may not be a valid transformation anymore. To be a valid transformation, it has to respect two conditions: (i) polynomial-time feasibility and (ii) mapping yes instances to yes instances and no instances to no instances. But the Schroeder-Bernstein theorem only states that, given two 1-1 functions (not transformations), there exists a bijection. We do not know that the bijection in question bears much relation to the two injections developed by padding; it may neither be computable in polynomial time nor respect the yes-yes, no-no dichotomy. So flaws may reside there. In fact, a flaw appears even earlier: our padding does result in an injective mapping, but does it respect the yes-yes, no-no dichotomy? This is not at all certain; it may not even produce legal instance descriptions.
- The flaw here is in the glib assertion that, if *Satisfiability* can be solved in $O(n^k)$ time, so can any problem in NP, simply because any problem in NP reduces to *Satisfiability* in polynomial time. The time spent in the polynomial-time reduction could prove a problem (if it is bigger than n^k), but a more fundamental flaw is present: the polynomial-time reduction need not preserve the size of the input! An instance of size n of some problem in NP will be transformed in $O(n^m)$ time (for some value of m) into an instance of *Satisfiability* of size $f(n)$—where the best we can say is that $f(n)$ is $O(n^m)$. Then the assumed solution to *Satisfiability* solves the transformed instance in time $O(f(n)^k)$, but that may be as large as n^{m+k} and thus not $O(n^k)$.

Exercise 6.27

It suffices to show that Turing reductions are no more powerful within P than many-one reductions; thus, if they are more powerful within NP, it must be the case that P and NP are different.

Assume then that problem B in P Turing-reduces in polynomial time to problem A in P; we want to show that B also many-one reduces to A in polynomial time. But that is trivially easy. Let x be a "no" instance of A and y be a "yes" instance of A; given an instance z of B, our mapping f begins by solving z (using the polynomial-time algorithm for B, which must exist since B is in P), then, if the answer is yes, maps z to y and otherwise maps z to x. The mapping takes polynomial time (all it requires is the running time of the solution algorithm for B) and is a valid many-one reduction from A to B. Hence Turing reductions are no more powerful than many-one reductions within P.

Exercise 6.28

Intuitively, NP is closed under conjunctive polynomial-time reductions because requiring "yes" answers to a fixed pattern of calls to an oracle is not that different from requiring a "yes" answer to a single call, at least not where nondeterminism is concerned—we are basically just adding a collection of existential quantifiers, but such a collection can easily be condensed down to a single existential quantifier.

More formally, assume that problem A is in NP and that we have a conjunctive polynomial-time reduction from some (decision) problem B to A. We need to prove that B is then also in NP. Let x be some "yes" instance of B and assume that the reduction makes k calls (where k is no larger than some polynomial function of $|x|$) to the oracle for A. Each call may use a somewhat different mapping of x, so let us denote the values passed to the oracle for A as $f_1(x)$, dots, $f_k(x)$. For each $f_i(x)$, the oracle for A answers "yes," meaning that there is a certificate c_i for each such value that can be verified in polynomial time. But then x, the "yes" instance of B, has a certificate as well, the collection $c_1\#c_2\#\ldots\#c_k$, and a polynomial-time checker, basically the checker for A embedded within a loop that runs it on each of the k pieces of the certificate and verifies that all answers are "yes." This checker cannot accept a "no" instance of B, because such an instance would give rise to at least one $f_i(x)$ that has no certificate acceptable to the checker for A—and thus our checker for B will reject at least one piece of any certificate proposed for the "no" instance of B.

Note that the restriction to a conjunction of acceptances is not the only one that will keep NP closed under the reduction. Requiring a given style of majority (of "yes" answers) would be another one; many others are possible. However, note that we cannot in general define a polynomial-time truth-table reduction, for two reasons. First, specifying the truth table for a polynomial (or even linear) number of variables (the calls to the oracle) may take exponential space! Secondly, the idea used above will only work for certificates and thus for "yes" answers in the calls to the oracle; it is thus essential that the “no” answers returned by the oracle may be safely discarded.

Solution Manual
Chapter 6, Additional Problems

Additional Problem 1

Closure under union and intersection means simply that the union or intersection of two sets in P or NP is also in P or NP, respectively.

Let A and B be two sets in P and let M_A be the solver for set A and M_B the solver for set B. We can combine the two solvers to obtain a solver for $A \cup B$ simply by running each solver in turn on the instance and accepting whenever at least one of the two solvers accepted. Similarly, we can combine the two solvers to obtain a solver for $A \cap B$ simply by running each solver in turn on the instance and accepting whenever both solvers accepted.

Let A and B be two sets in NP and let M_A be the checker for set A and M_B the checker for set B. Every instance in either set has a concise certificate that can be verified in polynomial time. But then every element in $A \cup B$ has such a certificate as well (the same certificate, or one of the two for elements of $A \cap B$), so that the set $A \cup B$ is in NP. In order to be in $A \cap B$, an instance must be in both A and B and thus must possess two concise certificates, one for each set. Thus we can define the set $A \cap B$ by a checker M that runs first M_A on the first part of the certificate and then M_B on the second part of the certificate, accepting only if both parts are verified. Thus $A \cap B$ is in NP.

Additional Problem 2

Additional Problem 3

One direction is easy. If a language is regular, it is accepted by a DFA; the DFA can be viewed as a highly restricted off-line Turing machine that uses no space at all on its scratch tape and so certainly runs in constant space.

Reducing constant space to zero space in the other direction is also easy: if our off-line Turing machine uses space k, we can produce a new off-line Turing machine with a control state set that encodes the $|\Sigma|^k$ possible configurations of the scratch tape of our original machine—in effect, we simply multiply the number of states by $|\Sigma|^k$, a constant. The problem is that the resulting machine is still a Turing machine and can do something that no FA can do, namely go back and reread some input it has already seen. What we have is, in effect, a *two-way finite automaton*.

So the trick is to show that whatever such a machine accepts can also be accepted by a regular FA. This result was proved as early as 1959 by J.C. Shepherdson, "The reduction of two-way automata to one-way automata," *IBM J. Res. & Dev.* **3**, pp. 198–200. Since the reduction is rather tricky and since the original paper is quite clear and concise, we simply refer the reader to the original write-up.