Appendix III:

NP-Completeness and Higher Complexity Classes

III.1 Introduction

In Chapter 1 we introduced the complexity classes that are of greatest interest for us. However, there are slightly different ways of defining complexity classes, the most important upshot of which is the notion of NP-completeness. Since programmers may encounter references to NP-completeness and higher-complexity classes in the literature, it is useful to provide a brief sketch of these concepts.

III.2 NP-Completeness

We must first define the complexity classes P and NP; then we explain NP-completeness. Before we outline the formal definition involving Turing machines, it is useful to give an informal characterization. Intuitively, a problem is in P if there exists an (ordinary) algorithm solving it in polynomial time. Thus, all the complexity classes we defined in Chapter 1 except for the exponential class are subsets of P. NP is then the class of all problems where we can check a solution in polynomial time (even though we may not necessarily find it).

Formally, the definition involves Turing machines. These are abstract machines that encapsulate the notion of computation in the most comprehensive way. A Turing machine consists of a finite state control and an unbounded tape consisting of cells, each of which can contain one data item, together with a read/write head that can look at and change the contents of a cell. Thus, a Turing machine M can be written as a sextuple,

$$M = (Q,T,T_0,\delta,q_0,F),$$

where Q is a finite nonempty set of states, the initial state q_0 is an element of Q, and the set of final states F is a subset of Q. The alphabet[1] T contains all the tape symbols, including a distinguished character □ denoting the blank, and the alphabet T_0 is a subset of T consisting of all input symbols (in particular, the blank symbol is not contained in T_0: $\square \in T - T_0$). Finally, the move function δ is a (partial) function taking a state *p* and a tape symbol *t* and returning a triple, consisting of a state *q*, a tape symbol *s*, and a direction instruction dir that is either L or R:

$$\delta(p,t) = (q,s,dir).^2$$

The interpretation of this is as follows: The finite state control is in state *p* and the read/write head looks at a certain cell whose content is *t*; then the control changes to *q*, the contents of the cell the head inspects is changed from *t* to *s*, and the head is moved to the cell either immediately to the left of the inspected cell (dir = L) or to the right (dir = R). The Turing machine is initially presented with its (finite) input (a string over the alphabet T_0) on a portion of its tape, with the head looking at the first input symbol and the finite state control in the initial state q_0. All cells not occupied by the input string are assumed to be blank (that is, contain □). Then M executes one transition move after another in sequence. This process can terminate in two ways. Either M enters a final state (any state in F), in which case the input string is considered accepted and the Turing machine halts, or M reaches a point where $\delta(p,t)$ is not defined [$\delta(p,t) = \varnothing$] for the given actual state *p* and the contents *t* of the cell currently inspected, in which case the Turing machine rejects the input string and halts. It is possible that the Turing machine does not reach either of these two configurations; in this case the Turing machine does not halt (consequently, no statement can be made regarding acceptance or rejection of the given input string).

The definition above is for a deterministic Turing machine because there is at most one triple in each of the entries $\delta(p,t)$. In other words, given a state and a cell content, we know deterministically (without any guessing or choosing) where the next transition takes us. If we relax this requirement and permit $\delta(p,t)$ to contain more than one triple, we have a nondeterministic Turing machine. In this model, when carrying out a transition move, we must first select one of the alternatives.

Using Turing machines, one can define complexity classes as follows. For a given input string of length *n*, we determine the number of moves the Turing machine makes for this input, assuming the machine halts. If it does not, then the complexity is not defined. Here, the length of the input string *n* is the measure of the input we assumed in our discussion in Chapter 1.

---

[1] An alphabet is a finite nonempty set of atomic symbols called letters. The alphabet of decimal digits is an example (with 10 elements).

[2] The move function is partial since it is permitted that no result is specified for a given pair (p,t): $\delta(p,t) = \varnothing$ where ∅ denotes the empty set.

The number of moves is then the complexity f(n). It can be defined for average[3] and for worst case.[4]

Turing machines are useful because anything that can be computed can be computed using a Turing machine.[5] [6] Moreover, all generally used deterministic computational models can be simulated using deterministic Turing machines in such a way that the complexity of the Turing machine model is no more than a polynomial function of the complexity of the other model. This is referred to as polynomial reduction.

Now we can define the two classes P and NP. P is the set of all problems that can be solved in (worst-case) polynomial time using a deterministic Turing machine. NP is the set of all problems that can be solved in (worst-case) polynomial time using a nondeterministic Turing machine.

While algorithms are polynomially reducible to deterministic Turing machines, it is not known whether nondeterministic Turing machines are polynomially reducible to deterministic Turing machines. Thus, it is not known whether P = NP (although P is contained in NP, P $\subseteq$ NP, since any deterministic Turing machine can be viewed as a nondeterministic one). However, in practical terms, if we want to simulate a nondeterministic Turing machine using a deterministic one, the complexity increases exponentially.

Within the set of all problems in NP, there is subset, called NP-complete problems, consisting of all those that are maximally difficult in the following sense. If we find that one NP-complete problem has a polynomial time algorithm (in other words, if it is in P), then all problems in NP have polynomial time complexity algorithms. Thus, the open question P = NP could be solved affirmatively if one were able to devise a polynomial time algorithm for a single NP-complete problem. However, no such NP-complete problem is known, as of this writing (2005). Thus, the best algorithm for any NP-complete problem has exponential time complexity.

From a practical point of view, finding out that a problem is NP-complete is generally undesirable since it means the best algorithm solving it has exponential time complexity. However, one should note that NP-completeness is based on worst-case time complexity; occasionally, the average time complexity is much better. Moreover, using approximation algorithms, one

---

[3] This requires assigning probabilities to each input string of length n, determining the number of moves for each string, and then forming the weighted average (weighted with these probabilities) of these numbers to obtain the average complexity f(n).

[4] This requires determining the number of moves for each string of length n and finding the maximum of all these values to obtain the worst-case complexity f(n).

[5] It is quite difficult to come up with something intuitively understandable that cannot be effectively computed. My best candidate is the following instruction for finding a location. "Take the one-way street and turn left two miles before the rail-road crossing." While it can be described, it cannot be executed, since we would have to backtrack two miles, which is impossible with a one-way street.

[6] Strictly speaking, this is not a fact or an observation, but a thesis, known as Church's thesis. It is essentially not possible to prove it, since defining in its full generality what is "computable" is infeasible.

can frequently obtain a solution that may not be optimal but is nevertheless acceptable.

III.3 Higher Complexity Classes

The hierarchy of complexity classes is infinite, so neither NP nor the exponential time algorithms are the most complicated classes. There are infinitely many more time-consuming classes between NP and the class of undecidable problems. We have, for instance, doubly exponential classes, exemplified by the number 2^{2^n} of boolean functions in n variables. In the higher reaches of this complexity hierarchy are nonelementary problems, (decidable) problems whose time complexity is so large that it cannot be expressed as a bounded stack of exponentials. This means that given an arbitrary integer M, there exists a value n (dependent on M) such that the time complexity of solving this problem requires more time than the function denoted by a stack of M powers of 2 followed by n:

$$2^{2^{\cdot^{\cdot^{2^n}}}}.$$

This complexity is exemplified by extended regular expressions, regular expressions in whose formulation we admit not just the three operations involved in ordinary regular expressions, namely union, concatenation, and star,[7] but also the operation complementation.[8] While the smallest deterministic finite automaton for an ordinary regular expression of length n may have up to 2^n states, the smallest deterministic finite automata for an extended regular expression of length n may have a nonelementary number in n states.

None of these higher complexity classes has great practical significance for programmers. Essentially, finding out that a problem belongs to one of these classes means that for all but the smallest instances, trying to solve this problem is an exercise in futility.

[7] Union allows one to provide alternatives (e.g., "a constant is either an integer or a real"). Concatenation allows one to compose one regular expression by appending one to another (e.g., "an assignment statement consists of a variable, concatenated to the assignment operator, concatenated to an expression"). Star captures unbounded iteration (e.g., "an integer consists of a digit, concatenated to zero or more digits").

[8] Complementation captures negation. Instead of describing what we want, we describe what we do not want. For example, we may be interested in all strings that do not contain a certain substring. The extremely surprising aspect of complementation is that is has an incredible effect on the complexity of the resulting expressions, from singly exponential to nonelementary.

Bibliographical Notes

NP-completeness and higher complexity classes, including nonelementary problems, are covered in texts on algorithms and abstract computational complexity, for example in Kleinberg and Tardos: *Algorithm Design*; Garey and Johnson: *Computers and Intractability: A Guide to the Theory of NP-Completeness*; and Aho, Hopcroft, and Ullman: *The Design and Analysis of Computer Algorithms*. Turing machines and regular expressions are covered in texts on formal language and automata theory, for example in Hopcroft and Ullman: *Introduction to Automata Theory.*