Chapter 29

Computational Complexity

D.B. Shmoys and É. Tardos

School of Operations Research and Industrial Engineering,

Cornell University, Ithaca, NY 14853, USA

(January 1991)

Computational complexity theory attempts to understand the power of computation, by providing insight into the question why certain computational problems appear to be more difficult than others. Computation has added a dimension to the study of combinatorics. The theorem that, given a matching in a graph, *there exists* a larger matching if and only if there is an augmenting path, is not the complete answer; is it possible to *efficiently construct* a larger matching if one exists? Although such an algorithm is known for the matching problem, this is not the case for many combinatorial problems. Indeed, the greatest challenge confronting complexity theory is to provide techniques to prove that no efficient algorithm exists for a given problem.

Computational complexity theory provides the mathematical framework in which to discuss these questions, and while substantial progress has been made towards distinguishing the difficulty of computational problems, most of the basic issues remain unresolved. In this chapter, we will describe the fundamentals of this theory and give a brief survey of the results that have been obtained in its first quarter century. For a more detailed and complete exposition, the reader is referred to the textbooks by Garey & Johnson (1979) and Hopcroft & Ullman (1979) as well as to the more recent Handbook of Theoretical Computer Science edited by van Leeuwen (1990).

1 Complexity of Computational Problems

In this section, we will outline the essential machinery used to give formal meaning to the complexity of computational problems. This involves describing what is precisely meant by a computational problem, setting up a mathematical model of computation, and then formalizing the notion of the computational resources required for a problem with respect to that model. Unfortunately, there is no one standardized specification in which to discuss these questions. For this theory to produce meaningful results, it is essential that the definitions be robust enough so that theorems proved with respect to them apply equally to all reasonable variants of this framework.

Indeed, the particular definitions that we will rely on will be accompanied by evidence that these notions are sufficiently flexible.

Computational problems

Computation can be thought of as finding a suitable output for a given input. Therefore, a **computational problem** is specified by a relation between inputs and output; an algorithm to solve the problem takes an acceptable input, called an **instance**, and computes an output that satisfies the input-output relation; for example, given a directed graph, output a Hamiltonian circuit, if there is one, and otherwise indicate that none exists. This framework is very general, and we will focus attention on certain sorts of inputs and outputs.

The most common type of input (or output) is a string of characters over a finite alphabet. The previous example can be cast in this setting, since it is easy to represent a graph of order n as such a string of 0's and 1's; one might concatenate the rows of the $n \times n$ matrix $A = (a_{ij})$ where $a_{ij} = 1$ if and only if (i, j) is an arc. Alternatively, one might list the names of the nodes incident from node i, for $i = 1, \ldots, n$, where the node named j is encoded by the binary representation of j. Observe that the second representation will be more compact if the graph has few arcs, but this difference is limited, in that the length of the input in one format is at most the square of the length in the other.

Some combinatorial structures cannot be compactly represented as a string; for example, a matroid on n elements is most naturally represented by a string of length 2^n, where each character indicates whether a particular subset is independent. In such cases, it is customary to use an **oracle** to specify the input; the algorithm may write down queries, such as a particular subset to be tested for independence, and the oracle's answer may be used in the next step of the algorithm. Sometimes, there is no natural finite representation of the input, such as in the problem of optimizing over a convex body. For this example, one standard way to give the input is via an oracle that decides whether a given point is in the body, and if not, outputs a separating hyperplane. For any oracle, there is an associated parameter which is used as a measure of the size of the input.

In analyzing algorithms, we often view numbers as atomic units, without regard for their lengths, and so an input can also be a list of numbers, where the size of the input is the number of elements in the list. However, for the remainder of this chapter we will focus on inputs that are strings, although it is straightforward to extend the discussion of computational models and complexity to include these alternatives.

An important special case of a computational problem is a **decision problem**, where the output is restricted to either "yes" or "no", and for each input, there is exactly one related output. The set L of "yes" instances for a decision problem is often called the **language** associated with this problem. A decision version of the previous example is: given a directed graph, does it contain a Hamiltonian circuit? This type of problem will play a central role in our discussion, and it is important to realize that it is not a significant limitation to focus on it. For example, it is possible to answer the first **search** version of the Hamiltonian circuit problem as follows: for each arc in the graph, delete the arc, decide if the resulting graph is still Hamiltonian and replace the arc only in the case that the answer is "no". At the end of this procedure, the remaining graph is a circuit. Thus, we have shown that finding a circuit is not much harder than the decision problem, and this relationship remains true for all well-formulated decision versions of search problems.

Another important type of computational problem is an **optimization problem**; for example, given a directed graph G and two nodes s and t, we may wish to find the shortest path from s to t (or merely find the length). We shall rather treat these as decision problems by adding a bound b to the instance, and, for example, asking whether G has a path from s to t of length at most b. By using a **binary search** procedure that iteratively halves the range of possible optimal values, we see that an optimization problem can be solved with only somewhat more work than the corresponding decision problem.

Throughout this chapter, we will be dealing with computational problems involving a variety of structures, and we will not be specifying the nature of the encodings used. We will operate on the premise that any reasonable encoding produces strings of length that can be bounded by a polynomial of the length produced by any other encoding. When encoding numbers (which we will assume to be integral) there is an important distinction between the binary representation, which we will typically use, and the unary representation (*e.g.*, representing 5 as 11111). Notice that the latter could be exponentially bigger than the former, and thus size is a deceptive measure, since it makes instances of the problem larger than they need be. As we survey the complexity of computational problems that involve numbers, we will see that some are sensitive to this choice of encodings, whereas others are less affected.

Models of computation and computability

We next turn our attention to defining a mathematical model of a computer. In fact, we will present three different models, and although their superficial characteristics make them appear quite different, they will turn out to be formally equivalent. The first of these is the Turing machine, which is an extremely primitive model, and as a result, it is easier to prove results about what cannot be computed within this model. On the other hand, its extreme simplicity makes it ill-suited for algorithm design. As a result, it will be convenient to have an alternative model, the random access machine (RAM), within which to discuss algorithms.

The name, Turing machine, is a slight misnomer, since a Turing machine is a mathematical formulation of an algorithm, rather than a machine. A Turing machine $M = (Q, \Gamma, \delta, q_0, A)$ is a machine that has a finite main memory represented by a finite set of states Q, a read-only input tape, a finite set of work tapes each of which contains a countably infinite number of cells (corresponding to the integers) to store a character from a finite alphabet Γ, which at least includes the input alphabet $\{0, 1\}$ and a blank symbol B. For each of the k work tapes and the input tape, there is a "head" that can read one cell of the tape at a given time, and will be able to move cell-by-cell across the tape as the computation proceeds. Throughout the computation, the heads will read the contents of cells, and depending on what was read and the current state of the main memory, rewrite the cells and then move each head by one cell, either left or right, as well as cause a change in the state of the main memory. The basic step of a Turing machine, a **transition**, is modeled by a partial function

$$\delta : Q \times \Gamma^{k+1} \mapsto Q \times \Gamma^{k} \times \{L, R\}^{k+1}$$

that selects the new state, the contents of the cells currently scanned on the work tapes, and indicates the direction in which each head moves one cell as a **deterministic** function of the current state and the contents of the input and work tape cells currently being read. One may view the transition function as the program hardwired into this primitive machine. The computation is begun in state q_0, with the input head at the left end of the input, and all of the work tapes and the rest of the input tape blank. The machine **halts** if δ is undefined for the current state and the symbols read. An input is **accepted** if it halts in a state in the accepting state set $A \subseteq Q$. A Turing machine M solves a decision problem L if L is the set of inputs accepted by M, and M halts on every input; such a language L is said to be **decidable**. This definition of a Turing machine is similar to the one given by Turing (1936), and the reader should note that many equivalent definitions are possible.

We have defined a Turing machine so that it can only solve decision problems, but this definition can be easily extended to arbitrary computational problems by, for example, adding a write-only output tape, on which to print the output before halting. Although this appears to be a very primitive form of a computer, it has become routine to accept the following proposition.

Church's Thesis: Any function computed by an effective procedure can be computed by a Turing machine.

Although this is a *thesis*, in the sense that any attempt to characterize the inexplicit notion of effective procedure would destroy its intent, it is supported by a host of *theorems*, since for any known characterization of computable functions, it has been shown that these are Turing computable.

A random access machine (RAM) is a model of computation that is well-suited for specifying algorithms, since it uses an idealized, simplified programming language that closely resembles the assembly language of any modern-day digital computer. There is an infinite number of memory cells indexed by the integers, and there is no bound on the size of an integer that can be stored in any cell. A program can directly specify a cell to be read from or written in, without moving a head into position. Furthermore, there is an indirect addressing option, that uses the contents of a cell as an index to another cell that is then (seemingly randomly) accessed. All basic arithmetic operations can be performed. For further details on RAM's the reader is referred to Aho, Hopcroft & Ullman (1974).

Another model of computation closely tied to a practical setting is the logical circuit model. The simplicity of the circuit model makes it extremely attractive for proving lower bounds on the computational resources needed for particular functions, and research along these lines will be discussed in depth in Chapter 40. A circuit may be thought of as a directed acyclic graph, where the nodes of indegree 0 are the Boolean **input gates** (that can assume value 0 or 1), and the remainder correspond to **functional gates** of the circuit and are labeled with an operation, such as the logical or, negation, or logical and operations. The nodes of outdegree 0 are the **outputs**. A given circuit only handles inputs of a particular size, and so we specify a circuit for each input length. We say that the family of circuits solves a computational problem if for each input the corresponding circuit in the family generates a related output. Note that this model differs from the previous two, in that the circuit for inputs of length n can be tailored **nonuniformly** to the particular value of n, whereas a Turing machine or a RAM must run on inputs **uniformly**, independent of their length. We can make the notion of a family of circuits equivalent to the other models of computation by insisting that there be a Turing machine that, on input n, computes the description of the circuit for inputs of length n.

In spite of the apparent differences in these three models, any particular choice is one of convenience, and not of substance.

(1.1) Theorem. *The following classes of problems are identical:*

- *the class of computational problems solvable by a Turing machine;*
- *the class of computational problems solvable by a RAM.*
- *the class of computational problems solvable by a family of circuits that can be generated by a Turing machine.*

Theorem (1.1) is complemented by the following theorem, which is a consequence of the fact that there are an uncountable number of decision problems, but only a countable number of Turing machines.

(1.2) Theorem. *Not all decision problems are solvable by a Turing machine.*

The understanding of this inherent limitation on the power of computation was an outgrowth of results in mathematical logic. In particular, the first incompleteness theorem of Gödel (1931) contained the first sort of undecidability result and provided many of the essential ideas that would be used by Church, Post and Turing in their groundbreaking work on the nature of computation. In particular, Turing (1936) proposed what we call a Turing machine, and this enabled the discussion to be conveniently directed towards computation.

At the core of all of these results is Gödel's notion of encoding theorems as strings in some uniform way. Analogously, a Turing machine can be encoded as a string by first specifying the number of states that the machine has, followed by a list of all of the allowed transitions. Each such string can also be interpreted as an integer by using a binary encoding. Thus, each integer i represents a Turing machine M_i. Turing showed that the language $L_{hp} = \{i \mid M_i \text{ halts on input } i\}$ is not solvable by a Turing machine. His proof that this **halting problem** is undecidable uses the following diagonalization argument. Suppose that L_{hp} were solvable by a Turing machine M. Build another machine M' that first uses M to decide if the input $i \in L_{hp}$; if it is, then M' enters an infinite loop, and if not, M' halts. Of course, M' must be M_k for some integer k. But M' and M_k cannot accept the same language, since M' halts on k if M_k does not, and vice versa.

A problem L_1 (many-one) **reduces** to a problem L_2 if there exists a function f computable by a Turing machine such that $x \in L_1$ if and only if $f(x) \in L_2$. Note that if L_1 is undecidable and L_1 reduces to L_2, then L_2 must also be undecidable. This provides a strategy for proving additional undecidable problems. As a simple example, consider the language $L_e = \{i \mid M_i \text{ accepts on an empty input}\}$. It is a simple exercise to convert the description of a given Turing machine M_i to the description of another (rather trivial) machine $M' = M_j$ that on every input, first runs M_i with input i and accepts if M_i halts on i. Clearly, M_j accepts an empty input if and only if M_i halts on i.

A similar strategy can be used to prove Gödel's undecidability theorem in the context of Turing computability, although the details of the reduction are more involved. The theory of arithmetic for non-negative integers with addition and multiplication can be defined as follows. Consider first-order formulas that can be constructed from variables and the constants 0 and 1 with the logical connectives $\neg$, $\vee$, $\wedge$, $\rightarrow$, $\exists$, and $\forall$, along with the operations $\cdot$,$+$ and the binary relations $=$ and $<$.

A sentence is a formula in which all of the variables are bound. We consider the standard model of number theory (as defined by the Peano axioms). A sentence is **provable** if it can be deduced from these axioms. The theory of arithmetic $L_a = (\mathbf{Z}_+, +, \cdot, =, <, O, 1)$ is the collection of provable sentences. A relatively straightforward construction shows that L_e reduces to L_a, which yields the following fundamental result.

(1.3) Theorem. *L_a is undecidable.*

This theorem implies the incompleteness of this model of number theory, *i.e.*, there are sentences such that neither it nor its negation is provable. Every complete model is decidable, since a Turing machine can generate all possible deductions and stop if either the statement or its negation is proved.

These results were only the first steps in a rich area of research that can be viewed as the ancestor of modern-day complexity theory. One of its pinnacles of achievement is the solution of Hilbert's tenth problem, which asked for a procedure to decide if a given multivariate polynomial has an integer-valued root. Culminating years of progress in this area, Matijasevič proved that this problem is undecidable. (For an introduction to this history and the many results on which this proof builds, the reader is referred to the survey of Davis (1977).)

An important generalization of a Turing machine that will play a fundamental role in complexity theory is that of a **nondeterministic** Turing machine. Here, the transition function δ is no longer a function, but rather a relation, in that at each step there is a finite set of possible next moves of which exactly one is made. The notion of acceptance by a nondeterministic Turing machine is central to its definition: an input is accepted if there exists a sequence of transitions of δ that cause the machine to halt in an accepting state. A nondeterministic Turing machine M solves a decision problem L if L is the set of inputs accepted by M, and for every input M halts on each sequence of transitions. An equivalent formulation is to think of a nondeterministic Turing machine as a deterministic Turing machine with an additional **guess tape**, that is a read-only tape, where the head only moves to the right. The contents of the guess tape are magically constructed and presented to the machine as it begins the computation. For simplicity, we shall assume that the machine makes the same number of transitions for any guess. The set of inputs in L is the set of inputs for which there is a guess that allows the machine to halt in an accepting state. Note that a nondeterministic Turing machine accepting L can be converted into a deterministic machine for L by trying all of the (exponentially many) guesses. We can view a particular computation as proving (or disproving) the theorem, "$x \in L$"; in this context, the difference between determinism and nondeterminism is analogous to the difference between proving a theorem and verifying its proof.

Consider again the Hamiltonian circuit problem. A nondeterministic Turing machine for it might be constructed by letting the guess tape encode a sequence of n nodes of the graph. The Turing machine simply verifies that there is an arc between each consecutive pair of nodes in the guessed sequence, as well as between the first and last nodes. If the graph is Hamiltonian, then there is a correct guess, but otherwise, for any sequence of n nodes there will be some pair that is not adjacent. The correct guess, in essence the Hamiltonian circuit, is a **certificate** that the graph is Hamiltonian. Observe that this definition is one-sided, since the requirements for instances in L and not in L are quite different.

Computational resources and complexity classes

Now that we have mathematical formulations of both problems and machines, we can describe what is meant by the computational resources required to solve a certain problem.

In considering the execution of a deterministic Turing machine, it is clear that the number of transitions before halting corresponds to the running time of the machine on a particular input. In discussing the running time of an algorithm, within any of the models, we do not want to simply speak of particular instances, and so we must make some characterization of the running time for all instances. The criterion that we will focus on for most of this chapter is the worst-case running time as a function of the input size. When we say that a Turing machine takes n^2 steps, this means that for any input of size n, the Turing machine always halts within n^2 transitions. Unless otherwise specified, we will let $n = |x|$ denote the length of the input string x.

We will not be interested in the precise count of the number of transitions, but rather in the order of the running time. A function $f(n)$ is $O(g(n))$ if there are constants N and c such that for all $n \geq N$, $f(n) \leq cg(n)$. Thus, rather than say that a Turing machine has worst-case running time $3n^2 + 5n - 17$, we say simply that it is $O(n^2)$. This simplification makes it possible to discuss complicated algorithms without being overwhelmed by details. Furthermore, for any Turing machine with superlinear running time and any constant c, there exists another Turing machine to solve the same problem that runs c times faster than the original, that can be constructed by using an expanded work tape alphabet.

We will also use a notation analogous to $O(\cdot)$ to indicate lower bounds. A function $f(n)$ is $\Omega(g(n))$ if there are constants N and c such that for all $n \geq N$, $f(n) \geq cg(n)$. A function $f(n)$ is $\Theta(g(n))$ if it is both $O(g(n))$ and $\Omega(g(n))$.

For a nondeterministic Turing machine, we define the running time to be the number of transitions in any computation path generated by the input. (Recall that we added the restriction that all computation paths must have the same length.) For a circuit, we will want to characterize the number of operations performed as a measure of time, so that the relevant parameter is the **size** of the circuit, which is the order of the graph representing it. For a RAM, there are two standard ways in which to count the running time on a particular input. In each operation, a RAM can, for example, add two numbers of unbounded length. In the **unit-cost** model, this action takes one step, independent of the lengths of the numbers being added. In the **log-cost** model, this operation takes time proportional to the lengths of the numbers added. These measures can have radically different values. Take 17, and repeatedly square it k times. With each squaring, the number of bits in the binary representation essentially doubles. Thus, although we have taken only k steps in the unit-cost model, the time required according to the log-cost model is exponential in k. This may seem artificial, but this problem can occur, for example, in Gaussian elimination, if it is not implemented carefully. Technically, we will use the log-cost model, in order to ensure that the RAM is equivalent to the Turing machine in the desired ways. But, when speaking of the running time of an algorithm, it is traditional to state the running time in the unit-cost model, since for all standard algorithms one can prove that the pathological behavior of the above example does not come into play.

Time is not the only computational resource in which we will be interested; we will see that the space complexity of certain combinatorial problems gives more insight into the structure of these problems. In considering the space requirements, we will focus on the Turing machine model, and will only count the number of cells used on the work tapes of the machine. Furthermore, we will again be interested in the asymptotic worst-case analysis of the space used. As was true for time bounds, the space used by a Turing machine can be compressed by any constant factor. The space used by a nondeterministic Turing machine is the maximum space used on any computation path. For circuits, it will also be interesting to study their **depth**, the longest path from a node of indegree 0 to one of outdegree 0.

The notion of the **complexity** of a problem is the order of a given computational resource, such as time, that is necessary and sufficient to solve the problem. Consider the following **directed reachability problem**: given a directed graph G, and two specified nodes s and t, does there exist a path from s to t? When we say that the complexity of the directed reachability problem for a graph with m arcs is $\Theta(m)$, this means that there is a (unit-cost RAM) algorithm that has worst-case running time $O(m)$ and there is no algorithm with running time of lower order. Tight results of this kind are extremely rare, since the tremendous progress in the design of efficient algorithms has not been matched, or even approached, by the slow progress in techniques for proving lower bounds on the complexity of these problems in general models of computation. For example, consider the **3-colorability problem**: given an undirected graph G with m edges, can the nodes be colored with three colors so that no two adjacent nodes are given the same color; *i.e.*, is $\chi(G) \leq 3$? The best lower bound is only $\Omega(m)$, in spite of substantial evidence that it cannot be solved in time bounded by a polynomial.

In order to study the relative power of particular computational resources, we introduce the notion of a **complexity class**, which is the set of problems that have a specified upper bound on their complexity. It will be convenient to define the complexity class $\boldsymbol{DTIME(T(n))}$ to be the set of all languages L that can be recognized by a deterministic Turing machine within time $O(T(n))$. $\boldsymbol{NTIME(T(n))}$ denotes the analogous class of languages for nondeterministic Turing machines. Throughout this chapter, it will be convenient to make certain assumptions about the sorts of time bounds that define complexity classes. A function $T(n)$ is called **fully time-constructible** if there exists a Turing machine that halts after exactly $T(n)$ steps on any input of length n. All common time bounds, such as $n \log n$ or n^2, are fully time-constructible. We will *implicitly* assume that any function $T(n)$ used to define a time-complexity class is fully time-constructible.

The single most important complexity class is $\boldsymbol{\mathcal{P}}$, the class of decision problems solvable in polynomial time. Two of the most well-known algorithms, the Euclidean algorithm for finding the greatest common divisor of two integers and Gaussian elimination for solving a system of linear equations, are classical examples of polynomial-time algorithms. In fact, Lamé observed as early as 1844 that the Euclidean algorithm was a polynomial-time algorithm. In 1953, Von Neumann contrasted the running time for an algorithm for the assignment problem that "turn[ed] out [to be] a moderate power of n, *i.e.*, considerably smaller than the 'obvious' estimate $n!$" for a complete enumeration of the solutions. Edmonds (1965) and Cobham (1965) were the first to introduce $\mathcal{P}$ as an important complexity class, and it was through the pioneering work of Edmonds that polynomial solvability became recognized as a theoretical model of efficiency. With only a few exceptions, the discovery of a polynomial-time algorithm has proved to be a important first step in the direction of finding truly efficient algorithms. Polynomial time has proved to be very fruitful as a theoretical model of efficiency both in yielding a deep and interesting theory of algorithms and in designing efficient algorithms.

There has been substantial work over the last 25 years in finding polynomial-time algorithms for combinatorial problems. It is a testament to the importance of this development that much of this handbook is devoted to discussing these algorithms. This work includes algorithms for graph connectivity and network flow (see Chapter 2), for graph matchings (see Chapter 3), for matroid problems (see Chapter 11), for point lattice problems (see Chapter 19), for testing isomorphism (see Chapter 27), for finding disjoint paths in graphs (see Chapter 5) as well as for problems connected with linear programming (see Chapters 28 and 30).

Another, more technical reason for the acceptance of $\mathcal{P}$ as the theoretical notion of efficiency, is its mathematical robustness. Recall the discussion of encodings where we remarked that any reasonable encoding will have length bounded by a polynomial in the length of another. As a result, any polynomial-time algorithm which expects its input in one form can be converted to a polynomial-time algorithm for the other. In particular, note that the previous discussion of the two different encodings of a graph can be swept aside and we can assume that the size of the input for a graph of order n is n. Notice further that the informal definition of $\mathcal{P}$ given above does not rely on any model of (deterministic) computation. One justification for this statement is the following theorem.

(1.4) Theorem. *The following classes of problems are identical:*

- *the class of computational problems solvable by a Turing machine in polynomial-time;*
- *the class of computational problems solvable by a RAM in polynomial-time under the log-cost measure;*
- *the class of computational problems solvable by a family of circuits of polynomial size, where the circuit for inputs of size n can be generated by a Turing machine with running time bounded by a polynomial in n.*

The importance of the class $\boldsymbol{\mathcal{NP}} = \cup_k NTIME(n^k)$ is due to the wide range of important problems that are known to be in the class, and yet are not known to lie in $\mathcal{P}$. For example, the nondeterministic algorithm given for the Hamiltonian circuit problem is clearly polynomial, and so this problem lies in $\mathcal{NP}$. However, it is not known whether this, or any problem is in $\mathcal{NP} \setminus \mathcal{P}$, and this is, undoubtably the central question in complexity theory.

Open Problem Is $\mathcal{P} = \mathcal{NP}$?

The following reformulation of $\mathcal{NP}$ is often useful: $L \in \mathcal{NP}$ if there exists a language $L' \in \mathcal{P}$ and a polynomial $p(n)$ such that $x \in L \Leftrightarrow \exists y$ such that $|y| = p(|x|)$ and $(x, y) \in L'$. (We will denote this polynomially bounded quantification by $\exists_p$.)

For each decision problem L, there is a complementary problem, $\bar{L}$, such as the problem of recognizing non-Hamiltonian graphs. For any complexity class $\mathcal{S}$, let $\boldsymbol{co - \mathcal{S}}$ denote the class of languages whose complement is in $\mathcal{S}$. The definition of $\mathcal{P}$ is symmetric with respect to membership and non-membership in L, so that $\mathcal{P} = co - \mathcal{P}$. $\mathcal{NP}$ is very different in this respect. In fact, it is unknown whether the Hamiltonian circuit problem is in $co - \mathcal{NP}$.

Open Problem Is $\mathcal{NP} = co - \mathcal{NP}$?

Edmonds (1965) brought attention to the class $\mathcal{NP} \cap co - \mathcal{NP}$, and called problems in this class **well-characterized**, since there is both a short certificate to show that the property holds, as well as a short certificate that it does not. Edmonds was working on algorithms for non-bipartite maximum matching at the time, and this problem serves as a good example of a problem in this class. If the instance consists of a graph G and a bound k and we wish to know if there is a matching of size at least k, the matching itself serves as a certificate for an instance in L, whereas an odd-set cover serves as a certificate for an instance not in L (see Chapter 3). Note that there is a min-max theorem characterizing the size of the maximum matching that is at the core of the fact that matching is in $\mathcal{NP} \cap co - \mathcal{NP}$, and indeed min-max theorems often serve this role. As mentioned above, matching is known to be in $\mathcal{P}$, and this raises the following question.

Open Problem Is $\mathcal{P} = \mathcal{NP} \cap co - \mathcal{NP}$?

We will also be concerned with complexity classes defined by the space complexity of problems. As for time, let $\boldsymbol{DSPACE(S(n))}$ and $\boldsymbol{NSPACE(S(n))}$ denote, respectively, the class of languages accepted by deterministic and nondeterministic Turing machines within space $O(S(n))$. We will implicitly assume the following condition for all space bounds used to define complexity classes: a function $S(n)$ is **fully space-constructible** if there is a Turing machine that, on any input of length n delimits $S(n)$ tape cells and then halts. Three space complexity classes will receive the most prominent attention:

- $\boldsymbol{\mathcal{L}} = DSPACE(\log n)$;
- $\boldsymbol{\mathcal{NL}} = NSPACE(\log n)$; and
- $\boldsymbol{\mathcal{PSPACE}} = \cup_k DSPACE(n^k)$.

One might be tempted to add a fourth class, $\mathcal{NPSPACE}$, but we shall see that nondeterminism does not add anything in this case. We will see that the chain of inclusions

$$\mathcal{L} \subseteq \mathcal{NL} \subseteq \mathcal{P} \subseteq \mathcal{NP} \subseteq \mathcal{PSPACE}$$

holds, and the main thrust of complexity theory is to understand which of these inclusions is proper. At the extremes, a straightforward diagonalization argument due to Hartmanis, Lewis & Stearns shows that $\mathcal{L} \neq \mathcal{PSPACE}$, and a result of Savitch further implies that $\mathcal{NL} \neq \mathcal{PSPACE}$, but after nearly a quarter century more work, these are the only sets in this chain known to be distinct.

Randomized computation

In this subsection, we will consider models of computation that exploit the power of randomization in the design and analysis of algorithms *without* making probabilistic assumptions about the inputs. A randomized algorithm is an algorithm that can flip coins during the computation; that is, we consider a fixed input, and study the algorithm's behavior as a random variable depending only on the coin flips used. For the algorithms discussed below, the algorithm is allowed to make mistakes, but for each input the probability of error must be very small.

The most well-known randomized algorithm is for testing primality. In the **primality testing problem**, we are given a natural number N, and we wish to decide if it is prime. It is not known whether primality testing is in $\mathcal{P}$. The input size of the number N is $\log N$, and therefore algorithms that simplistically search for divisors of N do not run in polynomial time. Consider Fermat's theorem: if N is prime then a^N is congruent to a modulo N for every integer a. This provides a way to conclude that a number is not prime without actually exhibiting a factor. That is, if we find an integer a such that a^N is not congruent to a modulo N, denoted $a^N \not\equiv a \bmod N$, then we can conclude that N is not prime. (Note that $a^N \bmod N$ can be computed in polynomial time by repeatedly squaring modulo N.) Let such an a be called a **witness** for N's compositeness. The advantage of this kind of witness, compared to exhibiting a divisor, is that if there exists an integer a such that $a^N \not\equiv a \bmod N$, then at least half of the integers in the range from 1 to N have this property.

Unfortunately, there are composite numbers, the so-called **Carmichael numbers** that are not prime, but for which no witness exists. If we momentarily forget about the existence of these numbers, we get the following algorithm for testing primality: given an integer N, choose an integer a in the range 1 to N at random, and check if a is a witness for N. If a witness is found, then we know that N is not prime (and not even a Carmichael number). On the other hand, if N is not a prime (and also not a Carmichael number), then a random a is a witness with probability at least one half. Running this test k times with independent random choices, we either find a witness or can fairly safely conclude that no witness exists (with error probability 2^{-k}). This gives a randomized polynomial-time algorithm to recognize the language of all primes and Carmichael numbers. Rabin and Solovay & Strassen, by using a somewhat more sophisticated variant of Fermat's theorem, gave randomized polynomial-time algorithms that accept the language of all primes.

The above idea actually gives a polynomial-time algorithm if the extended Riemann hypothesis is true. Miller has proved that if the extended Riemann hypothesis holds, then there exists a witness for N (in more or less the above sense) that is at most $O((\log N)^2)$. By trying all the integers up to this limit, we would get a polynomial-time deterministic algorithm for primality testing. For more details on this and other number-theoretic algorithms see the survey of Lenstra & Lenstra (1990).

The formal definition of a **randomized Turing machine** is similar to the definition of a nondeterministic Turing machine in the sense that at every point during the computation there could be several different next steps. Randomized Turing machines have a read-only **randomizing tape** similar to the guess tape of the nondeterministic Turing machine. We can think of this tape as providing the outcomes of the coin flips to be used by the algorithm. We shall assume that for a given input length n, the algorithm reads a fixed number of bits, $f(n)$, from the randomizing tape. The **probability** that the randomized Turing machine accepts an input x of length n is defined to be the fraction of all possible strings of length $f(n)$ that, when used as the initial segment of the randomizing tape, cause the Turing machine to accept x. For the randomized Turing machine, we take the simplifying approach that the running time for an input is the maximum number of transitions in some sequence of allowed transitions (*i.e.*, for some contents of the randomizing tape). Unlike the nondeterministic Turing machine, a randomized Turing machine is not just a convenient mathematical model; randomized algorithms can be implemented in practical settings.

We define $\boldsymbol{\mathcal{BPP}}$, the class of languages accepted by a randomized polynomial-time algorithm, as follows. A language L is in $\mathcal{BPP}$ if there exists a polynomial-time randomized Turing Machine that accepts each $x \in L$ with probability at least 2/3; and rejects each $x \notin L$ with probability at least 2/3. We can think of the outcomes of the computation as follows: if the Turing machine accepts x this means that "x is probably in L", whereas if it rejects, that means that "x is probably not in L". Note that the choice of the number 2/3 in the definition was rather arbitrary: if we run the algorithm k times independently, and take the majority decision, we can decrease the probability of error exponentially in k. If k is fairly large, then one can accept the answer given by the algorithm without any reasonable shadow of doubt. (The letters BPP stand for a Probabilistic

Polynomial-time algorithm with probabilities Bounded away from 1/2.)

Other problems not known to be in $\mathcal{P}$ for which there is a randomized polynomial-time algorithm include computing the square root of an integer x modulo a prime p, and deciding whether the determinant of a matrix whose entries are multivariate polynomials is the zero polynomial. The algorithm for the latter problem assigns random values to the variables in the polynomials, and computes the resulting determinant (of numbers). If this determinant is non-zero, then certainly the determinant of variables is non-zero as a polynomial. On the other hand, if the determinant of variables is non-zero, then a random evaluation will yield a non-zero determinant with very high probability. This can be used to show that given a graph G, a subset of edges F and an integer k, determining whether a graph has a perfect matching with exactly k edges in F can be solved by a randomized polynomial-time algorithm (see Chapter 3).

Notice that the randomized primality testing algorithm has a property stronger than required by the formal definition. The conclusion that N is not a prime was certain; uncertainty arose only in the case of the conclusion, "N is probably prime". The complexity class $\boldsymbol{\mathcal{RP}}$ is defined to reflect this asymmetry. A language L is in $\mathcal{RP}$ if there exists a polynomial-time randomized Turing Machine RM such that each input that RM can accept (along any computation path) is in L and for each input $x \in L$, the probability that RM accepts x is at least 1/2. Note again that the choice of the number 1/2 is arbitrary.

The above mentioned algorithms show that primality testing is in $co-\mathcal{RP}$. There are very few problems known to be in $\mathcal{BPP}$ but not in $\mathcal{RP}$ or $co-\mathcal{RP}$. Bach, Miller & Shallit provided the first "natural" examples of problems in $\mathcal{BPP}$ that are not obviously in $\mathcal{RP}$ or $co-\mathcal{RP}$. They proved that the set of perfect numbers is in $\mathcal{BPP}$. (A natural number N is **perfect** if the sum of all its natural divisors is $2N$; for example, 6 is perfect.)

In some sense, an $\mathcal{RP}$ algorithm is more satisfying than a $\mathcal{BPP}$ algorithm, since at least one of the two conclusions reached can be claimed with certainty. A randomized algorithm that *never makes mistakes* would be even more desirable. This can be defined in the following way: an algorithm to compute a function $f(x)$ is a **Las Vegas** algorithm if, given an input x, this randomized algorithm either correctly computes $f(x)$, or it halts without coming to a conclusion, and the probability of the latter outcome is less than 1/2 for each input x. It is easy to see that a language L is in $\mathcal{RP} \cap co-\mathcal{RP}$ if and only if there exists a polynomial-time Las Vegas algorithm to decide membership in L. Observe also that if we repeat any Las Vegas algorithm until it gives an answer, the resulting algorithm always gives the correct answer and for any input, it is expected to run in polynomial time. Recent results of Goldwasser & Kilian and Adleman & Huang yield a sophisticated Las Vegas algorithm for testing primality.

There is evidence that suggests that randomized polynomial time is not that different from $\mathcal{P}$. For example, consider a nonuniform analog of $\mathcal{P}$ by considering the class of languages accepted by a family of polynomial-size circuits. Alternatively, one can make the Turing machine model nonuniform, by allowing the machine free access to a prespecified polynomial-length advice string s_n, when processing any input of length n. This class is denoted $\boldsymbol{\mathcal{P}/poly}$. For any language L in $\mathcal{BPP}$, we can assume without loss of generality that there is a machine RM that accepts each $x \in L$ with probability $1-2^{-(n+1)}$ and rejects each $x \notin L$ with probability $1-2^{-(n+1)}$, where n denotes the length of x (since taking the majority decision of repeated trials decreases the error probability exponentially). However, this implies that the probability that a random string used by RM for an input of length n would work correctly for *all* strings of length n is at least $1/2$. Thus, there must exist such a good string to serve as the advice, and we have shown the following theorem, which is based on an idea of Adleman.

(1.5) Theorem. $\mathcal{BPP} \subseteq \mathcal{P}/poly$.

2 Shades of Intractability

In this section we will consider many computational problems, and see that the universe does not appear to be divided simply into tractable and intractable problems. Current evidence suggests that there are a variety of different classes of problems, each characterizing its own particular shade of intractability. Much of the work in complexity theory is aimed at understanding the correct framework in which to place these problems.

The subsections here reflect three types of approaches for characterizing the difficulty of these problems. The nicest sort of result places absolute limits on our ability to solve problems; for example, the most severe limit is to show that a problem is undecidable, and among decidable problems there are only a handful that can be proven intractable, in the sense that they require a certain (non-trivial) amount of time or space to be solved. Much more common is to provide a completeness result to show that a particular problem is a hardest problem within a given complexity class. If the class contains a great number of problems not known to be solvable with more modest resources, this provides evidence that the problem is intractable. Finally, in order to better understand a problem, it has frequently been useful to strengthen the basic Turing machine model in order define complexity classes that better characterize the problem. Such an alternative view has often made problems appear less intractable; the subsections on the polynomial-time hierarchy and on randomized proofs present results in this direction.

Evidence of intractability: $\mathcal{NP}$-completeness

The lack of lower bounds with respect to a general model of computation for either space or time complexity has led to the search for other evidence that suggests that lower bounds hold. One such type of evidence might be the implication: if the Hamiltonian circuit problem is in $\mathcal{P}$, then $\mathcal{P} = \mathcal{NP}$. $\mathcal{NP}$ contains a tremendous variety of problems that are not known to be in $\mathcal{P}$, and so by proving such a claim, one shows that the Hamiltonian circuit problem is a hardest problem in $\mathcal{NP}$, in that any polynomial-time algorithm to solve it would in fact solve thousands of other problems.

The principal tool in providing evidence of this form is that of reduction. We will say that a problem L_1 **polynomial-time reduces** to L_2 if there exists a polynomial-time computable function f that maps instances of L_1 into instances of L_2 such that x is a "yes" instance of L_1 if and only if $f(x)$ is a "yes" of L_2. We shall denote this by $L_1 \propto_p L_2$. Notice that if there were a polynomial-time algorithm for L_2, we could then obtain a polynomial-time algorithm for L_1 by first computing $f(x)$ and then running the assumed algorithm on $f(x)$. This composite procedure is polynomial-time, since the composition of two polynomials is itself a polynomial.

Definition A problem L_1 is **$\boldsymbol{\mathcal{NP}}$-complete** if

1. $L_1 \in \mathcal{NP}$;

2. for all $L \in \mathcal{NP}$, $L \propto_p L_1$.

The composition argument given above yields the following results.

(2.1) Theorem. *For any $\mathcal{NP}$-complete problem L, $L \in \mathcal{P}$ if and only if $\mathcal{P} = \mathcal{NP}$.*

(2.2) Theorem. *If L_1 is $\mathcal{NP}$-complete, $L_2 \in \mathcal{NP}$ and $L_1 \propto_p L_2$, then L_2 is $\mathcal{NP}$-complete.*

The first result says that any $\mathcal{NP}$-complete problem completely characterizes $\mathcal{NP}$ in its relationship to $\mathcal{P}$, and that we can focus on any such problem without loss of generality in trying to prove that the two classes are different. The Hamiltonian circuit problem is $\mathcal{NP}$-complete, and section we will prove that several combinatorial problems are among the plethora having this property. The second result gives a strategy for proving that a problem is $\mathcal{NP}$-complete, provided that a "first" $\mathcal{NP}$-complete problem is known. Of course, to initiate this strategy, we must show that some natural problem has this property, and it was a landmark achievement in complexity theory when Cook (1971) showed that the problem of deciding the satisfiability of a formula in propositional logic is $\mathcal{NP}$-complete. The importance of this result was only fully recognized through the work of Karp (1972), whose seminal paper showed 21 well-known combinatorial problems to be $\mathcal{NP}$-complete. Independently, Levin (1973) discovered the same approach to studying the intractability of computational problems.

A Boolean formula in conjunctive normal form is the conjunction (and) of clauses $C_1, \ldots, C_s$, each of which is a disjunction (or) of literals $x_1, \bar{x}_1, \ldots, x_t, \bar{x}_t$, where each x_i is a Boolean variable and $\bar{x}_i$ denotes its negation. In the **satisfiability problem** (SAT) we are given such a Boolean formula, and asked to decide if there exists a truth assignment for the variables such that the formula evaluates to true.

(2.3) Theorem. *The satisfiability problem is $\mathcal{NP}$-complete.*

Proof: It is easy to see that the satisfiability problem is in $\mathcal{NP}$, since we can interpret the first t cells of the guess tape as providing the assignment, and then it is a simple matter to evaluate the formula for that assignment in polynomial time.

Next, we must show that for any language $L \in \mathcal{NP}$ there exists a polynomial-time function f that maps each instance x of the original problem into a Boolean formula such that $x \in L$ if and only if $f(x)$ is satisfiable. Before giving the reduction, we first argue that M may be assumed to have a form somewhat simpler than the original definition. Imagine that the Turing machine has only one tape, which serves both as the input tape and as the work tapes. A simple construction shows that if there exists a nondeterministic Turing machine M with running time $T(n)$, then there exists a nondeterministic machine of this simpler form that finishes within time $T^2(n)$ (by enlarging the alphabet so that each symbol denotes one character on all of the tapes of M). Furthermore, we can assume without loss of generality that for some l the machine M runs for exactly time n^l on every input of length n.

Let M be such a simplified machine that runs on input x for T steps before halting. We can describe the configuration of the machine at any instant of the computation by giving the contents of the tape, the position of the head and the current state. This can all be encoded as a string by using the alphabet $\Gamma' = \Gamma \times (Q \cup \{!\})$ where the first coordinate gives the contents of a cell of the tape, and the second is '!' unless the head is reading that cell of the tape, when it is the current state. We can then encode the entire computation as a matrix, where each row of the matrix corresponds to one step of the computation, and each column corresponds to a cell of the tape. (Note that there are no more than T cells in either direction of the initial head position that could be reached during the computation.) Acceptance of x by M boils down to the following question: does there exist a guess that causes the matrix to be filled in so that in the last row, the configuration contains an accepting state?

It is straightforward to construct a Boolean formula that represents this question. Let $g_1, \ldots, g_T$ be variables that represent the binary values of the guess tape. Let a_{ijk} represent the contents of the ijth cell of the matrix in the sense that it is 1 if and only if it contains the kth character of Γ'. The formula will be a conjunction of pieces that correspond to the following conditions that we wish to impose: the variables represent some matrix, in that exactly one character is stored in each entry; the first row corresponds to the initial configuration for input x; the last row contains an accepting state; the computation proceeds in the deterministic way specified by the guesses g_i. We will not give each of these in detail, but only sketch the main ideas. The first is easy: for each of the $O(T^2)$ entries, check that at least one of the associated variables is 1 (by their or) and for each pair, check that not both are 1 (by the or of their negations). The second and third are equally routine. The last condition takes a bit more work and is based on a principle of locality: if locally the computation (*i.e.*, the matrix) appears correct, then the entire computation was performed correctly. In fact, it is sufficient to check that each 2×2 submatrix appears correct. Furthermore, it is an easy exercise to encode that some 2×2 submatrix in the ith and $i+1$st rows behaves according to the guess g_i. By taking the conjunction of all $O(T^2)$ such pieces of local information, we enforce that the computation is done correctly. It is now routine to verify that formula constructed is satisfiable if and only if there are guesses that lead M to accept x. □

Now that we have our initial $\mathcal{NP}$-complete problem, we proceed to give a number of reductions to show that several important combinatorial problems are $\mathcal{NP}$-complete. Literally thousands of problems are now known to be $\mathcal{NP}$-complete, so that we will only present a small handful of examples that serve to illustrate an important phenomenon in complexity theory, or relate to important combinatorial problems discussed elsewhere in this chapter, as well as in the rest of this volume. Most of the problems that we consider were shown to be $\mathcal{NP}$-complete in the pioneering work of Karp (1972).

Many restricted cases of the satisfiability problem are also $\mathcal{NP}$-complete. One that is often used in further $\mathcal{NP}$-completeness proofs is the **3-SAT** problem, where each clause of the conjunctive normal form must contain exactly three literals. It is a simple task to show that by adding additional variables, longer clauses can be broken into clauses of length three, yielding a new formula that can be satisfied if and only if the original can.

For the **stable set problem**, we are given a graph G and a bound k, and asked to decide if $\alpha(G) \geq k$; that is, do there exist k pairwise non-adjacent nodes in G? It is easy to see that this problem is in $\mathcal{NP}$, and to show that it is complete, we reduce 3-SAT to it and invoke Theorem (2.2). Given a 3-SAT instance ϕ, we construct a graph G as follows: for each clause in ϕ, let there be 3 nodes in G, each representing a literal in the clause, and let these 3 nodes induce a **clique** (*i.e.*, they are pairwise adjacent); complete the construction by making adjacent any pair of nodes that represent a literal and its negation, and set k to be the number of clauses in ϕ. If there is a satisfying assignment for ϕ, pick one literal from each clause that is given the assignment true; the corresponding nodes in G form a stable set of size k. If there is a stable set of size k, then it must have exactly one node in each clique corresponding to a clause. Furthermore, the nodes in the stable set cannot correspond to both a literal and its negation, so that we can form an assignment by setting to true all literals selected in the stable set, and extending this assignment to the remaining variables arbitrarily. This is a satisfying assignment. This is a characteristic reduction, in that we build **gadgets** to represent the variables and clause structure within the framework of the new problem. The $\mathcal{NP}$-completeness of two other problems follow immediately: the **clique problem**, given a graph G and a bound k, decide if there is a clique in G of size k; and the **node cover problem**, given a graph G and a bound k, decide if there exists a set of k nodes such that every edge is incident to a node in the set. A somewhat more complicated reduction transforms 3-SAT into the Hamiltonian circuit problem to show that to be $\mathcal{NP}$-complete. A seemingly slight generalization of bipartite graph matching, the **3-dimensional matching problem**, can be shown to be $\mathcal{NP}$-complete: given disjoint node sets A, B and C, and a collection $\mathcal{F}$ of hyperedges of the form, (a, b, c) where $a \in A$, $b \in B$ and $c \in C$, does there exist a subset of $\mathcal{F}$ such that each node is contained in exactly one edge in the subset?

If we restrict the stable set problem to a particular constant value of k (*e.g.*, is $\alpha(G) \leq 100$?), then this problem can be solved in $\mathcal{P}$ by enumerating all possible sets of size 100. In contrast to this, Stockmeyer has shown that the 3-colorability problem is $\mathcal{NP}$-complete, by reducing 3-SAT to it. Let ϕ be a 3-SAT formula. We construct a graph G from it in the following way. First, construct a "reference" clique on three nodes, called true, false and undefined; these nodes will serve as a way of naming the colors in any 3-coloring of the graph. For each variable in ϕ, construct a pair of adjacent nodes, one representing the variable, and one representing its negation, and make them both adjacent to the undefined node. For each clause, $l_1 \vee l_2 \vee l_3$, construct the subgraph shown in Figure 1 below, where the nodes labeled with literals, as well as false(F) and undefined(U), are the nodes already described. It is easy to see that if ϕ has a satisfying assignment, we can get a proper 3-coloring of this graph as follows: color the nodes corresponding literals that are true in the assignment with the same color as is given node true; color analogously the nodes for false literals; and then extend this coloring to the remaining nodes in a straightforward manner. Furthermore, it involves only a little case-checking to see that if the graph is 3-colorable, then the colors can be interpreted as a satisfying assignment.

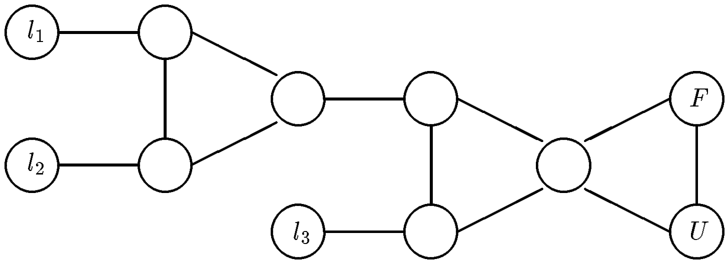

Figure 1:

The **integer programming problem**, defined as follows, is $\mathcal{NP}$-complete: given an $m \times n$ matrix A and an m-vector b, decide if there exists an integer n-vector x such that $Ax \leq b$. In this case, finding a reduction from 3-SAT is trivial: given a formula ϕ, represent each literal by an integer variable bounded between 0 and 1, and for each Boolean variable x, constrain the sum of the variables corresponding to x and its negation to be at most 1. The construction is completed by adding a constraint for each clause that forces the variables for the literals in the clause to sum to at least 1. On the other hand, to show that the problem is in $\mathcal{NP}$ requires more work, involving a calculation that bounds the length of a "smallest" solution satisfying the constraints (if one exists at all).

The **subset sum problem** is also $\mathcal{NP}$-complete: given a set of numbers S and a target number t, does there exist a subset of S that sums to t? This is the first problem that we have encountered that is a "number problem" in the sense that it is not the combinatorial structure, but rather the numbers that make this problem hard. If the numbers are given in unary, there is a polynomial-time algorithm (such an algorithm is called **pseudo-polynomial**): keep a (large, but polynomially bounded) table of all possible sums obtainable using a subset of the first i numbers; this is trivial to do for $i = 1$, and it is not hard to efficiently find the table for $i+1$ given the table for i; the table for $i = n$ gives us the answer. There are "number problems" that remain $\mathcal{NP}$-complete, even if the numbers are encoded in unary; such problems are called **strongly $\mathcal{NP}$-complete**. One example is the **3-partition problem**: given a set S of the $3n$ integers that sum to nB, does there exist a partition of S into n sets, $T_1, \ldots, T_n$, where each T_i has 3 elements that sum to B?

The complexity class $\mathcal{NP} \setminus \mathcal{P}$?

If we are to believe the conjecture that $\mathcal{P} \neq \mathcal{NP}$, then there exists a non-empty complexity class $\mathcal{NP} \setminus \mathcal{P}$. One might ask the question: is it true that every problem in $\mathcal{NP}$ is either $\mathcal{NP}$-complete or in $\mathcal{P}$? If $\mathcal{P} = \mathcal{NP}$ this question has a (trivial) affirmative answer, but a negative answer to it (under the assumption the $\mathcal{P} \neq \mathcal{NP}$) might help explain the reluctance of certain problems to be placed in one of those two classes. In fact, Ladner has shown, under the assumption that $\mathcal{P} \neq \mathcal{NP}$, that there is an extremely refined structure of equivalence between the two classes, $\mathcal{P}$ and $\mathcal{NP}$-complete.

(2.4) Theorem. *If L_1 is decidable and $L_1 \notin \mathcal{P}$, then there exists a decidable language L_2 such that $L_2 \notin \mathcal{P}$, $L_2 \propto_p L_1$ but $L_1 \not\propto_p L_2$.*

Note that if L_1 is $\mathcal{NP}$-complete, the language L_2 is "in between" the classes $\mathcal{P}$ and $\mathcal{NP}$-complete, if $\mathcal{P} \neq \mathcal{NP}$, and by repeatedly applying the result we see that there is a whole range of seemingly different complexity classes. Under the assumption that $\mathcal{P}$ is different from $\mathcal{NP}$, do we know of any candidate problems that may lie in this purgatory of complexity classes? The answer to this is "maybe". We will give four important problems that have not been shown to be either in $\mathcal{P}$ or $\mathcal{NP}$-complete. The problem for which there has been the greatest speculation along these lines is the **graph isomorphism problem**: given a pair of graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, decide if there is a bijection $\sigma : V_1 \mapsto V_2$ such that $ij \in E_1$ if and only if $\sigma(i)\sigma(j) \in E_2$. Later in this section we will provide evidence that it is *not* $\mathcal{NP}$-complete, and efforts to show that it is in $\mathcal{P}$ have so far fallen short (see Chapter 27 of this volume). A problem that mathematicians since the ancient Greeks have been trying to solve is that of **factoring** integers; a decision formulation that is polynomially equivalent to factoring is as follows: given an integer N and a bound k, does there exist a factor of N that is at most k? A problem that is no harder than factoring is the **discrete logarithm problem**: given a prime p, a generator g and an natural number $x < p$, find l such that $g^l \equiv x \bmod p$. Finally, there is the **shortest vector problem**, where we are given a collection of integer vectors, and we wish to find the shortest vector (in the Euclidean norm) that can be represented as a non-zero integral combination of these vectors. Here, current evidence makes it seem likely that this problem is really $\mathcal{NP}$-complete; related work is discussed elsewhere in this volume (see Chapter 19).

It is important to mention that there is an important subclass of $\mathcal{NP}$ which may also fall in this presumed gap. Edmonds' class of well-characterized problems, $\mathcal{NP} \cap co-\mathcal{NP}$, certainly contains $\mathcal{P}$ and is contained in $\mathcal{NP}$. Furthermore, unless $\mathcal{NP} = co-\mathcal{NP}$, it cannot contain any $\mathcal{NP}$-complete problem. On the other hand, the prevailing feeling is that showing a problem to be in this class is a giant step towards showing that the problem is in $\mathcal{P}$. A result of Pratt shows that primality is in $\mathcal{NP}$, so it lies in $\mathcal{NP} \cap co-\mathcal{NP}$, though as discussed above, there is additional evidence that it lies in $\mathcal{P}$. The factoring problem, which appears to be significantly harder, also lies in $\mathcal{NP} \cap co-\mathcal{NP}$, since a prime factorization can be guessed along with certificate that each of the factors is indeed prime. One interesting open question connected with $\mathcal{NP} \cap co-\mathcal{NP}$ is concerned with the existence of a problem that is complete for this class. One might hope that there is some natural problem that completely characterizes the relationship of $\mathcal{NP} \cap co-\mathcal{NP}$ with $\mathcal{P}$ (in the same manner that 3-SAT characterizes the $\mathcal{P} = \mathcal{NP}$ question).

One approach to shed light on the complexity of a problem that is not known to be either in $\mathcal{P}$ or $\mathcal{NP}$-complete, has been to consider weaker forms of completeness for $\mathcal{NP}$. In fact, Cook's notion of completeness, though technically a weaker definition of intractability, is no less damning. $L_1 \propto_p L_2$ can be thought of as solving L_1 by a restricted kind of subroutine call, where first some polynomial-time preprocessing is done, and then the subroutine for L_2 is called once. Cook (1971) proposed a notion of reducibility where L_1 is solved by using a polynomial-time Turing machine that can, in one step, get an answer to any query of the form, "is $x \in L_2$?" Note that any complete language L with respect to this reducibility still has the property that $L \in \mathcal{P}$ if and only if $\mathcal{P} = \mathcal{NP}$. Karp (1972) focused attention on the notion $\propto_p$, and was able to show that $\mathcal{NP}$-completeness was powerful enough to capture a wide range of combinatorial problems. On the other hand, it remains an open question to show that Cook's notion of reducibility is stronger than Karp's; is there a natural problem in $\mathcal{NP}$ that is complete with respect to "Cook" reducibility, but not with respect to "Karp" reducibility? Adleman & Manders showed that non-determinism and randomization can play a role in defining notions of reducibility, and used these notions to show that certain number theoretic problems were not, for example, in $\mathcal{NP} \cap co-\mathcal{NP}$ unless $\mathcal{NP} = co-\mathcal{NP}$.

Oracles and relativized complexity classes

In previous subsections, we have discussed techniques to provide evidence of the intractability of concrete problems in $\mathcal{NP}$ by proving completeness results. In this section, we will be concerned with extensions of the model of computation where the analogue of the $\mathcal{P}$ versus $\mathcal{NP}$ problem can be resolved.

We have mentioned earlier that oracles are used to compactly represent the input of some problems. One can define the analogue of the classes $\mathcal{P}$ and $\mathcal{NP}$ for problems whose inputs are oracles and it is easy to prove that these complexity classes are different. For example, if the oracle represents a set system on n elements, then the problem to decide if this set system is nonempty is clearly in $\mathcal{NP}$, but one has to ask 2^n queries from the oracle to resolve it deterministically. Similar results have also been proved for combinatorial problems that are naturally represented by oracles. For example, the matroid matching problem and the problem to decide if a matroid has girth at most k (*i.e.*, whether it has a cycle of length at most k) are both clearly in $\mathcal{NP}$, but it has been shown that any deterministic algorithm solving them has to ask an exponential number of queries (see Chapter 11).

Problems on graphs can also be given by an oracle. Suppose that a graph (*e.g.*, on $N = 2^n$ nodes) is given by an oracle, that can tell whether two nodes are adjacent. The question is whether all "reasonable" decision problems on graphs require to ask some constant fraction of the queries? This problem has a long history, both for directed and undirected graphs, and many attempts were made at giving sufficiently strong conditions before an accurate conjecture, due to Aanderaa and Rosenberg, was proved by Rivest & Vuillemin, and later strengthened by Kahn, Saks & Sturtevant. Consider a decision problem L where the instances are undirected graphs, and L has three important properties: (1) **nontriviality** – some graphs are in L, but not all; (2) **monotonicity** – if G is an instance in L and G' is formed by adding an edge, then G' is also in L ; (3) **invariance under isomorphism** – if G is an instance in L and its nodes are relabelled to form G', then G' is also in L. Then for any problem satisfying these two properties, any correct procedure uses essentially $N^2/4$ queries for some graph of order N. In fact, if N is a prime power, Kahn, Saks & Sturtevant have shown that all $N(N-1)/2$ queries must be asked (see Chapter 34). These results are also relevant in comparing the adjacency matrix form of encoding graphs to the adjacency list encoding.

Another extension of the classes $\mathcal{P}$ and $\mathcal{NP}$ uses oracles as a source of computational power rather than as a source of information. For a given language A we shall consider **oracle Turing machines** that, during the computation, may ask queries of the form: is $x \in A$? Here, the oracle A is considered as part of the model of computation, rather than as part of the input. This notion of an oracle can help, for example, in understanding the relative difficulty of some problems.

For an language A, we denote by $\boldsymbol{\mathcal{P}^A}$ and $\boldsymbol{\mathcal{NP}^A}$ the **relativized** analogues of $\mathcal{P}$ and $\mathcal{NP}$ that are defined by Turing machines that use an oracle that decides membership in A. In general, the relativized analogue of a complexity class $\mathcal{C}$ is denoted by $\mathcal{C}^A$. The main result concerning oracle complexity classes, due to Baker, Gill & Solovay (1975), is that the answer to the relativized version the $\mathcal{P} = \mathcal{NP}$ problem depends on the oracle. The intuition for the first alternative of the following theorem is given by the case when the oracle is an input. This is made rigorous by diagonalizing over all oracle Turing machines to construct an oracle A such that the language $L(A) = \{1^n : \exists x \in A \text{ such that } |x| = n\}$ is not in $\mathcal{P}^A$.

(2.5) Theorem. *There exist languages A and B such that $\mathcal{P}^A \neq \mathcal{NP}^A \neq \mathrm{co}-\mathcal{NP}^A$, and $\mathcal{P}^B = \mathcal{NP}^B = \mathrm{co}-\mathcal{NP}^B$.*

Several of the theorems and proof techniques discussed in this survey easily extend to relativized complexity classes (*i.e.*, they **relativize**). As a corollary to the above theorem, we can assert that techniques that relativize cannot settle the $\mathcal{P}$ versus $\mathcal{NP}$ problem.

One might wonder which one of the two alternatives provided by the theorem of Baker, Gill, & Solovay is more typical. We define a random language A to be one that contains each word x independently with probability 1/2. We say that a statement **holds for a random oracle** if the probability that the statement holds for a random language A in place of the oracle is 1. The Kolmogorov 0-1 law of probability theory states that if an event $\mathcal{A}$ is determined by the independent events, $\mathcal{B}_1$, $\mathcal{B}_2$, ... and $\mathcal{A}$ is independent of any event that is a finite combination of the events $\mathcal{B}_i$, then the probability of $\mathcal{A}$ is either 0 or 1. Applying this to the event $\mathcal{A}$ that $\mathcal{P}^A = \mathcal{NP}^A$ and the events $\mathcal{B}_i$ that the ith word is in the language A, we see that the probability of $\mathcal{P}^A = \mathcal{NP}^A$ for a random oracle A is either 0 or 1. Bennett & Gill have provided the answers to these questions.

(2.6) Theorem. *$\mathcal{P}^A \neq \mathcal{NP}^A$ and $\mathcal{NP}^A \neq co-\mathcal{NP}^A$ for a random oracle A.*

Evidence of intractability: $\mathcal{PSPACE}$-completeness

In this subsection, we turn to the question of the space complexity of problems. When we discuss space complexity, we may assume that the Turing machine has only one work tape (and a separate input tape), since by expanding the tape alphabet, any number of tapes can be simulated by one tape without using more space. We shall also assume that the Turing machine halts in a unique configuration when accepting the input, *e.g.*, it erases its work tape, moves both heads to the beginning of the tapes, and enters a special accepting state.

We remarked when introducing $\mathcal{PSPACE}$ that it was not an oversight that $\mathcal{NPSPACE}$ was not defined, since $\mathcal{PSPACE} = \mathcal{NPSPACE}$. This result is a special case of the following theorem of Savitch (1970).

(2.7) Theorem. *If L is accepted by a nondeterministic Turing machine using space $S(n) \geq \log n$, then it is accepted by a deterministic Turing machine using space $S(n)^2$.*

Proof: The proof is based on the idea of modeling the computation by a directed reachability problem and using a natural divide-and-conquer strategy. In any given computation, a nondeterministic Turing machine M can be completely described by specifying the input head position, the contents of the work tape, the work tape head position and the current state. Consider the directed graph G whose nodes correspond to these configurations and whose arcs correspond to possible transitions of M. The Turing machine accepts the input if and only if there is a path in G from the starting configuration to the (unique) accepting configuration.

Since M uses $S(n)$ space, there are at most $d^{S(n)}$ configurations for some constant d and hence the length of a simple path in G is at most $d^{S(n)}$. To solve the reachability problem in G, we build a procedure that recursively calls itself to check for any nodes i and k of G, and a bound l, whether there is a path from i to k of length at most 2^l. There is such a path if there exists a midpoint of the path j such that j can be reached from i, and k can be reached from j by paths of length at most 2^{l-1}. The existential quantifier can be implemented by merely trying all possible nodes j in some specified order. The basis of this recursion is the case $l = 0$, where we merely need to know if $i = k$, or if i and k are connected by an arc of G, and this can be easily checked in $O(S(n))$ space. In implementing this procedure, we need to keep track of the current midpoint at each level of the recursion, and so we need $lS(n)$ space to do this. To simulate M by a deterministic Turing machine, we run the procedure for the graph G with $l = S(n) \log d$ and the nodes corresponding to the starting and the accepting configurations. $\square$

A problem L_1 is ***$\mathcal{PSPACE}$*-complete** if $L_1 \in \mathcal{PSPACE}$ and for all $L \in \mathcal{PSPACE}$, $L \propto_p L_1$. The simplest $\mathcal{PSPACE}$-complete problem is the problem of determining if two nodes are connected in a directed graph G (of order 2^n) that is given by a circuit with $2n$ inputs where the first n specify in binary a node i and the second n specify a node j, and the output of the circuit is 1 if and only if (i, j) is an arc of G. This problem can easily be solved by a non-deterministic Turing machine using polynomial space, hence it is in $\mathcal{PSPACE}$. To prove that it is complete, consider a language $L \in \mathcal{PSPACE}$. L can be reduced to this **circuit-based directed reachability problem** by introducing the graph G used in the proof of Savitch's theorem. It is easy to construct in polynomial time a (polynomial-size) circuit that decides if one configuration follows another by one transition of M.

The idea of "guessing a midpoint" is also the main idea used to derive another $\mathcal{PSPACE}$-complete problem. The problem of the validity of quantified Boolean formulae is as follows: given a formula in first-order logic in prenix form, $\exists x_1 \forall x_2 \cdots Q_k x_k \phi(x_1, \ldots x_k) = \mathsf{true}$, decide if it is valid. This problem is clearly in $\mathcal{PSPACE}$. The proof of its completeness is a mixture of Theorem (2.7) and Theorem (2.3). We use the alternation of existential and universal quantifiers to capture the notion of the existence of a midpoint such that both (for all) the first and second halves of the computation are legitimate. The basis of the recursion is now solved by building a Boolean formula to express that two configurations are either the same or one is the result of a single transition from the other.

An instance of the previous problem can be viewed as a game between an existential player and a universal player; the existential player gets to choose values for x_1, then the universal player chooses for x_2, and so on. The decision question amounts to whether the first player has a strategy so that ϕ must evaluate to true. There are many other $\mathcal{PSPACE}$-complete problems known, and most of these have a game-like flavor. An example of a more natural $\mathcal{PSPACE}$-complete game is the **directed Shannon switching game**: given a graph G with two nodes s and t, two players alternately color the edges of G, where the first player, coloring with red, tries to construct a red path from s to t, whereas the second player, coloring with blue, tries to construct a blue (s,t)-cut; does the first player have a winning strategy for G? Note that this result is in stark contrast to the undirected case, which can be solved in polynomial time (see Chapter 11).

The role of games in $\mathcal{PSPACE}$-completeness suggests a new type of Turing machine, called an **alternating Turing machine**, which was originally proposed by Chandra, Kozen & Stockmeyer (1981). Consider a computation to be a sequence of moves made by two players, an existential player and a universal player. The current state indicates whose move it is, and in each configuration the specified player has several moves from which to choose. Each computation path either accepts or rejects the input. The input is accepted if the existential player has a winning strategy, that is, if there is a choice of moves for the existential player, so that for any choice of moves by the universal player, the input is accepted. For simplicity, assume again that each computation path has the same number of moves. As before, the time to accept an input x is this number of moves, and the space needed to accept x is the maximum space used on any computation path. Observe that a nondeterministic machine is an alternating machine where only the existential player moves.

The role of $\mathcal{PSPACE}$ in the definition of these machines suggests that alternating polynomial time is closely related to $\mathcal{PSPACE}$ and indeed this is just a special case of a general phenomenon. Let $ATIME(T(n))$ and $ASPACE(S(n))$ denote the classes accepted by an alternating Turing machine with $O(T(n))$ time and $O(S(n))$ space, respectively. Chandra, Kozen & Stockmeyer proved two fundamental results characterizing the relationship of alternating classes to deterministic and nondeterministic ones. Note that the first result, in essence, implies Savitch's theorem, and in fact, the proof of the last inclusion of Theorem (2.8) is very similar to the proof of Savitch's theorem.

(2.8) Theorem. *If $T(n) \geq n$, then*

$$ATIME(T(n)) \subseteq DSPACE(T(n)) \subseteq NSPACE(T(n)) \subseteq ATIME(T(n)^2).$$

(2.9) Theorem. *If $S(n) \geq \log n$, then $ASPACE(S(n)) = \cup_{c>0} DTIME(c^{S(n)})$.*

Among the consequences of these results, we see that $\mathcal{AP} = \mathcal{PSPACE}$. One can view an alternating Turing machine as extremely idealized parallel computer, since it can branch off an unbounded number of parallel processes that can be used in determining the final outcome. Therefore, one can consider these results as a proven instance of the **parallel computation thesis**: parallel time equals sequential space (up to a polynomial function).

The polynomial-time hierarchy

The definition of $\mathcal{PSPACE}$ using alternating Turing machines suggests a hierarchy of complexity classes between $\mathcal{NP}$ and $\mathcal{PSPACE}$, called the **polynomial-time hierarchy**. The kth level of the polynomial-time hierarchy can be defined in terms of polynomial-time alternating Turing machines where the number of "alternations" between existential and universal quantifiers is less than k. Equivalently, we can define $\boldsymbol{\Sigma}_k^{\mathcal{P}} = \{L | \exists L' \in \mathcal{P}$ such that $x \in L$ if and only if $\exists_p y_1 \forall_p y_2 \cdots Q_k y_k (x, y_1, y_2, \ldots, y_k) \in L'\}$, where Q_k is $\exists_p$ if k is odd, and $\forall_p$ if k is even. Note that $\Sigma_0^{\mathcal{P}} = \mathcal{P}$ and $\Sigma_1^{\mathcal{P}} = \mathcal{NP}$. We also define $\Pi_k^{\mathcal{P}}$ to be $co - \Sigma_k^{\mathcal{P}}$. Clearly, $\Sigma_k^{\mathcal{P}} \cup \Pi_k^{\mathcal{P}} \subseteq \Sigma_{k+1}^{\mathcal{P}}$. The following **generalized coloring problem** gives a natural example of a problem in $\Sigma_2^{\mathcal{P}}$. Given input graphs G and H, can we color the nodes of G with two colors so that the graph induced by each color does not contain H as a subgraph?

Alternatively, it is possible to define the polynomial-time hierarchy in terms of oracles, as was done in the original formulation by Meyer & Stockmeyer (1972). For complexity classes C and D let $\mathcal{C}^{\mathcal{D}}$ denote the union of $\mathcal{C}^A$ over all $A \in \mathcal{D}$. Consider again the generalized coloring problem; it is not hard to see that there is nondeterministic polynomial-time Turing machine to solve it, given an oracle for the following problem A: given G and H, decide if H is a subgraph of G. Since $A \in \mathcal{NP}$, we see that this coloring problem is in $\mathcal{NP}^{\mathcal{NP}}$. In fact, $\Sigma_2^{\mathcal{P}} = \mathcal{NP}^{\mathcal{NP}}$ and in general, $\Sigma_{k+1}^{\mathcal{P}} = \mathcal{NP}^{\Sigma_k^{\mathcal{P}}}$. Unfortunately, for each new complexity class, there is yet another host of unsettled questions.

Open Problems For each $k \geq 1$, is $\Sigma_k^{\mathcal{P}} = \Sigma_{k-1}^{\mathcal{P}}$? For each $k \geq 1$, is $\Sigma_k^{\mathcal{P}} = \Pi_k^{\mathcal{P}}$?

Contained in these, for $k = 1$, are the $\mathcal{P} = \mathcal{NP}$ and $\mathcal{NP} = co - \mathcal{NP}$ questions, and as was true for those questions, one might hope to find complete problems on which to focus attention in resolving these open problems. We define these notions of completeness with respect to polynomial-time reducibility, so that L is complete for $\mathcal{C}$ if and only if $L \in \mathcal{C}$ and all problems in $\mathcal{C}$ reduce ($\propto_p$) to L. As might be expected, analogues of the satisfiability problem which allow a particular number of alternations in the formulae can be used to provide a complete problem for each level of the hierarchy. On the other hand, it is more satisfying to have more natural complete problems, and Rutenberg showed that the generalized coloring problem is, in fact, complete for $\Sigma_2^{\mathcal{P}}$. Another problem of identical complexity is a similarly flavored node deletion problem: given graphs G and H and an integer k, decide if there is a subset of k nodes that can be deleted from G, so that the remaining graph no longer has H as a subgraph.

One piece of good news concerning this infinite supply of open problems, is that their answers may be related. There is a principle of upward inheritability that says that if $\Sigma_l^{\mathcal{P}} = \Pi_l^{\mathcal{P}}$ for some level l, then equality holds for all levels $k \geq l$. In fact, $\Sigma_l^{\mathcal{P}} = \Pi_l^{\mathcal{P}}$ implies that the entire hierarchy collapses to that level; *i.e.*, $\Sigma_k^{\mathcal{P}} = \Sigma_l^{\mathcal{P}}$ for all $k \geq l$. Note that $\mathcal{P} = \mathcal{NP}$ if and only if $\mathcal{P} = \mathcal{PH}$, where $\mathcal{PH} = \cup_{k\geq 0}\Sigma_k^{\mathcal{P}}$.

As we shall see, the polynomial-time hierarchy has helped to provide insight into the structure of several complexity classes. Perhaps the first result along these lines is due to Sipser, who used a beautiful "hashing function" technique to show that $\mathcal{BPP}$ is in the polynomial-time hierarchy, and in fact, can be placed within $\Sigma_2^{\mathcal{P}} \cap \Pi_2^{\mathcal{P}}$.

Furst, Saxe & Sipser (1981) discovered an interesting connection between constant-depth circuit lower bounds and separating relativized complexity classes. In particular, they showed that there are important consequences of an exponential lower bound on the size of a constant-depth circuit for the **parity function**, the sum modulo 2 of the bits of the input. Based on earlier results of Furst, Saxe & Sipser and Ajtai, Yao (1985) proved a sufficiently strong lower bound to yield the following theorem.

(2.10) Theorem. *There exists an oracle A that separates the polynomial-time hierarchy from $\mathcal{PSPACE}$; that is, $\cup_{k\geq 1}\Sigma_k^{\mathcal{P},A} \neq \mathcal{PSPACE}^A$.*

The idea of the proof is as follows. First one shows that it is sufficient to consider alternating Turing machines in which every branch of the computation has a single oracle question at the end of the branch. The computation tree of such an alternating Turing machine with k levels of alternation corresponds to a depth-k circuit where the oracle answers are the inputs. For any oracle A, we define the language $L(A) = \{1^n | A \text{ contains an odd number of strings of length } n\}$. $L(A)$ is in $\mathcal{PSPACE}^A$ for any oracle A. Using diagonalization and the result that for any constant c, a constant-depth circuit that computes the parity function has $\Omega(n^{\log^c n})$ gates, one can construct an oracle A such that $L(A)$ is not in $\cup_{k\geq 1}\Sigma_k^{\mathcal{P},A}$.

Another related problem is to separate the finite levels of the polynomial-time hierarchy using oracles. Baker & Selman proved the existence of an oracle A such that $\Sigma_2^{\mathcal{P},A} \neq \Pi_2^{\mathcal{P},A}$ (and consequently, $\Sigma_2^{\mathcal{P},A} \neq \Sigma_3^{\mathcal{P},A}$). Sipser showed that an oracle that separates finite levels of the polynomial-time hierarchy can be obtained via a connection to lower bounds on the size of constant-depth circuits, similar to the one used in Theorem (2.10). The following theorem is based on this as well as the requisite lower bounds for a certain function F_k, which is in some sense the generic function that is computable in depth k; these lower bounds were obtained through a series of results by Sipser, Yao and Hastad.

(2.11) Theorem. *For every k, there exists an oracle A_k such that $\Sigma_k^{\mathcal{P},A_k} \neq \Sigma_{k+1}^{\mathcal{P},A_k}$.*

Techniques for proving the circuit complexity lower bounds used in Theorems (2.10) and (2.11) are discussed in Chapters 33 and 40. Based on stronger circuit complexity results, Babai and Cai separated $\mathcal{PSPACE}$ from the finite levels of the hierarchy by random oracles. The question whether random oracles separate the finite levels of the polynomial-time hierarchy remains open. Another open question concerning random oracles is whether $\mathcal{P}^A = \mathcal{NP}^A \cap \text{co} - \mathcal{NP}^A$ for a random oracle A.

Evidence of intractability: $\#\mathcal{P}$-completeness

Consider the **counting problem** of computing the number of perfect matchings in a bipartite graph. If this is cast as a decision problem, it does not appear to be in $\mathcal{NP}$, since it seems that the number of solutions can only be certified by writing down possibly exponentially many matchings. However, consider modifying the definition of $\mathcal{NP}$ to focus on the *number* of good certificates, rather than the existence of one; let $\#\boldsymbol{\mathcal{P}}$ be the class of problems for which there exists a language $L' \in \mathcal{P}$ and a polynomial $p(n)$ such that for any input x, the only acceptable output is $z = |\{y : |y| = p(|x|) \text{ and } (x,y) \in L'\}|$. Clearly, the problem of counting perfect matchings in a bipartite graph is in $\#\mathcal{P}$. Any problem in $\#\mathcal{P}$ can be computed using polynomial space, since all possible certificates y may be tried in succession. A recent result of Toda gives evidence of the intractability of this class by showing that $\mathcal{PH} \subseteq \mathcal{P}^{\#\mathcal{P}}$.

We can also define a notion of a complete problem for $\#\mathcal{P}$. To do this, we use a reduction analogous to that used by Cook. Let P_1 and P_2 be counting problems, where the output for input x is denoted by $P_1(x)$ and $P_2(x)$, respectively. The problem P_1 reduces to P_2 if there exists a polynomial-time Turing machine that can compute P_1 if it has access to an oracle that, given x can compute $P_2(x)$ in one step. We define a counting problem to be **$\#\mathcal{P}$-complete** if it is in $\#\mathcal{P}$, and all problems in $\#\mathcal{P}$ reduce to it. A stronger notion of reducibility, analogous to the notion used by Karp, is called **parsimonious**: an instance x of P_1 is mapped in polynomial time to $f(x)$ for P_2, such that $P_1(x) = P_2(f(x))$. For either notion of reducibility, we see that any polynomial-time algorithm for P_2 yields a polynomial-time algorithm for P_1. It is not hard to see that the proof of Cook's theorem shows that the problem of computing the number of satisfying assignments of a Boolean formula in conjunctive normal form is $\#\mathcal{P}$-complete. Furthermore, by being only slightly more careful, it is easy to give parsimonious modifications of the reductions to the clique problem, the Hamiltonian circuit problem, and seemingly any $\mathcal{NP}$-complete problem, and so the counting versions of $\mathcal{NP}$-complete problems can be shown to be $\#\mathcal{P}$-complete.

Surprisingly, not all $\#\mathcal{P}$-complete counting problems need be associated with an $\mathcal{NP}$-complete problem. Computing the number the perfect matchings in a bipartite graph, (or equivalently, computing the **permanent** of a 0,1 matrix) is a counting version of a problem in $\mathcal{P}$, the perfect matching problem, and yet Valiant (1979) has shown that this problem is $\#\mathcal{P}$-complete. This made it possible to prove that a variety of counting problems are $\#\mathcal{P}$-complete although they are based on polynomially solvable decision problems.

There are many problems that are essentially $\#\mathcal{P}$-complete, although they do not have the appearance of a counting problem. An important example is the problem of computing the volume of a convex body. In the special case when the body is a polytope given by a system of linear inequalities, Khachiyan and Dyer & Frieze independently proved that this is $\#\mathcal{P}$-hard. For some problems, computing a good estimate suffices, and this might be much easier. If the convex body is given by an oracle that answers whether a given point is feasible, and if not produces a separating hyperplane (see Chapter 28), then it is most natural to only estimate the volume. However, Bárány & Füredi, extending work of Elekes, proved that if U and L are upper and lower bounds on the volume of a d-dimensional convex body that are computed by an algorithm that makes only a polynomial number of calls to the oracle, then $U/L = \Omega((d/\log d)^d)$. Surprisingly, randomization can overcome this intractability. Dyer, Frieze & Kannan showed that there is a randomized algorithm that, given any $\epsilon, \delta > 0$, computes upper and lower bounds U and L such that $U/L \leq 1+\epsilon$, uses a number of calls to the oracle that is bounded by polynomial in d, $1/\epsilon$, and $\log(1/\delta)$, and is correct with probability at least $1-\delta$.

Another problem that has been shown to be intimately connected with the estimation of the value of counting problems is that of uniformly generating combinatorial structures. As an example, suppose that a randomized algorithm requires a perfect matching in a bipartite graph G to be chosen uniformly from the set of all perfect matchings in G; how can this be done? Jerrum, Valiant & Vazirani have shown that a relaxed version of this problem, choosing perfect matchings that are selected with probability within an arbitrarily small constant factor of the uniform value, is equivalent to the problem of estimating the number of perfect matchings (using a randomized algorithm). In fact, their result carries through to most counting/generation problems related to problems in $\mathcal{NP}$, since it requires only a natural self-reducibility property, that says that an instance can be solved by handling some number of smaller instances of the same problem.

Stockmeyer has provided insight into estimating the value of $\#\mathcal{P}$ problems, by trying to place these problems within the polynomial-time hierarchy. Using Sipser's hashing function technique, Stockmeyer showed that for any problem in $\#\mathcal{P}$ and any fixed value of $d > 0$, there exists a polynomial-time Turing machine with a $\Sigma_2^{\mathcal{P}}$-complete oracle that computes a value that is within a factor of $(1 + n^{-d})$ of the correct answer.

Proof of intractability

It is, of course, far preferable to *prove* that a problem is intractable, rather than merely giving evidence that supports this belief. Perhaps the first natural question is, are there *any* decidable languages that require more time than $T(n)$? The diagonalization techniques used to show that there are undecidable languages can be used to show that for any Turing computable $T(n)$, there must exist such a language; consider the language L of all i such that if i is run on the Turing machine M_i that i encodes, either it is rejected, or it runs for more than $T(n)$ steps. It is easy to see that L is decidable, and yet no Turing machine that always halts within $T(n)$ steps can accept L. Using our stronger assumption about $T(n)$ (full time-constructibility) we are able to define such a language that is not only decidable, but can be recognized within only slightly more time than $T(n)$. The additional logarithmic factor needed in Theorem (2.12), which combines results of Hartmanis & Stearns (1965) and Hennie & Stearns, is due to the fact that the fastest known simulation of a (multitape) Turing machine by a machine with some constant number of tapes slows down the machine by a logarithmic factor.

(2.12) Theorem. *If* $\liminf_{n\to\infty} \dfrac{T_1(n)\log T_1(n)}{T_2(n)} = 0$, *then there exists* $L \in DTIME(T_2) \backslash DTIME(T_1)$.

Since any multitape machine can be simulated by a 1-tape machine without using any additional space, the corresponding space hierarchy theorem of Hartmanis, Lewis & Stearns does not require the logarithmic factor. Seiferas, Fischer & Meyer have proved an analogous, but much more difficult, hierarchy theorem for nondeterministic time.

(2.13) Theorem. *If* $\liminf_{n\to\infty} \dfrac{T_1(n+1)}{T_2(n)} = 0$, *then there exists* $L \in NTIME(T_2) \setminus NTIME(T_1)$.

Although the proofs of these theorems are non-constructive, Meyer & Stockmeyer developed the following strategy that makes use of completeness results in order to prove lower bounds on particular problems. Intuitively, a completeness result says that a problem is a "hardest" problem for some complexity class, and Theorems (2.12) and (2.13) can be used to show that certain complexity classes have provably hard problems. Consequently, these two pieces imply that the complete problem is provably hard.

As an example, consider the **circuit-based large clique problem**, L_{cc}, which is the problem analogous to the circuit-based directed reachability problem, that tests whether the (compactly represented) input graph on $N = 2^n$ nodes has a clique of size $N/2$. This problem is complete for the class $\mathcal{NEXPTIME} = \cup_k NTIME(2^{n^k})$ with respect to polynomial-time reducibility. This can be seen by introducing the circuit-based version of satisfiability; a formula is represented by a polynomial-size circuit that outputs 1 for input (i, j) when literal l_i is in the jth clause, where i and j are given in binary. The generic reduction of Cook's theorem translates exactly to show that circuit-based satisfiability is $\mathcal{NEXPTIME}$-complete, and the completeness of L_{cc} follows by using essentially the same reduction used to show the $\mathcal{NP}$-completeness of the ordinary clique problem. Now consider a language $L \in NTIME(2^{n^2}) - NTIME(2^n)$ specified by Theorem (2.13). Since $L \propto_p L_{cc}$, then $L_{cc} \in NTIME(2^{n^c})$ implies that $L \in NTIME(p(n) + 2^{p(n)^c})$, where $p(n) = n^k$ is the time bound for the reduction. By choosing $c = 1/k$, we obtain the following theorem.

(2.14) Theorem. *There exists a constant $c > 0$ such that $L_{cc} \notin NTIME(2^{n^c})$.*

One interpretation of this result is that there exist graphs specified by circuits such that proofs that these graphs have a large clique require an exponential number of steps (in terms of the size of the circuit). Observe that we would have been able to prove a stronger result if we would have had a better bound on the length of the string produced by the reduction. Lower bounds proved using this strategy are often based on completeness with respect to reducibilities that are further restricted to produce an output of length bounded by a linear function of the input length.

This strategy has been applied primarily to problems from logic and formal language theory. A result of Fischer & Rabin (1974) that contrasts nicely with Theorem (1.3) concerns L_{pa}, the language of all provable sentences in the theory of arithmetic for natural numbers without multiplication, which was shown to be decidable by Presburger.

(2.15) Theorem. *There is a constant $c > 0$ such that $L_{pa} \notin NTIME(2^{2^{cn}})$.*

A representative sample of problems treated in this fashion is surveyed by Stockmeyer (1987).

There are two other important results that separate complexity classes. Hopcroft, Paul and Valiant (1977) showed that $DTIME(T(n)) \subseteq DSPACE(T(n)/\log T(n))$, and thus time is not the same complexity measure as space. Paul, Pippenger, Szemerédi and Trotter (1983) showed that $DTIME(n) \neq NTIME(n)$.

Extensions of $\mathcal{NP}$: short proofs via randomization

In the same way that randomized algorithms give rise to an extended notion of efficiently computable, randomized proofs give rise to an extension of $\mathcal{NP}$, the class of languages for which membership is efficiently provable. Randomized proofs give overwhelming statistical evidence. In creating this branch of complexity theory, Babai (1985) and Goldwasser, Micali & Rackoff (1985) have given definitions to capture related notions of proof based on statistical evidence. The interested reader is directed to the surveys by Goldwasser (1989) and Johnson (1988).

Suppose that King Arthur has two graphs G_1 and G_2, and his magician Merlin wants to convince him that the two graphs are not isomorphic. The graph isomorphism problem is not known to be in $co-\mathcal{NP}$, but Goldreich, Micali & Wigderson have given the following way for Merlin to convince Arthur that the two graphs are not isomorphic. Merlin asks Arthur to choose one of the two graphs, randomly relabel the nodes, and show him this random isomorphic copy of the chosen graph; Merlin will tell which graph Arthur chose. If the two graphs are isomorphic, then Merlin has only a fifty percent chance of choosing the right graph, assuming that he cannot read Arthur's mind. If Merlin can successfully repeat this test several times, then Arthur can fairly safely conclude that Merlin can distinguish isomorphic copies of the two graphs; in particular, the two graphs are not isomorphic.

The above randomized proof is a prime example of the interactive proof defined by Goldwasser, Micali & Rackoff. An **interactive protocol** consists of two Turing machines: the Prover (Merlin) that has unrestricted power and the Verifier (Arthur) that is restricted to be a randomized polynomial-time Turing machine. The two machines operate in turns, and communicate only between turns via a special tape. The Prover is trying to convince the Verifier that a common input x is in a language L. A language L has an **interactive proof system** if there exists an interactive protocol such that if $x \in L$, then the Verifier accepts with probability at least 2/3; if $x \notin L$, then for any Turing machine used in place of the Prover, the Verifier rejects with probability at least 2/3. Let $\boldsymbol{\mathcal{IP}}$ denote the class of languages that have an interactive proof system with a polynomial number of turns. Note that, once again, the choice of probability 2/3 is arbitrary.

Babai has proposed a similar, but seemingly weaker, version of a randomized proof system, an **Arthur - Merlin game**, in trying to capture a complexity class just above $\mathcal{NP}$. It can be defined in the same way as an interactive proof system with the assumption that each machine may access the other's work and randomizing tapes. Note the importance of privacy in the above interactive protocol. Goldwasser & Sipser proved, however, that interactive proofs and Arthur–Merlin games define the same complexity class.

One can define randomized proof hierarchies in a way analogous to the polynomial-time hierarchy. We consider the class of languages accepted by an interactive proof system (or an Arthur–Merlin game) with less than k alternations of turns. Babai introduced $\boldsymbol{\mathcal{AM}}$ to denote the class of languages that have an Arthur–Merlin game where a single move of Arthur is followed by a single move of Merlin. It would be natural to conjecture that these hierarchies are strict. However, Babai proved that if a problem has an Arthur–Merlin game with a finite number of turns than it is in $\mathcal{AM}$. Since the equivalence between interactive proofs and Arthur–Merlin games increases the number of turns only by a constant, the same is true for interactive proofs with a finite number of turns.

On the other hand, it appears that $\mathcal{IP}$, which allows a polynomially bounded number of alternations, is a significantly richer class than $\mathcal{AM}$. Lund, Fortnow, Karloff & Nisan showed that $\mathcal{PH} \subseteq \mathcal{IP}$, and Shamir extended their techniques to prove the following theorem.

(2.16) Theorem. $\mathcal{IP} = \mathcal{PSPACE}$.

One way to view this result is that it is possible to convince someone of a theorem in polynomial time, if it can be proven using a polynomial-sized blackboard. An interesting aspect of these results is that they do not relativize, since, for example, Fortnow & Sipser have constructed an oracle A for which $co-\mathcal{NP}^A$ is not contained in $\mathcal{IP}^A$. There is evidence that $\mathcal{AM}$ is a more restrictive class. Just as $\mathcal{BPP} \subseteq \mathcal{P}/poly$, one can show that $\mathcal{AM}$ is contained in $\mathcal{NP}/poly$, a non-uniform extension of $\mathcal{NP}$. Babai has shown that $\mathcal{AM} \subseteq \Pi_2^{\mathcal{P}}$ by extending the proof that $\mathcal{BPP} \subseteq \Sigma_2^{\mathcal{P}} \cap \Pi_2^{\mathcal{P}}$.

It appears that $\mathcal{AM}$ does not contain $co-\mathcal{NP}$, but to prove this would imply that $\mathcal{NP} \neq co-\mathcal{NP}$. Nonetheless, the following theorem, due to Boppana, Hastad & Zachos, provides some evidence.

(2.17) Theorem. *If $co-\mathcal{NP} \subseteq \mathcal{AM}$ then $\Sigma_2^{\mathcal{P}} = \Pi_2^{\mathcal{P}} = \mathcal{AM}$.*

We have seen that the graph non-isomorphism problem is in $\mathcal{AM}$. Therefore, Theorem (2.17) implies that if the graph isomorphism problem is $\mathcal{NP}$-complete, then $\Sigma_2^{\mathcal{P}} = \Pi_2^{\mathcal{P}} = \mathcal{AM}$.

Goldwasser, Micali & Rackoff introduced the interactive proof system in order to characterize the minimum amount of "knowledge" that needs to be transferred in a proof. Interactive proofs make it possible to "prove", for example, that two graphs are isomorphic, without giving any further clue about the isomorphism between them. These aspects of interactive proofs shall be discussed in Chapter 40.

Note added in galleys: In the years since this survey was written, there have been quite a number of very significant developments beyond the results mentioned here. We have already mentioned the result that $\mathcal{IP} = \mathcal{PSPACE}$; this result was proved just in time to be included in the final revised version sent to the publisher, but it has turned out that this was more a beginning than a conclusion. Ben-Or, Goldwasser, Kilian, & Wigderson introduced an analogous notion of multiprover *interactive proofs in which the Verifier can be convinced by several independent Provers that cannot communicate with each other during the protocol. Fortnow, Rompel, & Sipser gave an alternate characterization of this class $\mathcal{MIP}$, which was used by Babai, Fortnow, & Lund to prove that $\mathcal{MIP} = \mathcal{NEXPTIME}$. Fiege, Goldwasser, Lovász, Safra, & Szegedy showed, using ideas from the proof that $\mathcal{MIP} = \mathcal{NEXPTIME}$, that there is a fundamental connection between randomized complexity classes and proving that certain optimization problems are hard to solve even approximately. Extensions of this result by Arora & Safra and Arora, Lund, Motwani, Sudan, & Szegedy, led to the ultimate result along these lines: $\mathcal{NP}$ is exactly the class of languages L for which there is the following type of* probabilistically checkable proof; *for any input x, the Verifier is given a certificate of polynomial length of which it may query $O(1)$ bits based on $O(\log|x|)$ random coin flips; for each $x \in L$, there exists a certificate such that the Verifier always accepts; for each $x \notin L$, given any certificate the Verifier rejects with probability at least 1/2. For a survey of these results, the reader is referred to Johnson (1992).*

3 Living with Intractability

The knowledge that a problem is $\mathcal{NP}$-complete is little consolation for the algorithm designer who needs to solve it. Contrary to their theoretical equivalence, all $\mathcal{NP}$-complete problems are not equally hard from a practical perspective. In this section, we will examine two approaches to these intractable problems that, while not overcoming their inherent difficulty, make them appear more manageable. In this process, finer theoretical distinctions will appear among these problems that help to explain the empirical evidence.

The complexity of approximate solutions

Most of the 21 $\mathcal{NP}$-complete problems in Karp's original paper are decision versions of optimization problems; this is also true for a great majority of the problems catalogued by Garey & Johnson (1979). Although the combinatorial nature of these problems makes it natural to focus on optimal solutions, for most practical settings in which these problems arise it is nearly as satisfying to obtain solutions that are guaranteed to be nearly optimal. In this subsection we will briefly survey the sorts of **performance guarantees** that can and cannot be obtained for particular combinatorial problems. For further discussion of the algorithmic techniques used in obtaining near-optimal solutions, the reader is referred to Chapter 28. Throughout this section, $\boldsymbol{OPT(I)}$ will denote the optimal value of an instance I of a particular combinatorial optimization problem.

It is possible that deciding if $OPT(I) \leq k$ is $\mathcal{NP}$-complete, and yet a solution of value no more than $OPT(I) + 1$ can be computed in polynomial time. In fact, this is true for the **edge coloring problem**, where we are given an undirected simple graph G and an integer k, and we wish to color the edges with as few colors as possible so that no two edges incident to the same node receive the same color. Holyer showed that it is $\mathcal{NP}$-complete to decide if a cubic graph can be 3-edge-colored. On the other hand, Vizing proved that the minimum number of colors needed is at most one more than maximum degree of G, and his proof immediately yields a polynomial-time algorithm that uses at most $OPT(I) + 1$ colors. Since edge-coloring graphs with chromatic index 2 is trivial, this also yields an algorithm that always uses no more than $4/3 \cdot OPT(I)$; a polynomial-time algorithm with such an **absolute performance guarantee** is often called a 4/3-approximation algorithm. On the other hand, it is easy to see that this is the best possible performance guarantee, unless $\mathcal{P} = \mathcal{NP}$. Suppose that there exists a ρ-approximation algorithm for edge coloring with $\rho < 4/3$. If this algorithm is run on a cubic graph that can be colored with three colors, then the algorithm must return a coloring that uses fewer than four colors; it returns an optimal coloring.

One type of strong approximation result is called a **fully polynomial approximation scheme**; this is a family of algorithms $\{A_\epsilon\}$, where A_ϵ is a $(1+\epsilon)$-approximation algorithm and the dependence of the running time on ϵ is bounded by a polynomial in $1/\epsilon$. By solving rounded instances with only a limited number of significant digits, Ibarra & Kim gave such a scheme for the **knapsack problem**: given n pieces to be packed into a knapsack of size B, where piece j has size s_j and value v_j, pack a subset of pieces of total size $\leq B$ with maximum total value. Note that it is impossible to improve the dependence of the running time on ϵ to a polynomial in $\log(1/\epsilon)$ since it is always possible to choose ϵ of polynomial length, such that $A_\epsilon(I) - OPT(I) < 1$, and thus by integrality, $A_\epsilon(I) = OPT(I)$. The same argument implies an important result of Garey & Johnson; if a problem is strongly $\mathcal{NP}$-complete, then there is no fully polynomial approximation scheme for it unless $\mathcal{P} = \mathcal{NP}$. If the running time of A_ϵ may depend arbitrarily on $1/\epsilon$, it sometimes possible to obtain such a **polynomial approximation scheme** for strongly $\mathcal{NP}$-complete problems. Hochbaum & Shmoys showed that this is the case for the following **machine scheduling problem**: each of n jobs is to be scheduled on one of m machines, where job j takes time p_j on any machine, and the aim is to minimize the time by which all jobs are completed. In fact, the idea of studying the performance guarantees of heuristics for optimization problems was first proposed by Graham (1966) in the context of this problem, who gave a 2-approximation algorithm.

It is sometimes too restrictive to focus on ρ-approximation algorithms. A good illustration of this is the **bin-packing problem**: given a collection of n pieces, where piece j has size s_j, how many bins of size B are needed to pack all of the pieces? Since it is easy to formulate the subset sum problem as a question of whether 2 bins suffice, we see that a ρ-approximation algorithm with $\rho < 3/2$ would imply that $\mathcal{P} = \mathcal{NP}$. However, Johnson showed that a simple heuristic uses at most $11/9 \cdot OPT(I) + 4$ bins. This suggests that an **asymptotic performance guarantee** may also be interesting, where we consider the infimum of the absolute performance guarantee for instances with $OPT(I) \geq k$ (as $k \to \infty$). It was a great surprise when Fernandez de la Vega & Lueker not only substantially improved this 11/9 bound, but gave a polynomial approximation scheme with respect to asymptotic guarantees. Perhaps even more surprisingly, Karmarkar & Karp extended this to give such a scheme where the dependence of the running time on $1/\epsilon$ was bounded by a polynomial.

If it is possible to scale up the data to generate an equivalent problem, such as using processing times $M \cdot p_j$ in the machine scheduling problem, any distinction between the absolute and asymptotic performance guarantees disappears. For node coloring, the following "graph composition" accomplishes this scaling: take M copies of the graph, and make each pair of nodes in different copies adjacent. The $\mathcal{NP}$-completeness of 3-colorability again implies that an absolute performance guarantee better than 4/3 is unlikely. In fact, Garey & Johnson (1976) use a more intricate composition technique to *increase* this ratio, and thereby prove that an asymptotic performance guarantee less than 2 would imply that $\mathcal{P} = \mathcal{NP}$.

For some problems, such as the **traveling salesman problem**, Gonzalez & Sahni observed that no constant performance guarantee is possible unless $\mathcal{P} = \mathcal{NP}$. In this example, where the aim is find a minimum length Hamiltonian circuit in a complete graph where edge e has length c_e, one can use a ρ-approximation algorithm to decide the Hamiltonian circuit problem for a graph $G = (V, E)$; set $c_e = 1$ for $e \in E$ and $\rho|V| + 1$ otherwise. However, Christofides has given a 3/2-approximation algorithm for instances that satisfy the triangle inequality.

For the great majority of problems, such as node coloring, there is both no constant performance guarantee, nor any evidence that such an algorithm does not exist. In the case of the maximum stable set problem, for which the best known algorithm has performance guarantee little better than $O(n/\log n)$, there is some evidence that no polynomial approximation scheme exists. Garey & Johnson have given another graph composition technique to show that if performance guarantee ρ^2 is obtained, then this can be used to obtain a ρ-approximation algorithm. By repeatedly applying this technique, it is possible to convert any ρ-approximation algorithm, where ρ is a constant, into a polynomial approximation scheme. There are few other techniques that provide evidence for the intractability of computing near-optimal solutions. Recently, Papadimitriou & Yannakakis have proposed a complexity class, along with a notion of completeness, that attempts to characterize those problems that have a constant performance guarantee, but do not have a polynomial approximation scheme.

Note added in galleys: In the years since this survey was written, there have been dramatic advances in proving that certain problems are also hard to approximate. Feige, Goldwasser, Lovász, Safra, & Szegedy showed that unless $\mathcal{NP} \subset DTIME(n^{\log\log n})$, there does exist a ρ-approximation algorithm for maximum stable set problem for any constant $\rho > 1$. Arora & Safra strengthened this to show that achieving such an approximation is $\mathcal{NP}$-hard, and this was strengthed even further by Arora, Lund, Motwani, Sudan, & Szegedy, who proved that there exists an $\epsilon > 0$ such that there does not exist an n^ϵ-approximation algorithm unless $\mathcal{P} = \mathcal{NP}$. These techniques have yielded signficant results for other problems as well. Lund & Yannakakis proved that any absolute performance guarantee that can be obtained for the minimum node coloring problem can also be obtained for the maximum stable set problem. Arora, Lund, Motwani, Sudan, & Szegedy also showed that, unless $\mathcal{P} = \mathcal{NP}$, there does not exist a polynomial approximation scheme for any complete problem in the class $\mathcal{MAX}\ \mathcal{SNP}$ proposed by Papadimitriou & Yannakakis. For example, a corollary of this result is that there does not exist a polynomial approximation scheme for the traveling salesman *problem with the triangle inequality, unless $\mathcal{P} = \mathcal{NP}$. For a survey of this research, the reader is referred to Johnson (1992).*

Probabilistic analysis of algorithms

One justified criticism of complexity theory is that it focuses on worst-case possibilities, and this may in fact be unrepresentative of practical reality. In this section, we will briefly indicate some of the results that are concerned with the probabilistic analysis of algorithms, where inputs are selected according to a specified probability distribution, and the average behavior is studied. Many related results are presented in Chapter 6, and the reader is referred there, as well as to the surveys of Karp, Lenstra, McDiarmid & Rinnooy Kan (1985) and Coffman, Lueker & Rinnooy Kan (1988). We shall also sketch the main ideas of an analogue of $\mathcal{NP}$-completeness, recently proposed by Levin (1986), to provide evidence that a problem is hard to solve in even a probabilistic sense.

For all of the problems mentioned in the subsection on the worst-case analysis of heuristics, it is possible to obtain much more optimistic results for the average-case analysis under the assumption that the input is drawn from a specified probability distribution. Unfortunately, these results are rely heavily on the particular distribution selected, and an approach to the average-case analysis of algorithms that is insensitive to this, would be an important advance. As an example, for the traveling salesman problem with edge lengths that are independently and identically distributed (i.i.d.) uniformly over the interval [0,1], Karp has given a heuristic where the expected value of its relative error is $O(n^{-1/2})$. On the other hand, the nodes may be selected i.i.d. uniformly in the unit square $[0,1]^2$, and the length of edge is given by the Euclidean distance between its endpoints. In this case, Karp (1976) has given a different algorithm that has the stronger property that the relative error converges to 0 almost surely as $n \to \infty$. This result, which stimulated much work in the area of probabilistic analysis, is based in part on a result of Beardwood, Halton & Hammersley, which proves that, for instances selected as above, there exists a constant c such that $OPT(I)/\sqrt{n} \to c$ almost surely.

Similar results are also known for such problems as node coloring and the Hamiltonian circuit problem. This work grew out of the theory of random graphs of Erdös and Rényi, which is treated in Chapter 6. A common way to choose a random graph is to include each possible edge independently with probability 1/2. For the first problem, it is possible to prove that a simple greedy method is, in probability, a factor of two more than optimal. For the latter, it is possible to give an algorithm that always gives a correct answer and runs in expected polynomial time.

Although the probabilistic analysis of algorithms has focused mainly on $\mathcal{NP}$-complete problems, it has often served as a useful tool to show that the average-case running time of certain algorithms is much better than the worst-case running time. The most important example of this is the simplex method for linear programming, which is a practically efficient algorithm that was shown to have exponential worst-case running time by Klee & Minty. Borgwardt and Smale, independently showed that variants of the simplex method take polynomial expected time under certain probabilistic assumptions. For a thorough survey of results in this area, the reader is referred to Shamir (1987).

Not all problems in $\mathcal{NP}$ have been solved efficiently even with probabilistic assumptions, and only the simplest sorts of distributions have been analyzed. This raises the specter of intractability: are there distributions for which certain $\mathcal{NP}$-complete problems remain hard to solve? Levin (1986) has proposed a notion of completeness in a probabilistic setting. Once again, evidence for hardness is given by showing that if a particular problem in $\mathcal{NP}$ with a specified input distribution can be solved in expected polynomial time, then every such problem and distribution pair can also be solved so efficiently. For a more complete discussion, the reader is encouraged to read the column of Johnson (1984). One of the motivations for studying such truly intractable problems come from the area of cryptography, which attempts to use the intractability of a problem to the algorithm designer's advantage (see Chapter 40).

A bit of care must be given in formulating the precise notion of polynomial expected time, so that it is insensitive to both the particular choice of machine and encoding. If $\mu(x)$ denotes the probability that a randomly selected instance of size n is x, and $T(x)$ is the running time on x, then one would typically define expected polynomial time to require that the sum of $\mu(x)T(x)$ over all instances of size n is $O(n^k)$ for some constant k. Instead, we consider μ to be the density function over the set of all instances I, and require that $\sum_{x \in I} \mu(x)T(x)^{1/k}/|x| < \infty$ for some constant k. Levin's notion of **random $\mathcal{NP}$** requires that the distribution function $M(x) = \sum_{i=1}^{x} \mu(i)$ can be computed in $\mathcal{P}$, where each instance is viewed as a natural number. This notion does not seem to be too restrictive, and includes all of the probability distributions discussed here. It remains only to define the notion of reducibility. As usual, "yes" instances must be mapped by a polynomial-time function f to "yes" instances, and analogously for "no" instances, but one must consider the density functions as well. The pair (L_1, μ_1) reduces to (L_2, μ_2) if in addition we require that $\mu_2(x)$ is at least a polynomial fraction of the total probability of elements that are mapped to x by f.

Levin showed that all of random $\mathcal{NP}$ reduces to a certain **random tiling problem**. Instances consist of integers $k \leq N$, a set of tile types, each of which is a unit square labeled with letters in its four corners, and a side-by-side sequence of k such tiles where consecutive tiles have matching labels in both adjoining corners. We wish to decide if there is a way of extending the sequence to fill out an $N \times N$ square, where adjacent tiles have matching labels in their corresponding corners, and the top row starts with the specified sequence of tiles. The instances are selected by first randomly choosing a value for N, where N is set equal to n with probability proportional to n^{-2}, $n = 1, 2, \ldots$; k is chosen uniformly between 1 and N, the tile types are chosen uniformly, and then the tiles in the sequence are selected in order uniformly among all possible extensions of the current sequence. More recently, Venkatesan & Levin have shown that a generalization of the problem of edge coloring digraphs (where for certain subgraphs, the number of edges given each color is specified) is also random $\mathcal{NP}$-complete.

4 Inside $\mathcal{P}$

In this section we shall focus on the complexity of problems in $\mathcal{P}$. After proving that a problem is in $\mathcal{P}$, the most important next step undoubtedly is to find an algorithm that is *truly* efficient, and not merely efficient in this theoretical sense. However, we will not address that issue, since that is best deferred to the individual chapters that discuss polynomial-time algorithms for particular problems. In this section, we shall address questions that relate to machine-independent complexity classes inside $\mathcal{P}$.

From a practical viewpoint, the most appealing complexity class inside $\mathcal{P}$ is the class of languages for which some polynomial-time algorithms can be speeded up significantly if several processors work simultaneously. We shall discuss parallel computation, and focus on the complexity class $\mathcal{NC}$, which serves as a theoretical model of efficient parallel computability, much as $\mathcal{P}$ serves as a theoretical model of efficient "sequential" computation.

We shall also consider the space complexity of problems in $\mathcal{P}$. Recall that the parallel computation thesis suggests that there is a close relationship between sequential space complexity and parallel time complexity. We will show another proven case of this thesis: every problem in $\mathcal{L}$, the class of problems solvable using logarithmic space, can be solved extremely efficiently in parallel. This is one source of interest in the complexity class $\mathcal{L}$. Another source is that this complexity class is the basis for the natural reduction that helps to distinguish among the problems in $\mathcal{P}$ in order to provide a notion of a hardest problem in $\mathcal{P}$.

Logarithmic space

The most general restriction on the space complexity of a language L that is known to imply that $L \in \mathcal{P}$ is logarithmic space. Observe that $\mathcal{L} \subseteq \mathcal{NL} \subseteq \mathcal{P}$, where the last inclusion follows, for example, from Theorem (2.9). The typical use of logarithmic space is to store a constant number of pointers, *e.g.*, the names of a constant number of nodes in the input graph, and in some sense, this restriction attempts to characterize such algorithms. Although $\mathcal{L}$ contains many interesting examples, we see the role of logarithmic space computation more as a natural means of reduction rather then providing interesting algorithms. Instead, we will focus on the nondeterministic and randomized analogues, for which there are languages that appear to be best characterized in terms of their space complexity.

To define the notion of a logarithmic-space reduction, we introduce a variant of logarithmic-space computation that can produce output of superlogarithmic size. We say that a function f can be **computed in logarithmic space** if there exists a Turing machine with a read-only input tape and a write-only output tape that, on input x, halts with $f(x)$ written on its output tape and uses at most logarithmic space on its work tapes. A problem L_1 **reduces in logarithmic space** to L_2 if there exists a function f computable in logarithmic space that maps instances of L_1 into instances of L_2 such that x is a "yes" instance of L_1 if and only if $f(x)$ is a "yes" instance of L_2. Let $L_1 \propto_{log} L_2$ denote such a logarithmic-space reduction (or log-space reduction, for short). It is fairly easy to show that the $\propto_{log}$ relation is transitive. A problem L_1 is $\boldsymbol{\mathcal{NL}}$**-complete** if $L_1 \in \mathcal{NL}$ and for all $L \in \mathcal{NL}$, $L \propto_{log} L_1$. The transitivity of $\propto_{log}$ yields the following result.

(4.1) Theorem. *For any $\mathcal{NL}$-complete problem L, $L \in \mathcal{L}$ if and only if $\mathcal{L} = \mathcal{NL}$.*

Savitch provided the most natural example of an $\mathcal{NL}$-complete language: the directed graph reachability problem. The problem is clearly in $\mathcal{NL}$ and can be shown to be $\mathcal{NL}$-complete along the same lines as the $\mathcal{PSPACE}$-completeness of the circuit-based directed reachability problem.

(4.2) Theorem. *The directed graph reachability problem is $\mathcal{NL}$-complete.*

In view of the above result, it was very surprising when Alelunias, Karp, Lipton, Lovász & Rackoff (1979) showed that the **undirected graph reachability problem**, the analogue of the directed graph reachability problem for undirected graphs, can be solved by a randomized Turing machine using logarithmic space. The algorithm attempts to find the required path by following a random walk in the graph, starting from the node s. It can be shown that a random walk is expected to use every edge with the same frequency, and if s and t are in the same connected component then the walk is expected to reach t in at most $O(nm)$ steps, where n and m denotes the number of nodes and edges. We define the class $\boldsymbol{\mathcal{RL}}$ to be the log-space analogue of $\mathcal{RP}$. A language L is in $\mathcal{RL}$

if there exists a randomized Turing Machine RM that works in logarithmic space, such that each input x that RM accepts along any computation path is in L and for every $x \in L$, the probability that RM accepts x is at least 1/2.

(4.3) Theorem. *Undirected graph reachability is in $\mathcal{RL}$.*

Note added in galleys: Recently, another piece of evidence has been discovered that suggests that undirected graph reachability is easier than its directed analogue. Nisan, Szemerédi, & Wigderson proved that the undirected graph reachability problem can be solved in $DSPACE(\log^{1.5} n)$.

One can think of the classes $\mathcal{L}$ and $\mathcal{NL}$ as lower-level analogues of $\mathcal{P}$ and $\mathcal{NP}$. By studying the relationships of $\mathcal{L}, \mathcal{NL}$ and $co{-}\mathcal{NL}$, one hopes to better understand the relationship of deterministic and nondeterministic computation. It is this point of view that makes the following theorem of Immerman (1988) and Szelepcsényi (1987) one of the biggest surprises in recent developments in complexity theory.

(4.4) Theorem. $\mathcal{NL} = co - \mathcal{NL}$

The proof uses a definition of nondeterministically computing a *function.* We say that a function $f(x)$ **can be computed in nondeterministic logarithmic space** if there is a nondeterministic log-space Turing machine that, on input x, outputs the value $f(x)$ on at least one branch of the computation and on every other branch either stops without an output or also outputs $f(x)$. If $f(x)$ is a Boolean function, then we say that the language L defined by $f(x) = 1$ is **decided in nondeterministic logarithmic space**, which is equivalent to L being in $\mathcal{NL} \cap co - \mathcal{NL}$.

We will prove Theorem (4.4) by showing that the $\mathcal{NL}$-complete directed graph reachability problem can be decided in nondeterministic logarithmic space. Given a directed graph $G = (V, A)$, a source node s and an integer k, let $f(G, s, k)$ denote the number of nodes reachable from the node s along paths of length at most k.

(4.5) Lemma. *The directed graph reachability problem is decidable in nondeterministic logarithmic space if and only if the function $f(G, s, k)$ can be computed in nondeterministic logarithmic space.*

Proof. To prove the *only if* direction, we use the fact that the directed graph reachability is $\mathcal{NL}$-complete. If it is decidable in logarithmic space, then so is the problem to recognize if there is a path of length at most k. To compute $f(G, s, k)$, we use the assumed nondeterministic machine for each node v, to decide if v is reachable from s by a path of length at most k, and count the number of reachable nodes.

To prove the opposite direction, we use the following nondeterministic log-space computation. First compute $f(G, s, n)$. Then for each node v, (nondeterministically) try to guess a path from s to v. Count the number of nodes for which a path has been found. If a path has been found to t, we accept the input. If $f(G, s, n)$ nodes have been reached without finding a path to t, this proves that t is not reachable from s, so we reject. Finally, if the number of nodes that have been reached is less than $f(G, s, n)$, then this is an incorrect branch of the computation, and the computation stops without producing an output. $\square$

To finish the proof of Theorem 4.4 we have to argue that the function $f(G, s, k)$ can be computed in nondeterministic logarithmic space. This is done by induction on k. Given $f(G, s, k)$ we can decide if there exists a path of length $k+1$ from s to a particular node v by checking if there is a path of length at most k to any of the predecessors of v using a variant of the algorithm given in the *if* part of Lemma (4.5). Counting these nodes gives $f(G, s, k+1)$.

The hardest problems in $\mathcal{P}$

One important application of log-space computation was introduced by Cook, who used log-space reducibility to introduce a notion of a hardest problem in $\mathcal{P}$. A problem L_1 is **$\mathcal{P}$-complete** if $L_1 \in \mathcal{P}$ and for all $L \in \mathcal{P}$, $L \propto_{log} L_1$. The transitivity of the log-space reduction gives the following theorem.

(4.6) Theorem. *For any $\mathcal{P}$-complete problem L, $L \in \mathcal{L}$ if and only if $\mathcal{L} = \mathcal{P}$.*

Later in this section we shall see that $\mathcal{P}$-completeness also provides evidence that a problem cannot be efficiently solved in parallel. This fact has greatly increased the interest in $\mathcal{P}$-completeness and a variety of problems have been shown to be $\mathcal{P}$-complete. Perhaps the most natural example is the circuit value problem, which was proved $\mathcal{P}$-complete by Ladner. Given a circuit with truth values assigned to its input gates, the **circuit value problem** is to compute the output of the circuit. This problem is clearly in $\mathcal{P}$. It can be proved $\mathcal{P}$-complete by building a circuit that simulates the computation of a Turing machine.

(4.7) Theorem. *The circuit value problem is $\mathcal{P}$-complete.*

Dobkin, Lipton & Reiss proved that each problem in $\mathcal{P}$ log-space reduces to the linear programming problem, and the celebrated result of Khachiyan showed that it is in $\mathcal{P}$. Valiant gave a straightforward reduction from a restricted circuit value problem that uses linear constraints to trace the value computed by the circuit.

Goldschlager, Shaw & Staples showed that the **maximum flow problem**, an important special case of the linear programming problem, is also $\mathcal{P}$-complete. In this problem, we are given a directed graph $G = (V, A)$ with two specified nodes, the source s and the sink t, and a non-negative capacity $u(a)$ assigned to each arc $a \in A$. A feasible flow is a vector $f \in \mathbf{R}^A$ that satisfies the capacity constraints, *i.e.*, $0 \leq f(a) \leq u(a)$ for each arc $a \in A$, and the flow conservation constraints, *i.e.*, the sum of the flow values on the arcs incident to a node $v \neq s, t$ is the same as the sum of the flow values on the arcs incident from v. The value of a flow is $\sum_{a=(v,t)\in A} f(a) - \sum_{a=(t,v)\in A} f(a)$. The decision problem that is proved to be $\mathcal{P}$-complete is deciding the parity of the maximum flow value.

There is a collection of $\mathcal{P}$-complete problems that are related to simple polynomial-time algorithms. The **maximal stable set problem**, in which the objective is to find a maximal (not maximum) stable set in an undirected graph, can clearly be solved by a simple greedy algorithm. When using this procedure, we usually select the first available node in each step, and so we find a specific solution, the lexicographically-first one. Cook showed that finding the lexicographically-first maximal stable set is $\mathcal{P}$-complete. This result might be surprising since this problem is easy to solve in polynomial time. However, $\mathcal{P}$-completeness also provides evidence that the problem is not solvable efficiently in parallel. Consequently, this completeness result supports the intuition that the greedy algorithm is inherently sequential.

Parallel computation

Parallel computation gives us the potential of substantially increasing the size of the instances for which certain problems are manageable by solving them with a large number of processors simultaneously. In studying parallel algorithms, we shall not be concerned with the precise number of parallel processors used, but rather their order as a function of the input size. We say that a parallel algorithm using $O(p(n))$ processors achieves **optimal speedup**, if it runs in $O(t(n))$ time and the best sequential algorithm known for solving the same problem runs in $O(t(n)p(n))$ time. Efficient algorithms that reach optimal (or near optimal) speedup with a significant number of processors have been found for many of the basic combinatorial problems. Another aspect of parallel computation is concerned with the inherent limitations of using many processors to speed up a computation. To be somewhat realistic, we shall only be interested in algorithms that use a polynomial number of processors. Consequently, we will focus on the possible speedup of polynomial-time sequential computation by parallel processing.

First we define a model of parallel computation. Although many such models have been proposed, one that seems to be the most convenient for designing algorithms is the **parallel random access machine (PRAM)**. The PRAM is the parallel analogue of the random access machine; it consists of a sequence of random access machines called processors, each with its own infinite local random access memory, in addition to an infinite shared random access memory where each memory cell can store any integer, and the input is stored in the shared memory. Each processor knows the input size and its identification number, although otherwise the processors are identical (*i.e.*, they run the same program). Different variants of the basic PRAM model are distinguished by the manner in which they handle read and write conflicts. In an **EREW PRAM** (exclusive-read exclusive-write PRAM), for example, it is assumed that each cell of the shared memory is only read from and written into by at most one processor at a time. At the other extreme, in a **CRCW PRAM** (concurrent-read, concurrent-write PRAM), each cell of the memory can be read from and written into by more than one processor at a time. If different processors attempt to write different things in the same cell, then the lowest numbered processor succeeds.

To illustrate the power of parallel computation, we give parallel algorithms for a problem that we have already discussed. Although finding the lexicographically-first maximal stable set is $\mathcal{P}$-complete, Karp & Wigderson have proved, surprisingly, that a maximal stable set can be found efficiently in parallel. Similar, much simpler and more efficient randomized algorithms have subsequently been independently discovered by Luby and by Alon, Babai & Itai.

Consider the most natural sequential algorithm for the problem: select a node v and include it in the stable set, delete v and all of its neighbors from the graph; repeat this procedure until all nodes have been deleted. Note that this algorithm requires n iterations for a path of length $2n$. A similar approach can still be used for a parallel algorithm. To make the algorithm fast, one selects a stable set in each iteration (rather than a single node), where the set is deleted along with its neighborhood. The following simple way to choose this stable set is due to Luby. A processor is assigned to each node and each edge of the graph. For a graph of order n, the processor assigned to node v picks a random integer $c(v)$ from 1 to n^4. Next, each processor assigned to an edge compares the values at the two nodes of the edge. The stable set selected consists of those nodes v for which $c(v)$ is strictly larger than the values assigned to any of its neighbors.

This algorithm clearly finds a maximal stable set, but it is less clear that few iterations are needed. It can be shown that each iteration is expected to remove a constant fraction of the edges, and consequently, the expected number of iterations is $O(\log n)$. The algorithm can be implemented on a randomized CRCW PRAM in $O(\log n)$ time (if we assume that a processor can choose a random number of size $O(\log n)$ in one step).

Karp & Wigderson introduced a technique that can be used to convert certain randomized algorithms into deterministic ones. The technique can be used if, in the analysis of the randomized algorithm, it is not necessary to assume mutual independence, but, for example, d-wise independent choices suffice for some constant d. One can appeal to known constructions to show that such variables can be chosen from a sample space of polynomial size. Each iteration can then be run for each point in the sample space simultaneously, and this ensures that a good sample point is used. This method can be used to convert the above randomized algorithm into a deterministic one.

When discussing parallel algorithms we shall assume that all arithmetic operations are restricted to polynomial-size numbers, and the number of processors used is polynomially bounded. We define the class $\boldsymbol{\mathcal{NC}}$ to consist of all languages L for which there exists a parallel algorithm that runs in time bounded by a polynomial in $\log n$. Note that in this definition the distinction between the different versions of the basic PRAM model are not relevant. If a problem can be solved by a CRCW PRAM using $p(n)$ processors in $O(\log^i n)$ time, then it can be solved by an EREW PRAM using $p(n)$ processors and $O(\log^i n \log p(n))$ time. The maximal stable set algorithm of Luby discussed earlier uses a randomized version of the CRCW PRAM. We define the complexity class $\boldsymbol{\mathcal{RNC}}$ to be the $\mathcal{NC}$ analog of $\mathcal{RP}$.

It is straightforward to see that Boolean product of two $n \times n$ $(0,1)$-matrices can be computed in constant time on a CRCW PRAM using $O(n^3)$ processors. By repeatedly squaring the adjacency matrix of a graph, the directed reachability problem can be solved in $O(\log n)$ time. This is, in some sense, the generic problem in $\mathcal{NL}$, and more generally, any problem in $\mathcal{NL}$ can be solved by a CRCW PRAM in $O(\log n)$ time. As a consequence, a log-space reduction can be simulated efficiently in parallel, and therefore $\mathcal{P}$-completeness provides evidence that a problem is not efficiently solvable in parallel.

(4.8) Theorem. *For any $\mathcal{P}$-complete problem L, $L \in \mathcal{NC}$ if and only if $\mathcal{NC} = \mathcal{P}$.*

We get the following chain of inclusions.

$$\mathcal{L} \subseteq \mathcal{NL} \subseteq \mathcal{NC} \subseteq \mathcal{P} \subseteq \mathcal{NP} \subseteq \mathcal{PSPACE}.$$

On the other hand, the computation of a CRCW PRAM that runs in $O(\log^i n)$ time can be simulated by a Turing machine in $O(\log^{i+1} n)$ space. This proves that $\mathcal{NC}$ is contained in $\cup_{i \geq 1} \mathcal{DSPACE}(\log^i n)$. By the analogue of Theorem (2.12) for space complexity, this implies that $\mathcal{NC} \neq \mathcal{PSPACE}$.

We have already seen that a simple parallel algorithm for the directed reachability problem is based on matrix multiplication, and in fact, many simple parallel graph algorithms are based on matrix operations. Csanky has given an $\mathcal{NC}$ algorithm to compute the rank and the determinant of a matrix over the reals in $O(\log^2 n)$ time. As a corollary, we get a parallel algorithm to solve systems of linear equations. Berkowitz, Chistov and Mulmuley have extended these results to matrices over arbitrary fields. One of the most beautiful connections between matrix operations and graph algorithms is that a perfect matching in a graph can be found by an efficient randomized parallel algorithm using only a single matrix inversion (see Chapter 3).

There has been substantial work over the past several years in finding efficient parallel algorithms for combinatorial problems. Some of these algorithms are mentioned elsewhere in this Handbook. For further results and more details the interested reader is referred to the survey of Karp & Ramachandran (1990).

Acknowledgments

In writing this survey, we have often been aided by the expertise of others. In particular, we would like to thank Laci Babai and Michael Sipser for sharing with us many of their insights into complexity theory. The surveys that we have cited in particular areas of complexity theory were extremely useful and we wish express our gratitude to their authors. In addition, we would like to thank Shafi Goldwasser, Leslie Hall, David Johnson, Jan Karel Lenstra, Laci Lovász, Albert Meyer, Carolyn Haibt Norton and Larry Stockmeyer for their many insightful comments on an earlier draft. The research of the first author was supported in part by an NSF Presidential Young Investigator award CCR-89-96272 with matching support from IBM, UPS and Sun and by Air Force Contract AFOSR-86-0078. The research of the second author was supported in part by Air Force Contract AFOSR-86-0078 and by a Packard Research Fellowship. Both were supported in part by the National Science Foundation, the Air Force Office of Scientific Research, and the Office of Naval Research, through NSF grant DMS-8920550.

References

Aho, A. V., J. E. Hopcroft and J. D. Ullman

[1974] *The Design and Analysis of Computer Algorithms*, (Addison–Wesley, Reading, Massachusetts).

Aleliunias, R., R. M. Karp, R. J. Lipton, L. Lovasz and C. Rackoff

[1979] Random walks, universal traversal sequences, and the complexity of maze problems, in: *Proceedings of the 20th Annual IEEE Symposium on Foundations of Computer Sciences*, (IEEE Computer Society Press, Washington, D.C.) pp. 218–223.

Babai, L.

[1985] Trading group theory for randomness, in: *Proceedings of the 17th Annual ACM Symposium on Theory of Computing*, (ACM Press, New York, NY) pp. 421–429.

Baker, T., J. Gill and R. Solovay

[1975] Relativizations of the $P = ?NP$ question, *SIAM J. on Comput.*, **4**, 431–442.

Chandra, A. K., D. C. Kozen and L. J. Stockmeyer

[1981] Alternation, *J. Assoc. Comput. Mach.*, **28**, 114–133.

Cobham, A.

[1965] The intrinsic computational difficulty of functions, in: *Proceedings of the 1964 International Congress for Logic, Methodology and Philosophy of Science*, edited by Y. Bar–Hillel, (North–Holland, Amsterdam) pp. 24–30.

Coffman, Jr., E. G., G. S. Lueker and A. H. G. Rinnooy Kan

[1988] Asymptotic methods in the probabilistic analysis of sequencing and packing heuristics, *Management Sci.*, **34**, 266–290.

Cook, S. A.

[1971] The complexity of theorem–proving procedures, in: *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing*, (ACM Press, New York, NY) pp. 151–158.

Davis, M.

[1977] Unsolvable problems, in: *Handbook of Mathematical Logic*, edited by J. Barwise (North–Holland, Amsterdam) pp. 567–594.

Edmonds, J.

[1965] Paths, trees and flowers, *Canad. J. Math.*, **17**, 449–467.

Fischer, M. J. and M. O. Rabin

[1974] Super–exponential complexity of Presburger arithmetic, in: *Complexity of Computation*, SIAM–AMS Proceedings, Vol. 7, edited by R.M. Karp, (American Mathematical Society, Providence, Rhode Island) pp. 27–41.

Furst, M., J. B. Saxe and M. Sipser

[1981] Parity, circuits, and the polynomial–time hierarchy, *Math. Systems Theory*, **17**, 13–27.

Garey, M. R. and D. S. Johnson

[1976] The complexity of near–optimal graph coloring, *J. Assoc. Comput. Mach.*, **23**, 43–49.

[1979] *Computers and Intractability: A Guide to the Theory of NP–Completeness*, (W. H. Freeman and Co., San Francisco).

Gödel, K.

[1931] Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme, I., *Monatsh. Math. Phys.*, **38**, 173–198.

GOLDWASSER, S.
[1989] Interactive proof systems, in: *Computational Complexity Theory*, AMS Symposia in Applied Mathematics, **38**, edited by J. Hartmanis (AMS, Providence, Rhode Island), pp. 108–128.

GOLDWASSER, S., S. MICALI and C. RACKOFF
[1989] The knowledge complexity of interactive proof–systems, *SIAM J. Comput.*, **18**, 186–208.

GRAHAM, R. L.
[1966] Bounds for certain multiprocessing anomalies, *Bell System Tech. J.*, **45**, 1563–1581.

HARTMANIS, J. and R. E. STEARNS
[1965] On the computational complexity of algorithms, *Trans. Amer. Math. Soc.*, **117**, 285–306.

HOPCROFT, J., W. PAUL and L. VALIANT
[1977] On time versus space, *J. Assoc. Comput. Mach.*, **24**, 332–337.

HOPCROFT, J. E. and J. D. ULLMAN
[1979] *Introduction to Automata Theory, Languages, and Computation*, (Addison–Wesley, Reading, MA).

IMMERMAN, N.
[1988] Nondeterministic space is closed under complementation, *SIAM J. Comput.*, **17**, 935–938.

JOHNSON, D. S.
[1984] The NP–Completeness Column: An Ongoing Guide, *J. Algorithms*, **5**, 284–299.
[1988] The NP–Completeness Column: An Ongoing Guide, *J. Algorithms*, **9**, 426–444.
[1992] The NP–Completeness Column: An Ongoing Guide, *J. Algorithms*, **13**, 502–524.

KARP, R. M.
[1972] Reducibility among combinatorial problems, in: *Complexity of Computer Computations*, edited by R. E. Miller and J. W. Thatcher, (Plenum Press, New York) pp. 85–103.
[1976] The probabilistic analysis of some combinatorial search algorithms, in: *Algorithms and Complexity: New Directions and Recent Results*, edited by J. F. Traub, (Academic Press, New York) pp. 1–19.

KARP, R. M., J. K. LENSTRA, C. J. H. MCDIARMID and A. H. G. RINNOOY KAN
[1985] Probabilistic analysis, in: *Combinatorial Optimization: Annotated Bibliographies*, edited by M. O'hEigeartaigh, J. K. Lenstra and A. H. G. Rinnooy Kan (Centre for Mathematics and Computer Science, Amsterdam and John Wiley & Sons, Chichester) pp. 52–88.

KARP, R. M. and V. RAMACHANDRAN
[1990] Parallel algorithms for shared–memory machines, in: *The Handbook of Theoretical Computer Science, Volume A, Algorithms and Complexity*, edited by J. van Leeuwen (Elsevier, Amsterdam) pp. 869–941.

VAN LEEUWEN, J., editor
[1990] *Handbook of Theoretical Computer Science* (Elsevier, Amsterdam).

LENSTRA, A. K. and H. W. LENSTRA, JR
[1990] Algorithms in number theory, in: *The Handbook of Theoretical Computer Science, Volume A, Algorithms and Complexity*, edited by J. van Leeuwen (Elsevier, Amsterdam) pp. 673–715.

LEVIN, L. A.
[1973] Universal sequential search problems, *Problemy Peredachi Informatsii*, **9**, 115–116 (in Russian)

[English translation: *Problems in Information Transmission*, **9**, 265–266].
[1986] Average case complete problems, *SIAM J. Comput.*, **15**, 285–286.

MEYER, A. R. and L. J. STOCKMEYER
[1972] The equivalence problem for regular expressions with squaring requires exponential time, in: *Proceedings of the 13th Annual Symposium on Switching and Automata Theory*, (IEEE Computer Society Press, Washington, D.C.) pp. 125–129.

PAUL, W. J., N. PIPPENGER, E. SZEMEREDI and W. T. TROTTER
[1983] On determinism versus non-determinism and related problems, in: *Proceedings of the 24th IEEE Symposium on Foundations of Computer Science*, (IEEE Computer Society Press, Washington, D.C.) pp. 429–438.

SAVITCH, W. J.
[1970] Relationships between nondeterministic and deterministic tape complexities, *J. Comput. System Sci.*, **4**, 177–192.

SHAMIR, R.
[1987] The efficiency of the simplex method: a survey, *Management Sci.*, **33**, 301–334.

STOCKMEYER, L. J.
[1987] Classifying the computational complexity of problems, *J. Symbolic Logic*, **52**, 1–43.

SZELEPCSENYI, R.
[1987] The method of forcing for nondeterministic automata, *Bull. European Assoc. Theoret. Comput. Sci.*, **33**, 96–100.

TURING, A. M.
[1936] On computable numbers, with an application to the Entscheidungsproblem, *Proc. London Math. Soc. II*, **2**, 230–265.

VALIANT, L. G.
[1979] The complexity of computing the permanent, *Theoret. Comput. Sci.*, **8**, 189–201.

YAO, A. C.–C.
[1985] Separating the polynomial–time hierarchy by oracles, in: *Proceedings of the 26th Annual IEEE Symposium on Foundations of Computer Science*, (IEEE Computer Society Press, Washington, D.C.) pp. 1–10.