Computational Complexity

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

March 19, 2000

1 Introduction

Computational Complexity (a.k.a Complexity Theory) is a central field of Computer Science[1] with a remarkable list of celebrated achievements as well as a very vibrant research activity. The field is concerned with the study of the *intrinsic complexity* of computational tasks, and this study tend 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.

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 (or effect) 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 (cf. [3]) and in proof complexity (cf. [2]) 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 on either of the problems (or notions) being related. The current exposition focuses on the latter effort (or direction), which may be viewed as a "high-level" study of computation. We present a few of the interesting notions, results and open problems in this direction.

We list several reasons for our choice to focus on the "high-level" direction. The first is the great *conceptual significance* of the know results; that is, as we

[1] The theoretical aspects of Computer Science can be viewed as consisting of two disciplines: the Theory of Computation (TOC) and the Theory of Programming (TOP). TOC is concerned with the process of computation and its effect, whereas TOP is concerned with the coding of programs that induce computations. Complexity Theory is but one field of TOC; other fields concern the design and analysis of algorithms for *specific* computational problems that arise from a huge variety of areas of mathematics and science.

shall see, many known results (and open problems) in this direction have an appealing conceptual message, which can be appreciated also by non-experts. Furthermore, these conceptual aspects may be explained without getting into too much technical details. Consequently, such material is more suitable for an exposition in a book of the current nature. Finally, we admit that the "high-level" direction is within the author's expertise, while this cannot be said about the "low-level" direction.

2 Preliminaries

This exposition considers only *finite* objects, encoded by binary finite sequences called strings. For a natural number n, we denote by $\{0,1\}^n$ the set of all n-bit long strings. The set of all strings is denoted $\{0,1\}^*$; that is, $\{0,1\}^* = \cup_{n\in\mathbb{N}}\{0,1\}^n$. 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 $\{0,1\}^*$; the reader should merely consider an adequate encoding (i.e., $(x_1 \cdots x_m, y_1 \cdots y_n) \in \{0,1\}^* \times \{0,1\}^*$ may be encoded by $x_1x_2 \cdots x_mx_m01y_1y_1 \cdots y_ny_n \in \{0,1\}^*$). Typically, natural numbers will be encoded by their binary expansion so that $b_{n-1} \cdots b_1b_0 \in \{0,1\}^n$ encodes the number $\sum_{i=0}^{n-1} b_i \cdot 2^i$.

Computability: We assume that the reader is familiar with the notion of a computation (cf. [13]). Loosely speaking, a computation is a process that modifies an environment via repeated applications of a predetermine rule that depends and effects only a (small) portion of the environment, called the active zone. The distinction is between the *a-priori bounded* size of the active zone and the *a-priori unbounded* size of the entire environment. Although each application of the (computation) rule has a very limited effect, the effect of the computational process induced by the rule (i.e., many successive applications of the rule) may be very complex.

We are interested in the transformation of the environment effected by the computational process (or computation). 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)[2] encodes an output string. We 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., application of the rule) taken by the computation (as a function of all possible inputs). The latter function is called the time complexity of the computational process.

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

To rigorously define computation (and computation time) 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); however a simpler abstract model that is commonly used is the one of Turing machines. We stress that most results in the theory of computation hold regardless of the specific computational model used, as long as it is "reasonable". This refers both to the class of functions that can be computed, and to the class of functions (resp., problems) that can be "efficiently" computed (resp., solved).

Efficient Computability: We associate efficient computations with computations that terminate within time polynomial in the length of their inputs. The functional treatment of running-time (i.e., running-time as function of the input length) is important for an easier development of the theory of efficient computation, but such theories could also be developed for fixed input-length (alas at much greater effort). 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 to consider as efficient all procedures that have practically admissible time complexity, and the closure properties of polynomials offer robustness of the notion of efficient computation.

Definition 1 (polynomial-time) *We say that a Turing machine M is* polynomial-time *if there exists a polynomial p so that for every $x \in \{0,1\}^*$, when invoked on input x, machine M halts after at most $p(|x|)$ steps.*

We stress again that the specific choice of Turing machines as a model of computation is immaterial, and polynomial-time Turing machines correspond to computations that can be carried out on a real computer within time polynomial in the input length.

What is being Computed? The above discussion has implicitly referred to computations and Turing machines as means for 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. However, computations can also be viewed as means for "solving problems" or making decisions. We may say that M solves a search problem $R \subseteq \{0,1\}^* \times \{0,1\}^*$ if for every $(x,y) \in R$ it holds that $(x, f_M(x)) \in R$; that is, given an instance x that has a valid solution y (i.e., $(x,y) \in R$), machine M finds some valid solution y' for x (i.e., $(x,y') \in R$). We say that M solves a decision problem $S \subseteq \{0,1\}^*$ 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 $x \in S$ or not.

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 $f_M(x)$.

3 The P versus NP Question

Our focus is on the concept of efficient computations, and on the question of which functions (resp., problems) can be efficiently computed (resp., efficiently solved). In terms of decision problems, a conservative approach to computing devices associates efficient computations with the complexity class $\mathcal{P}$ (where P stands for Polynomial-time). Jumping ahead, we note that a more liberal approach (pursued in Section 5) allows the computing devices to "toss coins" (be randomized).

Definition 2 (The complexity class $\mathcal{P}$) *A decision problem $S \subseteq \{0,1\}^*$ is* solvable in polynomial-time *if there exists a deterministic polynomial-time Turing machine M so that $M(x) = 1$ if and only if $x \in S$. The class of search problems that are solvable in polynomial-time is denoted $\mathcal{P}$.*

Similarly, we can define the class of functions (resp., search problems) that are computable (resp., solvable) in polynomial-time. Clearly, we should consider only *polynomially-bounded* functions and relations, where a function f (resp., relation R) is polynomially-bounded if there exists a polynomial p so that $|f(x)| \leq p(|x|)$ holds for every x (resp., $|y| \leq p(|x|)$ holds for every $(x,y) \in R$). An important fact is that for every polynomially-bounded relation R, *if $S_R \stackrel{\rm def}{=} \{(x,y') : \exists y''$ s.t. $(x,y'y'') \in R\}$ is in $\mathcal{P}$ then R is solvable in polynomial-time.*

The complexity class $\mathcal{NP}$ is associated with search problems having solutions that, once given, can be efficiently tested for validity. That is, a *polynomially-bounded* search problem $R \subseteq \{0,1\}^* \times \{0,1\}^*$ is of the NP-type if one can decide membership in R in polynomial-time. Thus, the following fundamental question arises:

Open Problem 3 (P versus NP – search version) *Is every NP-type search problem solvable in polynomial-time?*

That is, if there exists an efficient way to decide whether a given instance-solution pair is valid then does it follow that there exists an efficient way to find a valid solution to a given instance? The P versus NP Question has also a fundamental formulation in terms of decision problems:

Definition 4 (The complexity class $\mathcal{NP}$) *A set S is in $\mathcal{NP}$, if there exists a polynomially-bounded relation $R_S \subseteq \{0,1\}^* \times \{0,1\}^*$ such that R_S is in $\mathcal{P}$ and $x \in S$ if and only if there exists a y such that $(x,y) \in R_S$. Such a y is called a* proof of membership of $x \in S$.

Thus, $\mathcal{NP}$ consists of the class of sets for which there exist short proofs of membership that can be efficiently verified.[3] Recall that $\mathcal{P}$ is the class of sets for which membership can be efficiently decided (without being given a proof). Thus, the P versus NP Question can be casted as follows: *does the existence of an efficient verification procedure for proofs of membership in a set imply the ability to efficiently decide membership in the set*?

Open Problem 5 (P versus NP – decision version) *Is $\mathcal{NP}$ equal to $\mathcal{P}$?*

That is, *do proofs help* or *is it the case that one can find out the truth by oneself essentially as easily as one can be convinced of the fact by a proof*? Problems 3 and 5 are in fact equivalent:

Fact 6 *Every NP-type search problem is solvable in polynomial-time if and only if $\mathcal{P} = \mathcal{NP}$.*

Proof: Suppose that equality holds for the search version and let $S \in \mathcal{NP}$. Then R_S (as in Definition 4), being an NP-type search, is solvable in polynomial-time, and it follows that $S \in \mathcal{P}$. Suppose, on the other hand, that $\mathcal{NP} = \mathcal{P}$, and let R be an NP-type search. Then the set S_R (as defined above) is in $\mathcal{NP}$ and so in $\mathcal{P}$, and it follows that R is solvable in polynomial-time. ■

The Big Conjecture: It is widely believed that $\mathcal{P} \neq \mathcal{NP}$, and settling this conjecture is certainly the most intriguing open problem in Computer Science. The $\mathcal{P} \neq \mathcal{NP}$ Conjecture is supported by our intuition regarding its two formulations: we believe that solving problems is harder than verifying a given solution, and that proofs do help. Empirical evidence towards the conjecture is given by the fact that literally thousands of NP-type problems, coming from a wide variety of mathematical and scientific disciplines, are not known to be polynomial-time solvable in spite of extensive research attempts aimed at providing efficient solving procedures for them. A famous example is the Integer Factorization problem: given a natural number, find its prime factorization.

Another Big Question: Assuming that $\mathcal{P} \neq \mathcal{NP}$, it is not clear whether the existence of an efficient verification procedure for proofs of membership in a set implies the the existence of an efficient verification procedure for proofs of non-membership in that set. That is, let co$\mathcal{NP}$ denote the class of sets that are complements of sets in $\mathcal{NP}$ (i.e., co$\mathcal{NP} \stackrel{\rm def}{=} \{\{0,1\}^* \setminus S : S \in \mathcal{NP}\}$).

Open Problem 7 (NP versus coNP) *Is $\mathcal{NP}$ equal to co$\mathcal{NP}$?*

It is widely believed that co$\mathcal{NP} \neq \mathcal{NP}$. (Indeed, this implies $\mathcal{P} \neq \mathcal{NP}$.)

[3]In some sources the class $\mathcal{NP}$ is defined via a *fictitious* computing device called a non-deterministic machine, and NP stands for Non-deterministic Polynomial-time.

4 NP-Completeness

For the current discussions, it is more convenient to view decision problems as Boolean functions defined over the set of strings (i.e., $S : \{0,1\}^* \to \{0,1\}$) rather than as sets of strings (i.e., $S \subseteq \{0,1\}^*$). A general notion of (polynomially-time) reducibility among computational problems is obtained by considering a (polynomially-time) machine for solving one problem (e.g., computing a function f) that may ask queries to another problem (e.g., to a function g).[4] Below, we consider a restricted notion of (polynomially-time) reducibility in which the machine makes a single query and outputs the answer it obtains.

Definition 8 (NP-completeness) *A function f is* polynomially-reducible *to a function g if there exist a polynomial-time computable function h so that $f(x) = g(h(x))$ for every $x \in \{0,1\}^*$. Specifically, in case of decision problems, the set S is polynomially-reducible to the set S' if it holds that $x \in S$ if and only if $h(x) \in S'$. A decision problem S is* $\mathcal{NP}$-complete *if S is in $\mathcal{NP}$ and every decision problem in $\mathcal{NP}$ is polynomially-reducible to S.*

Thus, $\mathcal{NP}$-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 any problem in $\mathcal{NP}$ (and in particular *all* $\mathcal{NP}$-complete decision problems). Furthermore, in a sense, each of these ($\mathcal{NP}$-complete) problems "efficiently encodes" all the other problems, and in fact all NP-type search problems. For example, the problem of integer factorization can be "efficiently encoded" in any $\mathcal{NP}$-complete problem (which may have nothing to do with integers).[5] Thus, at first glance it seems very surprising that $\mathcal{NP}$-complete problems exist at all.

Theorem 9 *There exist $\mathcal{NP}$-complete decision problems. Furthermore, the following decision problems are $\mathcal{NP}$-complete:*

SAT : *Given a propositional formula, decide whether it is satisfiable.*[6]

3-Coloring : *Given a simple graph, decide whether it is 3-colorable.*[7]

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

[5] In particular, a polynomial-time decision procedure for any of the problems below yields a polynomial-time algorithm for factoring integers.

[6] The problem remains $\mathcal{NP}$-complete even when instances are restricted to be in Conjunctive Normal Form (CNF), and even when each clause has exactly 3 literals. In this case, the input is a set of clauses, each consisting of 3 literals, where each literal is either a Boolean variable or its negation. The question is whether there exists a truth assignment to the variables, so that each clause contains at least one literal that evaluates to `true`.

[7] The input consists of a set of unordered pairs, called edges, over a finite set V (of vertices). The question is whether there exists a mapping $\phi : V \to \{1,2,3\}$ so that $\phi(u) \neq \phi(v)$ for every edge (u,v).

Subset Sum: *Given a sequence of integers* $a_1, ..., a_n$ *and* b*, decide whether there exists a set* I *so that* $\sum_{i \in I} a_i = b$.

The decision problems mentioned above are but three examples out of literally thousands of $\mathcal{NP}$-complete problems, coming from a wide variety of mathematical and scientific disciplines. (Hundreds of such problems are listed in [4].)

Assuming that $\mathcal{P} \neq \mathcal{NP}$, no $\mathcal{NP}$-complete problem has a polynomial-time decision procedure. Consequently, the corresponding NP-type search problem (associated with the relation in Definition 4), cannot be solve in polynomial-time.

5 Randomized Computation

As hinted in Section 3, so far our approach to computing devises was somewhat conservative: we thought of them as (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* polynomial-time computations. Specifically, we allow probabilistic rules that choose uniformly among two predetermined possibilities, and observe that the effect of more general probabilistic rules can be efficiently approximated by a rule of the former type. We comment that probabilistic computations are believed to take place in real-life computations that are employed in a variety of applications (e.g., sampling, simulations, etc.).[8]

Rigorous models of probabilistic machines are defined by natural extensions of the basic model; for example, we will talk of probabilistic Turing machines. Again, the specific choice of model is immaterial; as long as it is "reasonable". We consider the output distribution of such probabilistic machines on fixed inputs; that is, for a probabilistic machine M and string $x \in \{0,1\}^*$, we denote by $M(x)$ the distribution of the output of M on input x, where the probability is taken over the machine's random moves. Considering decision problems, three natural types of machines arise:

1. Machines that never err, but may output a special `don't know` symbol (denoted $\perp$)[9];

2. Machines with one-sided error probability (see below);

3. Machines with two-sided error probability.

[8] In what sense do these applications really utilize random moves is a different question. The point is that the programmers and users treat these computations as if they are taking random moves. We further discuss this issue in Section 6.1.

[9] The latter relaxation is essential, or else one may obtain an equivalent deterministic machine by merely fixing all choices of the probabilistic machine (e.g., to be all 1).

Of course, the error probability needs to be bounded (or else the definition is meaningless). We focus on probabilistic polynomial-time machines, and on error probability that may be reduced to a "negligible" amount by polynomially many independent repetitions. This gives rise to natural complexity classes such as the following.

Definition 10 ($\mathcal{ZPP}$, $\mathcal{RP}$ and $\mathcal{BPP}$) *Let $S : \{0,1\}^* \to \{0,1\}$ be a decision problem, and M be a probabilistic polynomial-time machine.*

1. *Suppose that for every $x \in \{0,1\}^*$ it holds that $M(x) \in \{S(x), \perp\}$ and $\Pr[M(x) = S(x)] \geq \frac{1}{2}$. Then $S \in \mathcal{ZPP}$.*
2. *Suppose that for every $x \in \{0,1\}^*$ it holds that $S(x) = 1$ implies $\Pr[M(x) = 1] \geq \frac{1}{2}$ and $S(x) = 0$ implies $\Pr[M(x) = 0] = 1$. Then $S \in \mathcal{RP}$.*

   Similarly, if $S(x) = 1$ implies $\Pr[M(x) = 1] = 1$ and $S(x) = 0$ implies $\Pr[M(x) = 0] \geq \frac{1}{2}$ then $S \in \mathrm{co}\mathcal{RP}$.
3. *If for every $x \in \{0,1\}^*$ it holds that $\Pr[M(x) = S(x)] \geq \frac{2}{3}$ then $S \in \mathcal{BPP}$.*

Indeed, $\mathrm{co}\mathcal{RP} = \{\{0,1\}^* \setminus S : S \in \mathcal{RP}\}$, and $\mathcal{ZPP} = \mathcal{RP} \cap \mathrm{co}\mathcal{RP}$. We comment that, in all cases, the error (or *don't know*) probability can be reduced to, say, $\exp(-|x|)$ by invoking M for $O(|x|)$ times, where in each run M utilizes independent random choices.

A fundamental question that comes to mind refers to the effect of randomization on the computing power. Since $\mathcal{P} \subseteq \mathcal{ZPP} \subseteq \mathcal{RP} \subseteq \mathcal{BPP}$, the real question is whether these inclusions are strict. In Section 6 we discuss evidence to the contrary, still the following is a fundamental open problem:

Open Problem 11 *Does $\mathcal{P} = \mathcal{BPP}$?*

The set of prime numbers is known to be in $\mathcal{ZPP}$. It is also known that the Extended Riemann Hypothesis (ERH) implies that the set of primes is in $\mathcal{P}$, but an analogue unconditional result is not known. The current state of knowledge (by which a specific problem is in $\mathcal{P}$ if either ERH holds or a general computational conjecture holds but not unconditionally) seems fascinating.

5.1 Counting at Random

An interesting question regarding NP-type search problems is to determine how many solutions does a specific instance have. Clearly, counting the number of solutions (even approximately) allows to determine whether a solution exists at all. For example, approximately counting the number of satisfying assignments to a given propositional formula allows to determine whether the formula is satisfiable. Interestingly, approximately counting the number of satisfying assignments is not significantly harder than deciding if such exists:

Theorem 12 *There exists a probabilistic polynomial-time oracle*[10] *machine 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 of ψ.*

We comment that an analogous statement holds for any $\mathcal{NP}$-complete problem, and that it is not known whether a similar approximation can be obtained by a *deterministic* polynomial-time oracle machine. 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. Let us phrase this too as an important open problem.

Open Problem 13 *Does there exist a probabilistic polynomial-time oracle machine that, on input a formula ψ and oracle access to SAT, outputs an integer that with probability at least $\frac{2}{3}$ equals the number of satisfying assignments of ψ.*

Turning back to what is known, we mention that a machine as in Theorem 12 can generate a uniformly distributed satisfying assignment, provided that such exists: *There exists a probabilistic polynomial-time oracle machine that, on input a satisfiable formula ψ and oracle access to SAT, outputs a uniformly distributed satisfying assignment to ψ.*

5.2 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 gives proofs their value. Conceptually speaking, proofs are secondary to the verification procedure; indeed, proof systems are defined in terms of their verification procedures.

The notion of a verification procedure assumes the notion of computation and furthermore the notion of efficient computation. This implicit notion is made explicit in the definition of $\mathcal{NP}$, in which efficient computation is associated with (deterministic) polynomial-time procedures. Let us restate $\mathcal{NP}$ as a class of proof systems.

Definition 14 (NP-proof systems) *Let $S \subseteq \{0,1\}^*$ and $\nu : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}$ be a function so that $x \in S$ if and only if there exists a $w \in \{0,1\}^*$ such that $\nu(x,w) = 1$. If ν is computable in time bounded by a polynomial in the length of its first argument then we say that S is an* NP-set *and that ν defines an* NP-proof system.

The formulation of NP-proofs restricts the "effective" length of proofs to be polynomial in length of the corresponding assertions (since the running-time of the verification procedure is restricted to be polynomial in the length of the

[10] See Footnote 4. Here, upon making any query ψ' the machine is told whether ψ' is satisfiable or not.

assertion). However, longer proofs may be allowed by padding the assertion with sufficiently many blank symbols. So it seems that NP-proofs give a satisfactory formulation of proof systems (with efficient verification procedures). This is indeed the case if one associates efficient procedures with *deterministic* polynomial-time procedures. However, we can gain a lot if we are willing to take a somewhat non-traditional step and allow *probabilistic* verification procedures. In particular:

- Randomized and interactive verification procedures, giving rise to *interactive proof systems*, seem much more powerful (i.e., "expressive") than their deterministic counterparts. In particular, such interactive proof systems exists for any set in co$\mathcal{NP}$ (e.g., for the set of unsatisfied propositional formula), whereas it is widely believed that some sets in co$\mathcal{NP}$ do NOT have NP-proof systems (i.e., $\mathcal{NP} \neq$ co$\mathcal{NP}$).

  Loosely speaking, an interactive proof system is a game between a computationally bounded verifier and a computationally unbounded prover whose goal is to convince the verifier of the validity of some assertion. Specifically, the verifier is probabilistic and its time-complexity is polynomial in the length of the assertion. It is required that if the assertion holds then the verifier always accepts (when interacting with an appropriate prover strategy). On the other hand, if the assertion is false then the verifier must reject with probability at least $\frac{1}{2}$, no matter what strategy is being employed by the prover. Thus, a "proof" in this context is not a fixed and static object, but rather a randomized (dynamic) process in which the verifier interacts with the prover. Intuitively, one may think of this interaction as consisting of "tricky" questions asked by the verifier, to which the prover has to reply "convincingly".

- Such randomized procedures allow the introduction of *zero-knowledge proofs* which are of great theoretical and practical interest. Furthermore, under reasonable complexity assumptions (such as those in Section 6), every set in $\mathcal{NP}$ has a zero-knowledge proof system.

  Loosely speaking, zero-knowledge proofs are interactive proofs that yield nothing (to the verifier) beyond the fact that the assertion is indeed valid. That is, whatever the verifier can efficiently compute after interacting with a zero-knowledge prover, can be efficiently computed from the assertion itself (without interacting with anyone). Thus, zero-knowledge proofs exhibit an extreme contrast between being convinced of the validity of a statement and learning anything in addition (while receiving such a convincing proof).

- NP-proofs can be efficiently transformed into a (redundant) form that offers a trade-off between the number of locations examined in the NP-proof and the confidence in its validity. The latter redundant proofs are called *probabilistically checkable proofs* (or PCP).

Loosely speaking, a PCP system consists of a probabilistic polynomial-time verifier having access to an oracle which represents a proof in redundant form. Typically, the verifier accesses only few of the oracle bits, and these bit positions are determined by the outcome of the verifier's coin tosses. Again, it is required that if the assertion holds then the verifier always accepts (when given access to an adequate oracle); whereas, if the assertion is false then the verifier must reject with probability at least $\frac{1}{2}$, no matter which oracle is used.

It turns out that any set in $\mathcal{NP}$ has a PCP system in which the verifier asks only a constant number of (Boolean!) queries.

In all the abovementioned types of probabilistic proof systems, explicit bounds are imposed on the computational complexity of the verification procedure, which in turn is personified by the notion of a verifier. Furthermore, in all these proof systems, the verifier is allowed to toss coins and rule by statistical evidence. Thus, all these proof systems carry a probability of error; yet, this probability is explicitly bounded and, furthermore, can be reduced by successive application of the proof system.

6 The Bright Side of Hardness

The conjecture by which $\mathcal{P} \neq \mathcal{NP}$ means that there are computational problems of great interest that are inherently intractable. These are bad news, but there is a bright side to them: 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) phenomena for some efficiently-computable functions. That is, we assume that one-way functions (as defined below) do exist.

Definition 15 (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.*
2. hard to invert: *for every probabilistic polynomial-time machine, M, every positive polynomial $p(\cdot)$, and all sufficiently large n's*

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

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

For example, the widely believed conjecture by 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. *An 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 one hand, and two important computational theories on the other hand. Specifically, we refer to the theory of pseudorandomness and to the theory of cryptography. A 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[11] over bit strings rather than individual strings. We actually consider probability ensembles each being an infinite sequence of distributions, where each distribution assigns positive probability weight only to strings of length polynomial in the index of the distribution (within the ensemble).

Definition 16 (Computational Indistinguishability) *The probability ensembles $\{P_n\}_{n\in\mathbb{N}}$ and $\{Q_n\}_{n\in\mathbb{N}}$ are called* computationally indistinguishable *if for every probabilistic polynomial-time machine, M, every positive polynomial $p(\cdot)$, and all sufficiently large n's*

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

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 care about pairs of ensembles that are efficiently samplable.[12] Interestingly, such pairs exists if and only if one-way functions exist.

6.1 Pseudorandomness

Loosely speaking, a pseudorandom generator is an *efficient* (deterministic) procedure that *stretches* short random strings into longer strings that are *computationally indistinguishable from long random strings.*

Definition 17 (pseudorandom generators) *A deterministic polynomial-time machine G is called a* pseudorandom generator *if there exists a monotonely in-*

[11] We stress that when we talk of distributions we mean discrete probability distributions having a finite support that is a set of strings.

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

creasing function, $\ell : \mathsf{N} \to \mathsf{N}$, so that the following two probability ensembles, denoted $\{G_n\}_{n\in\mathsf{N}}$ and $\{R_n\}_{n\in\mathsf{N}}$, are computationally indistinguishable.

1. *Distribution G_n is defined as the output of G on a uniformly selected n-bit string.*

2. *Distribution R_n is defined as the uniform distribution over $\{0,1\}^{\ell(n)}$.*

The function ℓ is called the stretch measure *of the generator.*

That is, pseudorandom generators yield a particularly interesting case of computational indistinguishability: For every n, the distribution G_n is efficiently samplable using less that $|G_n|$ truly random coins, and yet it is computationally indistinguishable from the uniform distribution over $|G_n|$-bit long strings (i.e., the distribution R_n).

Theorem 18 *Pseudorandom generators exist if and only if one-way functions exist. Furthermore, in case pseudorandom generators exist they exists for any stretch measure that is a polynomial.*

Thus, in a sense, computational difficulty can be converted into randomness, and vice versa. Furthermore, *the proof of Theorem 18* links computational indistinguishability to computational unpredictability, hinting that computational difficulty (of predicting an information-theoretically determined event) is linked to randomness (or to appearance of being random).

Using pseudorandom generators. Pseudorandom generators allow to shrink the amount of "true randomness" used in any efficient randomized procedure. This is done by feeding the procedure with the output of a pseudorandom generator invoked on a truly random shorter string. The modified procedure needs a much smaller amount of "true randomness" but essentially maintains the functionality of the original procedure.[13] Still we need to start with some amount of "true randomness", and the question is from where to obtain it. The answer is that "true randomness" (or something that appears so) may be obtained from nature; that is, by sampling some physical phenomena. Indeed, such samples are not uniformly distributed over the set of strings of specific length, yet if they contain enough entropy then almost perfect randomness can be (efficiently) extracted from them.[14]

[13] Using seemingly stronger notions of pseudorandom generators, one may shrink the amount of "true randomness" to an even lower level, at which it is feasible to deterministically scan all possibilities. Such seemingly stronger pseudorandom generators imply that $\mathcal{BPP} = \mathcal{P}$, and exist under seemingly stronger (and yet very plausible) conjectures regarding computational difficulty.

[14] The construction of such randomness extractors is indeed a very active research direction, and the currently known results (although not optimal) are very satisfactory.

Pseudorandom functions. Pseudorandom generators allow one to efficiently generate long pseudorandom sequences from short random seeds. Pseudorandom functions are even more powerful: they allow efficient direct access to a huge pseudorandom sequence (which is infeasible to scan bit-by-bit). In other words, pseudorandom functions can replace truly random functions in any efficient application (e.g., most notably in cryptography). That is, pseudorandom functions are indistinguishable from random functions by any efficient procedure that may 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.*

6.2 Cryptography

The assumption that one-way functions exists is a necessary and sufficient condition for much of modern cryptography. Here we focus on the basic tasks of providing secret and authenticated communication. Ignoring several important issues, these tasks are reduced to the construction of encryption and signature schemes.

Encryption schemes. Such schemes are supposed to provide secret communication between parties in a setting in which these parties communicate over a channel that may be eavesdropped by an adversary. There are two cases differing by whether the communicating parties have agreed on a common secret prior to the communication or not. In both cases, the encryption scheme consists of three probabilistic polynomial-time procedures: *key generation*, *encryption* (denoted E), and *decryption* (D). Loosely speaking, on input a security parameter n (in unary), the key-generation outputs a pair of corresponding encryption and decryption keys, (e, d), so that for every string $x \in \{0,1\}^*$, it holds that $D_d(E_e(x)) = x$, where $E_e(x)$ (resp., $D_d(y)$) denotes the output of the encryption (resp., decryption) procedure on input (e, x) (resp., (d, y)).

The difference between the two cases is in the way the scheme is employed and this will be reflected in the definition of security. In the first case, known as the *private-key* case, a set of mutually trustful parties employ the key-generation process, prior to the actual communication, obtaining a pair of keys (e, d). We stress that, in this case, the encryption-key e is known to all trusted parties and only to them. Later, each trusted party may encrypt messages by applying E_e, and retrieve them (i.e., decrypt) by applying D_d. The information available to the adversary, in this case, is a sequence of encrypted messages sent over the channel, using a fixed encryption-key unknown to it. (We stress that the total amount of information encrypted using this encryption-key may be much larger than the length of the key, and so perfect information theoretic secrecy is not possible.)

In the second case, known as the *public-key* case, the receiver invokes the key-generation process, publicizes the encryption-key e (but not the decryption-key

d), and the sender uses e to generate encryptions as before. This allows everybody (not only parties that the receiver trusts) to send encrypted messages to the receiver, but in such a case also the adversary knows the encryption-key e. Thus, the information available to the adversary in this case is a sequence of encrypted messages sent over the channel, using a fixed encryption-key that is also known to it. In both cases, security amounts to asserting that the adversary does not learn anything from the information available to it. That is, whatever the adversary can efficiently compute from the public information, can be efficiently computed from scratch.[15]

Private-key encryption schemes exist if and only if one-way functions exists.[16] Public-key encryption schemes can be constructed based on a seemingly stronger assumption; yet this assumption is also implied by the abovementioned conjecture regarding the intractability of integer factorization.

Signature schemes. Here too we have two cases corresponding to whether a certain key (here it is the verification-key) is public or not. In both cases, the scheme consists of three probabilistic polynomial-time procedures: *key generation*, *signing* (S), and *verification* (V). On input a security parameter n (in unary), the key-generation outputs a pair of corresponding signing and verification keys, (s, v), so that for every string $x \in \{0,1\}^*$, it holds that $V_v(x, S_s(x)) = 1$, where $S_s(x)$ (resp., $V_d(x, y)$) denotes the output of the signing (resp., verification) procedure on input (s, x) (resp., (v, x, y)).

The difference between the two cases is in the way the scheme is employed and this will be reflected in the definition of security. In the *private-key* case (a.k.a message-authentication), the scheme is used to authenticate messages sent among mutually trustful parties that communicate over a channel that may be subject to message corruptions (and/or message insertion/deletion). It is assumed that the parties have invoked the key-generation process prior to the communication, obtaining a signing-key s (which may w.l.o.g equal the verification-key v). Subsequently, the sender authenticates each message x by appending $S_s(x)$ to it, and the receiver verifies the authenticity by applying V_v. In the *public-key* case, the scheme is used in order to allow universal verification of commitments done by parties. Towards this end, each party invokes the

[15]The actual formulation refers to the notion of computational indistinguishability. It asserts that for every distribution ensemble of the first type (representing what the aversary computes from the information available to it) there exists a distribution ensemble of the second type (representing what can be computed from scratch) so that the two ensembles are computationally indistinguishable. Note that in the private-key case, we may assume without loss of generality that $e = d$; whereas in the public-key case, d must be hard to compute from e.

[16]Specifically, a private-key encryption scheme may be constructed as follows. The key-generation procedure consists of selecting a pseudorandom function $f : \{0,1\}^n \to \{0,1\}^n$, which serves both as the encryption and decryption key. Subsequently, each message $x \in \{0,1\}^n$ is encrypted by uniformly selecting $r \in \{0,1\}^n$ and sending $(r, f(r) \oplus x)$, where $\oplus$ denotes the bit-by-bit exclusive-or of equal-length strings.

key-generation process, deposits the resulting verification-key v on a trusted public-file, and keeps the corresponding signing-key s secret. When the user later wishes to commit to a document, it applies S_s to it, and this commitment is universally verifiable with respect to its public verification-key.

In both cases, security amounts to asserting that it is infeasible for anybody given the public information (but not having the signing-key), to produce a valid signature (i.e., a commitment w.r.t the verification-key) to a document for which such a commitment was not supplied before by a party holding the signing-key. That is, forgery should be infeasible even if the forger may ask the legitimate user to sign documents of its choice; after such an attack the forger may indeed present valid signatures to all documents it has requested a signature for, but not for any other document. (We stress that in case of public-key schemes this is required to hold even if the forger has the verification-key.)

Private-key signature schemes exist if and only if one-way functions exists.[17] Public-key signature schemes can be constructed based on the same assumption.

Beyond encryption and signature schemes. We stress that cryptography encompasses much more than methods for providing secret and authenticated communication. In general, cryptography is concerned with the construction of schemes that maintain *any* desired functionality under malicious attempts aimed at making them deviate from their prescribed functionality. Loosely speaking, a secure implementation of a multi-party functionality is a multi-party protocol in which the *impact* of malicious parties is effectively restricted to applying the prescribed functionality on inputs chosen by the corresponding parties. A major result in the area states that under plausible assumptions regarding computational difficulty, *any efficiently computed functionality can be securely implemented*.

7 The Tip of an Iceberg

Even within the topics discussed above, many important results were not discussed. Some of these omissions will amazed experts in the field; but in view of space limitations we had no choice but to omit many interesting results regarding the above topics. Furthermore, other important topics and even wide areas were not mentioned at all. We briefly discuss some of these topics and areas.

Relaxing the requirements. The "$\mathcal{P}$ versus $\mathcal{NP}$" question, as well as much of the discussion in Sections 2–4, 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

[17] Specifically, a private-key signature scheme may be constructed as follows. The key-generation procedure consists of selecting a pseudorandom function $f : \{0,1\}^n \to \{0,1\}^n$. Subsequently, each message $x \in \{0,1\}^n$ is signed by the value $f(x)$.

procedures that "typically" give an "approximate" answer. Indeed, both terms in quotes require clarification:

1. *Average-case complexity.* Indeed, one may talk of procedures that answer correctly on a large fraction of the instances, but such a discussion assumes that all instances are equally interesting for practice, which is typically not the case.[18] A more appealing theory of average-case complexity must consider a wide class of "simple distributions" and measure the performance of procedures when instances are selected according to such distributions (cf. [5]). We warn that allowing arbitrary distributions would collapse average-case complexity to worst-case complexity (as discussed in Sections 2–4). A reasonable choice of a class of "simple distributions" is the class of distributions that can be efficiently sampled from.

2. *Approximation.* What do we mean by an approximation to a computational problem? There are many possible answers, and their meaningfulness depends on the specifics of the application. For example, in case of search problems, we may be satisfied with a solution that is close to be valid; e.g., for a search problem $R \subseteq \{0,1\}^* \times \{0,1\}^*$, given x we may be content with a y' that differs in relatively few bits from a string y satisfying $(x, y) \in R$. More generally, we may care about a payoff function $\pi : \{0,1\}^* \times \{0,1\}^*$ so that given x one should find a y with maximum (or close to maximum) value for $\pi(x, y)$. (See [8].) A natural notion of approximation is applicable also to decision problems (i.e., determining set membership): given an instance x we may ask how close is x (under some relevant distance measure) to an instance in the set (cf. [6]).

Other complexity measures. So far, 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 important too (cf. [13]). Another important issue is to what extent can a computation be performed in parallel; that is, speeding-up the computation by running – concurrently – several computing devices (which may exchange information during the course of computation). In addition to parallel-time, a fundamentally important complexity measure in such a case is the number of (parallel) computing devices used (cf. [10]).

Other notions of computation. A setting related to parallel computing is the one of distributed computing, with the difference being that in the latter case only parts of the input are given to each computing device. Furthermore,

[18] We mention that the formulation of one-way function does refer to one simple distribution of instances, which may be uniform in case the function is 1-1 over the set of strings of any length. However, there we deal with artificially generated (hard) instances, rather than with problem instances that arise from natural applications.

in typical studies one wishes to minimize the amount of communication between these devices (and certainly prohibit communicating the entire input among the devices). Consequently, measures of communication complexity arise and play a major role (cf. [1]). Communication complexity is also considered as a measure of the "complexity" of functions (cf. [12]), but in these abstract studies communication proportional to the length of the input is not ruled out (but rather appears frequently). An altogether different type of computational problems are investigated in the context of computational learning theory (cf. [11]).

Major areas we have ignored. As stated in the introduction, our exposition totally ignores two major areas of complexity theory: circuit complexity (cf. [3]) and proof complexity (cf. [2]). The activity in these areas is aimed towards developing proof techniques that may be used towards the resolution of the big problems (such as P vs NP), but the current achievements – though very impressive – seem far from reaching this goal. Current crown-jewel achievements in these areas take the form of tight (or strong) lower bounds on the complexity of computing (resp., proving) "relatively simple" functions (resp., claims) in restricted models of computation (resp., proof systems).

8 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. We believe that the coming century will witness even more exciting developments in this field, and urge the reader to try to contribute to them.

Bibliographic Notes: Providing even a minimal set of bibliographic notes for the material discussed in the main part of this exposition would have resulted in an extensive bibliography, which we cannot afford due to space limitations. Instead, we merely refer the reader to books containing such bibliographic notes: For Sections 2–4, see [9, 4]. For Sections 5 and 6, see [7].

References

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

[2] P. Beame and T. Pitassi: Propositional Proof Complexity: Past, Present, and Future. In *Bulletin of the European Association for Theoretical Computer Science*, Vol. 65, June 1998, pp. 66–89.

[3] R. Boppana and M. Sipser: The complexity of finite functions. In *Handbook of Theoretical Computer Science: Volume A– Algorithms and Complexity*, J. van Leeuwen editor, MIT Press/Elsevier, 1990, pp. 757–804.

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

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

[6] O. Goldreich: Combinatorial Property Testing – A Survey. In *DIMACS Series in Disc. Math. and Theoretical Computer Science*, Vol. 43 (Randomization Methods in Algorithm Design), 1998.

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

[8] D. Hochbaum (ed.): *Approximation Algorithms for NP-hard Problems.* PWS, 1996.

[9] J.E. Hopcroft and J.D. Ullman: *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, 1979.

[10] R.M. Karp and V. Ramachandran: Parallel Algorithms for Shared Memory Machines. In *Handbook of Theoretical Computer Science, Vol A: Algorithms and Complexity*, 1990.

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

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

[13] M. Sipser: *Introduction to the Theory of Computation.* PWS Publishing Company, 1997.