Computational Complexity

Oded Goldreich
Department of Computer Science
Weizmann Institute of Science
Rehovot, Israel.
oded.goldreich@weizmann.ac.il

Avi Wigderson
School of Mathematics
Institute for Advanced Study
Princeton, NJ, USA.
avi@ias.edu

October 3, 2004

Abstract

The strive for efficiency is ancient and universal, as time is always short for humans. Computational Complexity is a mathematical study of the what can be achieved when time (and other resources) are scarce.

In this brief article we will introduce quite a few notions: Formal models of computation, and measures of efficiency; the P vs. NP problem and NP-completeness; circuit complexity and proof complexity; randomized computation and pseudorandomness; probabilistic proof systems, cryptography and more. A glossary of complexity classes is included in an appendix. We highly recommend the given bibliography and the references therein for more information.

Contents

1 Introduction

Computational Complexity (or Complexity Theory) is a central subfield of the theoretical foundations of Computer Science. It is concerned with the study of the *intrinsic complexity* of computational tasks. This study tends to *aim at generality*: it focuses on natural computational resources, and considers the effect of limiting these resources on the class of problems that can be solved. It also tends to *asymptotics*: studying this complexity as the size of data grows. Another related subfield (represented in this volume) deals with the design and analysis of algorithms for *specific* (classes of) computational problems that arise in a variety of areas of mathematics, science and engineering.

The (half-century) history of Complexity Theory has witnessed two main research efforts (or directions). The first direction is aimed towards actually establishing concrete lower bounds on the complexity of problems, via an analysis of the evolution of the process of computation. Thus, in a sense, the heart of this direction is a "low-level" analysis of computation. Most research in circuit complexity and in proof complexity falls within this category. In contrast, a second research effort is aimed at exploring the connections among computational problems and notions, without being able to provide absolute statements. This effort may be viewed as a "high-level" study of computation. The theory of NP-completeness, the study of probabilistic proof systems as well as pseudorandomness and cryptography all falls within this category.

2 Preliminaries

This exposition considers only *finite* objects, encoded by finite binary sequences, called strings. For a natural number n, we denote by $\{0,1\}^n$ the set of all binary sequences of length n, hereafter referred to as n-bit strings. The set of all strings is denoted $\{0,1\}^*$; that is, $\{0,1\}^* = \cup_{n\in\mathsf{N}}\{0,1\}^n$. For $x \in \{0,1\}^*$, we denote by $|x|$ the length of x (i.e., $x \in \{0,1\}^{|x|}$). At times, we associate $\{0,1\}^* \times \{0,1\}^*$ with $\{0,1\}^*$. Natural numbers will be encoded by their binary expansion.

2.1 Computability and Algorithms

We are all familiar with computers, and the ability of computer programs to manipulate data. But how does one capture *all* computational processes? Before being formal, we offer a loose description, capturing many artificial as well as natural processes, and invite the reader to compare it with physical theories.

A *computation* is a process that modifies an environment via repeated applications of a predetermined rule. The key restriction is that this rule is *simple*: in each application it depends and affects only a (small) portion of the environment, called the active zone. We contrast the *a-priori bounded* size of the active zone (and of the modification rule) with the *a-priori unbounded* size of the entire environment. We note that, although each application of the rule has a very limited effect, the effect of many applications of the rule may be very complex. The computation rule (especially when designed to effect a desired computation) is often referred to as an algorithm.

Such processes naturally compute functions, and their complexity is naturally captured by the number of steps they apply. Let us elaborate.

We are interested in the transformation of the environment effected by the computational process. Typically, the initial environment to which the computation is applied encodes an input string, and the end environment (i.e., at termination of the computation)[1] encodes an output string. We

[1] We assume that, when invoked on any finite initial environment, the computation halts after a finite number of

consider the mapping from inputs to outputs induced by the computation; that is, for each possible input x, we consider the output y obtained at the end of a computation initiated with input x, and say that the computation maps input x to output y. We also consider the number of steps (i.e., applications of the rule) taken by the computation for each input. The latter function is called the time complexity of the computational process. While time complexity is defined per input, one often considers it per input length, taking the maximum over all inputs of the same length.

To define computation (and computation time) rigorously, one needs to specify some model of computation; that is, provide a concrete definition of environments and a class of rules that may be applied to them. Such a model corresponds to an abstraction of a real computer (be it a PC, mainframe or network of computers). One simple abstract model that is commonly used is that of *Turing machines* (see, e.g., [17]). Thus, specific algorithms (and their complexity) are typically formalized by corresponding Turing machines. We stress however that most results in the Theory of Computation hold regardless of the specific computational model used, as long as it is "reasonable" (i.e. satisfies the aforementioned simplicity condition).

The above discussion has implicitly referred to computations and Turing machines as a means of computing functions. Specifically, a Turing machine M computes the function $f_M : \{0,1\}^* \to \{0,1\}^*$ defined by $f_M(x) = y$ if, when invoked on input x, machine M halts with output y. (For example, we may refer to the computation of the integer multiplication function, which given an encoding of two integers returns the encoding of their product.) However, computations can also be viewed as a means of "solving problems" or "making decisions", which are captured (respectively) by relations and sets.

Search problems are captured by binary relations $R \subseteq \{0,1\}^* \times \{0,1\}^*$, with the semantics that y is called a (valid) solution for problem instance x if and only if $(x,y) \in R$. Machine M solves the search problem R if $(x, f_M(x)) \in R$ whenever a solution for x exists; that is, given an instance x that has a valid solution, machine M finds some valid solution for x. (For example, we may refer to a machine that, given a system of polynomial equations, returns a valid solution.)

Decision problems are captured by sets $S \subseteq \{0,1\}^*$, with the semantics that S is the set of "yes-instances" (of the problem). We say that M solves the decision problem S if it holds that $f_M(x) = 1$ if and only if $x \in S$; that is, given an instance x, machine M determines whether or not $x \in S$. (For example, we may refer to a machine that, given a natural number, determines whether or not it is prime.) At times, it will be convenient to view decision problems as Boolean functions defined on the set of all strings (i.e., $S : \{0,1\}^* \to \{0,1\}$) rather than as sets of strings (i.e., $S \subseteq \{0,1\}^*$).

In the rest of this exposition we associate the machine M with the function f_M computed by it; that is, we write $M(x)$ instead of $f_M(x)$.

2.2 Efficient Computability and the class P

So far we have mathematically defined all tasks that can be computed *in principle*, and the time such computations take. Now we turn to define what can be computed *efficiently*, and then discuss the choices made in this definition.

We call an algorithm *efficient* if it terminates within time that is *polynomial* in the length of its inputs. Understanding the class of problems (called P below) that have such algorithms is *the* major goal of Computational Complexity Theory.

Definition 1 (the complexity class $\mathcal{P}$) *A decision problem $S \subseteq \{0,1\}^*$ is* solvable in polynomial

steps.

time *if there exists a (deterministic) polynomial-time Turing machine M such that $M(x) = 1$ if and only if $x \in S$. The class of decision problems that are solvable in polynomial time is denoted $\mathcal{P}$.*

The *asymptotic analysis* of running-time (i.e., considering running-time as a function of the input length) turned out to be crucial for revealing structure in the theory of efficient computation. The choice of *polynomial time* may seem arbitrary (and theories could be developed with other choices), but again proved itself right. Polynomials are viewed as the canonical class of slowly growing functions that enjoy closure properties relevant to computation. Specifically, the class is closed under addition, multiplication and composition. The growth rate of polynomials allows us to consider as efficient essentially all problems for which practical computer programs exist, and the closure properties of polynomials guarantee robustness of the notion of efficient computation. Finally, while n^{100}-time algorithms are called efficient here despite their blatant impracticality, one rarely discovers even an n^{10}-time algorithm for a natural problem (and when this happens, improvements to n^3 or n^2-time, which border on the practical, typically follow).

It is important to contrast $\mathcal{P}$ to the class $\mathcal{EXP}$, of all problems solvable in time *exponential* in the length of their inputs. Exponential running time is considered blatantly *inefficient*, and if the problem has no faster algorithm, then it is deemed intractable. It is known (via a basic technique called *diagonalization* that $\mathcal{P} \neq \mathcal{EXP}$; furthermore, some problems in $\mathcal{EXP}$ do require exponential time. We note that almost all problems and classes considered in this paper will be in $\mathcal{EXP}$ via trivial, "brute force" algorithms, and the main question will be if much faster algorithms can be devised for them.

Note that so far we restricted computation to be a *deterministic* process. In Section 6 we pursue the important relaxation of allowing randomness (coin tosses) in computation, and its impact on efficiency and other computational notions.

3 The P versus NP Question

In a nutshell, the P versus NP question is *whether creativity can be automated.* This applies to all tasks for which a successful completion can be easily recognized. A particular special case, which is in fact quite general and has natural appeal to Mathematicians, is the task of determining if a mathematical statement is true. Here successful completion is a proof, so the P versus NP Question can be informally stated as whether verifying proofs (which we view as a mechanical process) is, or is not, much easier than finding a proof (which we view as creative). In general, the class NP captures all problems for which an adequate solution (when given) can be efficiently verified, while the class P captures all problems that can be solved efficiently (without such external help). We now turn to formally define these notions.

3.1 Efficient Verification and the class NP

The fundamental class $\mathcal{NP}$ of decision problems consists of the class of sets S for which there exist *short* proofs that $x \in S$ of membership, that which can be *efficiently verified.* These two ingredients are captured by two properties of an auxiliary binary relation $R_S \in \mathcal{P}$ in which all y for which $(x, y) \in R_S$ have polynomial (in $|x|$) length, and such a "proof" y exists iff $x \in S$ (thus certifying, or witnessing, or proving this fact).[2]

[2]The acronym NP stands for Non-deterministic Polynomial-time, where a *non-deterministic machine* is a *fictitious* computing device used in an alternative definition of the class $\mathcal{NP}$. The non-deterministic moves of such a machine correspond to guessing a "proof" in Definition 2.

Definition 2 (the complexity class $\mathcal{NP}$) *A binary relation $R \subseteq \{0,1\}^* \times \{0,1\}^*$ is called* polynomially bounded *if there exists a polynomial p such that $|y| \leq p(|x|)$ holds for every $(x,y) \in R$. A decision problem S is in $\mathcal{NP}$ if there exists a polynomially bounded binary relation R_S such that R_S is in $\mathcal{P}$ and $x \in S$ if and only if there exists y such that $(x,y) \in R_S$. Such a y is called a* proof (or witness) of membership *of $x \in S$.*

We note that trivially $\mathcal{NP} \subseteq \mathcal{EXP}$, since we can go over all possible y's in exponential time. Can this trivial algorithm be improved? Since $\mathcal{P}$ is the class of sets for which membership can be efficiently decided (without being given a proof), it follows that $\mathcal{P} \subseteq \mathcal{NP}$. Thus, the P versus NP Question can be cast as follows: *does the existence of an efficient verification procedure for proofs of membership in a certain set imply the ability to efficiently decide membership in the same set?*

Open Problem 3 *Is $\mathcal{NP}$ equal to $\mathcal{P}$?*

Natural search problems arise from *every* polynomially bounded relation $R \in \mathcal{P}$; namely, given x, *find* any y for which $(x,y) \in R$ (if such a solution exists). Note that the polynomial bound on the length of y guarantees that the search problem is not trivially intractable (as would be the case if all solutions had length that is super-polynomial in the length of the instance). Furthermore, $R \in \mathcal{P}$ implies that the search problem is natural in the sense that one can (efficiently) recognize the validity of a solution to a problem instance. One often views $\mathcal{NP}$ as the class of all such search problems; that is, the class of search problems referring to relations $R \in \mathcal{P}$ that are polynomially bounded. The search analog of the P versus NP question is *whether the efficient verification of candidate solutions necessarily entails that valid solutions are easy to find.* Indeed, the search and decision versions of the P versus NP question are equivalent.

3.2 The Big Conjecture

It is widely believed that $\mathcal{P} \neq \mathcal{NP}$. Settling this conjecture is certainly the most important open problem in Computer Science, and among the most significant in Mathematics. The $\mathcal{P} \neq \mathcal{NP}$ Conjecture is supported by our strong intuition, developed over centuries in a variety of human activities, that finding solutions is far harder than verifying their adequacy. Further empirical evidence in favor of the conjecture is given by the fact that literally thousands of NP problems, in a wide variety of mathematical and scientific disciplines, are not known to be solvable in polynomial time, in spite of extensive research attempts aimed at providing efficient procedures for solving them. One famous example is the Integer Factorization problem: given a natural number, find its prime factorization.

The section on Circuit Complexity (Section 5.1) is devoted to attempts to prove this conjecture, discussing some partial results and limits of the techniques used so far.

3.3 NP versus coNP

Assuming that $\mathcal{P} \neq \mathcal{NP}$, it is unclear whether the existence of an efficient verification procedure for proofs of membership in a set implies the existence of an efficient verification procedure for proofs of non-membership in that set. Let co$\mathcal{NP}$ denote the class of sets that are complements of sets in $\mathcal{NP}$ (i.e., co$\mathcal{NP} \stackrel{\text{def}}{=} \{\{0,1\}^* \setminus S : S \in \mathcal{NP}\}$).

Open Problem 4 *Is $\mathcal{NP}$ equal to co$\mathcal{NP}$?*

It is widely believed that co$\mathcal{NP} \neq \mathcal{NP}$. (Indeed, this implies $\mathcal{P} \neq \mathcal{NP}$). Here again intuition from Mathematics is extremely relevant: to verify that a set of logical constraints is mutually *inconsistent*, that a family of polynomial equations have *no* common root, that a set of regions in space has *empty* intersection, seems far harder to prove than their complements (exhibiting the consistent valuation, root, point resp.). Indeed, only when (rare) extra mathematical structure is available do we have duality theorems, or complete systems of invariants, implying (computational) equivalence of the set and its complement. The section on Proof Complexity (Section 5.3) deals further with this conjecture, and attempts to resolve it.

4 Reducibility and NP-Completeness

In this section we attempt to identify the "hardest" problems in $\mathcal{NP}$. For this we shall define a natural partial order on decision problems, called polynomial-time reducibility, and define maximal elements in $\mathcal{NP}$ under this order to be "complete". We note that reductions and completeness are key concepts in Complexity Theory.

A *general notion* of (*polynomial-time*) reducibility among computational problems is obtained by considering a (polynomial-time) machine for solving one problem (e.g., computing a function f) that may issue queries to another problem (e.g., to a function g)[3]. Thus, if the latter problem can be solved efficiently then so can the former. One restricted notion of a reduction, which refers to decision problems, requires the reduction machine to issue a single query and output the answer it obtains. In this case, a simpler formulation follows:

Definition 5 (Polynomial-time Reducibility) *A set S is* polynomial-time reducible *to the set T if there exist polynomial-time computable function h such that $x \in S$ if and only if $h(x) \in T$.*

Definition 6 (NP-Completeness) *A decision problem S is* NP-complete *if S is in $\mathcal{NP}$ and every decision problem in $\mathcal{NP}$ is polynomial-time reducible to S.*

Thus, NP-complete (decision) problems are "universal" in the sense that providing a polynomial-time procedure for solving *any* of them will immediately imply polynomial-time procedures for solving all other problems in $\mathcal{NP}$ (and in particular *all* NP-complete decision problems). Furthermore, in a sense, each of these (NP-complete) problems "efficiently encodes" all the other problems and, in fact, all NP search problems. For example, the Integer Factorization problem can be "efficiently encoded" in any NP-complete problem (which may have nothing to do with integers). Thus, at first glance, it seems very surprising that NP-complete problems exist at all.

Theorem 7 *There exist NP-complete decision problems. Furthermore, the following decision problems are NP-complete:*

`SAT`: *Given a propositional formula, decide whether or not it is satisfiable.*[4]

`3-Coloring`: *Given a planar map, decide whether or not it is 3-colorable.*[5]

[3]Such a machine is called an oracle machine, and in the above case we say that it computes the function f by issuing queries to the oracle (function) g such that for query q the answer is $g(q)$.

[4]The problem remains NP-complete even when instances are restricted to be in Conjunctive Normal Form (CNF), and even when each clause has exactly three literals. These formulae are said to be in 3CNF form, and the set of satisfiable 3CNF formulae is denoted 3`SAT`.

[5]Recall that the celebrated *4-color Theorem* asserts that 4 colors always suffice. In contrast to the NP-completeness of deciding 3-colorability, it is easy to decide 2-colorability of arbitrary graphs (and in particular of planar maps).

`Subset Sum`: *Given a sequence of integers $a_1, ..., a_n$ and b, decide whether or not there exists a set I such that $\sum_{i \in I} a_i = b$.*

The decision problems mentioned above are but three examples among literally thousands of natural NP-complete problems, from a wide variety of mathematical and scientific disciplines. Hundreds of such problems are listed in [5].

Assuming that $\mathcal{P} \neq \mathcal{NP}$, no NP-complete problem has a polynomial-time decision procedure. Consequently, the corresponding NP search problems cannot be solved in polynomial time. Thus, proofs of NP-completeness are often taken as evidence to the intrinsic difficulty of a problem.

Positive applications of NP-completeness are also known: in some cases a claim regarding all NP-sets is proved by establishing the claim only for some NP-complete set (and noting that polynomial-time reductions preserve the claimed property). Famous examples include the existence of Zero-Knowledge proofs, established first for 3-coloring (see Section 6.2.2), and the PCP Theorem, established first for 3-SAT (see Section 6.2.3).

We note that almost every natural problem in $\mathcal{NP}$ ever considered turns out to be either NP-complete or in $\mathcal{P}$. Curiously, only a handful of natural problems, including Integer Factorization and Graph Isomorphism, are not known to belong to either of these classes (and indeed there is strong evidence they don't).

5 Lower Bounds

In this section we survey some basic attempts at proving lower bounds on the complexity of natural computational problems. In the first part, Circuit Complexity, we describe lower bounds for the *size* of circuits that solve natural computational problems. This can be viewed as a program whose long-term goal is proving that $\mathcal{P} \neq \mathcal{NP}$. In the second part, Proof Complexity, we describe lower bounds on the length of propositional proofs of natural tautologies. This can be viewed as a program whose long-term goal is proving that $\mathcal{NP} \neq \text{co}\mathcal{NP}$. Both models refer to the finite model of *directed acyclic graphs* (DAGs), which we define next.

A DAG $G(V, E)$ consists of a finite set of vertices V, and a set of ordered pairs called directed edges $E \subseteq V \times V$, in which there are no directed cycles. The vertices with no incoming edges are called the inputs of the DAG G, and the vertices with no outgoing edges are called the outputs. We will restrict ourselves to DAGs in which the number of *incoming* edges to every vertex is at most 2. If the number of *outgoing* edges from every node is at most 1, the DAG is called a tree. Finally, we assume that every vertex can be reached from some input via a directed path. The size of a DAG will be its number of edges.

To make a DAG into a computational device (or a proof), each non-input vertex will be marked by a rule, converting values in its predecessors to values at that vertex. It is easy to see that the vertices of every DAG can be linearly ordered, such that predecessors of every vertex (if any) appear before it in the ordering. Thus, if the input vertices are labeled with some values, we can label the remaining vertices (in that order), one at a time, till all vertices (and in particular all outputs) are labeled.

For computation, the non-input vertices will be marked by functions (called gates) which make the DAG a circuit. If we label the input vertices by specific values from some domain, the outputs will be determined by them, and the circuit will naturally define a function (from input values to output values).

For proofs, the non-input vertices will be marked by sound deduction (or inference) rules, which make the DAG a proof. If we label the inputs by formulae that are axioms in a given proof system, the output again will be determined by them, and will yield the tautology proved by this proof.

We note that both settings fit the paradigm of simplicity shared by computational models discussed in the previous section; the rules are simple by definition – they are applied to at most 2 previous values. The main difference is that this model is finite – each DAG can compute only functions/proofs with a fixed input length. To allow all input lengths, one must consider families of DAGs, one for each, thus significantly extending the power of the computation model beyond that of the notion of *algorithm* defined earlier. However, as we are interested in lower bounds here, this is legitimate, and one can hope that the finiteness of the model will potentially allow for combinatorial techniques to analyze its power and limitations. Furthermore, these models allow for the introduction (and study) of meaningful restricted classes of computations.

We use the following asymptotic notation: For $f, g : \mathsf{N} \to \mathsf{N}$, by $f = O(g)$ (resp., $f = \Omega(g)$) we mean that there exists a constant $c > 0$ such that $f(n) \leq c \cdot g(n)$ (resp., $f(n) \geq c \cdot g(n)$) for all $n \in \mathsf{N}$.

5.1 Boolean Circuit Complexity

In Boolean circuits all inputs, outputs, and values at intermediate nodes of the DAG are bits. The set of allowed gates is naturally taken to be a *complete basis* – one that allows the circuit to compute *all* Boolean functions. The specific choice of a complete basis hardly effects the study of circuit complexity. A typical choice is the set $\{\wedge, \vee, \neg\}$ of (respectively) conjunction, disjunction (each on 2 bits) and negation (on 1 bit).

Definition 8 *Denote by $\mathcal{S}(f)$ the size of the smallest Boolean circuit computing f.*

We will be interested in sequences of functions $\{f_n\}$, where f_n is a function on n input bits, and will study the complexity $\mathcal{S}(f_n)$ asymptotically as a function of n. With some abuse of notation, for $f(x) \stackrel{\text{def}}{=} f_{|x|}(x)$, we let $\mathcal{S}(f)$ denote the integer function that assigns to n the value $\mathcal{S}(f_n)$.

We note that different circuits (in particular having a different number of inputs) are used for each f_n. Still there may be a simple description of this sequence of circuits, say, an algorithm that on input n produces a circuit computing f_n. In case such an algorithm exists and works in time polynomial in the size of its output, we say that the corresponding sequence of circuits is uniform. Note that if f has a uniform sequence of polynomial-size circuits then $f \in \mathcal{P}$. On the other hand, it can be shown that any $f \in \mathcal{P}$ has (a uniform sequence of) polynomial-size circuits. Consequently, a super-polynomial circuit lower-bound on any function in $\mathcal{NP}$ would imply that $\mathcal{P} \neq \mathcal{NP}$.

But Definition 8 makes no reference to "uniformity" and indeed the sequence of smallest circuits computing $\{f_n\}$ may be highly "nonuniform". Indeed, non-uniformity makes the circuit model stronger than Turing machines (or, equivalently, than the model of uniform circuits): there exist functions f that cannot be computed by Turing machines (regardless of their running time), but do have linear-size circuits. So isn't proving circuit lower bounds a much harder task than we need to resolve the P vs. NP question?

The answer is that there is a strong sentiment that the extra power provided by non-uniformity is irrelevant to the P vs. NP question; that is, it is conjectured that NP-complete sets do not have polynomial-size circuits. This conjecture is supported by the fact that its failure will yield an unexpected collapse in the complexity of standard computations. Furthermore, the hope is that abstracting away the (supposedly irrelevant) uniformity condition will allow for combinatorial techniques to analyze the power and limitations of polynomial-size circuits (w.r.t NP-sets). This hope has materialized in the study of restricted classes of circuits (see Sections 5.1.2 and 5.1.3).

We also mention that Boolean circuits are a natural computational model, corresponding to "hardware complexity", and so their study is of independent interest. Moreover, some of the techniques for analyzing Boolean functions found applications elsewhere (e.g., in computational learning theory, combinatorics and game theory).

5.1.1 Basic Results and Questions

We have already mentioned several basic facts about Boolean circuits, in particular the fact that they can efficiently simulate Turing Machines. The next basic fact is that *most Boolean functions require exponential size circuits*, which is due to the gap between the number of functions and the number of small circuits.

So hard functions for circuits (and hence for Turing machines) abound. However, the hardness above is proved via a counting argument, and thus supplies no way of putting a finger on one hard function. Using more conventional language – we cannot prove such hardness for any *explicit* function f (e.g., for an NP-complete function like SAT or even for functions in $\mathcal{EXP}$). The situation is even worse – no *nontrivial* lower-bound is known for any explicit function. Note that for any function f on n bits (which depends on all its inputs), we trivially must have $\mathcal{S}(f) \geq n$, just to read the inputs. The main open problem of circuit complexity is beating this trivial bound.

Open Problem 9 *Find an explicit Boolean function f (or even a length-preserving function f) for which $\mathcal{S}(f)$ is not $O(n)$.*

A particularly basic special case of this problem, is the question whether addition is easier to perform than multiplication. Let $\texttt{ADD}:\{0,1\}^n \times \{0,1\}^n \to \{0,1\}^{n+1}$ and $\texttt{MULT}:\{0,1\}^n \times \{0,1\}^n \to \{0,1\}^{2n}$, denote, respectively, the addition and multiplication functions on a pair of integers (presented in binary). For addition we have an optimal upper bound; that is, $\mathcal{S}(\texttt{ADD}) = O(n)$. For multiplication, the standard (elementary school) quadratic-time algorithm can be greatly improved (via Discrete Fourier Transforms) to slightly super-linear, yielding $\mathcal{S}(\texttt{MULT}) = O(n \cdot (\log n)^2)$. Now, the question is *whether or not there exist linear-size circuits for multiplication* (i.e., is $\mathcal{S}(\texttt{MULT}) = O(n)$)?

Unable to prove any nontrivial lower bound, we now turn to restricted models. There has been some remarkable successes in developing techniques for proving strong lower bounds for natural restricted classes of circuits. We describe the most important ones.

General Boolean circuits, as described above, can compute every function and can do it at least as efficiently as general (uniform) algorithms. Restricted circuits may be only able to compute a subclass of all functions (e.g., monotone functions). The restriction makes sense when either the related classes of functions or the computations represented by the restricted circuits are natural, from a programming or a mathematical viewpoint. The models discussed below satisfy this condition.

5.1.2 Monotone Circuits

An extremely natural restriction comes by forbidding negation from the set of gates, namely allowing only $\{\wedge, \vee\}$. The resulting circuits are called monotone circuits and it is easy to see that they can compute every function $f:\{0,1\}^n \to \{0,1\}$ that is monotone with respect to the standard partial order on n-bit strings ($x \preceq y$ iff for every bit position i we have $x_i \leq y_i$).

It is as easy to see that most monotone functions require exponential size monotone circuits. Still, proving a super-polynomial lower bound on an explicit monotone function was open for over 40 years, till the invention of the so-called *approximation method.*

Let CLIQUE be the function that, given a graph on n vertices (by its adjacency matrix), outputs 1 iff it contains a complete subgraph of size (say) $\sqrt{n}$ (namely, all pairs of vertices in some $\sqrt{n}$ subset are connected by edges). This function is clearly monotone. Moreover, it is known to be NP-complete.

Theorem 10 *There are no polynomial-size monotone circuits for* CLIQUE.

We note that similar lower-bounds are known for functions in $\mathcal{P}$.

5.1.3 Bounded-Depth Circuits

The next restriction is structural: we allow all gates, but limit the depth of the circuit. The depth of a DAG is simply the length of the longest directed path in it. So in a sense, depth captures the *parallel time* to compute the function: if a circuit has depth d, then the function can be evaluated by enough processors in d phases (where in each phase many gates are evaluated at once). Parallel time is another important computational resource.

We will restrict d to be a constant, which still is interesting not only as parallel time, but also due to the relation of this model to expressibility in first order logic as well as to complexity classes above NP called the *Polynomial-time Hierarchy* (see section III). In the current setting (of constant-depth circuits), we allow *unbounded fan-in* (i.e., $\wedge$-gates and $\vee$-gates taking any number of incoming edges), as otherwise each output bit can depend only on a constant number of input bits.

Let PAR (for parity) denote the sum modulo two of the input bits, and MAJ (for majority) be 1 iff there are more 1's than 0's among the input bits. The invention of the *random restriction method* led to the following basic result.

Theorem 11 *For all constant d,* PAR *and* MAJ *have no polynomial size circuit of depth d.*

Interestingly, MAJ remains hard (for constant-depth polynomial-size circuits) even if the circuits are also allowed (unbounded fan-in) PAR-gates (this result is based on yet another proof technique: *approximation by polynomials*). However the "converse" does not hold, and the class of constant-depth polynomial-size circuits with MAJ-gates seems quite powerful.

5.1.4 Formula Size

The final restriction is again structural – we require the DAG to be a tree. Intuitively, this forbids the computation from reusing a previously computed partial result (and if it is needed again, it has to be recomputed). The resulting circuits are simply formulae, which are natural not only for their prevalent mathematical use, but also since their size can be related to the *memory* requirements of a Turing machine. Here we go back to the standard basis of negation, and 2-bit input $\wedge, \vee$ gates.

One of the oldest results on Circuit Complexity, is that PAR and MAJ are nontrivial in this model. The proof follows a simple combinatorial (or information theoretic) argument.

Theorem 12 *Boolean formuale for n-bit* PAR *and* MAJ *require size $\Omega(n^2)$ size.*

This should be contrasted with the linear-size circuits that exist for both functions. We comment that $\mathcal{S}(\texttt{PAR}) = O(n)$ is trivial, but $\mathcal{S}(\texttt{MAJ}) = O(n)$ is not.

Can we give super-polynomial lower bounds on formula size? One of the cleanest methods suggested is the *communication complexity method*, which we demonstrate informally with an example.

Consider two players, the first having a *prime* number $x < 2^n$, and the second having a *composite* number $y < 2^n$. Clearly, any two such numbers must differ on at least one bit position in their binary expansion (i.e., there exists an i s.t. $x_i \neq y_i$), and it is the goal of the parties to find such an i. To that end, the party exchange messages, according to a pre-determined protocol, and the question is what is the communication complexity (in terms of total number of bits exchanged on the worst-case input pair) of the best such protocol. Proving a super-logarithmic lower-bound will establish (the widely believed conjecture) that testing primality has no polynomial size formulae. Note that a lower bound of purely information theoretic nature (no computational restriction were paced on the parties) implies a computational one!

5.1.5 Why Is It Hard to Prove Lower Bounds?

The failure to obtain (nontrivial) lower bounds for general circuit in a span of 60 years raises the question of whether there is a fundamental reason for this failure. The same may be asked about any long standing mathematical problem (e.g. the Riemann Hypothesis), and the typical (vague!) answer would be that, probably, the current tools and ideas (which may well have been successful at attacking related, easier problems) do not suffice. Complexity Theory can make this vague statement into a theorem! Thus we have a "formal excuse" for our failure so far: we can classify a general set of ideas and tools, which are responsible for virtually all restricted lower bounds known, yet must necessarily fail for proving general ones. This introspective result suggests a framework called *Natural Proofs*, which encapsulates all known lower bound techniques. It shows that natural proofs of general circuit lower bounds for explicit functions surprisingly imply (...) efficient algorithms of a type conjectured not to exist (e.g., for integer factoring).

One interpretation of the aforementioned result, is an "independence result" of general circuit lower bounds from a certain natural fragment of Peano Arithmetic.[6] This may hint that the P vs. NP problem may be independent from PA or even Set Theory, although few believe the latter to be the case.

5.2 Arithmetic Circuits

We now leave the Boolean rind, and discuss circuits over general fields. Fix any field F. The gates of the DAG will now be the standard $+$ and $\times$ operations in the field. This requires two immediate clarifications. First, to allow using constants of the field, one adds a special input vertex whose value is the constant '1' of the field. Moreover, multiplication by any field element (e.g., -1) is free. Second, one may wonder about division. However, we will be mainly interested in computing polynomials, and for computing polynomials (over infinite fields) division can be efficiently emulated by the other operations.

Now the inputs of the DAG will hold elements of the field F, and hence so will all computed values at vertices. Thus an arithmetic circuit computes a polynomial map $p : F^n \to F^m$, and every such polynomial map is computed by some circuit. We denote by $\mathcal{S}_F(p)$ the size of a smallest circuit computing p (when no subscript is given, $F = \mathcal{Q}$ the field of rational numbers). As usual, we'll be interested in sequences of polynomials, one for every input size, and will study size asymptotically.

It is easy to see that over any *fixed* finite field, arithmetic circuits can simulate Boolean circuits on Boolean inputs with only constant factor loss in size. Thus the study of arithmetic circuits focuses more on infinite fields, where lower bounds may be easier to obtain.

[6]This result as the aforementioned one rely on the existence of one-way functions, see Section 7.

As in the Boolean case, the existence of hard functions is easy to establish (via dimension considerations, rather than counting argument), and we will be interested in *explicit* (families of) polynomials. However, the notion of explicitness is more delicate here (e.g., allowing polynomials with algebraically independent coefficients would yield strong lower bounds, which are of no interest whatsoever). Very roughly speaking, polynomials are called explicit if the mapping from monomials to (a finite description of) their coefficients has an efficient program.

An important parameter, which is absent in the Boolean model, is the *degree* of the polynomial(s) computed. It is obvious, for example, that a degree d polynomial (even in one variable, i.e., $n = 1$) requires size at least $\log d$. We briefly consider the univariate case (in which d is the only measure of input size), which already contains striking and important problems. Then we move to the general multivariate case, in which as usual n, the number of inputs will be the main parameter.

5.2.1 Univariate Polynomials

How tight is the $\log d$ lower bounds for the size of an arithmetic circuit computing a degree d polynomial? A simple dimension argument shows that for most degree d polynomials p, $\mathcal{S}(p) = \Omega(d)$. However, we know of no explicit one:

Open Problem 13 *Find an explicit polynomial p of degree d, such that $\mathcal{S}(p)$ is not $O(\log d)$.*

Two concrete examples are illuminating. Let $p(x) = x^d$, and $q(x) = (x+1)(x+2)\cdots(x+d)$. Clearly $\mathcal{S}(p) \leq 2\log d$ (via repeated squaring), so the trivial lower bound is tight. On the other hand, it is a major open problem to determine $\mathcal{S}(q)$, and the conjecture is that $\mathcal{S}(q) \gg (\log d)^{O(1)}$. To realize the importance of this question, we state the following fact: *If $\mathcal{S}(q) \leq (\log d)^{O(1)}$, then Integer Factorization can be done in polynomial-time.*

5.2.2 Multivariate Polynomials

We are now back to polynomials with n variables. To make n our only input size parameter, it is convenient to restrict ourselves to polynomials whose total degree is at most n.

Once again, almost every polynomial p in n variables requires size $\mathcal{S}(p) \geq \exp(n/2)$, via a dimension argument, and we seek explicit polynomial (families) that are hard. Unlike in the Boolean world, here there are slightly nontrivial lower bounds (via elementary tools from algebraic geometry).

Theorem 14 $\mathcal{S}(x_1^n + x_2^n + \cdots + x_n^n) = \Omega(n \log n)$.

The same techniques extend to prove a similar lower-bound for other natural polynomials such as the symmetric polynomials and the determinant. Establishing a stronger lower-bound for any explicit polynomial is a major open problem. Another is obtaining a super-linear lower bound for a polynomial map of constant (even 1) total degree. Outstanding candidates for the latter are the *linear* maps computing the Discrete Fourier Transform over the Complex numbers, or the Walsh transform over the Rationals (for both $O(n \log n)$ algorithms are known, but no super-linear lower bounds).

We now focus on specific polynomials of central importance. The most natural and well studied candidate for the last open problem is the matrix multiplication function $\mathtt{MM}$: let A, B be two $m \times m$ matrices of variables over F, and define $\mathtt{MM}(A, B)$ to be the $n = m^2$ entries of the matrix $A \times B$. Thus, $\mathtt{MM}$ is a set of n explicit bilinear forms over the $2n$ input variables. It is known that $\mathcal{S}_{\mathrm{GF}(2)}(\mathtt{MM}) \geq 3n$. On the other hand, the obvious $m^3 = n^{3/2}$ algorithm can be improved.

Theorem 15 *For every field F, $\mathcal{S}_F(\mathtt{MM}) = O(n^{1.19})$.*

So what is the complexity of MM (even if one counts only multiplication gates)? Is it linear or almost-linear or is it the case that $\mathcal{S}(\mathtt{MM}) > n^{\alpha}$ for some $\alpha > 1$? This is indeed a famous open problem.

We next consider the determinant and permanent polynomials (DET and PER, resp.) over the $n = m^2$ variables representing an $m \times m$ matrix. While DET plays a major role in classical mathematics, PER is somewhat esoteric (though it appears in Statistical Mechanics and Quantum Mechanics). In the context of complexity theory both polynomials are of great importance, because they capture natural complexity classes. DET has relatively low complexity (and is closely related to the class of polynomials having polynomial-sized arithmetic formulae), whereas PER seems to have high complexity (and it is complete for the counting class $\#\mathcal{P}$ which contains $\mathcal{NP}$). Thus, it is conjectured that PER is *not* polynomial-time reducible to DET. A specific type of reduction that makes sense in this algebraic context is by projection.

Definition 16 *Let X and Y be two disjoint finite sets of variables. Let $p \in F[X]$ and $q \in F[Y]$ be two polynomials. We say that there is a* projection from p to q over F, *denoted $p \propto q$ if there exists a function $h : X \to Y \cup F$ such that $p(\bar{x}) \equiv q(h(\bar{x}))$.*

Clearly, if $p \propto q$ then $\mathcal{S}_F(p) \leq \mathcal{S}_F(q)$. Let $\mathtt{DET}_m$ and $\mathtt{PER}_m$ denote these functions restricted to m-by-m matrices. It is known that $\mathtt{PER}_m \propto \mathtt{DET}_{3^m}$, but to yield a polynomial-time reduction one would need a projection of $\mathtt{PER}_m$ to $\mathtt{DET}_{\mathrm{poly}(m)}$. It is conjectured that no such projection exists.

Open Problem 17 *Is $\mathtt{PER}_m \propto \mathtt{DET}_{m^{O(1)}}$?*

5.3 Proof Complexity

The concept of *proof* is what distinguishes the study of Mathematics from all other fields of human inquiry. Mathematicians have gathered millennia of experience to attribute such adjectives to proofs as "insightful, original, deep" and most notably, "difficult". Can one quantify, mathematically, the difficulty of proving various theorems? This is exactly the task undertaken in Proof Complexity. It seeks to classify theorems according to the difficulty of proving them, much like Circuit Complexity seeks to classify functions according to the difficulty of computing them. In proofs, just like in computation, there will be a number of models, called *proof systems* capturing the power of reasoning allowed to the prover.

We will consider only propositional proof systems, and so our theorems will be *tautologies.* We will see soon why the complexity of proving tautologies is highly nontrivial and amply motivated.

The formal definition of a proof system spells out what we take for granted: the efficiency of the verification procedure.[7]

Definition 18 *A* (propositional) proof system *is a polynomial-time Turing machine M with the property that T is a tautology if and only if there exist a* ("proof") *π such that $M(\pi, T) = 1$.*[8]

[7]Here efficiency of the verification procedure refers to its running-time measured in terms of the *total length of the alleged theorem and proof.* In contrast, in Sections 3.1 and 6.2, we consider the running-time as a function of the *length of the alleged theorem.*

[8]In agreement with standard formalisms (see below), the proof is seen as coming before the theorem.

Note that the definition guarantees completeness and soundness, as well as verification efficiency of the proof system. It judiciously ignores the size of the proof π (of the tautology T), which is a measure of how complex it is to prove T in the system M. For each tautology T, let $\mathsf{s}_M(T)$ denote the size of the shortest proof of T in M (i.e., the length of the shortest string π such that M accepts (π, T)). Abusing notation, we let $\mathsf{s}_M(n)$ denotes the maximum $\mathsf{s}_M(T)$ over all tautologies T of length n.

The following simple observation provides a basic connection of this concept with computational complexity, and the major question of Section 3.3.

Theorem 19 *There exists a proof system M such that s_M is polynomial if and only if $\mathcal{NP} = \text{co}\mathcal{NP}$.*

It is natural to start attacking this formidable problem by considering first simple (and thus weaker) proof systems, and then move on to more and more complex ones. Moreover, natural proof systems, capturing basic (restricted) types and "primitives" of reasoning, as well as natural tautologies, suggest themselves as objects for this study. In the rest of this section we focus on such restricted proof systems.

Different branches of Mathematics such as logic, algebra and geometry provide different such systems, often implicitly. A typical system would have a set of axioms, and a set of deduction rules. A proof would proceed to derive the desired tautology in a sequence of steps, each producing a formula (often called a line of the proof), which is either an axiom, or follows from previous formulae via one of the deduction rules. (Clearly, a Turing machine can easily verify the validity of such a proof).

This perfectly fit our DAG model.[9] The inputs will be labeled by the axioms, the internal vertices by deduction rules, which in turn "infer" a formula for that vertex from the formulae at the vertices pointing to it.

There is an equivalent and somewhat more convenient view of (simple) proof systems, namely as (simple) refutation systems. First, recalling that 3SAT is NP-complete (see Footnote 4), note that every (negation of a) tautology can be written as a conjunction of clauses, with each clause being a disjunction of only 3 literals (variables or their negation). Now, if we take these clauses as axioms, and derive (using the rules of the system) a contradiction (e.g., the negation of an axiom, or better yet the empty clause), then we have proved the tautology (since we have proved that its negation yields a contradiction). We will use the refutation viewpoint throughout, and often exchange "tautology" and its negation, "contradiction".

So we turn to study the proof length $\mathsf{s}_\Pi(T)$ of tautologies T in proof systems Π. The first observation, revealing a major difference between proof complexity and circuit complexity, is that the trivial counting argument *fails*. The reason is that, while the number of functions on n bits is 2^{2^n}, there are at most 2^n tautologies of this length. Thus in proof complexity, even the *existence* of a hard tautology, not necessarily explicit, would be of interest. As we shall see, however, most known lower bounds (in restricted proof systems) apply to very natural tautologies.

The rest of this section is divided to three parts, on logical, algebraic and geometric proof systems. We will briefly describe important representatives and basic results in each.

5.3.1 Logical Proof Systems

The proof systems in this section will all have lines that are Boolean formulae, and the differences will be in the structural limits imposed on these formulae.

[9] General proof systems as in Definition 18 can also be adapted to this formalism, by considering a deduction rule that corresponds to a single step of the machine M. However, the deduction rules considered below are even simpler, and more importantly they are natural.

The most basic proof system, called Frege system, puts no restriction on the formulae manipulated by the proof. It has one derivation rule, called the cut rule: $A \vee C, B \vee \neg C \vdash A \vee B$ (adding any other sound rule, like *modus ponens*, has little effect on the length of proofs in this system). Frege systems are basic in the sense that they (in several variants) are the most common in Logic, and in that polynomial length proofs in these systems naturally corresponds to "polynomial-time reasoning" about feasible objects.

The major open problem in proof complexity is to find any tautology (as usual we mean a family of tautologies) that has no polynomial-size proof in the Frege system.

As lower bounds for Frege are hard, we turn to subsystems of Frege which are interesting and natural. The most widely studied system is Resolution, whose importance stems from its use by most propositional (as well as first order) automated theorem provers. The formulae allowed in Resolution refutations are simply clauses (disjunctions), and so the derivation cut rule simplifies to the "resolution rule": $A \vee x, B \vee \neg x \vdash A \vee B$, for clauses A, B and variable x.

An example of a tautology that is easy for Frege and hard for Resolution, is the pigeonhole principle, $\mathtt{PHP}_n^m$, expressing the fact that there is no one-to-one mapping of m pigeons to $n < m$ holes.

Theorem 20 $\mathsf{s}_{\mathrm{Frege}}(\mathtt{PHP}_n^{n+1}) = n^{O(1)}$ *but* $\mathsf{s}_{\mathrm{Resolution}}(\mathtt{PHP}_n^{n+1}) = 2^{\Omega(n)}$

5.3.2 Algebraic Proof Systems

Just as a natural contradiction in the Boolean setting is an unsatisfiable collection of clauses, a natural contradiction in the algebraic setting is a system of polynomials without a common root. Moreover, CNF formulae can be easily converted to a system of polynomials, one per clause, over any field. One often adds the polynomials $x_i^2 - x_i$ which ensure Boolean values.

A natural proof system (related to Hilbert's Nullstellensatz, and to computations of Grobner bases in symbolic algebra programs) is Polynomial Calculus, abbreviated PC. The lines in this system are polynomials (represented explicitly by all coefficients), and it has two deduction rules: For any two polynomials g, h, the rule $g, h \vdash g + h$, and for any polynomial g and variable x_i, the rule $g, x_i \vdash x_i g$. Strong size lower bounds (obtained from degree lower bounds) are known for this system. For example, encoding the pigeonhole principle as a contradicting set of constant degree polynomials, we have

Theorem 21 *For every n and every $m > n$, $\mathsf{s}_{\mathrm{PC}}(\mathtt{PHP}_n^m) \geq 2^{n/2}$, over every field.*

5.3.3 Geometric Proof Systems

Yet another natural way to represent contradictions is a by a set of regions in space that have empty intersection. Again, we care mainly about discrete (say, Boolean) domains, and a wide source of interesting contradictions are Integer Programs from Combinatorial Optimization. Here, the constraints are (affine) linear inequalities with integer coefficients (so the regions are subsets of the Boolean cube carved out by halfspaces). The most basic system is called Cutting Planes (CP). Its lines are linear inequalities with integer coefficients. Its deduction rules are (the obvious) addition of inequalities, and the (less obvious) dividing the coefficients by a constant (and rounding, taking advantage of the integrality of the solution space).

While $\mathtt{PHP}_n^m$ is easy in this system, exponential lower bounds are known for other tautologies. We mention that they are obtained from the *monotone circuit* lower bounds of Section 5.1.2.

6 Randomized Computation

As hinted in Section 3, until now we restricted computations to (repeatedly) executing a deterministic rule. A more liberal approach pursued in this section considers computing devices that use a probabilistic (or randomized) rule. We still focus on polynomial-time computations, but these are *probabilistic* (i.e., can "toss coins"). Specifically, we allow probabilistic rules that choose uniformly among two outcomes. We comment that probabilistic computations are believed to take place in real-life algorithms that are employed in a variety of applications (e.g., *random* sampling, *Monte-Carlo* simulations, etc.).[10]

Rigorous models of probabilistic machines are defined by natural extensions of the basic model, yielding probabilistic Turing machines. For a probabilistic machine M and string $x \in \{0,1\}^*$, we denote by $M(x)$ the distribution of the output of M when invoked on input x, where the probability is taken over the machine's random moves. Considering decision problems, we want this distribution to yield the correct answer with high probability *for every input*. This leads to the definition of $\mathcal{BPP}$ (for Bounded error, Probabilistic Polynomial time):

Definition 22 ($\mathcal{BPP}$) *A Boolean function f is in $\mathcal{BPP}$ if there exists a probabilistic polynomial-time machine M such that for every $x \in \{0,1\}^*$, $\Pr[M(x) \neq f(x)] \leq 1/3$.*

The error bound $1/3$ is arbitrary; for any $k = \mathrm{poly}(|x|)$, the error can be reduced to 2^{-k} by invoking the program $O(k)$ times and taking a majority vote of the answers. We stress that the random moves in the different invocations are independent.

Again, it is trivial that $\mathcal{BPP} \subseteq \mathcal{EXP}$, via enumerating all possible outcomes of coin tosses and taking a majority vote. The relation of $\mathcal{BPP}$ to $\mathcal{NP}$ is not known, but it is known that if $\mathcal{NP} = \mathcal{P}$ then also $\mathcal{BPP} = \mathcal{P}$. Finally, non-uniformity can replace randomness: every function in $\mathcal{BPP}$ has polynomial-size circuits. But the fundamental question is whether or not randomization adds computing power over determinism (for decision problems).

Open Problem 23 *Does $\mathcal{P} = \mathcal{BPP}$?*

While quite a few problems[11] are known to be in $\mathcal{BPP}$ but not known to be in $\mathcal{P}$, there is overwhelming evidence that the answer to the question above is positive (namely, randomization does not add extra power in the context of decision problems): we elaborate a bit in Section 7.1.

6.1 Counting at Random

One important question regarding NP search problems is that of determining *how many* solutions a specific instance has. This captures a host of interesting problems from various disciplines, e.g. counting the number of solutions to a system of multivariate polynomials, counting the number of perfect matchings of a graph, computing the volume of a polytope (given by linear inequalities) in high dimension, computing various parameters of physical systems, etc.

In most of these problems, even approximate counting would suffice. Clearly, approximate counting allows one to determine whether a solution exists at all. For example, counting the number of satisfying assignments for a given propositional formula (even approximately) allows one to determine whether the formula is satisfiable. Interestingly, the converse is also true.

[10] The sense in which these applications actually utilize random moves is a different question. The point is that one analyzes these computations as though they are taking random moves.

[11] A central example is Identity Testing: given an arithmetic circuit over $\mathcal{Q}$, decide if it computes the identically zero polynomial.

Theorem 24 *There exists a probabilistic polynomial-time oracle machine*[12] *that, on input a formula ψ and oracle access to* `SAT`*, outputs an integer that with probability at least $\frac{2}{3}$ is within a factor of 2 of the number of satisfying assignments for ψ.*

We comment that an analogous statement holds for any NP-complete problem.

The approximation factor can be reduced to $1 + |\psi|^{-c}$, for any fixed constant c. However, it is believed that an *exact* count cannot be obtained via a probabilistic polynomial-time oracle with oracle access to `SAT`. We mention that computing the aforementioned quantity (or computing the number of solutions to any NP-search problem) is polynomial-time reducible to computing the permanent of positive integer matrices.[13]

For some of the problems mentioned above, approximate counting can be done *without* the `SAT` oracle: There are polynomial-time probabilistic algorithms for approximating the permanent of positive matrices, approximating the volume of polytopes, and more. These follow a connection of approximate counting to the related problem of *uniform generation* of solutions, and the construction and analysis of adequate Markov chains for solving the related sampling problems (see [9, Chap. 12]).

6.2 Probabilistic Proof Systems

The glory attributed to the creativity involved in finding proofs, makes us forget that it is the less glorified process of verification that *defines* proof systems.

The notion of a verification procedure presupposes the notion of computation[14] and furthermore the notion of efficient computation (because verification, unlike coming up with proofs, is supposed to be easy). It will be convenient here to view a proof system for a set S (e.g., of satisfiable formulae) as a game between an all-powerful prover and an efficient verifier: Both receive an input x, and the prover attempts to convince the verifier that $x \in S$. Completeness dictates that the prover succeeds for every $x \in S$, and soundness dictates that *any* prover fails for every $x \notin S$.

When taking the most natural choice of efficiency requirement, namely restricting the verifier to be a deterministic polynomial-time machine, we get back the definition of the class $\mathcal{NP}$ (slightly rephrased): *a set S is in $\mathcal{NP}$ if and only if membership in S can be verified by a deterministic polynomial-time machine when given an alleged proof of polynomial length* (i.e., polynomial in $|x|$).

Now we relax the efficiency requirement, and let the verifier be a *probabilistic* polynomial-time machine, allowing it to "rule by statistical evidence" and hence to err (with low probability, which is explicitly bounded and can be reduced via repetitions). This relaxation is not suggested as a substitute to the notion Mathematical truth; however, as we shall below, it turns out to yield enormous advance in computer science.

6.2.1 Interactive Proof Systems

When the verifier is deterministic, we can always assume that the prover simply sends it a single message (the purported "proof"), and based on this message the verifier decides whether to accept or reject the common input x as a member of the target set S.

When the verifier is probabilistic, *interaction* may add power. We thus consider a (randomized) interaction between the parties, which may be viewed as an "interrogation" by a persistent student,

[12] See Footnote 3. Here, upon issuing any query ψ' the machine is told whether or not ψ' is satisfiable.

[13] We stress that this reduction does *not* preserve the quality of an approximation.

[14] This may explain the historical fact that notions of computation were first *rigorously formulated* in the context of logic.

asking the teacher "tough" questions in order to be convinced of correctness.[15] Since the verifier ought to be efficient (i.e., run in time polynomial in $|x|$), this interaction is bounded to have at most these many rounds. The class $\mathcal{IP}$ (for Interactive Proofs) contains all sets S for which there is a verifier that accepts every $x \in S$ with probability 1 (after interacting with an adequate prover), but rejects any $x \notin S$ with probability at least $1/2$ (no matter what strategy is employed by the prover).

A major result asserts that interactive proofs exists for every set in $\mathcal{PSPACE}$ (i.e., having a decision procedure that uses a polynomial amount of memory, but possibly working in exponential-time).

Theorem 25 $\mathcal{IP} = \mathcal{PSPACE}$.

While it is not known if $\mathcal{NP} \neq \mathcal{PSPACE}$, it is widely believed to be the case, and so it seems that interactive proofs are more powerful than standard non-interactive and deterministic proofs (i.e., NP-proofs). In particular, since co$\mathcal{NP} \subseteq \mathcal{PSPACE}$, Theorem 25 implies that there are such interactive proofs for every set in co$\mathcal{NP}$ (e.g., the set of propositional tautologies), whereas some coNP-sets are believed not to have NP-proofs.

6.2.2 Zero-Knowledge Proof Systems

Here the thrust is not to prove more theorems, but rather to have proofs with additional properties. Randomized and interactive verification procedures as in Section 6.2.1 allow the (meaningful) introduction of *zero-knowledge proofs*, which are proofs that yield nothing beyond their own validity. Such proofs seem counter-intuitive and undesirable for educational purposes, but they are very useful in cryptography.

For example, a *zero-knowledge proof* that a certain propositional formula is satisfiable does not reveal a satisfying assignment to the formula nor any partial information regarding such an assignment (e.g., whether the first variable can assume the value `true`). In general, whatever the verifier can efficiently compute after interacting with a zero-knowledge prover, can be efficiently reconstructed from the assertion itself (without interacting with anyone).

Clearly, any set in $\mathcal{BPP}$ has a zero-knowledge proof, in which the prover says nothing (and the verifier decides by itself). What is surprising is that zero-knowledge proofs seem to exist also for sets that are not in $\mathcal{BPP}$. In particular:

Theorem 26 *Assuming the existence of one-way functions* (see Section 7)*, every set in $\mathcal{NP}$ has a zero-knowledge proof system.*

6.2.3 Probabilistically Checkable Proof systems

Let us return to the non-interactive mode, in which the verifier receives a (alleged) written proof. But now we restrict its access to the proof so as to read only a small part of it (which may be randomly selected). An excellent analogy is to imagine a referee trying to decide the correctness of a long proof by sampling a few lines of the proof. It seems hopeless to detect a single "bug" unless the entire "proof" is read; but this intuition is valid only for the "natural" way of writing down proofs and fails when "robust" formats of proofs are used and one is willing to settle for statistical evidence.

[15] Interestingly, it turns out that asking "tough" questions is not better than asking random questions!

Such "robust" proof systems are called PCPs (for Probabilistically Checkable Proofs). Loosely speaking, a PCP system for a set S consists of a probabilistic polynomial-time verifier having access to an oracle that represents a proof in redundant form, where (as in case of NP-proofs) the length of the proof is polynomial in the length of the input. The verifier accesses only a constant number of the oracle bits, and accepts every $x \in S$ with probability 1 (when given access to an adequate oracle), but rejects any $x \notin S$ with probability at least 1/2 (no matter to which oracle it is given access).

Theorem 27 (The PCP Theorem) *Each set in $\mathcal{NP}$ has a PCP system. Furthermore, there exists a polynomial-time procedure for converting any NP-proof to the corresponding PCP-oracle.*

Indeed, the proof of the PCP Theorem suggests a new way of writing "robust" proofs, in which any bug must "spread" all over[16]. One important application of the PCP Theorem (and its variants) is the connection to the complexity of combinatorial approximation. For example, it is NP-complete to decide if, for a given linear system of equations over GF(2), the fraction of mutually satisfiable equations is greater than 99% or smaller than 51%.

6.3 Weak Random Sources

We now return to the question of how to obtain the assumed randomness for all the probabilistic computations discussed in this section. Although randomness seems to be present in the world (e.g., the perceived randomness in the weather, Geiger counters, Zener diodes, real coin flips, etc.), it does not seem to be in the perfect form of unbiased and independent coin tosses (as postulated above). Thus, to actually use randomized procedures, we need to convert weak sources of randomness into almost perfect ones. Very general mathematical models capturing such weak sources have been proposed. Algorithms converting the randomness in them into a distribution that in close to uniform (namely unbiased, independent stream of bits) are called *randomness extractors*, and near optimal ones have been constructed. This large body of work is surveyed, e.g., in [16]. We mention that this question turned out to be related to certain types of pseudorandom generators (cf. Section 7.1) as well as to combinatorics and coding theory.

7 The Bright Side of Hardness

The Big Conjecture according to which $\mathcal{P} \neq \mathcal{NP}$ means that there are computational problems of great interest that are inherently intractable. This is bad news, but there is a bright side to the matter: computational hardness (alas in a stronger form than known to follow from $\mathcal{P} \neq \mathcal{NP}$) has many fascinating conceptual consequences as well as important practical applications. Specifically, in accordance with our intuition, we shall assume that not all efficient processes can be efficiently reversed (or inverted). Furthermore, we shall assume that hardness to invert is a typical (rather than pathological) phenomenon for some efficiently computable functions. That is, we assume that *one-way functions* exist.

Definition 28 (One-Way Functions) *A function $f : \{0,1\}^* \to \{0,1\}^*$ is called* one-way *if the following two conditions hold*

1. easy to compute: *the function f is computable in polynomial time.*

[16] The analogy to error correcting codes is indeed in place, and the cross fertilization between these two areas has been very significant.

2. hard to invert: *for every probabilistic polynomial-time machine, M, every positive polynomial $p(\cdot)$, and all sufficiently large n*

$$\Pr_x\left[M(1^n, f(x)) \in f^{-1}(f(x))\right] < \frac{1}{p(n)}$$

where x is uniformly distributed in $\{0,1\}^n$.

For example, the widely believed conjecture according to which integer factorization is intractable (for a noticeable fraction of the instances) implies the existence of one-way functions. On the other hand, if $\mathcal{P} = \mathcal{NP}$ then no one-way functions exist. *One important open problem is whether $\mathcal{P} \neq \mathcal{NP}$ implies the existence of one-way functions.*

Below, we discuss the connection between computational difficulty (in the form of one-way functions) on the one hand, and two important computational theories on the other hand: the theory of *Pseudorandomness* and the theory of *Cryptography*.

One fundamental concept, which is pivotal to both these theories, is the concept of *computational indistinguishability*. Loosely speaking, two objects are said to be computationally indistinguishable if no efficient procedure can tell them apart. Here objects will be probability distributions (on finite binary sequences). We actually consider probability ensembles, where an ensemble is a family of distributions, each on strings of different length (e.g., the *uniform* ensemble is the family $\{U_n\}_{n\in\mathsf{N}}$, where U_n is the uniform distribution on all n-bit strings).

Definition 29 (Computational Indistinguishability) *The probability ensembles $\{P_n\}_{n\in\mathsf{N}}$ and $\{Q_n\}_{n\in\mathsf{N}}$ are called* computationally indistinguishable *if for every probabilistic polynomial-time machine, M, every positive polynomial $p(\cdot)$, and all sufficiently large n*

$$|\Pr[M(1^n, P_n)=1] - \Pr[M(1^n, Q_n)=1]| < \frac{1}{p(n)}\ .$$

Computational indistinguishability is a (strict) coarsening of statistical indistinguishability. We focus on the non-trivial cases of pairs of ensembles that are computationally indistinguishable although they are statistically very different. It is easy to show that such pairs do exist, but we further focus on pairs of such ensembles that are *efficiently samplable*[17]. Interestingly, such pairs exists if and only if one-way functions exist.

7.1 Pseudorandomness

We call an ensemble pseudorandom if it is computationally indistinguishable from the random (i.e., uniform) ensemble. A pseudorandom generator is an *efficient* (deterministic) procedure that *stretches* short random strings into longer pseudorandom strings.

Definition 30 (Pseudorandom Generators) *A deterministic polynomial-time machine G is called a* pseudorandom generator *if there exists a monotonically increasing function, $\ell:\mathsf{N}\rightarrow\mathsf{N}$, such that the probability ensembles $\{U_{\ell(n)}\}_{n\in\mathsf{N}}$ and $\{G(U_n)\}_{n\in\mathsf{N}}$ are computationally indistinguishable.*[18] *The function ℓ is called the* stretch measure *of the generator, and the n-bit input of the generator is called its* seed.

[17] The ensemble $\{P_n\}_{n\in\mathsf{N}}$ is *efficiently samplable* if there exists a probabilistic polynomial-time machine M such that $M(1^n)$ and P_n are identically distributed, for every n.

[18] Recall that U_m denotes the uniform distribution over $\{0,1\}^m$. Thus, $G(U_n)$ is defined as the output of G on a uniformly selected n-bit input string.

That is, pseudorandom generators yield a particularly interesting case of computational indistinguishability: the distribution $G(U_n)$, which is efficiently samplable using only n truly random coins (and so has entropy n), is computationally indistinguishable from the uniform distribution over $\ell(n)$-bit long strings (having entropy $\ell(n) > n$). The major question which we turn to deal with is of course: do pseudorandom generators exist?

7.1.1 Hardness versus Randomness

By its very definition, the notion of a pseudorandom generator is connected to computational difficulty (i.e., the computational difficulty of determining that the generator's output is *not* truly random). It turns out that the connection between the two notions is even stronger.

Theorem 31 *Pseudorandom generators exist if and only if one-way functions exist. Furthermore, if pseudorandom generators exist then they exist for any stretch measure that is a polynomial.*

Theorem 31 converts computational difficulty (hardness) into pseudorandomness, and vice versa. Furthermore, its proof links computational indistinguishability to computational unpredictability, hinting that computational difficulty (of predicting an information theoretically determined event) is linked to randomness (or to the appearance of being random).

Pseudorandom generators allow for a drastic reduction in the amount of "true randomness" used in any *efficient* randomized procedure. Rather than using independent coin tosses, such procedures can use the output of a pseudorandom generator, which in turn can be generated deterministically based on many fewer coin tosses (used to select the generator's seed). The effect of this replacement on the behavior of such procedures will be negligible. In algorithmic applications, where it is possible to invoke the same procedure many times and rule by a majority vote, one can derive deterministic procedures by trying all possible seeds. In particular, using a seemingly stronger notion of pseudorandom generators (which work in time exponential in their seeds and produce sequences that look random to tests of a fixed polynomial-time complexity), allows to convert any probabilistic polynomial-time algorithm into a deterministic one (implying that $\mathcal{BPP} = \mathcal{P}$). Such pseudorandom generators exist under plausible conjectures regarding computational difficulty which seem far weaker than the existence of one-way functions. Thus for example:

Theorem 32 *If, for some constant $\epsilon > 0$, $\mathcal{S}(\texttt{SAT}) > 2^{\epsilon n}$ then $\mathcal{BPP} = \mathcal{P}$. Moreover,* SAT *can be replaced by any problem computable in $2^{O(n)}$-time.*

7.1.2 Pseudorandom Functions

Pseudorandom generators allow for the efficient generation of long pseudorandom sequences from short random seeds. Pseudorandom functions are even more powerful: they allow for *efficient* direct access to a huge pseudorandom sequence (which is infeasible to scan bit-by-bit). That is, pseudorandom functions are efficiently computable (ensembles of) functions that are indistinguishable from truly random functions by any *efficient* procedure that can obtain the function values at arguments of its choice. We refrain from presenting a precise definition, but do mention a central result: *pseudorandom functions can be constructed given any pseudorandom generator.* We also mention that pseudorandom functions have many applications (most notably in cryptography).

7.2 Cryptography

Cryptography has existed for millennia. However, in the past it was focused on one basic problem – that of providing secret communications. By contrast, the modern computational theory of cryptography is interested in *all* tasks involving several communicating agents in which the following (often conflicting) desires are crucial: *privacy*, namely the protection of secrecy, and *resilience*, namely the ability to withstand malicious behavior of participants. Perhaps the best example to illustrate these difficulties is playing a game of Poker over the telephone (i.e., the "new age" players cannot rely on physical implements such as cards dealt from a deck that is visible by all players). In general, cryptography is concerned with the construction of schemes that maintain *any* desired functionality under malicious attempts aimed at making these schemes deviate from their prescribed functionality.

As with pseudorandomness, there are two key assumptions underlying the new theory. First, that all parties (including the adversary) are computationally limited: they are modeled as probabilistic polynomial-time machines and hence computationally indistinguishable distributions are equivalent as far as these parties are concerned. Second, that a certain type of computationally hard problem exists, namely, one-way functions and in some cases stronger versions called *trapdoor permutations*, which in turn are implied by the hardness of integer factorization. In fact, all the results mentioned below hold if trapdoor permutations exist, and cannot hold if one-way functions do not exist.

Starting with the traditional problem of providing secret communication over insecure channels, we note that pseudorandom functions (which can be constructed based on any one-way function) provide a satisfactory solution for this problem: The communicating parties, sharing a pseudorandom function, may exchange information in secrecy by masking it with the values of the function evaluated at adequately selected arguments (which may be agreed-upon a priori or transmitted in the clear). That is, the parties use a pseudorandom function as a secret key in (predetermined) encryption and decryption procedures. Still, the communicating parties have to agree on this key beforehand (or transmit this key through an auxiliary secret channel).

The need for a priori agreement on a secret key is removed when using "public-key" encryption schemes, in which the key used for encryption can be made public while only the (different) key used for decryption is kept secret. In particular, in such schemes, it is infeasible to recover the decryption-key from the encyption-key, although such random pairs of keys can be generated efficiently. Secure public-key encryption schemes (i.e., providing for secret communication without any prior secret agreement) can be constructed based on trapdoor permutations.

A general framework for casting cryptographic problems consists of specifying a random process which maps m inputs to m outputs. The inputs to the process are to be thought of as local inputs of m parties, and the m outputs are their corresponding local outputs. The random process describes the desired functionality. That is, if the m parties were to trust each other (or trust some outside party), then they could each send their local input to the trusted party, who would compute the outcome of the process and send each party the corresponding output. Loosely speaking, a secure implementation of such a functionality is an m-party protocol in which the *impact* of malicious parties is effectively restricted to application of the prescribed functionality to inputs chosen by the corresponding parties. One major result in this area is the following.

Theorem 33 *Assuming the existence of trapdoor permutations, any efficiently computed functionality can be securely implemented.*

8 The Tip of an Iceberg

Even within the topics discussed above, many important notions and results have not been discussed for space reasons. Furthermore, other important topics and even wide areas have not been mentioned at all. Here we briefly discuss some of these topics and areas.

8.1 Relaxing the Requirements

The P vs. NP Question, as well as most of the discussion so far, focuses on a simplified view of the goals of (efficient) computations. Specifically, we have insisted on efficient procedures that *always* give the *exact* answer. In practice, one may be content with efficient procedures that "typically" give an "approximate" answer. Indeed, both terms in quotation marks require clarification.

8.1.1 Average-Case Complexity

One may consider procedures that answer correctly on a large fraction of the instances. But this assumes that all instances are equally interesting in practice, which is typically not the case. On the other hand, demanding success under all input distributions gives back worst-case complexity. A very appealing theory of average-case complexity (cf. [6]) demands success only for the family of all input distributions that can be efficiently sampled.

8.1.2 Approximation

What do we mean by an approximation to a computational problem? There are many possible answers, and their significance depends on the specifics of the application. For optimization problems, the answer is obvious: we'd like to get "close" to the optimum (see [9]). For search problems, we may be satisfied with a solution that is close in some metric to being valid. For decision problems (i.e., determining set membership), we may ask how close the input is (under some relevant distance measure) to an instance in the set (cf. [15]).

8.2 Other Complexity Measures

Until now, we have focused on the *running time* of procedures, which is arguably the most important complexity measure. However, other complexity measures such as the amount of *work-space* consumed during the computation are also important (cf. [17]). Another important issue is the extent to which a computation can be performed in parallel; that is, speeding-up the computation by splitting the work among several computing devices, which are viewed as components of the same (parallel) machine and they are provided with direct access to the same memory module. In addition to the *parallel time*, a fundamentally important complexity measure in such a case is the number of (parallel) computing devices used (cf. [10]).

8.3 Other Notions of Computation

Following are a few of the computational models we did not discuss. Models of *distributed computing* refer to distant computing devices, each given a local input (which may be viewed as a part of a global input). In typical studies one wishes to minimize the amount of communication between these devices (and certainly avoid the communication of the entire input). In addition to measures of communication complexity, a central issue is asynchrony (cf. [1]). We note that the *communication complexity* of two-argument (and many-argument) functions is studied as a measure of their “complexity” (cf. [13]), but in these studies communication proportional to the length of the input is not ruled out (but rather appears frequently). While being “information theoretic” in nature, this model has many connections to complexity theory. Altogether different types of computational problems are investigated in the context of *computational learning theory* (cf. [11]) and the study of *on-line* (cf. [2]). Finally, *Quantum Computation* investigates the possibility of using quantum mechanics to speed up computation (cf. [12]).

9 Concluding Remarks

We hope that this ultra-brief survey conveys the fascinating flavor of the concepts, results and open problems that dominate the field of computational complexity. One important feature of the field we did not do justice to, is the remarkable web of (often surprising) connections between different subareas, and its impact on progress. For further details on the material discussed in Sections 2–4, the reader is referred to standard textbooks such as [5, 17]. For further details on the material discussed in Sections 5.1, 5.2 and 5.3, the reader is referred to [4], [18] and [3], respectively. For further details on the material discussed in Sections 6 and 7, the reader is referred to [7] (and also to [8] for further details on Section 7.2).

References

[1] H. Attiya and J. Welch: Distributed Computing: Fundamentals, Simulations and Advanced Topics, McGraw-Hill, 1998.

[2] A. Borodin and R. El-Yaniv: On-line Computation and Competitive Analysis, Cambridge University Press, 1998.

[3] P. Beame and T. Pitassi: *Propositional Proof Complexity: Past, Present, and Future*, in Bulletin of the EATCS, Vol. 65, June 1998.

[4] R. Boppana and M. Sipser: *The complexity of finite functions*, in [14].

[5] M.R. Garey and D.S. Johnson: Computers and Intractability: A Guide to the Theory of NP-Completeness, W.H. Freeman and Company, 1979.

[6] O. Goldreich: *Notes on Levin's Theory of Average-Case Complexity*, In ECCC, TR97-058, 1997.

[7] O. Goldreich: Modern Cryptography, Probabilistic Proofs and Pseudorandomness, Algorithms and Combinatorics series (Vol. 17), Springer, 1999.

[8] O. Goldreich: Foundation of Cryptography (in two volumes: Basic Tools and Basic Applications), Cambridge University Press, 2001 and 2004.

[9] D. Hochbaum (ed.): Approximation Algorithms for NP-Hard Problems, PWS, 1996.

[10] R.M. Karp and V. Ramachandran: *Parallel Algorithms for Shared-Memory Machines*, in [14].

[11] M.J. Kearns and U.V. Vazirani: An introduction to Computational Learning Theory, MIT Press, 1994.

[12] A. Kitaev, A. Shen, M Vyalyi: Classical and Quantum Computation, AMS, 2002.

[13] E. Kushilevitz and N. Nisan: Communication Complexity, Cambridge University Press, 1996.

[14] J. van Leeuwen (ed.): Handbook of Theoretical Computer Science, Vol A: Algorithms and Complexity, MIT Press/Elsevier, 1990.

[15] D. Ron: *Property Testing* (*A Tutorial*), in Handbook on Randomized Computing (Volume II), Kluwer Academic Publishers, 2001.

[16] R. Shaltiel: *Recent Developments in Explicit Constructions of Extractors*, in Bulletin of the EATCS, Vol. 77, 2002.

[17] M. Sipser: Introduction to the Theory of Computation, PWS, 1997.

[18] V. Strassen: *Algebraic Complexity Theory*, in [14].

Appendix: Glossary of Complexity Classes

Complexity classes are sets of computational problems, where each class contains problems that can be solved with specific computational resources. Examples of such classes (e.g., $\mathcal{P}$ and $\mathcal{NP}$) are presented in the essay "Computational Complexity" (Sec. IV) and the reader is referred there for further discussion of the notions of computation and complexity.

To define a complexity class one specifies a model of computation, a complexity measure (like time or space), and a bound on it. The prevailing model of computation is that of Turing machines, which in turn capture the notion of (uniform) algorithms. Another important model is the one of non-uniform circuits. The term uniformity refers to whether the algorithm is the same one for every input length or whether a different "algorithm" (or rather a "circuit") is considered for each input length. Recall (from Sec. IV) that complexity is always measured as a function of the input length.

We focus on natural complexity classes, obtained by considering natural complexity measures and bounds, which contain natural computational problems. Furthermore, almost all of these classes can be "characterized" by natural problems, which capture every problem in the class. Such problems are called complete for the class, which means that they are in the class and every problem in the class can be "easily" reduced to them, where "easily" means that the reduction takes less resources than what each of the problems seems to require individually. We stress the fact that complete problem not only exist, but rather are natural and make no reference to computational models or resources. Efficient algorithm for a complete problem implies an algorithm of similar efficiency for *all* problems in the class.

A.1 Algorithm-based classes

The two main complexity measures considered in the context of (uniform) algorithms are the number of steps taken by the algorithm (i.e., its time complexity) and the amount of "memory" or "work-space" consumed by the computation (i.e., its space complexity). In our Sec. IV essay, we define the time-complexity classes $\mathcal{P}$ and $\mathcal{NP}$ (cf. Sec. 3.1), co$\mathcal{NP}$ (cf. Sec. 3.4), and $\mathcal{BPP}$ (cf. Sec. 5). In addition, we mention a couple of other classes associated with probabilistic polynomial-time:

- The set S is in $\mathcal{RP}$ if there exists a probabilistic polynomial-time machine M such that $x \in S$ implies $\Pr[M(x)=1] \geq \frac{1}{2}$, while $x \notin S$ implies $\Pr[M(x)=1] = 0$. Also, co$\mathcal{RP} = \{\{0,1\}^* \setminus S : S \in \mathcal{RP}\}$. The latter class contains the problem of deciding whether a given arithmetic circuit over $\mathcal{Q}$ computes the identically zero polynomial.
- The decision problem $S : \{0,1\}^* \to \{0,1\}$ is in $\mathcal{ZPP}$ if there exists a probabilistic polynomial-time machine M such that for every x it holds that $M(x) \in \{S(x), \perp\}$ and $\Pr[M(x)=S(x)] \geq \frac{1}{2}$, where $\perp$ s a special failure symbol. Equivalently, $\mathcal{ZPP}$ is the class of all sets which have a probabilistic algorithm which always returns the correct andswer, and runs in *expected* polynomial time.

Clearly, $\mathcal{ZPP} = \mathcal{RP} \cap \text{co}\mathcal{RP} \subseteq \mathcal{RP} \subseteq \mathcal{NP} \cap \mathcal{BPP}$.

When defining space-complexity classes, one counts *only* the space consumed by the actual computation, and *not* the space occupied by the input and output. This is formalized by postulating that the input is read from a read-only device (resp., the output is written on a write-only device). Four important classes of decision problems are:

- The class $\mathcal{L}$ consists of problems solvable in logarithmic space. That is, a set S is in $\mathcal{L}$ if there exists a standard (i.e., deterministic) algorithm of logarithmic space-complexity for deciding membership in S. This class contains some simple computational problems (e.g., matrix multiplication), and arguably captures the most space-efficient computations.

- The class $\mathcal{RL}$ consists of problems solvable by a randomized algorithm of logarithmic space-complexity. This class contains the problem of deciding whether a given undirected graph is connected. This problem is not known to be in $\mathcal{L}$.

- The class $\mathcal{NL}$ is the non-deterministic analogue of $\mathcal{L}$, and is traditionally defined in terms of non-deterministic machines of logarithmic space-complexity. Alternatively, analogously to the definition of $\mathcal{NP}$, a set S is in $\mathcal{NL}$ if there exists a polynomially bounded binary relation $R_S \in \mathcal{L}$ such that $x \in S$ if and only if there exists y such that $(x, y) \in R_S$. The class $\mathcal{NL}$ contains the problem of deciding whether there exists a directed path between two given vertexes in a given directed graph. In fact, the latter problem is complete for the class (under logarithmic-space reductions). Interestingly, co$\mathcal{NL} \stackrel{\text{def}}{=} \{\{0,1\}^* \setminus S : S \in \mathcal{NL}\}$ equals $\mathcal{NL}$.

- The class $\mathcal{PSPACE}$ consists of (decision) problems solvable in polynomial space. This class contains very difficult problems, including the computation of winning strategies for any efficient 2-party games (as discussed below).

Clearly, $\mathcal{L} \subseteq \mathcal{RL} \subseteq \mathcal{NL} \subseteq \mathcal{P}$ and $\mathcal{NP} \subseteq \mathcal{PSPACE}$.

Turning back to time-complexity, we mention the classes $\mathcal{E}$ and $\mathcal{EXP}$ corresponding to problems that can be solved (by a deterministic algorithm) in time $2^{O(n)}$ and $2^{\mathrm{poly}(n)}$, respectively, for n-bit long inputs. Clearly, $\mathcal{PSPACE} \subseteq \mathcal{EXP}$.

Two classes related to the class $\mathcal{NP}$ are the "counting class" $\#\mathcal{P}$ and the Polynomial-time hierarchy. Functions in $\#\mathcal{P}$ count the number of solutions to an NP-type search problem (e.g.. compute the number of satisfying assignments of a given formula). Formally, a function f is in $\#\mathcal{P}$ if there exists an NP-type relation R such that $f(x) = |\{y : (x, y) \in R\}|$. Clearly, $\#\mathcal{P}$ problems are solvable in polynomial space. Surprisingly, the permanent of positive integer matrices is $\#\mathcal{P}$-complete (i.e., it is in $\#\mathcal{P}$ and any function in $\#\mathcal{P}$ is polynomial-time reducible to it).

The Polynomial-time hierarchy, $\mathcal{PH}$, consists of sets S such that there exists a constant k and a $(k+1)$-ary polynomially bounded relation $R_S \in \mathcal{P}$ such that $x \in S$ if and only if $\exists y_1 \forall y_2 \exists y_3 \forall y_4 ...$ such that $(x, y_1, y_2, y_3, y_4, ..., y_k) \in R_S$. Indeed, $\mathcal{NP}$ corresponds to the special case where $k = 1$. Interestingly, $\mathcal{PH}$ is polynomial-time reducible to $\#\mathcal{P}$.

Oded's Note: *I prefer to omit the next paragraph.*

Sets in the Polynomial-time hierarchy and in the class $\mathcal{PSPACE}$ capture the complexity of finding winning strategies in certain *efficient 2-party game*. In such games, the two players compute their next move (from any given position) in polynomial time (in terms of the initial position) and a winning position can be recognized in polynomial-time. For example, a set S as above can be viewed via a k-move game in which, starting from a given position x, the first party takes the first move y_1, the second responds with y_2, etc, and the winner is determined by whether or not the transcript $(x, y_1, \cdots y_k)$ of the game is in R_S. That is, $x \in S$ if, starting at the initial position x, the first party has a winning strategy in the k-move game determined by R_S. Thus, sets in $\mathcal{PH}$ (resp., $\mathcal{PSPACE}$) corresponds to games with a constant number of (resp., polynomailly many) moves.

A.2 Circuit-based classes

See Sec. IV for discussion of circuits as computing devices. The two main complexity measures considered in the context of (non-uniform) circuits are the number of gates (or wires) in the circuit (i.e., the circuit's size) and the length of the longest directed path from an input to an output (i.e., the circuit's depth).

The main motivation for the introduction of complexity classes based on circuits is the development of lower-bounds. For example, the class of problems solvable by polynomial-size circuits, denoted $\mathcal{P}$/poly, is a super-set of $\mathcal{P}$ (because it clearly contains $\mathcal{P}$ as well as any subset of $\{1\}^*$, whereas there exists such sets that represents decision problems that are not solvable (i.e., by any uniform algorithm)). Thus, showing that $\mathcal{NP}$ is not contained in $\mathcal{P}$/poly would imply $\mathcal{P} \neq \mathcal{NP}$. For further discussion see Sec. IV.

The class $\mathcal{AC}^0$, discussed in our Sec. IV article (cf. Sec. 5.1.3), consists of sets recognized by constant-depth polynomial-size circuits of *unbounded fan-in*. The analogue class that allows also (unbounded fan-in) majority-gates (or, equivalently, threshold-gates) is denoted $\mathcal{TC}^0$. For any non-negative integer k, the class of sets recognized by polynomial-size circuits of *bounded fan-in* (resp., unbounded fan-in) having depth $O(\log^k n)$, where n is the input length, is denoted $\mathcal{NC}^k$ (resp., $\mathcal{AC}^k$). Clearly, $\mathcal{NC}^k \subseteq \mathcal{AC}^k \subseteq \mathcal{NC}^{k+1}$ and $\mathcal{NC} \stackrel{\text{def}}{=} \cup_{k \in \mathbb{N}} \mathcal{NC}^k$.

We mention that the class $\mathcal{NC}^2 \supseteq \mathcal{NL}$ is the habitat of most natural computational problems of Linear Algebra: solving a linear system of equations as well as computing the rank, inverse and determinant of a matrix. The class $\mathcal{NC}^1$ contains all symmetric functions, regular languages as well as word problems for finite groups and monoids. The class $\mathcal{AC}^0$ contains all properties of finite objects expressible by first-order logic.