COMPUTATIONAL COMPLEXITY THEORY

Complexity theory is the part of theoretical computer science that attempts to prove that certain transformations from input to output are impossible to compute using a reasonable amount of resources. Theorem 1 below illustrates the type of "impossibility" proof that can sometimes be obtained (1); it talks about the problem of determining whether a logic formula in a certain formalism (abbreviated WS1S) is true.

Theorem 1. *Any circuit of* AND, OR, *and* NOT *gates that takes as input a WS1S formula of 610 symbols and outputs a bit that says whether the formula is true must have at least* 10^{125}*gates*.

This is a very compelling argument that no such circuit will ever be built; if the gates were each as small as a proton, such a circuit would fill a sphere having a diameter of 40 billion light years! Many people conjecture that somewhat similar intractability statements hold for the problem of factoring 1000-bit integers; many public-key cryptosystems are based on just such assumptions.

It is important to point out that Theorem 1 is specific to a particular circuit technology; to prove that there is no efficient way to compute a function, it is necessary to be specific about *what* is performing the computation. Theorem 1 is a compelling proof of intractability precisely because every deterministic computer that can be purchased today can be simulated efficiently by a circuit constructed with AND, OR, and NOT gates. The inclusion of the word "deterministic" in the preceding paragraph is significant; some computers are constructed with access to devices that are presumed to provide a source of random bits. Probabilistic circuits (which are allowed to have some small chance of producing an incorrect output) might be a more powerful model of computing. Indeed, the intractability result for this class of circuits (1) is slightly weaker:

Theorem 2. *Any probabilistic circuit of* AND, OR, *and* NOT *gates that takes as input a WS1S formula of 614 symbols and outputs a bit that says whether the formula is true (with error probability at most 1/3) must have at least* 10^{125}*gates*.

The underlying question of the appropriate *model of computation* to use is central to the question of how relevant the theorems of computational complexity theory are. Both deterministic and probabilistic circuits are examples of "classical" models of computing. In recent years, a more

Wiley Encyclopedia of Computer Science and Engineering, edited by Benjamin Wah.

powerful model of computing that exploits certain aspects of the theory of quantum mechanics has captured the attention of the research communities in computer science and physics. It seems likely that some modification of theorems 1 and 2 holds even for quantum circuits. For the factorization problem, however, the situation is different. Although many people conjecture that classical (deterministic or probabilistic) circuits that compute the factors of 1000-bit numbers must be huge, it is known that small quantum circuits can compute factors (2). It remains unknown whether it will ever be possible to build quantum circuits, or to simulate the computation of such circuits efficiently. Thus, complexity theory based on classical computational models continues to be relevant.

The three most widely studied general-purpose "realistic" models of computation today are deterministic, probabilistic, and quantum computers. There is also interest in *restricted* models of computing, such as *algebraic circuits* or *comparison-based* algorithms. Comparison-based models arise in the study of sorting algorithms. A comparison-based sorting algorithm is one that sorts n items and is not allowed to manipulate the representations of those items, other than being able to test whether one is greater than another. Comparison-based sorting algorithms require time $\Omega(n \log n)$, whereas faster algorithms are sometimes possible if they are allowed to access the bits of the individual items. Algebraic circuits operate under similar restrictions; they cannot access the individual bits of the representations of the numbers that are provided as input, but instead they can only operate on those numbers via operations such as $+$, $\times$, and $\div$. Interestingly, there is also a great deal of interest in "unrealistic" models of computation, such as nondeterministic machines. Before we explain why unrealistic models of computation are of interest, let us see the general structure of an intractability proof.

DIAGONALIZATION AND REDUCIBILITY

Any intractability proof has to confront a basic question:

How can one prove that there is *not* a clever algorithm for a certain problem? Here is the basic strategy that is used to prove theorems 1 and 2. There are three steps.

Step 1 involves showing that there is program A that uses roughly 2^n bits of memory on inputs of size n such that, for every input length n, the function that A computes on inputs of length n requires circuits *as large as are required by any function on n bits*. The algorithm A is presented by a "diagonalization" argument (so-called because of similarity to Cantor's "diagonal" argument from set theory). The same argument carries through essentially unchanged for probabilistic and quantum circuits. The problem computed by A is hard to compute, but this by itself is not very interesting, because it is probably not a problem that anyone would ever *want* to compute.

Step 2 involves showing that there is an efficiently computable function f that transforms any input instance x for A into a WS1S formula ϕ(i.e., $f(x) = \phi$) with the property that A outputs "1" on input x if and only if the formula is true. If there were a small circuit deciding whether a formula is true, then there would be a small circuit for the problem computed by A. As, by step 1, there is no such small circuit for A, it follows that there is no small circuit deciding whether a formula is true.

Step 3 involves a detailed analysis of the first two steps, in order to obtain the concrete numbers that appear in Theorem 1.

Let us focus on step 2. The function f is called a *reduction*. Any function g such that x is in B if and only if $g(x)$ is in C is said to *reduce* B to C. (This sort of reduction makes sense when the computational problems B and C are problems that require a "yes or no" answer; thus, they can be viewed as *sets*, where x is in B if B outputs "yes" on input x. Any computational problem can be viewed as a set this way. For example, computing a function h can be viewed as the set $\{(x, i) :$ the ith bit of $f(x)$ is $1\}$.)

Efficient reducibility provides a remarkably effective tool for classifying the computational complexity of a great many problems of practical importance. The amazing thing about the proof of step 2 (and this is typical of many theorems in complexity theory) is that it makes *no use at all* of the algorithm A, other than the fact that A uses at most 2^n memory locations. *Every* problem that uses at most this amount of memory is efficiently reducible to the problem of deciding whether a formula is true. This provides motivation for a closer look at the notion of efficient reducibility.

EFFICIENT COMPUTATION, POLYNOMIAL TIME

Many notions of efficient reducibility are studied in computational complexity theory, but without question the most important one is polynomial-time reducibility. The following considerations explain why this notion of reducibility arises.

Let us consider the informal notion of "easy" functions (functions that are easy to compute). Here are some conditions that one might want to satisfy, if one were trying to make this notion precise:

- If f and g are easy, then the composition $f \circ g$ is also easy.
- If f is computable in time n^2 on inputs of length n, then f is easy.

These conditions might seem harmless, but taken together, they imply that some "easy" functions take time n^{100} to compute. (This is because there is an "easy" function f that takes input of length n and produces output of length n^2. Composing this function with itself takes time n^4, etc.) At one level, it is clearly absurd to call a function "easy" if it requires time n^{100} to compute. However, this is precisely what we do in complexity theory! When our goal is to show that certain problems require *superpolynomial* running times, it is safe to consider a preprocessing step requiring time n^{100} as a "negligible factor".

A polynomial-time reduction f is simply a function that is computed by some program that runs in time bounded by $p(n)$ on inputs of length n, for some polynomial p. (That is, for some constant k, the running time of the program

computing f is at most $n^k + k$ on inputs of length n.) Note that we have not been specific about the programming language in which the program is written. Traditionally, this is made precise by saying that f is computed by a *Turing machine* in the given time bound, but exactly the same class of polynomial-time reductions results, if we use any other reasonable programming language, with the usual notion of running time. This is a side-benefit of our overly generous definition of what it means to be "easy" to compute.

If f is a polynomial-time reduction of A to B, we denote this $A \leq_m^p B$. Note that this suggests an ordering, where B is "larger" (i.e., "harder to compute") than A. Any efficient algorithm for B yields an efficient algorithm for A; if A is hard to compute, then B must also be hard to compute. If $A \leq_m^p B$ and $B \leq_m^p A$, this is denoted $A \equiv_m^p B$.

One thing that makes complexity theory useful is that naturally-arising computational problems tend to clump together into a shockingly small number of equivalence classes of the $\equiv_m^p$ relation. Many thousands of problems have been analyzed, and most of these fall into about a dozen equivalence classes, with perhaps another dozen classes picking up some other notably interesting groups of problems.

Many of these equivalence classes correspond to interesting time and space bounds. To explain this connection, first we need to talk about *complexity classes*.

COMPLEXITY CLASSES AND COMPLETE SETS

A *complexity class* is a set of problems that can be computed within certain resource bounds on some model of computation. For instance, P is the class of problems computable by programs that run in time at most $n^k + k$ for some constant k; "P" stands for "polynomial time." Another important complexity class is EXP: the set of problems computed by programs that run for time at most 2^{n^k+k} on inputs of length n. P and EXP are both defined in terms of time complexity. It is also interesting to bound the amount of memory used by programs; the classes PSPACE and EXPSPACE consist of the problems computed by programs whose space requirements are polynomial and exponential in the input size, respectively.

An important relationship exists between EXP and the game of checkers; this example is suitable to introduce the concepts of "hardness" and "completeness."

Checkers is played on an 8-by-8 grid. When the rules are adapted for play on a 10-by-10 grid, the game is known as "draughts" (which can also be played on boards of other sizes). Starting from any game position, there is an *optimal strategy*. The task of finding an optimal strategy is a natural computational problem. $N \times N$-*Checkers* is the function that takes as input a description of a $N \times N$ draughts board with locations of the pieces, and returns as output the move that a given player should make, using the optimal strategy. It is known that there is a program computing the optimal strategy for $N \times N$-*Checkers* that runs in time exponential in N^2; thus, $N \times N$-*Checkers* $\in$ EXP.

More interestingly, it is known that for every problem $A \in$ EXP, $A \leq_m^p N \times N$-*Checkers*. We say that $A \leq_m^p N \times N$-*Checkers* is *hard* for EXP (3).

More generally, if $\mathcal{C}$ is any class of problems, and B is a problem such that $A \leq_m^p B$ for every $A \in \mathcal{C}$, then we say that B is hard for $\mathcal{C}$. If B is hard for $\mathcal{C}$ and $B \in \mathcal{C}$, then we say that B is *complete* for $\mathcal{C}$. Thus, in particular, $N \times N$-*Checkers* is complete for EXP. This means that the complexity of $N \times N$-*Checkers* is well understood, in the sense that the fastest program for this problem cannot be too much faster than the currently known program. Here is why: We know (via a diagonalization argument) that there is some problem A in EXP that cannot be computed by any program that runs in time asymptotically less than 2^n. As $N \times N$-*Checkers* is complete for EXP, we know there is a reduction from A to $N \times N$-*Checkers* computable in time n^k for some k, and thus $N \times N$-*Checkers* requires running time that is asymptotically at least $2^{n^{1/k}}$.

It is significant to note that this yields only an *asymptotic* lower bound on the time complexity of $N \times N$-*Checkers*. That is, it says that the running time of any program for this problem must be very slow *on large enough inputs*, but (in contrast to Theorem 1) it says *nothing* about whether this problem is difficult for a given fixed input size. For instance, it is still unknown whether there could be a handheld device that computes optimal strategies for 100 $\times$ 100-Checkers (although this seems very unlikely). To mimic the proof of Theorem 1, it would be necessary to show that there is a problem in EXP that requires large circuits. Such problems are known to exist in EXPSPACE; whether such problems exist in EXP is one of the major open questions in computational complexity theory.

The complete sets for EXP (such as $N \times N$-*Checkers*) constitute one of the important $\equiv_m^p$-equivalence classes; many other problems are complete for PSPACE and EXPSPACE (and of course every nontrivial problem that can be solved in polynomial time is complete for P under $\leq_m^p$ reductions). However, this accounts for only a few of the several $\equiv_m^p$-equivalence classes that arise when considering important computational problems. To understand these other computational problems, it turns out to be useful to consider *unrealistic* models of computation.

UNREALISTIC MODELS: NONDETERMINISTIC MACHINES AND THE CLASS NP

Nondeterministic machines *appear* to be a completely unrealistic model of computation; if one could *prove* this to be the case, one would have solved one of the most important open questions in theoretical computer science (and even in all of mathematics).

A *nondeterministic Turing machine* can be viewed as a program with a special "guess" subroutine; each time this subroutine is called, it returns a random bit, zero or one. Thus far, it sounds like an ordinary program with a random bit generator, which does not sound so unrealistic. The unrealistic aspect comes with the way that we define how the machine produces its output. We say that a nondeterministic machine *accepts* its input (i.e., it outputs one) if there is some sequence of bits that the "guess" routine could return that causes the machine to output one; otherwise it is said to *reject* its input. If we view the "guess" bits as independent coin tosses, then the machine rejects its input

if and only if the probability of outputting one is zero; otherwise it accepts. If a nondeterministic machine runs for t steps, the machine can flip t coins, and thus, a nondeterministic machine can do the computational equivalent of finding a needle in a haystack: If there is even one sequence r of length t (out of 2^t possibilities) such that sequence r leads the machine to output one on input x, then the nondeterministic machine will accept x, and it does it in time t, rather than being charged time 2^t for looking at all possibilities.

A classic example that illustrates the power of nondeterministic machines is the Travelling Salesman Problem. The input consists of a labeled graph, with nodes (cities) and edges (listing the distances between each pair of cities), along with a bound B. The question to be solved is as follows: *Does there exist a cycle visiting all of the cities, having length at most B?* A nondeterministic machine can solve this quickly, by using several calls to the "guess" subroutine to obtain a sequence of bits r that can be interpreted as a list of cities, and then outputting one if r visits all of the cities, and the edges used sum up to at most B.

Nondeterministic machines can also be used to factor numbers; given an n-bit number x, along with two other numbers a and b with $a < b$, a nondeterministic machine can accept whether there is a factor of x that lies between a and b.

Of course, this nondeterministic program is of no use at all in trying to factor numbers or to solve the Traveling Salesman Problem on realistic computers. In fact, it is hard to imagine that there will ever be an efficient way to simulate a nondeterministic machine on computers that one could actually build. This is *precisely* why this model is so useful in complexity theory; the following paragraph explains why.

The class NP is the class of problems that can be solved by *nondeterministic* machines running in time at most $n^k + k$ on inputs of size n, for some constant k; NP stands for Nondeterministic Polynomial time. The Traveling Salesman Problem is one of many hundreds of very important computational problems (arising in many seemingly unrelated fields) that are *complete* for NP. Although it is more than a quarter-century old, the volume by Garey and Johnson (4) remains a useful catalog of NP-complete problems. The NP-complete problems constitute the most important $\equiv_m^p$-equivalence class whose complexity is unresolved. If any one of the NP-complete problems lies in P, then P = NP. As explained above, it seems much more likely that P is *not* equal to NP, which implies that any program solving any NP-complete problem has a worst-case running time greater than $n^{100,000}$ on all large inputs of length n.

Of course, even if P is not equal to NP, we would still have the same situation that we face with $N \times N$-*Checkers*, in that we would not be able to conclude that instances of some fixed size (say $n = 1{,}000$) are hard to compute. For that, we would seem to need the stronger assumption that there are problems in NP that require very large circuits; in fact, this is widely conjectured to be true.

Although it is conjectured that deterministic machines require exponential time to simulate nondeterministic machines, it is worth noting that the situation is very different when memory bounds are considered instead. A classic theorem of complexity theory states that a nondeterministic machine using space $s(n)$ can be simulated by a deterministic machine in space $s(n)^2$. Thus, if we define NPSPACE and NEXPSPACE by analogy to PSPACE and EXPSPACE using nondeterministic machines, we obtain the equalities PSPACE = NPSPACE and EXPSPACE = NEXPSPACE.

We thus have the following six complexity classes:

$$\mathrm{P} \subseteq \mathrm{NP} \subseteq \mathrm{PSPACE} \subseteq \mathrm{EXP} \subseteq \mathrm{NEXP} \subseteq \mathrm{EXPSPACE}$$

Diagonalization arguments tell us that $\mathrm{P} \neq \mathrm{EXP}$, $\mathrm{NP} \neq \mathrm{NEXP}$, and $\mathrm{PSPACE} \neq \mathrm{EXPSPACE}$. All other relationships are unknown. For instance, it is unknown whether P = PSPACE, and it is also unknown whether PSPACE = NEXP (although at most one of these two equalities can hold). Many in the community conjecture that all of these classes are distinct, and that no significant improvement on any of these inclusions can be proved. (That is, many people conjecture that there are problems in NP that require exponential time on deterministic machines, that there are problems in PSPACE that require exponential time on nondeterministic machines, that there are problems in EXP that require exponential space, etc.) These conjectures have remained unproven since they were first posed in the 1970s.

A THEORY TO EXPLAIN OBSERVED DIFFERENCES IN COMPLEXITY

It is traditional to draw a distinction between mathematics and empirical sciences such as physics. In mathematics, one starts with a set of assumptions and derives (with certainty) the consequences of the assumptions. In contrast, in a discipline such as physics one starts with external reality and formulates theories to try to explain (and make predictions about) that reality.

For some decades now, the field of computational complexity theory has dwelt in the uncomfortable region between mathematics and the empirical sciences. Complexity theory is a mathematical discipline; progress is measured by the strength of the theorems that are proved. However, despite rapid and exciting progress on many fronts, the fundamental question of whether P is equal to NP remains unsolved.

Until that milestone is reached, complexity theory can still offer to the rest of the computing community some of the benefits of an empirical science, in the following sense. All of our observations thus far indicate that certain problems (such as the Traveling Salesman Problem) are intractible. Furthermore, we can observe that with surprisingly few exceptions, natural and interesting computational problems can usually be shown to be complete for one of a handful of well-studied complexity classes. Even though we cannot currently *prove* that some of these complexity classes are distinct, the fact that these complexity classes correspond to natural or unnatural models of computation gives us an intuitively appealing explanation for *why* these classes appear to be distinct. That is, complexity theory gives us a vocabulary and a set of plausible conjectures that

helps explain our observations about the differing computational difficulty of various problems.

NP AND PROVABILITY

There are important connections between NP and mathematical logic. One equivalent way of defining NP is to say that a set A is in NP if and only if there are short *proofs* of membership in A. For example, consider the Traveling Salesman Problem. If there is a short cycle that visits all cities, then there is a short *proof* of this fact: Simply present the cycle and compute its length. Contrast this with the task of trying to prove that there is *not* a short cycle that visits all cities. For certain graphs this is possible, but nobody has found a general approach that is significantly better than simply listing all (exponentially many) possible cycles, and showing that all of them are too long. That is, for NP-complete problems A, it seems to be the case that the complement of A (denoted co-A) is not in NP.

The complexity class coNP is defined to be the set of all complements of problems in NP; $\text{coNP} = \{\text{co} - A : A \in \text{NP}\}$. This highlights what appears to be a fundamental difference between deterministic and nondeterministic computation. On a deterministic machine, a set and its complement always have similar complexity. On nondeterministic machines, this does not appear to be true (although if one could prove this, one would have a proof that P is different from NP).

To discuss the connections among NP, coNP, and logic in more detail, we need to give some definitions related to propositional logic. A propositional logic formula consists of variables (which can take on the values TRUE and FALSE), along with the connectives AND, OR, and NOT. A formula is said to be *satisfiable* if there is some assignment of truth values to the variables that causes it to evaluate to TRUE; it is said to be a *tautology* if *every* assignment of truth values to the variables causes it to evaluate to TRUE. SAT is the set of all satisfiable formulas; TAUT is the set of all tautologies. Note that the formula ϕ is in SAT if and only if "NOTϕ" is not in TAUT.

SAT is complete for NP; TAUT is complete for coNP. [This famous theorem is sometimes known as "Cook's Theorem" (5) or the "Cook-Levin Theorem" (6).]

Logicians are interested in the question of how to *prove* that a formula is a tautology. Many proof systems have been developed; they are known by such names as *resolution, Frege systems*, and *Gentzen calculus*. For some of these systems, such as resolution, it is known that certain tautologies of n symbols require proofs of length nearly 2^n(7). For Frege systems and proofs in the Gentzen calculus, it is widely suspected that similar bounds hold, although this remains unknown. Most logicians suspect that for *any* reasonable proof system, some short tautologies will require very long proofs. This is equivalent to the conjecture that NP and coNP are different classes; if every tautology had a short proof, then a nondeterministic machine could "guess" the proof and accept if the proof is correct. As TAUT is complete for coNP, this would imply that NP = coNP.

The P versus NP question also has a natural interpretation in terms of logic. Two tasks that occupy mathematicians are as follows:

1. Finding proofs of theorems.
2. Reading proofs that other people have found.

Most mathematicians find the second task to be considerably simpler than the first one. This can be posed as a computational problem. Let us say that a mathematician wants to prove a theorem ϕ and wants the proof to be at most 40 pages long. A nondeterministic machine can take as input ϕ followed by 40 blank pages, and "guess" a proof, accepting if it finds a legal proof. If P = NP, the mathematician can thus determine fairly quickly whether there is a short proof. A slight modification of this idea allows the mathematician to efficiently *construct* the proof (again, assuming that P = NP). That is, the conjecture that P is different than NP is consistent with our intuition that finding proofs is more difficult than verifying that a given proof is correct.

In the 1990s researchers in complexity theory discovered a very surprising (and counterintuitive) fact about logical proofs. Any proof of a logic statement can be *encoded* in such a way that it can be verified by picking a few bits at random and checking that these bits are sufficiently consistent. More precisely, let us say that you want to be 99.9% sure that the proof is correct. Then there is some constant k and a procedure such that, no matter how long the proof is, the procedure flips $O(\log n)$ coins and picks k bits of the encoding of the proof, and then does some computation, with the property that, if the proof is correct, the procedure accepts with probability one, and if the proof is incorrect, then the procedure detects that there is a flaw with probability at least 0.999. This process is known as a *probabilistically checkable proof*. Probabilistically checkable proofs have been very useful in proving that, for many optimization algorithms, it is NP-complete not only to find an optimal solution, but even to get a very rough approximation to the optimal solution.

Some problems in NP are widely believed to be intractable to compute, but are not believed to be NP-complete. Factoring provides a good example. The problem of computing the prime factorization of a number can be formulated in several ways; perhaps the most natural way is as the set FACTOR = $\{(x, i, b)$: the ith bit of the encoding of the prime factorization of x is $b\}$. By making use of the fact that primality testing lies in P (8), set FACTOR is easily seen to lie in NP $\cap$ coNP. Thus, FACTOR cannot be NP-complete unless NP = coNP.

OTHER COMPLEXITY CLASSES: COUNTING, PROBABILISTIC, AND QUANTUM COMPUTATION

Several other computational problems appear to be intermediate in complexity between NP and PSPACE that are related to the problem of *counting* how many accepting paths a nondeterministic machine has. The class #P is the class of functions f for which there is an NP machine M with the property that, for each string x, $f(x)$ is the number

of guess sequences r that cause M to accept input x. #P is a class of *functions*, instead of being a class of *sets* like all other complexity classes that we have discussed. #P is equivalent in complexity to the class PP (probabilistic polynomial time) defined as follows. A set A is in PP if there is an NP machine M such that, for each string x, x is in A if and only if more than half of the guess sequences cause M to accept x. If we view the guess sequences as flips of a fair coin, this means that x is in A if and only the probability that M accepts x is greater than one half. It is not hard to see that both NP and coNP are subsets of PP; thus this is not a very "practical" notion of probabilistic computation.

In practice, when people use probabilistic algorithms, they want to receive the correct answer with *high* probability. The complexity class that captures this notion is called BPP (bounded-error probabilistic polynomial time). Some problems in BPP are not known to lie in P; a good example of such a problem takes two algebraic circuits as input and determines whether they compute the same function.

Early in this article, we mentioned quantum computation. The class of problems that can be solved in polynomial time with low error probability using quantum machines is called BQP (bounded-error quantum polynomial time). FACTOR (the problem of finding the prime factorization of a number) lies in BQP (2). The following inclusions are known:

$$\mathrm{P} \subseteq \mathrm{BPP} \subseteq \mathrm{BQP} \subseteq \mathrm{PP} \subseteq \mathrm{PSPACE}$$
$$\mathrm{P} \subseteq \mathrm{NP} \subseteq \mathrm{PP}$$

No relationship is known between NP and BQP or between NP and BPP. Many people conjecture that neither NP nor BQP is contained in the other.

In contrast, many people now conjecture that BPP = P, because it has been proved that if there is any problem computable in time 2^n that requires circuits of nearly exponential size, then there is an efficient deterministic simulation of any BPP algorithm, which implies that P = BPP (9). This theorem is one of the most important in a field that has come to be known as *derandomization*, which studies how to simulate probabilistic algorithms deterministically.

INSIDE P

Polynomial-time reducibility is a very useful tool for clarifying the complexity of seemingly intractible problems, but it is of no use at all in trying to draw distinctions among problems in P. It turns out that some very useful distinctions can be made; to investigate them, we need more refined tools.

Logspace reducibility is one of the most widely used notions of reducibility for investigating the structure of P; a logspace reduction f is a polynomial-time reduction with the additional property that there is a Turing machine computing f that has (1) a read-only input tape, (2) a write-only output tape, and (3) the only other data structure it can use is a read/write worktape, where it uses only $O(\log n)$ locations on this tape on inputs of length n. If A is logspace-reducible to B, then we denote this by $A \leq_m^{log} B$. Imposing this very stringent memory restriction seems to place severe limitations on polynomial-time computation; many people conjecture that many functions computable in polynomial time are not logspace-computable. However, it is also true that the full power of polynomial time is not exploited in most proofs of NP-completeness. For essentially all natural problems that have been shown to be complete for the classes NP, PP, PSPACE, EXP, and so on using polynomial-time reducibility, it is known that they are also complete under logspace reducibility. That is, for large classes, logspace reducibility is essentially as useful as polynomial-time reducibility, but logspace reducibility offers the advantage that it can be used to find distinctions among problems in P.

Logspace-bounded Turing machines give rise to some natural complexity classes inside P: If the characteristic function of a set A is a logspace reduction as defined in the preceding paragraph, then A lies in the complexity class L. The analogous class, defined in terms of nondeterministic machines, is known as NL. The class #P also has a logspace analog, known as #L. These classes are of interest primarily because of their complete sets. Some important complete problems for L are the problem of determining whether two trees are isomorphic, testing whether a graph can be embedded in the plane, and the problem of determining whether an undirected graph is connected (10). Determining whether a *directed* graph is connected is a standard complete problem for NL, as is the problem of computing the length of the shortest path between two vertices in a graph. The complexity class #L characterizes the complexity of computing the determinant of an integer matrix as well as several other problems in linear algebra.

There are also many important complete problems for P under logspace reducibility, such as the problem of evaluating a Boolean circuit, linear programming, and certain network flow computations. In fact, there is a catalog of P-complete problems (11) that is nearly as impressive as the list of NP-complete problems (4). Although many P-complete problems have very efficient algorithms in terms of time complexity, there is a sense in which they seem to be resistent to extremely fast parallel algorithms. This is easiest to explain in terms of circuit complexity. The *size* of a Boolean circuit can be measured in terms of either the number of gates or the number of wires that connect the gates. Another important measure is the *depth* of the circuit: the length of the longest path from an input gate to the output gate. The problems in L, NL, and #L all have circuits of polynomial size and very small depth ($O(\log^2 n)$). In contrast, all polynomial-size circuits for P-complete problems seem to require a depth of at least $n^{1/k}$.

Even a very "small" complexity class such as L has an interesting structure inside it that can be investigated using a more restricted notion of reducibility than $\leq_m^{log}$ that is defined in terms of very restricted circuits. Further information about these small complexity classes can be found in the textbook by Vollmer (12).

We have the inclusions $L \subseteq NL \subseteq P \subseteq NP \subseteq PP \subseteq PSPACE$. Diagonalization shows that $NL \neq PSPACE$, but no other separations are known. In particular, it remains unknown whether the "large" complexity class PP actually coincides with the "small" class L.

TIME-SPACE TRADEOFFS

Logspace reducibility (and in general the notion of Turing machines that have very limited memory resources) allows the investigation of another aspect of complexity: the tradeoff between time and space. Take, for example, the problem of determining whether an undirected graph is connected. This problem can be solved using logarithmic space (10), but currently all "space-efficient" algorithms that are known for this problem are so slow that they will never be used in practice, particularly because this problem can be solved in linear time (using linear space) using a standard depth-first-search algorithm. However, there is no strong reason to believe that no fast small-space algorithm for graph connectivity *exists* (although there have been some investigations of this problem, using "restricted" models of computation, of the type that were discussed at the start of this article).

Some interesting time-space tradeoffs have been proved for the NP-complete problem SAT. Recall that it is still unknown whether SAT lies in the complexity class L. Also, although it is conjectured that SAT is not solvable in time n^k for any k, it remains unknown whether SAT is solvable in time $O(n)$. However, it *is* known that if SAT *is* solvable in linear time, then any such algorithm must use much more than logarithmic space. In fact, any algorithm that solves SAT in time $n^{1.7}$ must use memory $n^{1/k}$ for some k(13,14).

CONCLUSION

Computational complexity theory has been very successful in providing a framework that allows us to understand why several computational problems have resisted all efforts to find efficient algorithms. In some instances, it has been possible to prove very strong intractibility theorems, and in many other cases, a widely believed set of conjectures explains why certain problems appear to be hard to compute. The field is evolving rapidly; several developments discussed here are only a few years old. Yet the central questions (such as the infamous P vs. NP question) remain out of reach today.

By necessity, a brief article such as this can touch on only a small segment of a large field such as computational complexity theory. The reader is urged to consult the texts listed below, for a more comprehensive treatment of the area.

FURTHER READING

D.-Z. Du and K.-I. Ko, *Theory of Computational Complexity*. New York: Wiley, 2000.

L. A. Hemaspaandra and M. Ogihara, *The Complexity Theory Companion*. London: Springer-Verlag, 2002.

D. S. Johnson, A catalog of complexity classes, in J. van Leeuwen, (ed.), *Handbook of Theoretical Computer Science, Vol. A: Algorithms and Complexity*. Cambridge, MA: MIT Press, 1990, pp. 69–161.

D. Kozen, *Theory of Computation*. London: Springer-Verlag, 2006.

C. Papadimitriou, *Computational Complexity*. Reading, MA: Addison-Wesley, 1994.

I. Wegener, *Complexity Theory: Exploring the Limits of Efficient Algorithms*. Berlin: Springer-Verlag, 2005.

BIBLIOGRAPHY

1. L. Stockmeyer and A. R. Meyer, Cosmological lower bound on the circuit complexity of a small problem in logic, *J. ACM*, **49**: 753–784, 2002.
2. P. Shor, Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer, *SIAM J. Comput*. **26**: 1484–1509, 1997.
3. J. M. Robson, N by N Checkers is EXPTIME complete, *SIAM J. Comput.***13**: 252–267, 1984.
4. M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco, CA: Freeman, 1979.
5. S. Cook, The complexity of theorem proving procedures, *Proc. 3rd Annual ACM Symposium on Theory of Computing (STOC)*, 1971, pp. 151–158.
6. L. Levin, Universal search problems, *Problemy Peredachi Informatsii*, **9**: 265–266, 1973 (in Russian). English translation: B. A. Trakhtenbrot, A survey of Russian approaches to perebor (brute-force search) algorithms, *Ann. History Comput.*, **6**: 384–400, 1984.
7. A. Haken, The intractability of resolution, *Theor. Comput. Sci.*, **39**: 297–308, 1985.
8. M. Agrawal, N. Kayal, and N. Saxena, PRIMES is in P, *Ann. Math.*, **160**: 781–793, 2004.
9. R. Impagliazzo and A. Wigderson, P=BPP unless E has subexponential circuits, *Proc. 29th ACM Symposium on Theory of Computing (STOC)*, 1997, pp. 220–229.
10. O. Reingold, Undirected ST-connectivity in log-space, *Proc. 37th Annual ACM Symposium on Theory of Computing (STOC)*, 2005, pp. 376–385.
11. R. Greenlaw, H. J. Hoover, and W. L. Ruzzo, *Limits to Parallel Computation: P-Completeness Theory*. New York: Oxford University Press, 1995.
12. H. Vollmer, *Introduction to Circuit Complexity*. Berlin: Springer-Verlag, 1999.
13. L. Fortnow, R. Lipton, D. van Melkebeek, and A. Viglas, Time-space lower bounds for satisfiability, *J. ACM*, **52**: 835–865, 2005.
14. R. Williams, Better time-space lower bounds for SAT and related problems, *Proc. 20th Annual IEEE Conference on Computational Complexity (CCC)*, 2005, pp. 40–49.

ERIC ALLENDER
Rutgers University
Piscataway, New Jersey