APPENDIX B

NP-Completeness

In this appendix, we briefly review the concepts of NP-completeness and reductions. We will use the knapsack problem of Section 3.1 as a running example. Recall that in the knapsack problem, we are given a set of n items $I = \{1, \dots, n\}$, where each item i has a value v_i and a size s_i. All sizes and values are positive integers. The knapsack has capacity B, where B is also a positive integer. The goal is to find a subset of items $S \subseteq I$ that maximizes the value $\sum_{i \in S} v_i$ of items in the knapsack subject to the constraint that the total size of these items is no more than the capacity; that is, $\sum_{i \in S} s_i \leq B$.

Recall the definition of a polynomial-time algorithm.

Definition B.1. *An algorithm for a problem is said to run in polynomial time, or said to be a polynomial-time algorithm, with respect to a particular model of computer (such as a RAM) if the number of instructions executed by the algorithm can be bounded by a polynomial in the size of the input.*

More formally, let x denote an *instance* of a given problem; for example, an instance of the knapsack problem is the number n of items, the numbers s_i and v_i giving the sizes and values of the items, and the number B giving the size of the knapsack. To present the instance as an input to an algorithm A for the problem, we must encode it in bits in some fashion; let $|x|$ be the number of bits in the encoding of x. Then $|x|$ is called the *size* of the instance or the *instance size*. Furthermore, we say that A is a polynomial-time algorithm if there exists a polynomial $p(n)$ such that the running time of A is $O(p(|x|))$.

We will need the concept of a decision problem. A decision problem is one whose output is either “Yes” or “No.” It is not difficult to think of decision problems related to optimization problems. For instance, consider a decision variant of the knapsack problem in which, in addition to inputs B, and v_i and s_i for every item i, there is also an input C, and the problem is to output “Yes” if the optimum solution to the knapsack instance has value at least C, and “No” otherwise. The instances of a decision problem can be divided into “Yes” instances and “No” instances, that is, instances in which the correct output for the instance is “Yes” (or “No”). The class P contains all *decision problems* that have polynomial-time algorithms.

Roughly speaking, the class NP is the set of all decision problems such that for any "Yes" instance of the problem, there is a short, easily verifiable "proof" that the answer is "Yes." Additionally, for each "No" instance of the problem, no such "proof" is convincing. What kind of "short proof" do we have in mind? Take the example of the decision variant of the knapsack problem given above. For any "Yes" instance, in which there is a feasible subset of items of value at least C, a short proof of this fact is a list of the items in the subset. Given the knapsack instance and the list, an algorithm can quickly verify that the items in the list have total size at most B, and total value at least C. Note that for any "No" instance, no possible list of items will be convincing.

We now attempt to formalize this rough idea as follows. A short proof is one whose encoding is bounded by some polynomial in the size of the instance. An easily verifiable proof is one that can be verified in time bounded by a polynomial in the size of the instance and the proof. This gives the following definition.

Definition B.2. *A decision problem is said to be in the problem class* NP *if there exists a verification algorithm $A(\cdot, \cdot)$ and two polynomials, p_1 and p_2, such that*

1. *for every "Yes" instance x of the problem, there exists a proof y with $|y| \le p_1(|x|)$ such that $A(x, y)$ outputs "Yes";*
2. *for every "No" instance x of the problem, for all proofs y with $|y| \le p_1(|x|)$, $A(x, y)$ outputs "No";*
3. *the running time of $A(x, y)$ is $O(p_2(|x| + |y|))$.*

NP stands for *nondeterministic polynomial time*.

Observe that nothing precludes a decision problem in NP from having a polynomial-time algorithm. However, the central problem of complexity theory is whether *every* problem in NP has a polynomial-time algorithm. This is usually expressed as the question of whether the class P of decision problems with polynomial-time algorithms is the same as the class NP or, more succinctly, whether P = NP.

As an approach to this problem, it has been shown that there are problems in NP that are representative of the entire class, in the sense that if they have polynomial-time algorithms, then P = NP, and if they do not, then P $\neq$ NP. These are the NP-*complete* problems. To define NP-completeness, we will need the notion of a *polynomial-time reduction*.

Definition B.3. *Given two decision problems A and B, there is a polynomial-time reduction from A to B (or A reduces to B in polynomial time) if there is a polynomial-time algorithm that takes as input an instance of A and produces as output an instance of B and has the property that a "Yes" instance of B is output if and only if a "Yes" instance of A is input.*

We will use the symbol $\preceq$ to denote a polynomial-time reduction so that we write $A \preceq B$ if A reduces to B in polynomial time. Sometimes the symbol $\le_m^P$ is used in the literature to denote a polynomial-time reduction. We can now give a formal definition of NP-completeness.

Definition B.4 (NP-complete). *A problem B is* NP-*complete if B is in* NP*, and for every problem A in* NP*, there is a polynomial-time reduction from A to B.*

The following theorem is now easy to show.

Theorem B.5. *Let B be an* NP*-complete problem. If B has a polynomial-time algorithm, then* $\mathrm{P} = \mathrm{NP}$.

Proof. It is easy to see that $\mathrm{P} \subseteq \mathrm{NP}$. To show that $\mathrm{NP} \subseteq \mathrm{P}$, pick any problem $A \in \mathrm{NP}$. For any instance of the problem A, we can run our polynomial-time reduction from A to B and use the polynomial-time algorithm for B. We return "Yes" if this algorithm returns "Yes," and "No" otherwise. By the properties of the polynomial-time reduction, this algorithm correctly decides whether the instance is a "Yes" instance of A, and does so in polynomial time. □

A useful property of NP-complete problems is that once we have an NP-complete problem B it is often easy to prove that other problems are also NP-complete. As we will see, all we have to do is show that a problem A is in NP, and that $B \preceq A$. This follows as an easy corollary of the transitivity of polynomial-time reductions.

Theorem B.6. *Polynomial-time reductions are transitive: that is, if $A \preceq B$ and $B \preceq C$, then $A \preceq C$.*

Corollary B.7. *If A is in* NP, *B is* NP*-complete, and $B \preceq A$, then A is also* NP*-complete.*

Proof. All we need to show is that for each problem C in NP, there is a polynomial-time reduction from C to A. Because B is NP-complete, we know that $C \preceq B$. By hypothesis, $B \preceq A$. By Theorem B.6, $C \preceq A$. □

Many thousands of problems have been shown to be NP-complete. We list two of them here. In the *partition problem*, we are given as input positive integers $a_1, \ldots, a_n$ such that $\sum_{i=1}^{n} a_i$ is even. We must decide whether there exists a partition of $\{1, \ldots, n\}$ into sets S and T such that $\sum_{i \in S} a_i = \sum_{i \in T} a_i$. In the *3-partition problem*, we are given as input positive integers $a_1, \ldots, a_{3n}, b$, such that $b/4 < a_i < b/2$ for all i, and such that $\sum_{i=1}^{3n} a_i = nb$. We must decide whether there exists a partition of $\{1, \ldots, 3n\}$ into n sets T_j such that $\sum_{i \in T_j} a_i = b$ for all $j = 1, \ldots, n$. By the condition on the a_i, each T_j must contain exactly three elements. The decision version of the knapsack problem given at the beginning of the section is also NP-complete. However, as shown in Section 3.1, we know that this problem has a pseudopolynomial-time algorithm. This brings up an interesting distinction among the NP-complete problems. Some NP-complete problems, such as the knapsack and partition problems, are NP-complete only when it is assumed that their numeric data are encoded in binary. As we have seen, the knapsack problem has a polynomial-time algorithm if the input is encoded in unary (recall that in a unary encoding the number 7 would be encoded as 1111111). The partition problem also has a polynomial-time algorithm if the input is encoded in unary. Other problems, however, such as the 3-partition problem above, are NP-complete even when their numeric data are encoded in unary. We call such problems *strongly* NP*-complete* or, sometimes, *unary* NP*-complete*. In contrast, problems such as the knapsack and partition problems are called *weakly* NP*-complete* or *binary* NP*-complete*.

Definition B.8. *A problem B is* strongly NP-complete *if it is* NP*-complete even when its numeric data are encoded in unary. A problem C is* weakly NP*-complete if it has* *a pseudopolynomial-time algorithm (that is, it has a polynomial-time algorithm if its numeric data are encoded in unary).*

We conclude this section by defining the term NP-*hard*, which can be applied to either optimization or decision problems. Roughly speaking, it means "as hard as the hardest problem in NP." To be more precise, we need to define an *oracle*. Given a decision or optimization problem A, we say that an algorithm has A as an oracle (or has *oracle access* to A) if we suppose that the algorithm can solve an instance of A with a single instruction.

Definition B.9 (NP-hard). *A problem A is* NP-hard *if there is a polynomial-time algorithm for an* NP*-complete problem B when the algorithm has oracle access to A.*

For example, the knapsack problem is NP-hard because given oracle access to it, we can solve the decision version of the knapsack problem in polynomial time: we simply check whether the value of the optimal solution is at least the value C for the decision problem, and output "Yes" if so, and otherwise "No."

The term "NP-hard" is most frequently applied to optimization problems whose corresponding decision problems are NP-complete; it is easy to see that such optimization problems are indeed NP-hard, as we saw for the knapsack problem above. It is also easy to see that if A is NP-hard and there is a polynomial-time algorithm for A, then P = NP.