Lecture Series III

The less traditional material

These lectures are based on research done in the 1980's and the 1990's.

The lectures on *Probabilistic Proof Systems* and *Pseudorandomness* are related to lectures that may be given as part of other courses (i.e., *Foundations of Cryptography* and *Randomness in Computation*, respectively). But the choice of material for the current course as well as the perspective would be different here.

Lecture 12

Probabilistic Proof Systems

Various types of *probabilistic* proof systems have played a central role in the development of computer science in the last decade. In these notes, we concentrate on three such proof systems: *interactive proofs*, *zero-knowledge proofs*, and *probabilistic checkable proofs*.

The notes for this lecture were adapted from various texts that I wrote in the past (see, e.g., [26, Chap. 2]). In view of the fact that that zero-knowledge proofs are covered at Weizmann in the Foundation of Cryptography course, I have only discussed IP and PCP in the current course. The actual notes I have used in the current course appear in Section 12.5.

12.1 Introduction

The glory given to the creativity required to find proofs, makes us forget that it is the less glorified procedure of verification which gives proofs their value. Philosophically speaking, proofs are secondary to the verification procedure; whereas technically speaking, 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 assumption is made explicit in the definition of $\mathcal{NP}$, in which efficient computation is associated with (deterministic) polynomial-time algorithms.

Traditionally, NP is defined as the class of NP-sets. Yet, each such NP-set can be viewed as a proof system. For example, consider the set of satisfiable Boolean formulae. Clearly, a satisfying assignment τ for a formula ϕ constitutes an NP-proof for the assertion "ϕ is satisfiable" (the verification procedure consists of substituting the variables of ϕ by the values assigned by τ and computing the value of the resulting Boolean expression).

The formulation of NP-proofs restricts the "effective" length of proofs to be polynomial in length of the corresponding assertions. However, longer proofs may be considered by padding the assertion with sufficiently many blank symbols. So it seems that NP gives a satisfactory formulation of proof systems (with efficient verification procedures). This is indeed the case if one associates efficient procedures with *deterministic* polynomial-time algorithms. 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.
- Such randomized procedures allow the introduction of *zero-knowledge proofs*, which are of great theoretical and practical interest.

- 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 (which is captured in the notion of *probabilistically checkable proofs*).

In all 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.

12.2 Interactive Proof Systems

In light of the growing acceptability of randomized and distributed computations, it is only natural to associate the notion of efficient computation with probabilistic and interactive polynomial-time computations. This leads naturally to the notion of interactive proof systems in which the verification procedure is interactive and randomized, rather than being non-interactive and deterministic. 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". The above discussion, as well as the actual definition, makes explicit reference to a prover, whereas a prover is only implicit in the traditional definitions of proof systems (e.g., NP-proofs).

12.2.1 The Definition

The main new ingredients in the definition of interactive proof systems are:

- *Randomization* in the verification process.
- *Interaction* between the verifier and the prover, rather than uni-directional communication (from the prover to the verifier) as in the case of NP-proof systems.

The combination of both new ingredients is the source of power of the new definition: If the verifier does not toss coins then the interaction can be collapsed to a single message. (On the other hand, combining randomization with uni-directional communication yields a randomized version of NP-proof systems, called $\mathcal{MA}$.) We stress several other aspects:

- *The prover is computationally unbounded*: As in NP, we start by not considering the complexity of proving.
- *The verifier is probabilistic polynomial-time*: We maintain the paradigm that verification ought to be easy, alas we allow random choices (in our notion of easiness).
- Completeness and Soundness: We relax the traditional soundness condition by allowing small probability of being fooled by false proofs. The probability is taken over the verifier's random choices. (We still require "perfect completeness"; that is, that correct statements are proven with probability 1). Error probability, being a parameter, can be further reduced by successive repetitions.

We denote by $\mathcal{IP}$ the class of sets having interactive proof systems.

Variations: Relaxing the "perfect completeness" requirement yields a two-sided error variant of $\mathcal{IP}$ (i.e., error probability allowed also in the completeness condition). Restricting the verifier to send only "random" (i.e., uniformly chosen) messages yields the restricted notion of Arthur-Merlin interactive proofs (aka public-coins interactive proofs, and denoted $\mathcal{AM}$). However, both variants are essentially as powerful as the original one.[1]

12.2.2 An Example: interactive proof of Graph Non-Isomorphism

The problem (not known to be in $\mathcal{NP}$)**:** Proving that two graphs are isomorphic can be done by presenting an isomorphism, but how do you prove that no such isomorphism exists?

The construction – the "two object protocol": If you claim that two objects are different then you should be able to tell which is which (when I present them to you in random order). In the context of the Graph Non-Isomorphism interactive proof, two (supposedly) different objects are defined by taking random isomorphic copies of each of the input graphs. If these graphs are indeed non-isomorphic then the objects are different (the distributions have distinct support) else the objects are identical.

12.2.3 Interactive proof of Non-Satisfiability

We show that co$\mathcal{NP} \subseteq \mathcal{IP}$ by presenting an interactive proof for Non-Satisfiability (i.e., $\overline{SAT}$).

Arithmetization of Boolean (CNF) formulae: Given a Boolean (CNF) formula, we replace the Boolean variables by integer variables (their negations by 1 minus the variable), OR-clauses by sums, and the top level conjunction by a product. Note that `false` is associated with zero, whereas `true` is associated with a positive integer. To prove that the given formula is not satisfiable, we consider the sum over all 0-1 assignments of the resulting integer expression. Observe that the resulting arithmetic expression is a low degree polynomial (i.e., the degree is at most the number of clauses), and that its value is bounded (i.e., exponentially in the number of clauses).

Moving to a Finite Field: Whenever we check equality between two integers in $[0, M]$, it suffices to check equality mod q, where $q > M$. The benefit is that the arithmetic is now in a finite field (mod q) and so certain things are "nicer" (e.g., uniformly selecting a value). Thus, proving that a CNF formula is not satisfiable reduces to proving an equality of the following form

$$\sum_{x_1=0,1} \cdots \sum_{x_n=0,1} \phi(x_1, ..., x_n) \equiv 0 \pmod q$$

where ϕ is a low degree multi-variant polynomial (and q is exponential in n).

The actual construction: stripping summations in iterations. In each iteration the prover is supposed to supply the polynomial describing the expression in one (currently stripped) variable. (By the above observation, this is a low degree polynomial and so has a short description.) The verifier checks that the polynomial is of low degree, and that it corresponds to the current value being claimed (i.e., $p(0) + p(1) \equiv v$). Next, the verifier randomly instantiates the variable, yielding a new value to be claimed for the resulting expression (i.e., $v \leftarrow p(r)$, for uniformly chosen

[1] See [21] and [36], respectively. Specifically, we can get rid of the completeness error by adapting the proof of Theorem 7.1 (cf. [21]). The proof that $\mathcal{AM} = \mathcal{IP}$ is significantly more involved (cf. [36]).

$r \in \mathrm{GF}(q)$). The verifier sends the uniformly chosen instantiation to the prover. (At the end of the last iteration, the verifier has a fully specified expression and can easily check it against the claimed value.) That is, for $i = 1, ..., n$, the ith iteration is intended to provide evidence that $\sum_{x_i=0,1} \cdots \sum_{x_n=0,1} \phi(r_1, ..., r_{i-1}, x_i, ..., x_n) \equiv v_{i-1} \pmod q$, where $r_1, ..., r_{i-1}, v_{i-1}$ are as determined in the previous $i-1$ iterations (and $v_0 \stackrel{\text{def}}{=} 0$). The prescribed prover is supposed to set $p_i(z) = \sum_{x_{i+1}=0,1} \cdots \sum_{x_n=0,1} \phi(r_1, ..., r_{i-1}, z, x_{i+1}, ..., x_n)$, and send p to the verifier, which checks that $p_i(0) + p_i(1) \equiv v_{i-1} \pmod q$ (rejecting immediately if the equivalence does not hold), selects r_i at random in $\mathrm{GF}(q)$, sends it to the prover, and sets $v_i = p_i(r_1) \bmod q$. (In the next iteration, the verifier expects to get evidence that $\sum_{x_{i+1}=0,1} \cdots \sum_{x_n=0,1} \phi(r_1, ..., r_{i-1}, r_i, x_{i+1}, ..., x_n) \equiv v_{i-1} \pmod q$.)

Completeness of the above: When the claim holds, the prover has no problem supplying the correct polynomials, and this will lead the verifier to always accept.

Soundness of the above: It suffices to bound the probability that for a particular iteration the initial claim is false whereas the ending claim is correct. Both claims refer to the current summation expression being equal to the current value, where 'current' means either at the beginning of the iteration or at its end. Let $T(\cdot)$ be the actual polynomial representing the expression when stripping the current variable, and let $p(\cdot)$ be any potential answer by the prover. We may assume that $p(0) + p(1) \equiv v$ and that p is of low-degree (as otherwise the verifier will reject). Using our hypothesis (that the initial claim is false), we know that $T(0) + T(1) \not\equiv v$. Thus, p and T are different low-degree polynomials and so they may agree on very few points. In case the verifier instantiation does not happen to be one of these few points, the ending claim is false too.

Open Problem 1: alternative proof of co$\mathcal{NP} \subseteq \mathcal{IP}$. Polynomials play a fundamental role in the above construction and this trend has even deepened in subsequent works on PCP. It does not seem possible to abstract that role, which seems to be very annoying. I consider it important to obtain an alternative proof of co$\mathcal{NP} \subseteq \mathcal{IP}$; a proof in which all the underlying ideas can be presented at an abstract level.

12.2.4 The Power of Interactive Proofs

Theorem 12.1 (The IP Characterization Theorem): $\mathcal{IP} = \mathcal{PSPACE}$.

Interactive Proofs for PSPACE: Recall that PSPACE languages can be expressed by Quantified Boolean Formulae. The number of quantifiers is polynomial in the input, but there are both existential and universal quantifiers, and furthermore these quantifiers may alternate. Considering the arithmetization of these formulae, we face two problems: Firstly, the value of the formulae are only bounded by a double-exponential function (in the length of the input), and secondly when stripping out summations, the expression may be a polynomial of high degree (due to the universal quantifiers which are replaced by products). The first problem is easy to deal with by using the Chinese Reminder Theorem (i.e., if two integers in $[0, M]$ are different then they must be different modulo *most of the primes* in the interval $[1, \mathrm{poly}(\log M)]$). The second problem is resolved by "refreshing" variables after each universal quantifier (e.g, $\exists x \forall y \exists z \, \phi(x, y, z)$ is transformed into $\exists x \forall y \exists x' (x = x') \wedge \exists z \, \phi(x', y, z)$). That is, in the resulting formula, all variables appearing in a residual formula are quantified (within the residual formula).

IP is in PSPACE: We show that for every interactive proof system there exists an optimal prover strategy, and furthermore that this strategy can be computed in polynomial-space. This follows by looking at the tree of all possible executions. Thus, the acceptance probability of the verifier (when interacting with an optimal prover) can be computed in polynomial-space.

12.2.5 Advanced Topics

The IP Hierarchy

Let $\mathcal{IP}(r(\cdot))$ denote the class of languages having an interactive proof in which at most $r()$ messages are exchanges. Then, $\mathcal{IP}(0) = \text{co}\mathcal{RP} \subseteq \mathcal{BPP}$. The class $\mathcal{IP}(1)$ is a randomized version of $\mathcal{NP}$; witnesses are verified via a probabilistic polynomial-time procedure, rather than a deterministic one.[2] The class $\mathcal{IP}(2)$ seems to be fundamentally different; the verification procedure here is truly interactive. Still, this class seems relatively close to $\mathcal{NP}$; specifically, it is contained in the polynomial-time hierarchy (which seems 'low' when contrasted with $\mathcal{PSPACE} = \mathcal{IP}(\text{poly})$). Interestingly, $\mathcal{IP}(2r(\cdot)) = \mathcal{IP}(r(\cdot))$, and so in particular $\mathcal{IP}(O(1)) = \mathcal{IP}(2)$. (Note that "$\mathcal{IP}(2r(\cdot)) = \mathcal{IP}(r(\cdot))$" can be applied successively a constant number of times, but not more.)

Open Problem 2: the structure of the $\mathcal{IP}(\cdot)$ hierarchy: Suppose that $L \in \mathcal{IP}(r)$. What can be said about $\overline{L}$? Currently, we only know to argue as follows: $L \in \mathcal{IP}(r) \subseteq \mathcal{IP}(\text{poly}) \subseteq \mathcal{PSPACE}$, and so $\overline{L} \in \mathcal{PSPACE}$ and is in $\mathcal{IP}(\text{poly})$. This seems ridiculous: we do not use the extra information (i.e., $L \in \mathcal{IP}(r)$ and not merely $L \in \mathcal{IP}$). On the other hand, we do not expect $\overline{L}$ to be in $\mathcal{IP}(g(r))$, for any function g, since this would put $\text{co}\mathcal{NP} \subseteq \text{co}\mathcal{IP}(1)$ in $\mathcal{IP}(g(1)) \subseteq \mathcal{IP}(2)$. Other parameters of interest are the total lengths of the messages exchanged in the interaction and the total number of bits sent by the prover.[3] In general, it would be interesting to get a better understanding of the $\mathcal{IP}(\cdot)$ Hierarchy.

How Powerful Should the Prover be?

Here we consider the complexity of proving valid statements; that is, the complexity of the prescribed prover referred to in the completeness condition.

The Cryptographic Angle: Interactive proofs occur inside "cryptographic" protocols and so the prover is merely a probabilistic polynomial-time machine; yet it may have access to an auxiliary input (given to it or generated by it in the past). Such provers are relatively weak (i.e., they can only prove languages in $\mathcal{IP}(1)$); yet, they may be of interest for other reasons (e.g., see zero-knowledge).

The Complexity Theoretic Angle: It make sense to try to relate the complexity of proving a statement (to another party) to the complexity of deciding whether the statement holds. This gives rise to two related approaches:

1. Restricting the prover to be a probabilistic polynomial-time oracle machine with oracle access to the language (in which membership is proven). This approach can be thought of as extending the notion of self-reducibility (of NP-languages): these languages have an NP-proof

[2]This class is also denoted $\mathcal{MA}$. Observe that the proof of Theorem 7.1 can be adapted to give $\mathcal{BPP} \subseteq \mathcal{MA}$. Thus, $\mathcal{BPP} \cup \mathcal{NP} \subseteq \mathcal{MA}$.

[3]For a study of the latter complexity measure see On interactive proofs with a laconic provers (by Goldreich, Vadhan and Wigderson) in *Proc. of the 28th ICALP*, Springer's LNCS 2076, pages 334–345, 2001.

system in which the prover is a polynomial-time machine with oracle access to the language. Indeed, alike NP-complete languages, the IP-complete languages also have such a "relatively efficient" prover. (Recall that an optimal prover strategy can be implemented in polynomial-space, and thus by a polynomial-time machine having oracle access to a PSPACE-complete language.)

2. Restricting the prover to run in time that is polynomial in the complexity of the language (in which membership is proven).

Open Problem 3: Further investigate the power of the various notions, and in particular the one extending the notion of self-reducibility of NP languages. Better understanding of the latter is also long due. A specific challenge: provide an NP-proof system for Quadratic Non-Resideousity (QNR), using a probabilistic polynomial-time prover with access to the QNR language.[4]

Computationally-Sound Proofs

Computationally-sound proofs systems are fundamentally different from the above discussion (which did not effect the soundness of the proof systems): here we consider relaxations of the soundness conditions – false proofs may exist (even with high probability) but are hard to find. Variants may correspond to the above approaches; specifically, the following have been investigated:

Argument Systems: Here one only considers prover strategies implementable by (possibly non-uniform) polynomial-size circuits (equiv., probabilistic polynomial-time machines with auxiliary inputs). Under some reasonable assumptions there exist argument systems for $\mathcal{NP}$ having poly-logarithmic communication complexity. Analogous interactive proofs cannot exists unless $\mathcal{NP}$ is contained in Quasi-Polynomial Time (i.e., $\mathcal{NP} \subseteq \textsc{Dtime}(\exp(\text{poly}(\log n)))$).

CS Proofs: Here one only considers prover strategies implementable in time that is polynomial in the complexity of the language. In an non-interactive version one asks for "certificates of the NP-type" that are only computationally-sound. In a model allowing both prover and verifier access to a random oracle, one can convert interactive proofs (alike CS proofs) into non-interactive ones. As a heuristics, it was also suggested to replace the random oracle by use of "random public functions" (a fuzzy notion, not to be confused with pseudorandom functions).

Open Problem 4: Try to provide firm grounds for the heuristics of making proof systems non-interactive by use of "random public functions": I advise not to try to define the latter notion (in a general form), but rather devise some ad-hoc method, using some specific but widely believed complexity assumptions (e.g., hardness of deciding Quadratic Residucity modulo a composite number), for this specific application.[5]

12.3 Zero-Knowledge Proofs

Zero-knowledge proofs are central to cryptography. Furthermore, zero-knowledge proofs are very intriguing from a conceptual point of view, since they exhibit an extreme contrast between being

[4]We mention that QNR has a constant-round interactive proof in which the prover is a probabilistic polynomial-time prover with access to QNR. This proof system is similar to the one presented above for Graph Non-Isomorphism.

[5]The reasons for this recommendation are explained in The Random Oracle Methodology, Revisited (by Canetti, Goldreich and Halevi), in *Proc. of the 30th STOC*, pp. 209–218, 1998.

convinced of the validity of a statement and learning anything in addition while receiving such a convincing proof. Namely, zero-knowledge proofs have the remarkable property of being both convincing while yielding nothing to the verifier, beyond the fact that the statement is valid.

The zero-knowledge paradigm: Whatever can be efficiently computed after interacting with the prover on some common input, can be efficiently computed from this input alone (without interacting with anyone). That is, the interaction with the prover can be efficiently *simulated* in solitude.

A Technical Note: I have deviated from other presentation in which the simulator works in expected (probabilistic) polynomial-time and require that it works in strict probabilistic polynomial-time. Yet, I allow the simulator to halt without output with probability at most $\frac{1}{2}$. Clearly this implies an expected polynomial-time simulator, but the converse is not known. In particular, some known positive results regarding perfect zero-knowledge (with average polynomial-time simulators) are not known to hold under the above more strict notion.[6]

12.3.1 Perfect Zero-Knowledge

The Definition: A simulator can produce *exactly* the same distribution as occurring in an interaction with the prover. Furthermore, in the general definition this is required with respect to any probabilistic polynomial-time verifier strategy (not necessarily the one specified for the verifier). Thus, the zero-knowledge property protects the prover from any attempt to obtain anything from it (beyond conviction in the validity of the assertion).

Zero-Knowledge NP-proofs: Extending the NP-framework to interactive proof is essential for the non-triviality of zero-knowledge. It is easy to see that zero-knowledge NP-proofs exist only for languages in $\mathcal{RP}$. (Actually, that's a good exercise.)[7]

A perfect zero-knowledge proof for Graph Isomorphism: The prover sends the verifier a random isomorphic copy of the first input graph. The verifier challenges the prover by asking the prover to present an isomorphism (of graph sent) to either the first input graph or to the second input graph. The verifier's choice is made at random. The fact that this interactive proof system is zero-knowledge is more subtle than it seems; for example, (many) parallel repetitions of the proof system are unlikely to be zero-knowledge.

Statistical (or almost-perfect) Zero-Knowledge: Here the simulation is only required to be statistically close to the actual interaction. The resulting class, denoted $\mathcal{SZK}$, lies between Perfect ZK and general (or Computational) ZK. For further details see [78].

12.3.2 General (or Computational) Zero-Knowledge

This definition is obtained by substituting the requirement that the simulation is identical to the real interaction, by the requirement that the two are *computational indistinguishable*.

[6] See further details in strict polynomial-time in simulation and extraction (by Barak and Lindell), *34th STOC*, pages 484–493, 2002.

[7] An NP-proof system for a language L yields an NP-relation for L (defined using the verifier). On input $x \in L$ a perfect zero-knowledge simulator either halts without output or outputs an accepting conversation (i.e., an NP-witness for x).

Computational Indistinguishability is a fundamental concept of independent interest. Two ensembles are considered indistinguishable by an algorithm A if A's behavior is almost invariant of whether its input is taken from the first ensemble or from the second one. We interpret "behavior" as a binary verdict and require that the probability that A outputs 1 in both cases is the same up-to a negligible difference (i.e., smaller than $1/p(n)$, for any positive polynomial $p(\cdot)$ and all sufficiently long input lengths (denoted by n)). Two ensembles are computational indistinguishable if they are indistinguishable by all probabilistic polynomial-time algorithms.

A zero-knowledge proof for NP – an abstract (boxes) setting: It suffices to construct such a proof system for 3-Colorability (3COL). (To obtain a proof system for other NP-languages use the fact that the (standard) reduction of $\mathcal{NP}$ to 3COL is polynomial-time invertible.)
The prover uses a fixed 3-coloring of the input graph and proceeds as follows. First, it uniformly selects a relabeling of the colors (i.e., one of the 6 possible ones) and puts the resulting color of each vertex in a locked box (marked with the vertex name). All boxes are sent to the verifier who response with a uniformly chosen edge, asking to open the boxes corresponding to the endpoint of this edge. The prover sends over the corresponding keys, and the verifier opens the two boxes and accepts iff he sees two different legal colors.

A zero-knowledge proof for NP – the real setting: The locked boxes need to be implemented digitally. This is done by a *commitment scheme*, a cryptographic primitive designed to implement such locked boxes. Loosely speaking, a commitment scheme is a two-party protocol which proceeds in two phases so that at the end of the first phase (called the commit phase) the first party (called sender) is committed to a single value (which is the only value he can later reveal in the second phase), whereas at this point the other party gains no knowledge on the committed value. Commitment schemes exist if (and actually iff) one-way functions exist. Thus, the mildest of all cryptographic assumptions suffices for constructing zero-knowledge proofs for $\mathcal{NP}$ (and actually for all of $\mathcal{IP}$). That is:

Theorem 12.2 (The ZK Characterization Theorem): *If one-way functions exist then every set in $\mathcal{IP}$ has a zero-knowledge interactive proof system.*

Furthermore, zero-knowledge proofs for languages that are "hard on the average" imply the existence of one-way functions; thus, the above construction essentially utilizes the minimal possible assumption.

12.3.3 Concluding Remarks

The prover's strategy in the above zero-knowledge proof for NP can be implemented by a probabilistic polynomial-time machine which is given (as auxiliary input) an NP-witness for the input. (This is clear for 3COL, and for other NP-languages one needs to use the fact that the relevant reductions are coupled with efficient witness transformations.) The efficient implementation of the prover strategy is essential to the applications below.

Applications to Cryptography: Zero-knowledge proofs are a powerful tool for the design of cryptographic protocols, in which one typically wants to guarantee proper behavior of a party without asking him to reveal all his secrets. Note that proper behavior is typically a polynomial-time computation based on the party's secrets as well as on some known data. Thus, the claim that the party behaves consistently with its secrets and the known data can be casted as an NP-statement, and the above result can be utilized. More generally, using additional ideas, one can provide a secure protocol for any functional behavior. These general results have to be considered as plausibility arguments; you would not like to apply these general constructions to specific practical problems, yet you should know that these specific problems are solvable.

Open Problems do exists, but seem more specialized in nature. For example, it would be interesting to figure out and utilize the minimal possible assumption required for constructing "zero-knowledge protocols for NP" in various models like constant-round interactive proofs, the "non-interactive" model, and perfect zero-knowledge arguments.

Further Reading: See chapter on Zero-Knowledge in [27].

12.4 Probabilistically Checkable Proof (PCP) Systems

When viewed in terms of an interactive proof system, the probabilistically checkable proof setting consists of a prover that is memoryless. Namely, one can think of the prover as being an oracle and of the messages sent to it as being queries. A more appealing interpretation is to view the probabilistically checkable proof setting as an alternative way of generalizing $\mathcal{NP}$. Instead of receiving the entire proof and conducting a deterministic polynomial-time computation (as in the case of $\mathcal{NP}$), the verifier may toss coins and probe the proof only at location of its choice. Potentially, this allows the verifier to utilize very long proofs (i.e., of super-polynomial length) or alternatively examine very few bits of an NP-proof.

12.4.1 The Definition

The Basic Model: A probabilistically checkable proof 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. Completeness and soundness are defined similarly to the way they were defined for interactive proofs: for valid assertions there exist proofs making the verifier always accepts, whereas no oracle can make the verifier accept false assertions with probability above $\frac{1}{2}$. (We've specified the error probability since we intend to be very precise regarding some complexity measures.)

Additional complexity measures of fundamental importance are the *randomness* and *query* complexities. Specifically, $\mathcal{PCP}(r(\cdot), q(\cdot))$ denotes the set of languages having a probabilistic checkable proof system in which the verifier, on any input of length n, makes at most $r(n)$ coin tosses and at most $q(n)$ oracle queries. (As usual, unless stated otherwise, the oracle answers are always binary (i.e., either 0 or 1).)

Observed that the "effective" oracle length is at most $2^r \cdot q$ (i.e., locations that may be accessed on some random choices). In particular, the effective length of oracles in a $\mathcal{PCP}(\log, \cdot)$ system is polynomial. (Exercise: Show that $\mathcal{PCP}(\log, \text{poly}) \subseteq \mathcal{NP}$.)

PCP augments the traditional notion of a proof: An oracle that always makes the pcp-verifier accept constitutes a proof in the standard mathematical sense. However a pcp system has the extra property of enabling a lazy verifier, to toss coins, take its chances and "assess" the validity of the proof without reading all of it (but rather by reading a tiny portion of it).

12.4.2 The power of probabilistically checkable proofs

Theorem 12.3 (The PCP Characterization Theorem): $\mathcal{PCP}(\log, O(1)) = \mathcal{NP}$.

Thus, probabilistically checkable proofs in which the verifier tosses only logarithmically many coins and makes only a constant number of queries exist for every NP-language. It follows that NP-proofs can be transformed into NP-proofs which offer a trade-off between the portion of the proof being read and the confidence it offers. Specifically, if the verifier is willing to tolerate an error probability of ϵ then it suffices to let it examine $O(\log(1/\epsilon))$ bits of the (transformed) NP-proof. These bit locations need to be selected at random. Furthermore, an original NP-proof can be transformed into an NP-proof allowing such trade-off in polynomial-time. (The latter is an artifact of the proof of the PCP Theorem.)

The Proof of the PCP Characterization Theorem is one of the most complicated proofs in the Theory of Computation. Its main ingredients are:

1. A $\mathsf{pcp}(\log, \mathrm{poly}(\log))$ proof system for $\mathcal{NP}$. Furthermore, this proof system has additional properties which enable proof composition as in Item (3) below.

2. A $\mathsf{pcp}(\mathrm{poly}, O(1))$ proof system for $\mathcal{NP}$. This proof system also has additional properties enabling proof composition as in Item (3).

3. The proof composition paradigm: Suppose you have a $\mathsf{pcp}(r(\cdot), O(\ell(\cdot)))$ system for $\mathcal{NP}$ in which a constant number of queries are made (non-adaptively) to an 2^ℓ-valued oracle and the verifier's decision regarding the answers may be implemented by a $\mathrm{poly}(\ell)$-size circuit. Further suppose that you have a $\mathsf{pcp}(r'(\cdot), q(\cdot))$-like system for $\mathcal{P}$ in which the input is given in encoded form via an additional oracle so that the system accepts input-oracles that encode inputs in the language and reject any input-oracle which is "far" from the encoding of any input in the language. In this latter system access to the input-oracle is accounted in the query complexity. Furthermore, suppose that the latter system may handle inputs which result from concatenation of a constant number of sub-inputs each encoded in a separate sub-input oracle. Then, $\mathcal{NP}$ has a $\mathsf{pcp}(2(r(\cdot) + r'(s(\cdot))), 2q(s(\cdot)))$, where $s(n) \stackrel{\text{def}}{=} \mathrm{poly}(\ell(n))$. [The extra factor of 2 is an artifact of the need to amplify each of the two pcp systems so that the total error probability sums up to at most 1/2.]

In particular, the proof system of Item (1) is composed with itself [using $r = r' = \log$, $\ell = q = \mathrm{poly}(\log)$, and $s(n) = \mathrm{poly}(\log(n))$] yielding a $\mathsf{pcp}(\log, \mathrm{poly}(\log\log))$ system for $\mathcal{NP}$, which is then composed with the system of Item (2) [using $r = \log$, $\ell = \mathrm{poly}(\log\log)$, $r' = \mathrm{poly}$, $q = O(1)$, and $s(n) = \mathrm{poly}(\log\log(n))$] yielding the desired $\mathsf{pcp}(\log, O(1))$ system for $\mathcal{NP}$.

The $\mathsf{pcp}(\log, \mathrm{poly}(\log))$ system for $\mathcal{NP}$: We start with a different arithmetization of CNF formulae (than the one used for constructing an interactive proof for co$\mathcal{NP}$). Logarithmically many variables are used to represent (in binary) the names of variables and clauses in the input formula, and an oracle from variables to Boolean values is supposed to represent a satisfying assignment. An arithmetic expression involving a logarithmic number of summations is used to represent the value of the formula under the truth assignment represented by the oracle. This expression is a low-degree polynomial in the new variables and has a cubic dependency on the assignment-oracle. Small-biased probability spaces are used to generate a polynomial number of such expressions so that if the formula is satisfiable then all these expressions evaluate to zero, and otherwise at most half of them evaluate to zero. Using a summation test (as in the interactive proof for co$\mathcal{NP}$) and a low-degree test, this yields a $\mathsf{pcp}(t(\cdot), t(\cdot))$ system for $\mathcal{NP}$, where $t(n) \stackrel{\rm def}{=} O(\log(n) \cdot \log\log(n))$. [We use a finite field of poly$(\log(n))$ elements, and so we need $(\log n) \cdot O(\log\log n)$ random bits for the summation test.] To obtain the desired pcp system, one uses $\frac{O(\log n)}{\log\log n}$-long sequences over $\{1, ..., \log n\}$ to represent variable/clause names (rather than logarithmically-long binary sequences). [We can still use a finite field of poly$(\log(n))$ elements, and so we need only $\frac{O(\log n)}{\log\log n} \cdot O(\log\log n)$ random bits for the summation test.] All this is relatively easy compared to what is needed in order to transform the pcp system so that only a constant number of queries are made to a (multi-valued) oracle. This is obtained via (randomness-efficient) "parallelization" of pcp systems, which in turn depends heavily on efficient low-degree tests.

Open Problem 5: As a first step towards the simplification of the proof of the PCP Characterization Theorem, one may want to provide an alternative "parallelization" procedure that does not rely on polynomials or any other algebraic creatures.[8]

The $\mathsf{pcp}(\text{poly}, O(1))$ system for $\mathcal{NP}$: It suffices to prove the satisfiability of a systems of quadratic equations over GF(2), because this problem is NP-complete. The oracle is supposed to hold the values of all quadratic expressions under a satisfying assignment to the variables. We distinguish two tables in the oracle: One corresponding to the (2^n) linear expressions and the other to the $(2^{n^2}$ pure) bilinear expressions. Each table is tested for self-consistency (via a linearity test) and the two tables are tested to be consistent (via a matrix-equality test which utilizes "self-correction"). Each of these tests utilizes a constant number of Boolean queries, and randomness which is logarithmic in the size of the corresponding table.

12.4.3 PCP and Approximation

PCP-Characterizations of $\mathcal{NP}$ play a central role in recent developments concerning the difficulty of approximation problems. To demonstrate this relationship, we first note that the PCP Characterization Theorem can be rephrased without mentioning the class $\mathcal{PCP}$ altogether. Instead, a new type of polynomial-time reductions, which we call *amplifying*, emerges.

Amplifying reductions: There exists a constant $\epsilon > 0$, and a (polynomial-time) Karp-reduction f, of 3SAT to itself so that f maps non-satisfiable 3CNF formulae to 3CNF formulae for which every truth assignment satisfies at most a $1 - \epsilon$ fraction of the clauses. I call the reduction f *amplifying*, and its existence follows from the PCP Characterization Theorem. On the other hand, any amplifying reduction for $3SAT$ yields a proof of the PCP Characterization Theorem. (The proofs of both directions are left as an exercise.)[9]

[8] A first step towards this partial goal was taken in A Combinatorial Consistency Lemma with application to the PCP Theorem (by Goldreich and Safra), *SICOMP*, Volume 29, Number 4, pages 1132–1154, 1999.

[9] Hint: To prove the first direction, consider the guaranteed pcp system for 3SAT, associate the bits of the oracle with Boolean variables, and introduce a (constant size) Boolean formula for each possible outcome of the sequence of $O(\log n)$ coin tosses (describing whether the verifier would have accepted given this outcome). For the other direction, consider a pcp system that is given oracle access to a truth assignment for the formula resulting from the amplified reduction.

Amplifying reductions and Non-Approximability: The above amplifying reduction of 3SAT implies that it is NP-Hard to distinguish satisfiable 3CNF formulae from 3CNF formulae for which every truth assignment satisfies less than a $1-\epsilon$ fraction of its clauses. Thus, Max-3SAT is NP-Hard to approximate to within a $1-\epsilon$ factor.

Stronger Non-Approximability Results were obtained via alternative PCP Characterizations of NP. For example, the NP-Hardness of approximating Max-Clique to within $N^{1-\epsilon}$, $\forall \epsilon > 0$, was obtained via $\mathcal{NP} = \overline{\mathcal{FPCP}}(\log, \epsilon)$, where the second parameter in $\overline{\mathcal{FPCP}}$ measures the "amortized free-bit" complexity of the pcp system.

12.5 The actual notes that were used

I have focused on interactive proofs and probabilistically checkable proofs.

12.5.1 Interactive Proofs (IP)

Unfortunately, this part of my notes was lost. I have defined and discussed the basic model, examplified it with the Graph Non-Isomorphism protocol, and showed that co$\mathcal{NP} \subseteq \mathcal{IP}$.

12.5.2 Probabilistically Checkable Proofs (PCP)

Unfortunately, the first part of my notes (introducing the basic model and the complexity measures) was lost.

Adaptivity versus non-adaptivity in the context of PCP. Whenever one discusses oracle machines, there is a distinction between adaptive machines that may select their queries based on answers to prior queries and non-adaptive machines that determine all their queries as a function of their initial input (and coin tosses). Adaptive machines can always be converted to non-adaptive ones at the cost of an exponential increase in their query complexity (i.e., by considering a-priori all possible answers). In our case, where the query complexity is an unspecified constant, this difference is immaterial. Thus, whenever it is convenient, we will assume that the verifier (in the PCP scheme) is non-adaptive.

The PCP Characterization Theorem: The theorem states that $\mathcal{NP} = \mathcal{PCP}[O(\log n), O(1)]$. The easy direction consists of showing that $\mathcal{PCP}[O(\log n), O(1)]$ is contained in $\mathcal{NP}$. This follows by observing that the effective length of the oracle (i.e., the number of bits read from the oracle under all possible settings of the random-tape) is polynomial. The other direction is much more complex. Here we will only sketch a proof of a much easier result: that is, $\mathcal{NP} \subset \mathcal{PCP}[\text{poly}(n), O(1)]$. We stress that this result is very interesting by itself, because it states that NP-assertions can be verified probabilistically by making only a constant number of probes into the (possibly exponentially-long) proof.

Let QE_2 be the set of satisfiable systems of quadratic equations modulo 2 (i.e., quadratic equations over $GF(2)$). That is, $((c_{i,j}^{(k)})_{i,j\in[n]}, b^{(k)})_{k\in[m]}$ is in QE_2 if the quadratic system of equations $\{\sum_{i,j\in[n]} c_{i,j}^{(k)} x_i x_j = b^{(k)}\}_{k\in[m]}$ modulo 2 has a solution in the x_i's. Exercise: Prove that QE_2 is NP-complete, and that this holds also when $m = n$. (Also note that linear terms can be replaced by quadratic terms.) We will show that QE_2 is in $\mathcal{PCP}[O(n^2), O(1)]$. (Below, all arithmetic operations are modulo 2.)

The oracle in the PCP system that we will present is supposed to encode a satisfying assignment to the system of equations, where the encoding will be very redundant. As we will see redundant encodings may be very easy to check. Specifically, a satisfying assignment $\tau = (\tau_1, ..., \tau_n)$ will be encoded by providing all the partial sums of the τ_i's (i.e., an encoding of τ via the Hadamard code) as well as all the partial sums of the $\tau_i \tau_j$'s That is, the first (resp., second) part of the encoding of $\tau = (\tau_1, ..., \tau_n)$ is the 2^n-bit long string in which entry $\alpha \in \{0,1\}^n$ corresponds to $\sum_i \alpha_i \tau_i$ (resp., the 2^{n^2}-bit long string in which entry $\beta \in \{0,1\}^{n^2}$ corresponds to $\sum_{i,j} \beta_{i,j} \tau_i \tau_j$), where $\alpha = (\alpha_1, ..., \alpha_n)$ (resp., $\beta = \beta_{1,1}, \beta_{1,2}, ..., \beta_{n,n}$).

On input $((c_{i,j}^{(k)})_{i,j \in [n]}, b^{(k)})_{k \in [n]}$ and oracle access to $\Pi = (\Pi^{(1)}, \Pi^{(2)})$, where $|\Pi^{(i)}| = 2^{n^i}$, the verifier will perform the following four tests:

1. *Test that $\Pi^{(1)}$ is close to an encoding of some $\tau \in \{0,1\}^n$ under the Hadamard code*: That is, we check whether there exists a $\tau \in \{0,1\}^n$ such that

$$\Pr_\alpha \left[\Pi^{(1)}(\alpha) = \sum_i \alpha_i \tau_i \right] \; > \; 0.99 \tag{12.1}$$

   This checking is performed by invoking the so-called linearity test for a constant number of times where in each invocation we select uniformly and independently $\alpha', \alpha'' \in \{0,1\}^n$ and check whether $\Pi^{(1)}(\alpha') + \Pi^{(1)}(\alpha'') = \Pi^{(1)}(\alpha' + \alpha'')$. holds, where $\alpha' + \alpha''$ denote bit-by-bit addition. (Although very appealing, the analysis of the linearity test is quite non-trivial and thus omitted.)

2. *Test that $\Pi^{(2)}$ is close to an encoding of some $\sigma \in \{0,1\}^{n^2}$ under the Hadamard code*: That is, we check whether there exists a $\sigma \in \{0,1\}^{n^2}$ such that

$$\Pr_\beta \left[\Pi^{(2)}(\beta) = \sum_{i,j} \beta_{i,j} \sigma_{i,j} \right] \; > \; 0.99 \tag{12.2}$$

   Indeed, we just use the linearity test on $\Pi^{(2)}$.

3. *Test that the string encoded in $\Pi^{(1)}$ match the one encoded in $\Pi^{(2)}$*: Recall that the Hadamard code has relative distance equal to 1/2 (i.e., the encodings of two different strings agree in exactly 1/2 of the coordinates). Thus, Eq. (12.1) may hold only for one τ, and similarly Eq. (12.2) may hold only for one σ. In the current step we want to test whether the string τ that satisfies Eq. (12.1) is consistent with the string σ that satisfies Eq. (12.2); that is, that $\sigma_{i,j} = \tau_i \tau_j$ holds for all $i, j \in [n]$.

   A detour: Suppose we want to test that two n-by-n matrices, A and B are equal, by making few queries to a suitable encoding. This case be done by uniformly selecting a row vector r and a column vector s and checking whether $rAs = rBs$ (i.e., bit equality). Let $C = A - B$. We are actually checking whether C is all zeros by checking whether $rCs = 0$. Clearly, if C is all zeros then equality will always hold. On the other hand, if C is a non-zero matrix then it has rank $d \geq 1$ in which case the probability that (for a randomly chosen r) the vector rC is an all-zero vector is exactly 2^{-d}. (The proof is left as an exercise, but do the next exercise first.) Furthermore, for a non-zero vector $v = rC$, the probability that (for a randomly chosen s) it holds that $vs = 0$ is exactly 1/2. (Prove this too.) We conclude that for any non-zero matrix C, it holds that $\Pr_{r,s}[rCs = 0] \leq 3/4$.

Considering the matrices $A = (\tau_i\tau_j)_{i,j}$ and $B = (\sigma_{i,j})_{i,j}$, we want to check whether they are identical. By the above detour, this calls for uniformly selecting $r, s \in \{0,1\}^n$ and checking whether $rAs = rBs$. Now, observe that $rAs = (r\tau^\top)\tau s$ equals the product of $\sum_i r_i\tau_i$ and $\sum_i s_i\tau_i$. On the other hand, $rBs = \sum_{i,j} r_i s_j \sigma_{i,j}$. So it seems that all we need to check is whether $\Pi^{(1)}(r) \cdot \Pi^{(1)}(s)$ equals $\Pi^{(2)}(z)$, where z is the outer-product of r and s. This is not quite true. Steps 1 and 2 only guarantee that $\Pi^{(1)}(\alpha) = \sum_i \alpha_i\tau_i$ and $\Pi^{(2)}(\beta) = \sum_{i,j}\beta_{i,j}\sigma_{i,j}$ with high probability for uniformly distributed α and β. This is fine with respect to what we want to retrieve from $\Pi^{(1)}$, but not for what we want to retrieve from $\Pi^{(2)}$ (because the outer-product of r and s is not uniformly distributed even if r and s are uniformly distributed). Thus, instead of querying $\Pi^{(2)}$ on z, we uniformly select $z' \in \{0,1\}^{n^2}$, query $\Pi^{(2)}$ on z' and $z+z'$ (which are both uniformly distributed), and use the value $\Pi^{(2)}(z+z') - \Pi^{(2)}(z')$. This process is called self-correction.

4. *Test that the string encoded in* $\Pi^{(1)}$ *satisfies the quadratic system*: That is, for τ as in Eq. (12.1), we want to check whether $\sum_{i,j\in[n]} c_{i,j}^{(k)}\tau_i\tau_j = b^{(k)}$ holds for all $k \in [n]$. Rather than performing n tests (which we cannot afford), we uniformly select $r \in \{0,1\}^n$, and check whether
$$\sum_{k\in[n]} r_k \sum_{i,j\in[n]} c_{i,j}^{(k)}\tau_i\tau_j = \sum_{k\in[n]} r_k b^{(k)}$$
The left-hand side can be written as $\sum_{i,j}(\sum_k r_k c_{i,j}^{(k)})\tau_i\tau_j$, and so we merely need to retrieve that value, which by Steps 1–3 can be obtained, via self-correction, from $\Pi^{(2)}$. That is, assuming we did not reject in any of Steps 1–3, it holds that, with high probability over a uniformly chosen $\beta' \in \{0,1\}^{n^2}$, the value of $\sum_{i,j}\beta_{i,j}^{(k)}\tau_i\tau_j$ equals $\Pi^{(2)}(\beta+\beta') - \Pi^{(2)}(\beta')$, where we will set β such that $\beta_{i,j} = \sum_k r_k c_{i,j}$.

We conclude that if the original system of equations is not satisfiable then every $\Pi = (\Pi^{(1)}, \Pi^{(2)})$ is rejected with probability at least $1/2$ (by one of the above four steps), whereas the original system is satisfiable then there exists a $(\Pi^{(1)}, \Pi^{(2)})$ that is accepted with probability 1 (by all the above steps).

Amplifying reductions: For sake of concreteness, we focus on a specific NP-complete problem (i.e., 3SAT), but similar statements can be made about some other (but not all) natural NP-complete problems. We say that a Karp-reduction f is an amplifying reduction of 3SAT to itself if there exists a constant $\epsilon > 0$ such that the following holds:

- If $\phi \in 3SAT$ then $f(\phi) \in 3SAT$.
- If $\phi \notin 3SAT$ then (not only that $f(\phi) \notin 3SAT$ but rather) every truth assignment to $\phi' \stackrel{\rm def}{=} f(\phi)$ satisfies at most $1-\epsilon$ fraction of the clauses of ϕ'.

  That is, the reduction "amplifies" the unsatisfiability of ϕ (i.e., it may be that there exists a truth assignment that satisfies all but one of the clauses of ϕ, still all truth assignments fail to satisfy a constant fraction of the clauses of ϕ).

Interestingly, the notion of amplifying reductions captures the entire contents of the PCP Theorem (and so you should not expect to be able to see a simple amplifying reduction).

Theorem 12.4 *The following two are equivalent:*

1. $3SAT \in \mathcal{PCP}[O(\log n), O(1)]$.

2. *There exists an amplifying reduction of 3SAT to itself.*

Note that $3SAT \in \mathcal{PCP}[O(\log n), O(1)]$ if and only if $\mathcal{NP} \subseteq \mathcal{PCP}[O(\log n), O(1)]$.

Proof sketch: We first show that amplifying reductions imply the PCP Theorem. Suppose that f is an amplifying reduction of 3SAT to itself (and $\epsilon > 0$ be the corresponding constant). Consider a verifier that on input a 3CNF formula ϕ, computes $\phi' = f(\phi)$, selects at random a clause of ϕ', probe the oracle for the values of the corresponding three variables, and decide accordingly. This verifier uses a logarithmic amount of randomness, always accepts $\phi \in 3SAT$ (when provided an adequate oracle), and rejects each $\phi \notin 3SAT$ with probability at least ϵ (not matter what oracle is presented). Clearly, the error can be reduced to $1/2$ (as required) by invoking this verifier $1/\epsilon$ times.

On the other hand, given a PCP system as in Item 1, we construct an amplifying reduction as follows. On input a 3CNF formula ϕ, we construct a 3CNF formula ϕ' as follows. The variables of ϕ' will correspond to the bits of the oracle used by the PCP verifier. (Recall, that the number of effective oracle bits is polynomial in $|\phi|$.) For each possible random-tape $r \in \{0,1\}^{O(\log |\phi|)}$ of the verifier, consider the verifier's verdict as a function of the $O(1)$ answers obtained from the oracle. Thus, the verifier's decision on input ϕ and random-tape r can be represented as a constant-size formula in $O(1)$ variables (representing the corresponding oracle bits). Using auxiliary variables, such a formula can be represented in 3CNF (of constant size). The conjunction of these $2^{O(\log |\phi|)}$ formulae (each constructible in polynomial time from ϕ) yields ϕ'. Observe that if ϕ is satisfiable then so is ϕ'. On the other hand, if ϕ is not satisfiable then every truth assignment to the variables of ϕ' satisfies at most $1/2$ of the constant-size CNFs (which correspond to individual values of the random-tape). Thus, for each of at least $1/2$ of the constant-size CNFs, at least one of the clauses is not satisfied. It follows that the reduction constructed above is amplifying (with ϵ that depends on the constant number of clauses in each of the small CNFs). ■

Amplifying reductions and the difficulty of approximation: Max3SAT is typically defined as a search problem in which given a 3CNF formula, one seeks a truth assignment satisfying as many clauses as possible. In the $(1-\epsilon)$-approximation version, given a formula ϕ, one is only required to find a truth assignment that satisfies at least $(1-\epsilon) \cdot \mathrm{opt}(\phi)$ clauses, where $\mathrm{opt}(\phi)$ denotes the maximum number of clauses that can be satisfied by any truth assignment to ϕ. Observe that the existence on an amplifying reduction of 3SAT to itself, with constant ϵ, implies that the $(1-\epsilon)$-approximation version of Max3SAT is NP-hard. (Proving this fact is left as an exercise.)

Lecture 13

Pseudorandomness

A fresh view at the *question of randomness* was taken in the theory of computing: It has been postulated that a distribution is pseudorandom if it cannot be told apart from the uniform distribution by any efficient procedure. The paradigm, originally associating efficient procedures with polynomial-time algorithms, has been applied also with respect to a variety of limited classes of such distinguishing procedures.

Loosely speaking, pseudorandom generators are efficient procedures that stretch short random seeds into (significantly longer) pseudorandom sequences. Again, the original approach has required that the generation be done in polynomial-time, but subsequent works have demonstrated the fruitfulness of alternative requirements.

The notes for this lecture were adapted from various texts that I wrote in the past (see, e.g., [26, Chap. 3]). In view of the fact that that the archetypical case of pseudorandom generators is covered at Weizmann in the Foundation of Cryptography course, I focused in the current course on the derandomization aspect. The actual notes I have used in the current course appear in Section 13.6.

13.1 Introduction

The second half of this century has witnessed the development of three theories of randomness, a notion which has been puzzling thinkers for ages. The first theory (cf., [18]), initiated by Shannon [71], is rooted in probability theory and is focused at distributions that are not perfectly random. Shannon's Information Theory characterizes perfect randomness as the extreme case in which the *information content* is maximized (and there is no redundancy at all). Thus, perfect randomness is associated with a unique distribution – the uniform one. In particular, by definition, one cannot generate such perfect random strings from shorter random seeds.

The second theory (cf., [58, 60]), due to Solomonov [74], Kolmogorov [53] and Chaitin [15], is rooted in computability theory and specifically in the notion of a universal language (equiv., universal machine or computing device). It measures the complexity of objects in terms of the shortest program (for a fixed universal machine) that generates the object. Like Shannon's theory, Kolmogorov Complexity is quantitative and perfect random objects appear as an extreme case. Interestingly, in this approach one may say that a single object, rather than a distribution over objects, is perfectly random. Still, Kolmogorov's approach is inherently intractable (i.e., Kolmogorov Complexity is uncomputable), and – by definition – one cannot generate strings of high Kolmogorov Complexity from short random seeds.

The third theory, initiated by Blum, Goldwasser, Micali and Yao [34, 12, 81], is rooted in complexity theory and is the focus of this lecture. This approach is explicitly aimed at providing a notion of perfect randomness that allows to efficiently generate perfect random strings from shorter random seeds. The heart of this approach is the suggestion to view objects as equal if they cannot be told apart by any efficient procedure. Consequently, a distribution that cannot be efficiently distinguished from the uniform distribution will be considered as being random (or rather called pseudorandom). Thus, randomness is not an "inherent" property of objects (or distributions) but rather relative to an observer (and its computational abilities). To demonstrate this approach, let us consider the following mental experiment.

> Alice and Bob play "head or tail" in one of the following four ways. In all of them Alice flips a coin high in the air, and Bob is asked to guess its outcome *before* the coin hits the floor. The alternative ways differ by the knowledge Bob has before making his guess. In the first alternative, Bob has to announce his guess before Alice flips the coin. Clearly, in this case Bob wins with probability 1/2. In the second alternative, Bob has to announce his guess while the coin is spinning in the air. Although the outcome is *determined in principle* by the motion of the coin, Bob does not have accurate information on the motion and thus we believe that also in this case Bob wins with probability 1/2. The third alternative is similar to the second, except that Bob has at his disposal sophisticated equipment capable of providing accurate *information* on the coin's motion as well as on the environment effecting the outcome. However, Bob cannot process this information in time to improve his guess. In the fourth alternative, Bob's recording equipment is directly connected to a *powerful computer* programmed to solve the motion equations and output a prediction. It is conceivable that in such a case Bob can improve substantially his guess of the outcome of the coin.

We conclude that the randomness of an event is relative to the information and computing resources at our disposal. Thus, a natural concept of pseudorandomness arises – a distribution is *pseudorandom* if no efficient procedure can distinguish it from the uniform distribution, where efficient procedures are associated with (probabilistic) polynomial-time algorithms.

13.2 The General Paradigm

A generic formulation of *pseudorandom generators* consists of specifying three fundamental aspects: the *stretching measure* of the generators, the class of distinguishers that the generators are supposed to fool (i.e., the algorithms with respect to which the *computational indistinguishability* requirement should hold), and the resources that the generators are allowed to use (i.e., their own *computational complexity*).

Stretching function: A necessary requirement from any notion of a pseudorandom generator is that it is a deterministic algorithm that stretches short strings, called *seeds*, into longer output sequences. Specifically, it stretches k-bit long seeds into $\ell(k)$-bit long outputs, where $\ell(k) > k$. The function ℓ is called the *stretching measure* (or *stretching function*). In some settings the specific stretching measure is immaterial (e.g., see Section 13.3).

Computational Indistinguishability: A necessary requirement from any notion of a pseudorandom generator is that it "fools" some non-trivial algorithms. That is, any algorithm taken from some class of interest cannot distinguish the output produced by the generator (when the generator is fed with a uniformly chosen seed) from a uniformly chosen sequence. Typically, we consider a class $\mathcal{D}$ of distinguishers and a class $\mathcal{F}$ of noticeable functions, and require that the generator G satisfies the following: For any $D \in \mathcal{D}$, any $f \in \mathcal{F}$, and for all sufficiently large k's

$$|\Pr[D(G(U_k)) = 1] - \Pr[D(U_{\ell(k)}) = 1]| < f(k)$$

where U_n denotes the uniform distribution over $\{0,1\}^n$ and the probability is taken over U_k (resp., $U_{\ell(k)}$) as well as over the coin tosses of algorithm D in case it is probabilistic.[1] The archetypical choice is that $\mathcal{D}$ is the set of probabilistic polynomial-time algorithms, and $\mathcal{F}$ is the set of functions which are the reciprocal of some positive polynomial.

Complexity of Generation: The archetypical choice is that the generator has to work in polynomial-time (i.e., time that is polynomial in length of its input – the seed). Other choices will be discussed as well. We note that placing no computational requirements on the generator (or, alternatively, putting very mild requirements such as a double-exponential running-time upper bound), yields "generators" that can fool any subexponential-size circuit family.

13.3 The Archetypical Case

As stated above, the most natural notion of a pseudorandom generator refers to the case where both the generator and the potential distinguisher work in polynomial-time. Actually, the distinguisher is more complex than the generator: The generator is a fixed algorithm working within *some fixed* polynomial-time, whereas a potential distinguisher is *any* algorithm that runs in polynomial-time. Thus, for example, the distinguisher *may* always run in time cubic in the running-time of the generator. Furthermore, to facilitate the development of this theory, we allow the distinguisher to be probabilistic (whereas the generator remains deterministic as above). In the role of the set of noticeable functions we consider all functions that are the reciprocal of some positive polynomial.[2] This choice is naturally coupled with the association of efficient computation with polynomial-time algorithms: An event that occurs with noticeable probability occurs almost always when the experiment is repeated a "feasible" (i.e., polynomial) number of times.

13.3.1 The actual definition

The above discussion leads to the following instantiation of the generic framework presented in Section 13.2.

Definition 13.1 (pseudorandom generator – archetypical case): *A deterministic polynomial-time algorithm G is called a* pseudorandom generator *if there exists a* stretching function, $\ell : \mathbb{N} \to \mathbb{N}$, *so that for any probabilistic polynomial-time algorithm D, for any positive polynomial p, and for all sufficiently large k's*

$$|\Pr[D(G(U_k)) = 1] - \Pr[D(U_{\ell(k)}) = 1]| < \frac{1}{p(k)}$$

[1] Thus, we require certain functions (i.e., the absolute difference between the above probabilities), to be smaller than any noticeable function *on all but finitely many integers*. We call such functions negligible. Note that a function may be neither noticeable nor negligible (e.g., it may be smaller than any noticeable function on infinitely many values and yet larger than some noticeable function on infinitely many other values).

[2] The definition below asserts that the distinguishing gap of certain machines must be smaller than the reciprocal of any positive polynomial for all but finitely many n's. Such functions are called *negligible*. The notion of negligible probability is robust in the sense that an event that occurs with negligible probability occurs with negligible probability also when the experiment is repeated a "feasible" (i.e., polynomial) number of times.

where U_n denotes the uniform distribution over $\{0,1\}^n$ and the probability is taken over U_k (resp., $U_{\ell(k)}$) as well as over the coin tosses of D.

Thus, pseudorandom generators are efficient (i.e., polynomial-time) deterministic programs that expand short randomly selected seeds into longer pseudorandom bit sequences, where the latter are computationally indistinguishable from truly random sequences by efficient (i.e., polynomial-time) algorithms. It follows that any efficient randomized algorithm maintains its performance when its internal coin tosses are substituted by a sequence generated by a pseudorandom generator.

Amplifying the stretch function. Pseudorandom generators as defined above are only required to stretch their input a bit; for example, stretching k-bit long inputs to $(k+1)$-bit long outputs will do. Clearly generator of such moderate stretch function are of little use in practice. In contrast, we want to have pseudorandom generators with an arbitrary long stretch function. By the efficiency requirement, the stretch function can be at most polynomial. It turns out that pseudorandom generators with the smallest possible stretch function can be used to construct pseudorandom generators with any desirable polynomial stretch function. (Thus, when talking about the existence of pseudorandom generators, we may ignore the stretch function.)

Theorem 13.2 *Let G be a pseudorandom generator with stretch function $\ell(k) = k+1$, and ℓ' be any polynomially bounded stretch function, which is polynomial-time computable. Let $G_1(x)$ denote the $|x|$-bit long prefix of $G(x)$, and $G_2(x)$ denote the last bit of $G(x)$ (i.e., $G(x) = G_1(x)\,G_2(x)$). Then*

$$\begin{aligned} G'(s) \;&\stackrel{\text{def}}{=}\; \sigma_1\sigma_2\cdots\sigma_{\ell'(|s|)}\,, \\ &\quad \textit{where } x_0 = s,\ \sigma_i = G_2(x_{i-1}) \textit{ and } x_i = G_1(x_{i-1}),\ \textit{for } i = 1, ..., \ell'(|s|) \end{aligned}$$

is a pseudorandom generator with stretch function ℓ'.

Proof Sketch: The theorem is proven using the *hybrid technique* (cf., [27, Sec. 3.2.3]): One considers distributions H_k^i (for $i = 0, ..., \ell'(k)$) defined by $U_i^{(1)} P_{\ell'(k)-i}(U_k^{(2)})$, where $U_i^{(1)}$ and $U_k^{(2)}$ are independent uniform distributions (over $\{0,1\}^i$ and $\{0,1\}^k$, respectively), and $P_j(x)$ denotes the j-bit long prefix of $G'(x)$. The extreme hybrids correspond to $G'(U_k)$ and $U_{\ell'(k)}$, whereas distinguishability of neighboring hybrids can be worked into distinguishability of $G(U_k)$ and U_{k+1}. Loosely speaking, suppose one could distinguish H_k^i from H_k^{i+1}. Then, using $P_j(s) = G_2(s)P_{j-1}(G_1(s))$ (for $j \geq 1$), this means that one can distinguish $H_k^i \equiv (U_i^{(1)}, G_2(U_k^{(2)}), P_{(\ell'(k)-i)-1}(G_1(U_k^{(2)})))$ from $H_k^{i+1} \equiv (U_i^{(1)}, U_1^{(1')}, P_{\ell'(k)-(i+1)}(U_k^{(2')}))$. Incorporating the generation of $U_i^{(1)}$ and the evaluation of $P_{\ell'(k)-i-1}$ into the distinguisher, one could distinguish $(G_1(U_k^{(2)}), G_2(U_k^{(2)})) \equiv G(U_k)$ from $(U_k^{(2')}, U_1^{(1')}) \equiv U_{k+1}$, in contradiction to the pseudorandomness of G. (For details see [27, Sec. 3.3.2].) ■

13.3.2 How to Construct Pseudorandom Generators

The known constructions transform computation difficulty, in the form of one-way functions (defined below), into pseudorandomness generators. Loosely speaking, a *polynomial-time computable* function is called one-way if any efficient algorithm can invert it only with negligible success probability. For simplicity, we consider only length-preserving one-way functions.

Definition 13.3 (one-way function): *A* one-way function*, f, is a polynomial-time computable function such that for every probabilistic polynomial-time algorithm A', every positive polynomial $p(\cdot)$, and all sufficiently large n's*

$$\Pr_{x\sim U_n}\left[A'(f(x))\in f^{-1}(f(x))\right] \;<\; \frac{1}{p(n)}$$

where U_n denotes the uniform distribution over $\{0,1\}^n$, and $x\sim X$ means that x is distributed according to X.

Popular candidates for one-way functions are based on the conjectured intractability of Integer Factorization, the Discrete Logarithm Problem, and decoding of random linear code. The infeasibility of inverting f yields a weak notion of unpredictability: Let $b_i(x)$ denotes the i^{th} bit of x. Then, for every probabilistic polynomial-time algorithm A (and sufficiently large n), it must be the case that $\Pr_{i,x}[A(i,f(x)) \neq b_i(x)] > 1/2n$, where the probability is taken uniformly over $i \in \{1,...,n\}$ and $x \in \{0,1\}^n$. A stronger (and in fact strongest possible) notion of unpredictability is that of a hard-core predicate. Loosely speaking, a *polynomial-time computable* predicate b is called a hard-core of a function f if all efficient algorithm, given $f(x)$, can guess $b(x)$ only with success probability which is negligible better than half.

Definition 13.4 (hard-core predicate): *A polynomial-time computable predicate $b : \{0,1\}^* \mapsto \{0,1\}$ is called a* hard-core *of a function f if for every probabilistic polynomial-time algorithm A', every positive polynomial $p(\cdot)$, and all sufficiently large n's*

$$\Pr_{x\sim U_n}[A'(f(x))=b(x)] \;<\; \frac{1}{2}+\frac{1}{p(n)}$$

Clearly, if b is a hard-core of a 1-1 polynomial-time computable function f then f must be one-way.[3] It turns out that any one-way function can be slightly modified so that it has a hard-core predicate.

Theorem 13.5 (A generic hard-core): *Let f be an arbitrary one-way function, and let g be defined by $g(x,r) \stackrel{\text{def}}{=} (f(x),r)$, where $|x|=|r|$. Let $b(x,r)$ denote the inner-product mod 2 of the binary vectors x and r. Then the predicate b is a hard-core of the function g.*

See proof in [26, Apdx C.2] or [27, Sec. 2.5.2]). Finally, we get to the construction of pseudorandom generators:

Theorem 13.6 (A simple construction of pseudorandom generators): *Let b be a hard-core predicate of a polynomial-time computable 1-1 function f. Then, $G(s) \stackrel{\text{def}}{=} f(s)\,b(s)$ is a pseudorandom generator.*

Proof Sketch: Clearly the $|s|$-bit long prefix of $G(s)$ is uniformly distributed (since f is 1-1 and onto $\{0,1\}^{|s|}$). Hence, the proof boils down to showing that distinguishing $f(s)b(s)$ from $f(s)\sigma$, where σ is a random bit, yields contradiction to the hypothesis that b is a hard-core of f (i.e., that $b(s)$ is *unpredictable* from $f(s)$). Intuitively, such a distinguisher also distinguishes $f(s)b(s)$ from $f(s)\overline{b(s)}$, where $\overline{\sigma} = 1-\sigma$, and so yields an algorithm for predicting $b(s)$ based on $f(s)$. ■

In a sense, the key point in the proof of the above theorem is showing that the (obvious by constriction) unpredictability of the output of G implies its pseudorandomness. The fact that (next bit) unpredictability and pseudorandomness are equivalent in general is proven explicitly in [27, Sec. 3.3.5].

[3]Functions that are not 1-1 may have hard-core predicates of information theoretic nature; but these are of no use to us here. For example, for $\sigma \in \{0,1\}$, the function $f(\sigma,x) = 0f'(x)$ has an "information theoretic" hard-core predicate $b(\sigma,x) = \sigma$.

A general condition for the existence of pseudorandom generators. Recall that given any one-way 1-1 function, we can easily construct a pseudorandom generator. Actually, the 1-1 requirement may be dropped, but the currently known construction – for the general case – is quite complex. Still we do have.

Theorem 13.7 (On the existence of pseudorandom generators): *Pseudorandom generators exist if and only if one-way functions exist.*

To show that the existence of pseudorandom generators imply the existence of one-way functions, consider a pseudorandom generator G with stretch function $\ell(k) = 2k$. For $x, y \in \{0,1\}^k$, define $f(x,y) \stackrel{\text{def}}{=} G(x)$, and so f is polynomial-time computable (and length-preserving). It must be that f is one-way, or else one can distinguish $G(U_k)$ from U_{2k} by trying to invert f and checking that the result is correct: Inverting f on its range distribution refers to experimenting with the distribution $G(U_k)$, whereas the probability that U_{2k} has an inverse under f is negligible.

The interesting direction is the construction of pseudorandom generators based on any one-way function. In general (when f may not be 1-1) the ensemble $f(U_k)$ may not be pseudorandom, and so Construction 13.6 (i.e., $G(s) = f(s)b(s)$, where b is a hard-core of f) cannot be used *directly*. Thus, one idea is to hash $f(U_k)$ to an almost uniform string of length related to its entropy, using Universal Hash Functions [14]. (This is done after guaranteeing, that the logarithm of the probability mass of a value of $f(U_k)$ is typically close to the entropy of $f(U_k)$.)[4] But "hashing $f(U_k)$ down to length comparable to the entropy" means shrinking the length of the output to, say, $k' < k$. This foils the entire point of stretching the k-bit seed. Thus, a second idea is to compensate for the $k - k'$ loss by extracting these many bits from the seed U_k itself. This is done by hashing U_k, and the point is that the $(k - k' + 1)$-bit long hash value does not make the inverting task any easier. Implementing these ideas turns out to be more difficult than it seems, and indeed an alternative construction would be most appreciated.

13.3.3 Pseudorandom Functions

Pseudorandom generators allow to efficiently generate long pseudorandom sequences from short random seeds. Pseudorandom functions (defined below) are even more powerful: They allow efficient direct access to a huge pseudorandom sequence (which is infeasible to scan bit-by-bit). Put 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 efficient machines that may obtain the function values at arguments of their choice. (Such machines are called oracle machines, and if M is such machine and f is a function, then $M^f(x)$ denotes the computation of M on input x when M's queries are answered by the function f.)

Definition 13.8 (pseudorandom functions): *A* pseudorandom function (ensemble)*, with length parameters* $\ell_{\mathrm{D}}, \ell_{\mathrm{R}} : \mathsf{N} \to \mathsf{N}$*, is a collection of functions* $F \stackrel{\text{def}}{=} \{f_s : \{0,1\}^{\ell_{\mathrm{D}}(|s|)} \mapsto \{0,1\}^{\ell_{\mathrm{R}}(|s|)}\}_{s \in \{0,1\}^*}$ *satisfying*

- (efficient evaluation): *There exists an efficient* (deterministic) *algorithm which given a* seed, *s, and an $\ell_{\mathrm{D}}(|s|)$-bit* argument*, x, returns the $\ell_{\mathrm{R}}(|s|)$-bit long value $f_s(x)$.*

[4] Specifically, given an arbitrary one way function f', one first constructs f by taking a "direct product" of sufficiently many copies of f'. For example, for $x_1, ..., x_{k^2} \in \{0,1\}^k$, we let $f(x_1, ..., x_{k^2}) \stackrel{\text{def}}{=} f'(x_1), ..., f'(x_{k^2})$.

- (pseudorandomness): *For every probabilistic polynomial-time oracle machine, M, for every positive polynomial p and all sufficiently large n's*

$$\left|\Pr_{f\sim F_n}[M^f(1^n)=1]-\Pr_{\rho\sim R_n}[M^\rho(1^n)=1]\right| < \frac{1}{p(n)}$$

where F_n denotes the distribution on F obtained by selecting s uniformly in $\{0,1\}^n$, and R_n denotes the uniform distribution over all functions mapping $\{0,1\}^{\ell_{\mathrm{D}}(n)}$ to $\{0,1\}^{\ell_{\mathrm{R}}(n)}$.

Suppose, for simplicity, that $\ell_{\mathrm{D}}(n)=n$ and $\ell_{\mathrm{R}}(n)=1$. Then a function uniformly selected among 2^n functions (of a pseudorandom ensemble) presents an input-output behavior which is indistinguishable in poly(n)-time from the one of a function selected at random among all the 2^{2^n} Boolean functions. Contrast this with the 2^n pseudorandom sequences, produced by a pseudorandom generator, which are computationally indistinguishable from a sequence selected uniformly among all the $2^{\mathrm{poly}(n)}$ many sequences. Still pseudorandom functions can be constructed from any pseudorandom generator.

Theorem 13.9 (How to construct pseudorandom functions): *Let G be a pseudorandom generator with stretching function $\ell(n)=2n$. Let $G_0(s)$ (resp., $G_1(s)$) denote the first (resp., last) $|s|$ bits in $G(s)$, and*

$$G_{\sigma_{|s|}\cdots\sigma_2\sigma_1}(s) \stackrel{\mathrm{def}}{=} G_{\sigma_{|s|}}(\cdots G_{\sigma_2}(G_{\sigma_1}(s))\cdots)$$

Then, the function ensemble $\{f_s:\{0,1\}^{|s|}\mapsto\{0,1\}^{|s|}\}_{s\in\{0,1\}^*}$, where $f_s(x)\stackrel{\mathrm{def}}{=}G_x(s)$, is pseudorandom with length parameters $\ell_{\mathrm{D}}(n)=\ell_{\mathrm{R}}(n)=n$.

The above construction can be easily adapted to any (polynomially-bounded) length parameters $\ell_{\mathrm{D}},\ell_{\mathrm{R}}:\mathsf{N}\to\mathsf{N}$.

Proof Sketch: The proof uses the hybrid technique: The i^{th} hybrid, H_n^i, is a function ensemble consisting of $2^{2^i\cdot n}$ functions $\{0,1\}^n\mapsto\{0,1\}^n$, each defined by 2^i random n-bit strings, denoted $\langle s_\alpha\rangle_{\alpha\in\{0,1\}^i}$. The value of such function at $x=\beta\alpha$, with $|\alpha|=i$, is $G_\beta(s_\alpha)$. The extreme hybrids correspond to our indistinguishability claim (i.e., $H_n^0\equiv f_{U_n}$ and $H_n^n\equiv R_n$), and neighboring hybrids correspond to our indistinguishability hypothesis (specifically, to the indistinguishability of $G(U_n)$ and U_{2n} under multiple samples). ■

13.3.4 The Applicability of Pseudorandom Generators

Randomness is playing an increasingly important role in computation: It is frequently used in the design of sequential, parallel and distributed algorithms, and is of course central to cryptography. Whereas it is convenient to design such algorithms making free use of randomness, it is also desirable to minimize the usage of randomness in real implementations. Thus, pseudorandom generators (as defined above) are a key ingredient in an "algorithmic tool-box" – they provide an automatic compiler of programs written with free usage of randomness into programs which make an economical use of randomness. In the context of complexity theory, this yields results of the following type.

Theorem 13.10 (Derandomization of $\mathcal{BPP}$): *If there exists non-uniformly strong pseudorandom generators then $\mathcal{BPP}$ is contained in $\cap_{\epsilon>0}\mathrm{D{\small TIME}}(t_\epsilon)$, where $t_\epsilon(n)\stackrel{\mathrm{def}}{=}2^{n^\epsilon}$.*

Proof Sketch: Given any $L \in \mathcal{BPP}$ and any $\epsilon > 0$, we let A denote the decision procedure for L and G denote a pseudorandom generator stretching n^ϵ-bit long seeds into poly(n)-long sequences (to be used by A on input length n). Combining A and G, we obtain an algorithm $A' = A_G$ that, on input x, first produces a poly$(|x|)$-long sequence by applying G to a uniformly selected $|x|^\epsilon$-bit long string, and next runs A using the resulting sequence as a random-tape. We note that A and A' may differ in their decision on at most finitely many inputs (or else we can incorporate such inputs, together with A, into a family of polynomial-size circuits which distinguishes $G(U_{n^\epsilon})$ from $U_{\text{poly}(n)}$). Incorporating these finitely many inputs into A', and more importantly – emulating A' on each of the 2^{n^ϵ} possible random choices (i.e., seeds to G), we obtain a deterministic algorithm A'' as required. ■

We comment that stronger results regarding derandomization of $\mathcal{BPP}$ are presented in Section 13.4.

Comment: Indeed, "pseudo-random number generators" have appeared with the first computers. However, typical implementations use generators which are not pseudorandom according to the above definition. Instead, at best, these generators are shown to pass SOME ad-hoc statistical test (cf., [52]). However, the fact that a "pseudo-random number generator" passes some statistical tests, does not mean that it will pass a new test and that it is good for a future (untested) application. Furthermore, the approach of subjecting the generator to some ad-hoc tests fails to provide general results of the type stated above (i.e., of the form "for ALL practical purposes using the output of the generator is as good as using truly unbiased coin tosses"). In contrast, the approach encompassed in Definition 13.1 aims at such generality, and in fact is tailored to obtain it: The notion of computational indistinguishability, which underlines Definition 13.1, covers all possible efficient applications postulating that for all of them pseudorandom sequences are as good as truly random ones.

13.3.5 The Intellectual Contents of Pseudorandom Generators

We shortly discuss some intellectual aspects of pseudorandom generators as defined above.

Behavioristic versus Ontological. Our definition of pseudorandom generators is based on the notion of computational indistinguishability. The behavioristic nature of the latter notion is best demonstrated by confronting it with the Kolmogorov-Chaitin approach to randomness. Loosely speaking, a string is *Kolmogorov-random* if its length equals the length of the shortest program producing it. This shortest program may be considered the "true explanation" to the phenomenon described by the string. A Kolmogorov-random string is thus a string which does not have a substantially simpler (i.e., shorter) explanation than itself. Considering the simplest explanation of a phenomenon may be viewed as an ontological approach. In contrast, considering the effect of phenomena (on an observer), as underlying the definition of pseudorandomness, is a behavioristic approach. Furthermore, there exist probability distributions which are not uniform (and are not even statistically close to a uniform distribution) that nevertheless are indistinguishable from a uniform distribution by any efficient method. Thus, distributions which are ontologically very different, are considered equivalent by the behavioristic point of view taken in the definitions above.

A relativistic view of randomness. Pseudorandomness is defined above in terms of its observer. It is a distribution which cannot be told apart from a uniform distribution by any efficient (i.e. polynomial-time) observer. However, pseudorandom sequences may be distinguished from random ones by infinitely powerful powerful (not at our disposal!). Specifically, an exponential-time machine can easily distinguish the output of a pseudorandom generator from a uniformly selected string of the same length (e.g., just by trying all possible seeds). Thus, pseudorandomness is subjective to the abilities of the observer.

Randomness and Computational Difficulty. Pseudorandomness and computational difficulty play dual roles: The definition of pseudorandomness relies on the fact that putting computational restrictions on the observer gives rise to distributions which are not uniform and still cannot be distinguished from uniform. Furthermore, the construction of pseudorandom generators rely on conjectures regarding computational difficulty (i.e., the existence of one-way functions), and this is inevitable: given a pseudorandom generator, we can construct one-way functions. Thus, (non-trivial) pseudorandomness and computational hardness can be converted back and forth.

13.4 Derandomization of BPP

The above discussion has focused mainly on one aspect of the pseudorandomness question: the resources or type of the observer (or potential distinguisher). Another important question is whether such pseudorandom sequences can be generated from much shorter ones, and at what cost (or complexity). So far, we have required the generation process to be at least as efficient as the efficiency limitations of the distinguisher.[5] Indeed, this seems "fair" and natural. Allowing the generator to be more complex (i.e., use more time or space resources) than the distinguisher seems unfair, but still yields interesting consequences in the context of trying to "de-randomize" randomized complexity classes. For example, as we shall see, one may benefit from considering generators that work in time exponential in the length of their seed.

In the context of derandomization, we typically lose nothing by (being more liberal and) allowing exponential-time generators. To see why, we consider a typical derandomization argument, proceeding in two steps (cf. the proof of Theorem 13.10): First one replaces the true randomness of the algorithm by pseudorandom sequences generated from much shorter seeds, and next one deterministically scans all possible seeds and looks for the most frequent behavior of the (modified) algorithm. Thus, in such a case, the deterministic complexity is anyhow exponential in the seed length. The question is whether we gain anything by allowing exponential-time generators. The answer seems to be positive, because with more time at their disposal the generators can perform better (e.g., output longer sequences and/or be based on weaker intractability assumptions). For example:

Theorem 13.11 *Let $\mathcal{E} \stackrel{\rm def}{=} \cup_c \mathrm{D{\scriptsize TIME}}(t_c)$, with $t_c(n) = 2^{cn}$. Suppose that there exists a language $L \in \mathcal{E}$ and a constant $\epsilon > 0$ such that, for all but finitely many n's, any circuit C_n which correctly decides L on $\{0,1\}^n$ has size at least $2^{\epsilon n}$. Then, $\mathcal{BPP} = \mathcal{P}$.*

Indeed, Theorem 13.11 is related to Theorem 13.10, but the pseudorandom generators underlying their proofs are very different.

Proof Sketch: Underlying the proof is a construction of an adequate pseudorandom generator. This generator operates in exponential-time, and generates an exponentially long output that fools

[5]If fact, we have require the generator to be more efficient than the distinguisher: The former was required to be a fixed polynomial-time algorithm, whereas the latter was allowed to be any algorithm with polynomial running time.

circuits of size that is a *fixed polynomial* in the length of the output (or a smaller exponential in the seed length). That is, for some constant $b > 0$ and all k's, the generator (running in time $2^{O(k)}$) stretches k-bit seeds into sequences of length 2^{bk} that cannot be distinguished from truly random sequences by any circuit of size 2^{bk}. (Note that $b < 1$, because a 2^k-time machine can easily distinguish the generated sequences from random ones, by trying all possible k-bit seeds.) The derandomization of $\mathcal{BPP}$ proceeds by setting the seed-length to be logarithmic in the input length, and utilizing the above generator.

> Specifically, let A be a randomized $p(\cdot)$-time algorithm that we wish to derandomize. On input x, we set $k \stackrel{\text{def}}{=} (1/b) \cdot \log_2 p(|x|) = O(\log |x|)$, and scan all possible k-bit seeds. For each seed, we produce the corresponding 2^{bk}-bit sequence, use it as a random-tape to A (invoked on input x), and record the output of A. (Each such invocation takes time $2^{O(k)} + p(|x|) = \text{poly}(|x|)$, and we have $2^k = \text{poly}(|x|)$ many invocations.) We output the most frequent output obtained in all 2^k invocations of $A(x)$.

We now turn to the construction of the generator. The construction utilizes a predicate computable in exponential-time but unpredictable, even to within a particular exponential advantage, by any circuit family of a particular exponential size.[6] (One main ingredient of the proof is supplying such a predicate, given the hypothesis, but we omit this part here.) Given such a predicate the generator works by evaluating the predicate on exponentially-many subsequences of the bits of the seed so that the intersection of any two subsets is relatively small. That is, for $\epsilon > 0$ as in the hypothesis and $\delta, b = \text{poly}(\epsilon)$, given a k-bit seed, the generator constructs (in $2^{O(k)}$-time) 2^{bk} subsets of $[k] \stackrel{\text{def}}{=} \{1, ..., k\}$ each of size δk such that the intersection of every two sets has size at most $2\delta^2 k$, and evaluates the predicate on the projection of the seed bits determined by each of these subsets.[7]

The above generator fools circuits of the stated size, even when these circuits are presented with the seed as auxiliary input. (These circuits are smaller than the running time of the generator and so they cannot just evaluate the generator on the given seed.) The proof that the generator fools such circuits refers to the characterization of pseudorandom sequences as unpredictable ones. Thus, one proves that the next bit in the generator's output cannot be predicted given all previous bits (as well as the seed). Assuming that a small circuit can predict the next bit (of the generator), we construct a circuit for predicting the hard predicate. The new circuit incorporates the best (for such prediction) augmentation of the input to the circuit into a seed for the generator (i.e., the bits not in the specific subset of the seed are fixed in the best way). The key observation is that all other bits in the output of the generator depend only on a small fraction of the input bits (i.e., recall the small intersection clause above), and so circuits for computing these other bits have relatively small size (and so can be incorporated in the new circuit). Using all these circuits, the new circuit forms the adequate input for the next-bit predicting circuit, and outputs whatever the latter circuit does.

> Specifically, using a circuit C for predicting the $i + 1^{\text{st}}$ bit of the generator (invoked on k-bit seeds), we describe a circuit for approximating the value of the predicate on inputs of length $\ell \stackrel{\text{def}}{=} \delta k$. Recall that C is given the first i bits output by the generator as well

[6]For future reference, say that for some constant $\epsilon' > 0$, no circuit of size $2^{\epsilon' \ell}$ can guess the value of the predicate on a random ℓ-bit input with success probability higher than $2^{-\epsilon' \ell}$.

[7]Thus, this generator is only "moderately more complex" than the distinguisher: Viewed in terms of its output, the generator works in time polynomial in the length of the output, whereas the output fools circuits of size which is a (smaller) polynomial in the length of the output.

as the (k-bit) seed, and predicts the said bit with advantage (say) 2^{-2bk}. We first fix the best setting (for C's prediction) of the seed bits that are not in the $i+1^{\text{st}}$ subset. (Certainly, C's prediction for a random setting of the bits of the $i+1^{\text{st}}$ subset and a best best setting of the rest is at least as good as its prediction on a random seed.) Next, for each of the first i bits (in the generator's output), we consider circuits for computing the value of these bits as a function of the undetermined seed bits (of the $i+1^{\text{st}}$ subset) and the fixed bits of the rest of the seed. Since the number of undetermined bits is at most $2\delta^2 k$, each such circuit has size $2^{2\delta^2 k}$. Incorporating these $i < 2^{bk}$ circuits into C, we obtain a circuit that predicts the $i+1^{\text{st}}$ output bit when only given the bits of the $i+1^{\text{st}}$ subset. In other words, the resulting circuit approximates the predicate on random inputs of length δk with correlation at least $2^{-2bk} > 2^{-\epsilon' \cdot \delta k}$ (for ϵ' as in Footnote 6). The size of the resulting circuit is at most $2^{bk} \cdot 2^{2\delta^2 k} + \text{size}(C) < 2^{\epsilon' \cdot \delta k}$. This contradicts the hypothesis regarding the predicate.

Recall that we have only showed how to use a predicate that is hard to approximate in order to obtain the desired pseudorandom generator. To complete the proof sketch, one has to show how the existence of predicates (in $\mathcal{E}$) that are hard in the (adequate) worst-case sense implies the existence of predicates (in $\mathcal{E}$) that are hard to approximate (in the adequate sense). This part is too complex to be treated here, and the interested reader is referred to [45]. ∎

13.5 On weaker notions of computational indistinguishability

Whenever the aim is to replace random sequences utilized by an algorithm with pseudorandom sequences, one may try to capitalize on knowledge of the target algorithm. Above we have merely used the fact that the target algorithm runs in polynomial-time. However, for example, if we know that the algorithm uses very little work-space then we may be able to do better. The same holds if we know that the analysis of the algorithm only depends on some specific properties of the random sequence it uses (e.g., pairwise independence of its elements). In general, weaker notions of computational indistinguishability such as fooling space-bounded algorithms, constant-depth circuits, and even specific tests (e.g., testing pairwise independence of the sequence), arise naturally: Generators producing sequences that fool such tests are useful in a variety of applications – if the application utilizes randomness in a restricted way then feeding it with sequences of low randomness-quality may do. Needless to say, we advocate a rigorous formulation of the characteristics of such applications and a rigorous construction of generators that fool the type of tests that emerge.

In the context of a course on complexity theory, it is most appropriate to mention the pseudorandom generators that fool space-bounded algorithms that have on-line access to the inspected sequence (which is analogous to the on-line access of randomized bounded-space machines to their random-tape). Such generators can be constructed without relying on any intractability assumptions, and yield strong derandomization results. Two such famous results are captured by the following theorems.

Theorem 13.12 *$\mathcal{BPL} \subseteq \mathcal{SL}$, where $\mathcal{BPL} \supseteq \mathcal{RL}$ in the class of sets recognized by two-sided log-space machines, and $\mathcal{SL}$ in the class of sets recognized by deterministic polynomial-time algorithms that use only poly-logarithmic amount of space.*

Theorem 13.13 *Suppose that L can be decided by a probabilistic polynomial-time algorithm of space complexity s. Then L can be decided by a probabilistic polynomial-time algorithm of space complexity $O(s)$ and randomness complexity $O(s')$, where $s'(n) = \max(s(n), n)$.*

Analogous results hold for search problems. The pseudorandom generator underlying Theorem 13.12 uses a logarithmic number of hashing functions (each having logarithmic description length) and a logarithmically-long string to define a polynomially-long sequence. The seed of the generator consists of the description of the hash functions and the additional string, but for a fixed log-space distinguisher one can determine a sequence of hashing functions for which the distinguisher is fooled (when on only varies the additional logarithmically-long string). The pseudorandom generator underlying Theorem 13.13 uses a "randomness extractor", which is a more sophisticated construct (which has been the focus on extensive research in the recent decade; see [69]).

13.6 The actual notes that were used

The general paradigm of pseudorandom generators refers to a *deterministic* program that *stretches* random seeds into longer sequences that *look random to a specified set of resource-bounded observers.* Thus, pseudorandomness is not generated deterministically, but rather from a short random seed, and the above formalism is aimed to make explicit the (relatively small) amount of randomness used in the generation process. That is, we refer to a deterministic function (or family of functions one per each value of k) of the form $G : \{0,1\}^k \to \{0,1\}^{\ell(k)}$, satisfying three properties:

1. *Stretching*: At the very least $\ell(k) > k$ for every k.

2. *Pseudorandomness*: For every observer D taken from an adequate class (which depends on the setting), D cannot distinguish a random output of G from a truly random string of the same length. That is,

$$\Pr_{s\in\{0,1\}^k}[D(G(s)) = 1] \approx \Pr_{r\in\{0,1\}^{\ell(k)}}[D(r) = 1] \tag{13.1}$$

   That is, D as a potential distinguisher, fails to do its job in a very strong sense. Throughout this lecture, we will focus on potential distinguishers that are implementable by polynomial-size circuits (i.e., a non-uniform family of circuits of size polynomial in the length of the input (i.e., $\ell(k)$)).[8]

3. *The complexity of generation*: This will vary from setting to setting. We mention two important cases:

   (a) *The archetypical case*: The natural requirement is that G be a polynomial-time algorithm. Using such a generator allows to shrink the amount of randomness used in any (efficient) application. Note that in this case, the stretch ℓ is polynomially-bounded, and so the shrinkage obtained is ℓ^{-1} (i.e., if the original application used m random bits then we can typically modify it to use only $\ell^{-1}(m)$ random bits).

   We stress that in this case the distinguisher, which may use any probabilistic polynomial-time procedure, is more complex than the generator, which has running-time equal a specific (fixed) polynomial.

   (b) *The case of derandomization*: As we'll see below, in the context of derandomization, we will anyhow scan all possible seeds. Thus, derandomization is always exponential in the seed, and so we gain nothing by requiring that the generation process (i.e., G) is

[8] Indeed, such a distinguisher may incorporate the output of G on a specific seed (or on a few seeds), but the probability that this seed will be chosen for the left-hand-side of Eq. (13.1) is negligible.

polynomial-time. Instead, we may allow G to work for exponential time (i.e., time that is exponential in the seed length). In this case, the stretch ℓ is only exponentially-bounded. We stress that in this case the generator may be more complex than the distinguisher. Specifically, whereas the generator is allowed time exponential in the seed length, this cannot be possibly allowed for the distinguisher (or else the latter may try all seeds and apply the generator to each such seed).

We stress that the archetypical case yields a *general-purpose* generator that can be used in *any application*. In particular, it yields a compiler for saving randomness in any probabilistic polynomial-time algorithm and is the type of thing needed in cryptography (where the adversary/distinguisher may be more complex than the legitimate strategy that uses the generator). In contrast, the type of generators used in case of derandomization are sometimes good only with respect to the specific algorithm being derandomized (or a specific resource bound).

To clarify the above, let us spell out how one typically uses a pseudorandom generator. Let A be a probabilistic polynomial-time algorithm, say running in time n^3 (where n denote its input length). Let G be a pseudorandom generator of the first type (i.e., G is polynomial-time computable), say, with stretch function $\ell(k) = k^5$. We derive a new algorithm A' by replacing the randomness of A with randomness generated out of a random seed of G. That is, let $A(x,r)$ denote the output of A on input x and randomness $r \in \{0,1\}^{|x|^3}$ (recall that $A(x)$ makes at most $|x|^3$ steps). Then on input x and randomness $s \in \{0,1\}^{|x|^{3/5}}$, algorithm A' computes $G(s)$ and outputs $A(x, G(s))$. Note that A' runs in polynomial-time because so do A and G. We claim that A' performs as well as A, while using significantly less random bits. The proof is left as an exercise (hint: use the fact that inputs on which A' differs significantly from A can be hard-wired into a distinguishing circuit). Note that using an adequate pseudorandom generator we can shrink the amount of randomness used by any probabilistic polynomial-time algorithm to n^ϵ, for any constant $\epsilon > 0$.

So far we have only shrinked the amount of randomness used by probabilistic polynomial-time algorithms. Full derandomization is obtained by scanning all possible random-tapes used by the resulting algorithm (or in other words scanning all possible seeds for the generator). That is, given A' as above, we derive a deterministic algorithm A'' by scanning all possible s's and outputting, on input x, the majority value of $A'(x,s)$ (taken over all relevant s's). If we use a generator G of running time t_G and stretch $\ell(k) \leq t_G(k)$, then the running-time of A'' on input an n-bit string will be

$$2^{\ell^{-1}(n)} \cdot \left(t_G(\ell^{-1}(n)) + \text{time}_A(n)\right)$$

For ℓ that is exponential (i.e., $\ell(k) = 2^{\Omega(k)}$), whenever A is polynomial-time and G is exponential-time (i.e., $t_G(k) = 2^{O(k)}$), we obtain a polynomial-time algorithm A'', because $\ell^{-1}(n) = O(\log n)$ and $t_G(\ell^{-1}(n)) = 2^{O(\ell^{-1}(n))} = \text{poly}(n)$. Let use take a closer look at what we need in order to obtain such a result. We need a generator ($G : \{0,1\}^k \to \{0,1\}^{\ell(k)}$) that (1) runs in at most exponential-time (i.e., $t_G(k) = 2^{O(k)}$), and (2) stretches its seed by an exponential amount (i.e., $\ell(k) = 2^{\Omega(k)}$), such that (3) these outputs are indistinguishable from random $\ell(k)$-bit long sequences by circuits of size, say, $\ell(k)^2$ (or even $\ell(k)$). (Note that the complexity of the distinguisher circuit is dominated by the complexity of A, but we have set $\ell(k) = \text{time}_A(n)$.)

The question is whether such generators exist. The answer depends on the existence of sufficiently hard problems. Note that this is not surprising, because the definition of pseudorandomness actually refers to a problem (i.e., the one of distinguishing) that should be hard (although it is "solvable" when waiving resource-bounds, because the pseudorandom sequences are not truly random).

Indeed, we have:

Theorem 13.14 (Theorem 13.11, restated): *Suppose that there exists a predicate f_0 that is computable in exponential-time and a constant $c_0 > 0$ such that, for all but finitely many m's, any circuit C_m that correctly compute f_0 on $\{0,1\}^m$ has size at least $2^{c_0 m}$. Then, there exists a constant $c > 0$ and an exponential-time generator $G : \{0,1\}^k \to \{0,1\}^{\ell(k)}$ such that $\ell(k) = 2^{\Omega(k)}$ and for circuit C of size 2^{ck} it holds that*

$$\left|\Pr_{s\in\{0,1\}^k}[C(G(s)) = 1] - \Pr_{r\in\{0,1\}^{\ell(k)}}[C(r) = 1]\right| \; < \; 1/10$$

Note that $c_0 < 1$ must hold or else the hypothesis cannot possibly hold (i.e., because a circuit of size 2^m may just incorporate the values of f_0 for all m-bit strings). Exercise: Show that Theorem 13.14 implies Theorem 13.11 (i.e., if for some $c_1 > 0$ the class $\mathcal{E}$ does not have $2^{c_0 n}$-size circuits then $\mathcal{BPP} = \mathcal{P}$). The proof of Theorem 13.14 consists of two steps:

1. *Hardness Amplification*: Given f_0 as in the hypothesis, we construct an exponential-time computable predicate f_1 that cannot be approximated (on random m-bit inputs) by $2^{c_1 m}$-sized circuits, where $c_1 > 0$ is a constant depending on c_0. Specifically, for any such circuit C, it holds that
$$\Pr_{x\in\{0,1\}^m}[C(x) = f_1(x)] \; < \; \frac{1}{2} + 2^{-c_1 m}$$
That is, whereas f_0 is "only" hard to compute in the worst-case, f_1 is even hard to guess with significant advantage (over the obvious random guess).

2. *The actual construction*: Average-case of the latter type is naturally linked to pseudorandomness. Specifically, given $f = f_1$ as above, $G(s) = s, f(s)$ is a pseudorandom generator (alas with "pitiful" stretch).[9] However, our goal is to obtain exponential stretch (rather than one-bit stretch). Clearly, we cannot just repeat the above (i.e., $G(s) = s, f(s), f(s), ..., f(s)$ is clearly *not* a pseudorandom generator, regardless how complex f is). One natural idea is to apply f to different parts of the seed; that is, to parts of the seed with small pairwise overlap. This is indeed the construction in use. Let $T_1, ..., T_{\ell(k)}$ be a collection of sets such that $T_i \subset \{1, ..., k\}$, $|T_i| = k' = \Omega(k)$, and $|T_i \cap T_j| \leq k'' = k'/O(1)$ for every $i \neq j$. On input a k-bit seed s, the generator will construct such a collection in exponential-time (details omitted), and will output the sequence
$$f(s[T_1]), f(s[T_2]), ..., f(s[T_{\ell(k)}])$$
where $s[T_i]$ is the projection of s on coordinates T_i.

Observe that since f is computable in exponential-time so is G, and that G has the desired stretch. The issue is to establish the pseudorandomness of G. An important theorem in that respect is the connection of pseudorandomness and unpredictability (i.e., hardness of guessing the next bit in the output sequence when given the previous bits). Clearly, pseudorandomness implies unpredictability (because ability to predict the next bit in the output of G yields ability to distinguish G's output from a truly random sequence). However, we care about the opposite direction (i.e., that unpredictability implies pseudorandomness, or put differently, ability to distinguish from random implies ability to predict).

[9] Exercise: Prove that $G(s) = s, f(s)$ is indeed a pseudorandom generator.

Unpredictability implies pseudorandomness: Suppose that a circuit C can distinguish with gap $\epsilon(k)$ between X_k (in our case the output of G on a random k-bit seed) and the uniform distribution over $\{0,1\}^{\ell(k)}$. Consider, for $i = 0, ..., \ell(k)$, the hybrid distributions H_k^i, where H_k^i consists of the first i bits of X_k augmented with an $(\ell(k) - i)$-bit long uniformly distributed string. Observe that $H_k^{\ell(k)} \equiv G(U_k)$ and $H_k^0 \equiv U_{\ell(k)}$, where U_m denotes the uniform distribution over $\{0,1\}^m$. Thus, although "not designed for that purpose", there exists an i such that C distinguishes with gap at least $\epsilon(k)/\ell(k)$ between H_k^{i+1} and H_k^i. On the other hand, H_k^{i+1} and H_k^i differ only in the distribution of the $i+1$st bit, and so C can be easily converted into a predictor of the $i+1$st bit of $G(U_k)$. (Exercise: Fill-up the details.)

Predictability of G implies approximation of f: By the above, it suffices to prove that the output of G is unpredictable (with the suitable parameters). Towards the contradiction, we consider a circuit C (of size at most 2^{ck}) predicting the $i+1$st bit of $G(U_k)$. Using the definition of G, we have

$$\Pr_{s\in\{0,1\}^k}[C(f(s[T_1]), ..., f(s[T_i])) = f(s[T_{i+1}])] \;>\; \frac{1}{2} + \epsilon(k)$$

For simplicity of notations, suppose that $T_{i+1} = \{1, ..., k'\}$, and write $s = \langle x, s' \rangle$, where $|x| = k'$. Using an averaging-argument (i.e., fixing the best s'), we infer that there exists a string $s' \in \{0,1\}^{k-k'}$ such that

$$\Pr_{x\in\{0,1\}^{k'}}[C(f(\langle x, s'\rangle[T_1]), ..., f(\langle x, s'\rangle[T_i])) = f(x)] \;>\; \frac{1}{2} + \epsilon(k)$$

The key observation is that, for $j \leq i$, the value of $f(\langle x, s'\rangle[T_j])$ depends only on at most k'' bits of x (i.e., the bits in positions $T_j \cap T_{i+1}$). Thus, there exists a circuit of size at most $\exp(k'')$ (which depends on the fixed s'') that given x computes $f(\langle x, s'\rangle[T_j])$ (i.e., by using a look-up table for the relevant bits of x). Combining all these circuits, we obtain a circuit C' (which is only $\ell(k) \cdot \exp(k'')$ bigger than C) such that $\Pr_{x\in\{0,1\}^{k'}}[C'(x) = f(x)] \;>\; \frac{1}{2} + \epsilon(k)$. For a suitable setting of the constants c, $c' = k'/k$ and $c'' = k''/k'$, we obtain a contradiction to the hypothesis regarding f (since C' has size at most $2^{ck} + 2^{ck+k''} < 2^{1+((c/c')+c'')k'}$ and approximates the value of f on random k'-bit inputs, whereas $(c/c') + c'' < c_1$).

Lecture 14

Average-Case Complexity

In 1984, Leonid Levin has initiated a theory of average-case complexity. We provide an exposition of the basic definitions suggested by Levin, and discuss some of the considerations underlying these definitions. The notes for this lecture were adapted from [24],

14.1 Introduction

The average complexity of a problem is, in many cases, a more significant measure than its worst-case complexity. This has motivated the development of a rich area in algorithmic research: the probabilistic analysis of algorithms (cf. [48, 46]). However, this line of research has so far been applicable only to specific algorithms and with respect to specific, typically uniform, probability distributions.

The general question of average-case complexity was addressed for the first time by Levin [59]. Levin's work can be viewed as the basis for a theory of average NP-completeness, much the same way as Cook's [17] (and Levin's [57]) works are the basis for the theory of NP-completeness. Subsequent works have provided few additional complete problems. Other basic complexity problems, such as decision versus search, were studied in [10].

Levin's average-case complexity theory in a nutshell. An average-case complexity class consists of pairs, called distributional problems. Each such pair consists of a decision (resp., search) problem and a probability distribution on problem instances. We focus on the class DistNP$\stackrel{\rm def}{=}\langle\mathcal{NP},\text{P-computable}\rangle$, defined by Levin [59], which is a distributional analogue of NP: It consists of NP decision problems coupled with distributions for which the accumulative measure is polynomial-time computable. That is, P-computable is the class of distributions for which there exists a polynomial time algorithm that on input x computes the total probability of all strings $y \leq x$. The easy distributional problems are those solvable in "average polynomial-time" (a notion which surprisingly require careful formulation). Reductions between distributional problems are defined in a way guaranteeing that if Π_1 is reducible to Π_2 and Π_2 is in average polynomial-time, then so is Π_1. Finally, it is shown that the class DistNP contains a complete problem.

Levin's average-case theory, revisited. Levin's laconic presentation [59] hides the fact that choices has been done in the development of the average-case complexity theory. We discuss some of these choices here. Firstly, we stress that the motivation here is to provide a theory of efficient computation, rather than a theory of infeasible computation (e.g., as in Cryptography). (The two are not the same!) Furthermore, we note that a theory of useful-for-cryptography infeasible computations does exist (cf., e.g., [27]). A key difference between the two theories is that in Cryptography we needs problems for which one may generate instance-solution pairs so that solving the problem given only the instance is hard. In the theory of average-case complexity considered below, we consider problems that are hard to solve, but do not require an efficient procedure for generating hard (on the average) *instances coupled with solutions*.

Secondly, one has to admit that the class DistNP (i.e., specifically, the choice of distributions) is somewhat problematic. Indeed P-computable distributions seem "simple", but it is not clear if they exhaust all natural "simple" distributions. A much wider class, which is easier to defend, is the class of all distributions having an efficient algorithm for generating instances (according to the distribution). One may argue that the instances of any problem we may need to solve are generated efficiently by some process, and so the latter class of P-samplable distribution suffices for our theory [10]. Fortunately, it was show [44] that any distributional problem that is complete for DistNP=$\langle \mathcal{NP}, \text{P-computable}\rangle$, is also complete with respect to the class $\langle \mathcal{NP}, \text{P-samplable}\rangle$. Thus, in retrospect, Levin's choice only makes the theory stronger: It requires to select complete distributional problems from the restricted class $\langle \mathcal{NP}, \text{P-computable}\rangle$, whereas hardness holds with respect to the wider class $\langle \mathcal{NP}, \text{P-samplable}\rangle$.

As hinted above, the definition of average polynomial-time is less straightforward than one may expect. The obvious attempt at formulation this notion leads to fundamental problems which, in our opinion, deem it inadequate. (For a detailed discussion of this point, the reader is referred to the Appendix.) We believe that once the failure of the obvious attempt is understood, Levin's definition (presented below) does look a natural one.

14.2 Definitions and Notations

In this section we present the basic definitions underlying the theory of average-case complexity. Most definitions originate from Levin [59], but the reader is advised not to look there for further explanations and motivating discussions.

For sake of simplicity, we consider the standard lexicographic ordering of binary strings. Any fixed efficient enumeration will do. (An *efficient enumeration* is a 1-1 and onto mapping of strings to integers that can be computed and inverted in polynomial-time.) By writing $x < y$ we mean that the string x precedes y in lexicographic order, and $y - 1$ denotes the immediate predecessor of y. Also, we associate pairs, triples etc. of binary strings with single binary strings in some standard manner (i.e. encoding).

Definition 14.1 (Probability Distribution Function): *A* distribution function $\mu : \{0,1\}^* \to [0,1]$ *is a non-decreasing function from strings to the unit interval* $[0,1]$ *that converges to one; that is,* $\mu(0) \geq 0$, $\mu(x) \leq \mu(y)$ *for each* $x < y$, *and* $\lim_{x\to\infty} \mu(x) = 1$. *The* density function *associated with the distribution function* μ *is denoted* μ' *and defined by* $\mu'(0) = \mu(0)$ *and* $\mu'(x) = \mu(x) - \mu(x-1)$ *for every* $x > 0$.

Clearly, $\mu(x) = \sum_{y \leq x} \mu'(y)$. For notational convenience, we often describe distribution functions converging to some $c \neq 1$. In all the cases where we use this convention it is easy to normalize the distribution, so that it converges to one. An important example is the *uniform* distribution function μ_0 defined as $\mu'_0(x) = \frac{1}{|x|^2} \cdot 2^{-|x|}$. (A minor modification that does converge to 1 is obtained by letting $\mu'_0(x) = \frac{1}{|x|\cdot(|x|+1)} \cdot 2^{-|x|}$.)

Definition 14.2 (A Distributional Problem): *A* distributional decision problem (resp., distributional search problem) *is a pair (D, μ) (resp. (S, μ)), where $D : \{0,1\}^* \to \{0,1\}$* (resp., $S \subseteq \{0,1\}^* \times \{0,1\}^*$) *and $\mu : \{0,1\}^* \to [0,1]$ is a distribution function.*

In the sequel we consider mainly decision problems. Similar formulations for search problems can be easily derived.

14.2.1 Distributional-NP

Simple distributions are identified with the P-computable ones. The importance of restricting attention to simple distributions (rather than allowing arbitrary ones) is demonstrated in [10, Sec. 5.2]: essentially, making no such restrictions would collapse the average-case theory to the standard worst-case theory.

Definition 14.3 (P-computable): *A distribution μ is in the class* P-computable *if there is a deterministic polynomial time Turing machine that on input x outputs the binary expansion of $\mu(x)$ (i.e., the running time is polynomial in $|x|$).*

It follows that the binary expansion of $\mu(x)$ has length polynomial in $|x|$. An necessary condition for distributions to be of interest is their putting noticeable probability weight on long strings (i.e., for some polynomial, p, and sufficiently big n the probability weight assigned to n-bit strings should be at least $1/p(n)$). Consider to the contrary the density function $\mu'(x) \stackrel{\text{def}}{=} 2^{-3|x|}$. An algorithm of running time $t(x) = 2^{|x|}$ will be considered to have constant on the average running-time w.r.t this μ (as $\sum_x \mu'(x) \cdot t(|x|) = \sum_n 2^{-n} = 1$).

If the distribution function μ is in P-computable then the density function, μ', is computable in time polynomial in $|x|$. The converse, however, is false, unless $\mathcal{P} = \mathcal{NP}$. In spite of this remark we usually present the density function, and leave it to the reader to verify that the corresponding distribution function is in P-computable.

We now present the class of distributional problems which corresponds to (the traditional) NP. Most of results in the literature refer to this class.

Definition 14.4 (The class DistNP): *A distributional problem (D, μ) belongs to the class* DistNP *if D is an NP-predicate and μ is in* P-computable*.* DistNP *is also denoted $\langle \mathcal{NP}, \text{P-computable} \rangle$.*

A wider class of distributions, denoted P-samplable, gives rise to a wider class of distributional NP problems, which was discussed in the introduction: A distribution μ is in the class P-samplable if there exists a polynomial p and a probabilistic algorithm A that outputs the string x with probability $\mu'(x)$ within $p(|x|)$ steps. That is, elements in a P-samplable distribution are generated in time polynomial in their length. We comment that any P-computable distribution is P-samplable, whereas the converse if false (provided one-way functions exist). For a detailed discussion see [10].

14.2.2 Average Polynomial-Time

The following definitions, regarding average polynomial-time, may seem obscure at first glance. It is important to point out that the naive formalizations of these definitions suffer from serious problems such as not being closed under functional composition of algorithms, being model dependent, encoding dependent etc. For a more detailed discussion, see Appendix.

Definition 14.5 (Polynomial on the Average): *A function* $f : \{0,1\}^* \to \mathbf{N}$ *is* polynomial on the average *with respect to a distribution* μ *if there exists a constant* $\epsilon > 0$ *such that*

$$\sum_{x \in \{0,1\}^*} \mu'(x) \cdot \frac{f(x)^\epsilon}{|x|} < \infty$$

The function $l(x) = f(x)^\epsilon$ *is* linear on the average w.r.t. μ.

Thus, a function is polynomial on the average if it is bounded by a polynomial in a function that is linear on the average. In fact, the basic definition is that of a function that is linear on the average; see [10, Def. 2].

Definition 14.6 (The class Average-P): *A distributional problem* (D, μ) *is in the class* Average-P *if there exists an algorithm* A *solving* D*, so that the running time of* A *is polynomial on the average with respect to the distribution* μ*.*

We view the classes Average-P and DistNP as the average-case analogue of P and NP (respectively).

14.2.3 Reducibility between Distributional Problems

We now present definitions of (average polynomial time) reductions of one distributional problem to another. Intuitively, such a reduction should be efficiently computable, yield a valid result and "preserve" the probability distribution. The purpose of the last requirement is to ensure that the reduction does not map very likely instances of the first problem to rare instances of the second problem. Otherwise, having a polynomial time on the average algorithm for the second distributional problem does not necessarily yield such an algorithm for the first distributional problem. Following is a definition of randomized Turing reductions. Definitions of deterministic and many-to-one reductions can be easily derived as special cases.

Definition 14.7 (Randomized Turing Reductions): *We say that the probabilistic oracle Turing machine* M randomly reduces *the distributional problem* (D_1, μ_1) *to the distributional problem* (D_2, μ_2) *if the following three conditions hold.*

1) Efficiency: *Machine* M *is polynomial time on the average taken over* x *with distribution* μ_1 *and the internal coin tosses of* M *with uniform probability distribution (i.e., let* $t_M(x, r)$ *be the running time of* M *on input* x *and internal coin tosses* r*, then there exists* $\epsilon > 0$ *such that* $\sum_{x,r} \mu_1'(x)\mu_0'(r) \cdot \frac{t_M(x,r)^\epsilon}{|x|} < \infty$*, where* μ_0 *is the uniform distribution).*

2) Validity: *For every* $x \in \{0,1\}^*$*,*

$$Prob(M^{D_2}(x) = D_1(x)) \geq \frac{2}{3}$$

where $M^{D_2}(x)$ *is the random variable (determined by* M*'s internal coin tosses) which denotes the output of the oracle machine* M *on input* x *and access to oracle for* D_2*.*

3) Domination: *There exists a constant* $c > 0$ *such that for every* $y \in \{0,1\}^*$*,*

$$\mu_2'(y) \geq \frac{1}{|y|^c} \cdot \sum_{x \in \{0,1\}^*} Ask_M(x, y) \cdot \mu_1'(x)$$

where $Ask_M(x, y)$ *is the probability (taken over* M*'s internal coin tosses) that "machine* M *asks query* y *on input* x*".*

In the definition of deterministic Turing reductions $M^{D_2}(x)$ is determined by x (rather than being a random variable) and $Ask_M(x,y)$ is either 0 or 1 (rather than being any arbitrary rational in $[0,1]$). In case of a many-to-one deterministic reduction, for every x, we have $Ask_M(x,y)=1$ for a unique y.

It can be proven[1] that if (D_1,μ_1) is deterministically (resp., randomly) reducible to (D_2,μ_2) and if (D_2,μ_2) is solvable by a deterministic (resp., randomized) algorithm with running time polynomial on the average then so is (D_1,μ_1).

Reductions are transitive in the special case in which they are *honest*; that is, on input x they ask queries of length at least $|x|^\epsilon$, for some constant $\epsilon>0$. All known reductions have this property.

14.2.4 A Generic DistNP Complete Problem

The following distributional version of *Bounded Halting*, denoted $\Pi_{BH}=(BH,\mu_{BH})$, is known to be DistNP-complete (see Section 14.3).

Definition 14.8 (distributional Bounded Halting):

- Decision*: $BH(M,x,1^k)=1$ iff there exists a computation of the non-deterministic machine M on input x which halts within k steps.*
- Distribution*: The distribution μ_{BH} is defined in terms of its density function*

$$\mu'_{BH}(M,x,1^k) \stackrel{\rm def}{=} \frac{1}{|M|^2\cdot 2^{|M|}}\cdot\frac{1}{|x|^2\cdot 2^{|x|}}\cdot\frac{1}{k^2}$$

Note that μ'_{BH} is very different from the uniform distribution on binary strings (e.g., consider relatively large k). Yet, as noted by Levin, one can easily modify Π_{BH} so that has a "uniform" distribution and is DistNP-complete with respect to randomized reduction. (Hint: replace the unary time bound by a string of equal length, assigning each such string the same probability.)

14.3 DistNP-completeness of Π_{BH}

The proof, presented here, is due to Guretich [37]. (An alternative proof is implied by Levin's original paper [59].)

In the traditional theory of NP-completeness, the *mere* existence of complete problems is almost immediate. For example, it is very easy to show that Bounded Halting is NP-complete.[2] In the case of distributional-NP an analogous theorem is much harder to prove. The difficulty is that we have to reduce all DistNP problems (i.e., pairs consisting of decision problems and simple

[1] Hint: Suppose that, for $\epsilon>0$, we have $\sum_x \mu'_1(x)\frac{t(x)^\epsilon}{|x|}=O(1)$, and for some $c\geq 1$ we have $\mu'_2(x)\leq |x|^c\mu'_1(x)$ $(\forall x)$. Then, let $S\stackrel{\rm def}{=}\{x: t(x)\leq |x|^{2c/\epsilon}\}$, and split the sum $\sum_x \mu'_2(x)\frac{t(x)^{\epsilon/2c}}{|x|}$ according to $x\in S$ or not. The sum $\sum_{x\in S}\mu'_2(x)\frac{t(x)^{\epsilon/2c}}{|x|}$ is bounded by 1, using $t(x)^{\epsilon/2c}\leq|x|$; whereas $\sum_{x\notin S}\mu'_2(x)\frac{t(x)^{\epsilon/2c}}{|x|}$ is bounded by $O(1)$, using $\mu'_2\leq |x|^c\mu'_1(x)$ and $|x|^c\leq t(x)^{\epsilon/2}$ (and $\sum_x\mu'_1(x)\frac{t(x)^\epsilon}{|x|}=O(1)$).

[2] Recall that Bounded Halting (BH) is defined over triples $(M,x,1^k)$, where M is a non-deterministic machine, x is a binary string and k is an integer (given in unary). The problem is to determine whether there exists a computation of M on input x which halts within k steps. Clearly, Bounded Halting is in $\mathcal{NP}$ (here its crucial that k is given in unary). Let D be an arbitrary $\mathcal{NP}$ problem, and let M_D be the non-deterministic machine solving it in time $P_D(n)$ on inputs of length n, where P_D is a fixed polynomial. Then the reduction of D to BH consists of the transformation $x\mapsto(M_D,x,1^{P_D(|x|)})$.

distributions) to one single distributional problem (i.e., Bounded Halting with a single simple distribution). Applying reductions as in Footnote 2 we end-up with many distributional versions of Bounded Halting, and furthermore the corresponding distribution functions will be very different and will not necessarily dominate one another. Instead, one should reduce each distributional problem, (D,μ), with an arbitrary P-computable distribution μ to the same distributional problem with a fixed (P-computable) distribution (e.g. Π_{BH}). The difficulty in doing so is that the reduction should have the domination property. Consider for example an attempt to reduce each problem in DistNP to Π_{BH} by using the standard transformation of D to BH (i.e., $x \mapsto (M_D, x, 1^{P_D(|x|)})$). This transformation fails when applied to distributional problems in which the distribution of (infinitely many) strings is much higher than the distribution assigned to them by the uniform distribution. In such cases, the standard reduction maps an instance x having probability mass $\mu'(x) \gg 2^{-|x|}$ to a triple $(M_D, x, 1^{P_D(|x|)})$ with much lighter probability mass (recall $\mu'_{BH}(M_D, x, 1^{P_D(|x|)}) < 2^{-|x|}$). This violates the domination condition, and thus an alternative reduction is required.

The key to the alternative reduction (of (D,μ) to Π_{BH}) is an (efficiently computable) encoding of strings taken from an arbitrary polynomial-time computable distribution by strings that have comparable probability mass under a fixed distribution. This encoding will map x into a codeword of length bounded above by the logarithm of $1/\mu'(x)$. Accordingly, the reduction will map x to a triple $(M_{D,\mu}, x', 1^{|x|^{O(1)}})$, where $|x'| < O(1) + \log_2 1/\mu'(x)$, and $M_{D,\mu}$ is a non-deterministic Turing machine that first retrieves x from x' and then applies the standard non-deterministic machine (i.e., M_D) of the problem D. Such a reduction will be shown to satisfy all three conditions (i.e. efficiency, validity, and domination). Thus, instead of forcing the structure of the original distribution μ on the target distribution μ_{BH}, the reduction will incorporate the structure of μ into the the reduced instance. The following technical lemma is the basis of the reduction.

Coding Lemma: Let μ be a polynomial-time computable distribution function. Then there exist a coding function C_μ satisfying the following three conditions.

1) *Compression:* For every $x \in \{0,1\}^*$

$$|C_\mu(x)| \leq 1 + \min\left(|x|, \log_2 \frac{1}{\mu'(x)}\right)$$

2) *Efficient Encoding:* The function C_μ is computable in polynomial-time.

3) *Unique Decoding:* The function C_μ is one-to-one (i.e. $C_\mu(x) = C_\mu(x')$ implies $x = x'$).

Proof: The function C_μ is defined as follows. If $\mu'(x) \leq 2^{-|x|}$ then $C_\mu(x) = 0x$ (i.e. in this case x serves as its own encoding). If $\mu'(x) > 2^{-|x|}$ then $C_\mu(x) = 1z$, where z is the longest common prefix of the binary expansions of $\mu(x-1)$ and $\mu(x)$ (e.g. if $\mu(1010) = 0.10000$ and $\mu(1011) = 0.10101111$ then $C_\mu(1011) = 1z$ with $z = 10$). Consequently, $0.z1$ is in the interval $(\mu(x-1), \mu(x)]$; that is, $\mu(x-1) < 0.z1 \leq \mu(x)$.

We now verify that C_μ so defined satisfies the conditions of the lemma. We start with the compression condition. Clearly, if $\mu'(x) \leq 2^{-|x|}$ then $|C_\mu(x)| = 1 + |x| \leq 1 + \log_2(1/\mu'(x))$. On the other hand, suppose that $\mu'(x) > 2^{-|x|}$ and let $z = z_1 \cdots z_\ell$ be as above (i.e., the longest common prefix of the binary expansions of $\mu(x-1)$ and $\mu(x)$). Then,

$$\mu'(x) \;=\; \mu(x) - \mu(x-1) \;\leq\; \left(\sum_{i=1}^{\ell} 2^{-i} z_i + \sum_{i=\ell+1}^{\mathrm{poly}(|x|)} 2^{-i}\right) - \sum_{i=1}^{\ell} 2^{-i} z_i \;<\; 2^{-|z|}$$

and $|z| \leq \log_2(1/\mu'(x))$ follows. Thus, $|C_\mu(x)| \leq 1+\log_2(1/\mu'(x))$ in both cases. Clearly, C_μ can be computed in polynomial-time by computing $\mu(x-1)$ and $\mu(x)$. Finally, note that C_μ is one-to-one by considering the two cases, $C_\mu(x) = 0x$ and $C_\mu(x) = 1z$. (In the second case, use the fact that $\mu(x-1) < 0.z1 \leq \mu(x)$). ∎

Using the coding function presented in the above proof, we introduce a non-deterministic machine $M_{D,\mu}$ so that the distributional problem (D,μ) is reducible to $\Pi_{BH} = (BH, \mu_{BH})$ in a way that all instances (of D) are mapped to triples with first element $M_{D,\mu}$. On input $y = C_\mu(x)$, machine $M_{D,\mu}$ computes $D(x)$, by first retrieving x from $C_\mu(x)$ (e.g., guess and verify), and next running the non-deterministic polynomial-time machine (i.e., M_D) that solves D.

The reduction maps an instance x (of D) to the triple $(M_{D,\mu}, C_\mu(x), 1^{P(|x|)})$, where $P(n) \stackrel{\text{def}}{=} P_D(n) + P_C(n) + n$, $P_D(n)$ is a polynomial bounding the running time of M_D on acceptable inputs of length n, and $P_C(n)$ is a polynomial bounding the running time of an algorithm for encoding inputs (of length n).

Proposition: The above mapping constitutes a reduction of (D,μ) to (BH, μ_{BH}).

Proof: We verify the three requirements.

- The transformation can be computed in polynomial-time. (Recall that C_μ is polynomial-time computable.)

- By construction of $M_{D,\mu}$ it follows that $D(x) = 1$ if and only if there exists a computation of machine $M_{D,\mu}$ that on input $C_\mu(x)$ halts outputting 1 within $P(|x|)$ steps. (Recall, on input $C_\mu(x)$, machine $M_{D,\mu}$ non-deterministically guesses x, verifies in $P_C(|x|)$ steps that x is encoded by $C_\mu(x)$, and non-deterministically "computes" $D(x)$.)

- To see that the distribution induced by the reduction is dominated by the distribution μ_{BH}, we first recall that the transformation $x \to C_\mu(x)$ is one-to-one. It suffices to consider instances of BH that have a preimage under the reduction (since instances with no preimage satisfy the condition trivially). All these instances are triples with first element $M_{D,\mu}$. By the definition of μ_{BH}
$$\mu'_{BH}(M_{D,\mu}, C_\mu(x), 1^{P(|x|)}) = c \cdot \frac{1}{P(|x|)^2} \cdot \frac{1}{|C_\mu(x)|^2 \cdot 2^{|C_\mu(x)|}}$$
where $c = \frac{1}{|M_{D,\mu}|^2 \cdot 2^{|M_{D,\mu}|}}$ is a constant depending only on (D,μ).
By virtue of the coding Lemma
$$\mu'(x) \leq 2 \cdot 2^{-|C_\mu(x)|}$$
It thus follows that
$$\begin{aligned} \mu'_{BH}(M_{D,\mu}, C_\mu(x), 1^{P(|x|)}) &\geq c \cdot \frac{1}{P(|x|)^2} \cdot \frac{1}{|C_\mu(x)|^2} \cdot \frac{\mu'(x)}{2} \\ &> \frac{c}{2 \cdot |M_{D,\mu}, C_\mu(x), 1^{P(|x|)}|^2} \cdot \mu'(x) \end{aligned}$$

The Proposition follows. ∎

14.4 Conclusions

In general, a theory of average-case complexity should provide

1. a specification of a broad class of *interesting* distributional problems;
2. a definition capturing the subclass of (distributional) problems that are *easy* on the average;
3. notions of reducibility that allow to infer the easiness of one (distributional) problem from the easiness of another;
4. and, of course, results...

It seems that the theory of average-case complexity, initiated by Levin and further developed in [37, 10, 44], satisfies these expectations to some extent. Following is my evaluation regarding its "performance" with respect to each of the above.

1. The scope of the theory, originally restricted to P-computable distributions has been significantly extended to cover all P-sampleable distributions (as suggested in [10]). The key result here is by Impagliazzo and Levin [44] show proved that every language that is $\langle \mathcal{NP}, \text{P-computable}\rangle$-complete is also $\langle \mathcal{NP}, \text{P-samplable}\rangle$-complete. This important result makes the theory of average-case very robust: It allows to reduce distributional problems from an utmost wide class to distributional problems with very restricted/simple type of distributions.
2. The definition of average polynomial-time does seem strange at first glance, but it seems that it (or similar alternative) does captures the intuitive meaning of "easy on the average".
3. The notions of reducibility are both natural and adequate.
4. Results did follow, but here indeed much more is expected. Currently, DistNP-complete problems are known for the following areas: Computability (e.g., Bounded-Halting), Combinatorics (e.g., Tiling and a generalization of graph coloring), Formal Languages and Algebra (e.g., of matrix groups). However the challenge of finding a really natural distributional problem that is complete in DistNP (e.g., subset sum with uniform distribution), has not been met so far. It seems that what is still lacking are techniques for design of "distribution preserving" reductions.

In addition to their central role in the theory of average-case complexity, reductions that preserve uniform (or very simple) instance distribution are of general interest. Such reductions, unlike most known reductions used in the theory of NP-completeness, have a range that is a non-negligible part of the set of all possible instances of the target problem (i.e. a part that cannot be claim to be only a "pathological subcase").

We note that Levin views the results in his paper [59] as an indication that all "simple" (i.e., P-computable) distributions are in fact related (or similar).

Appendix: Failure of a naive formulation

When asked to motivate his definition of average polynomial-time, Leonid Levin replies, non-deterministically, in one of the following three ways:

- "This is *the* natural definition".

- "This definition is *not important* for the results in my paper; only the definitions of reduction and completeness matter (and also they can be modified in many ways preserving the results)".

- "Any definition that *makes sense* is either equivalent or weaker".

For further elaboration on the first argument the reader is referred to Leonid Levin. The second argument is, of course, technically correct but unsatisfactory. We will need a definition of "easy on the average" when motivating the notion of a reduction and developing useful relaxations of it. The third argument is a thesis which should be interpreted along Wittgenstein's suggestion to the teacher: "say nothing and confine yourself to pointing out errors in the students' attempts to say something". We will follow this line here by arguing that the definition that seems natural to an average computer scientist suffers from serious problems and should be rejected.

Definition X (naive formulation of the notion of easy on the average): *A distributional problem (D, μ) is polynomial-time on the average if there exists an algorithm A solving D (i.e. on input x outputs $D(x)$) such that the running time of algorithm A, denoted t_A, satisfies $\exists c > 0 \forall n$:*

$$\sum_{x \in \{0,1\}^n} \mu'_n(x) \cdot t_A(x) < n^c$$

where $\mu'_n(x)$ is the conditional probability that x occurs given that an n-bit string occurs (i.e., $\mu'_n(x) = \mu'(x) / \sum_{y \in \{0,1\}^n} \mu'(y)$).

The problem which we consider to be most upsetting is that Definition X is not robust under functional composition of algorithms. Namely, if the distributional problem A can be solved in average polynomial-time given access to an oracle for B, and problem B can be solved in polynomial-time then it does **not** follow that the distributional problem A can be solved in average polynomial-time. For example, consider uniform probability distribution on inputs of each length and an oracle Turing machine M which given access to oracle B solves A. Suppose that M^B runs $2^{\frac{n}{2}}$ steps on $2^{\frac{n}{2}}$ of the inputs of length n, and n^2 steps on all other inputs of length n; and furthermore that M when making t steps asks a single query of length $\sqrt{t}$. (Note that machine M, given access to oracle for B, is polynomial-time on the average.) Finally, suppose that the algorithm for B has cubic running-time. The reader can now verify that although M given access to the oracle B is polynomial-time on the average, combining M with the cubic running-time algorithm for B *does not* yield an algorithm which is polynomial-time on the average according to Definition X. It is easy to see that this problem does not arise when using the definition presented in Section 2.

The source of the above problem with Definition X is the fact that the underlying definition of polynomial-on-the-average is not closed under application of polynomials. Namely, if $t : \{0,1\}^* \to \mathsf{N}$ is polynomial on the average, with respect to some distribution, it does not follow that also $t^2(\cdot)$ is polynomial on the average (with respect to the same distribution). This technical problem is also the source of the following problem, that Levin considers most upsetting: Definition X is *not* machine independent. This is the case since some of the simulations of one computational model on another square the running time (e.g., the simulation of two-tape Turing machines on a one-tape Turing machine, or the simulation of a RAM (Random Access Machine) on a Turing machine).

Another two problems with Definition X have to do with the fact that it deals separately with inputs of different length. The first problem is that Definition X is very dependent on the particular encoding of the problem instance. Consider, for example, a problem on simple undirected graphs for which there exist an algorithm A with running time $t_A(G) = f(n, m)$, where n is the number of vertices in G and m is the number of edges (in G). Suppose that if $m < n^{\frac{3}{2}}$ then $f(n,m) = 2^n$ and else $f(n,m) = n^2$. Consider the distributional problem which consists of the above graph problem with the uniform probability distribution on all graphs with the same number of vertices. Now, if the graph is given by its (incident) matrix representation then Definition X implies that A solves the problem in average polynomial-time (the average is taken on all graphs with n nodes). On the other hand, if the graphs are represented by their adjacency lists then the modified algorithm A (which transforms the graphs to matrix representation and applies algorithm A) is judged by Definition X to be non-polynomial on the average (here the average is taken over all graphs of m edges). This of course will not happen when working with the definition presented in Section 2. The second problem with dealing separately with different input lengths is that it does not allow one to disregard inputs of a particular length. Consider for example a problem for which we are only interested in the running-time on inputs of odd length.

After pointing out several weaknesses of Definition X, let us also doubt its "clear intuitive advantage" over the definition presented in Section 2. Definition X is derived from the formulation of worst-case polynomial-time algorithms which requires that $\exists c > 0\ \forall n$:

$$\forall x \in \{0,1\}^n \ : \quad t_A(x) \ < \ n^c$$

Definition X was derived by applying the expectation operator to the above inequality. But why not make a very simple algebraic manipulation of the inequality before applying the expectation operator? How about taking the c-th root of both sides and dividing by n; this yields $\exists c > 0\ \forall n$:

$$\forall x \in \{0,1\}^n \ : \quad \frac{t_A(x)^{\frac{1}{c}}}{n} \ < \ 1$$

Applying the expectation operator to the above inequality leads to the definition presented in Section 2... We believe that this definition demonstrates a better understanding of the effect of the expectation operator with respect to complexity measures!

Summary: Robustness under functional composition as well as machine independence seems to be essential for a coherent theory. So is robustness under efficiently effected transformation of the problem encoding. These are one of the primary reasons for the acceptability of $\mathcal{P}$ as capturing problems that can be solved efficiently. In going from worst-case analysis to average-case analysis we should not and would not like to lose these properties.

Lecture 15

Circuit Lower Bounds

See old survey by Boppana and Sipser [13].

15.1 Constant-depth circuits

15.2 Monotone circuits

Lecture 16

Communication Complexity

See textbook by Kushilevitz and Nisan [54].

16.1 Deterministic Communication Complexity

16.2 Randomized Communication Complexity

Historical Notes

Probabilistic Proof Systems

For a more detailed account of the history of the various types of probabilistic proof systems, we refer the reader to [26, Sec. 2.6.2].

Interactive Proofs: Interactive proof systems were introduced by Goldwasser, Micali and Rackoff [35], with the explicit objective of capturing the most general notion of efficiently verifiable proof systems. The original motivation was the introduction of zero-knowledge proof systems, which in turn were supposed to provide (and indeed do provide) a powerful tool for the design of complex cryptographic schemes (cf. [32, 33]).

First evidence that interactive proofs may be more powerful than NP-proofs was given by Goldreich, Micali and Wigderson [32], in the form of the interactive proof for Graph Non-Isomorphism presented above. The full power of interactive proof systems was discovered by Lund, Fortnow, Karloff, Nisan, and Shamir (in [61] and [70]). The basic technique was presented in [61] (where it was shown that co$\mathcal{NP} \subseteq \mathcal{IP}$) and the final result ($\mathcal{PSPACE} = \mathcal{IP}$) in [70]. Our presentation follows [70].

Public-coin interactive proofs (also known as Arthur-Merlin proofs) were introduced by Babai [5]. The fact that these restricted interactive proofs are as powerful as general ones was proved by Goldwasser and Sipser [36]. The linear speed-up (in number of rounds) of public-coin interactive proofs was shown by Babai and Moran [8].

Zero-knowledge proofs: The concept of zero-knowledge has been introduced by Goldwasser, Micali and Rackoff (in the very same paper quoted above; i.e., [35]). Their paper contained also a perfect zero-knowledge proof for Quadratic Non-Residuosity. The perfect zero-knowledge proof system for Graph Isomorphism is due to Goldreich, Micali and Wigderson [32]. More importantly, the latter paper presents a zero-knowledge proof systems for all languages in $\mathcal{NP}$, using any secure commitment scheme, which in turn can be constructed based on any one-way function [40, 63]. For the comprehensive discussion of zero-knowledge see [27, Chap. 4].

Probabilistically Checkable Proofs: The PCP Characterization Theorem is attributed to Arora, Lund, Motwani, Safra, Sudan and Szegedy (cf. [4] and [3]). These papers, in turn, built on numerous previous works; for details see the papers themselves or [26]. In general, our presentation of PCP follows [26, Sec. 2.4], and the interested reader is referred to the latter for a survey of further developments and more refined considerations.

The first connection between PCP and hardness of approximation was made by Feige, Goldwasser, Lovasz, Safra, and Szegedy [19]: They showed the connection to maxClique (presented above). The connection to max3SAT and other "MaxSNP approximation" problems was made later in [3].

We did not present the strongest known non-approximability results for max3SAT and max-Clique. These can be found in Hastad's papers, [39] and [38], respectively.

Pseudorandomness

The notion of computational indistinguishability was introduced by Goldwasser and Micali [34] (within the context of defining secure encryptions), and given general formulation by Yao [81]. Our definition of pseudorandom generators follows the one of Yao, which is equivalent to a prior formulation of Blum and Micali [12]. For more details regarding this equivalence, as well as many other issues, see [26]. The latter source presents the notion of pseudorandomness discussed here as a special case (or archetypical case) of a general paradigm.

The discovery that computational hardness (in form of one-wayness) can be turned into a pseudorandomness was made by Blum and Micali [12]. Theorem 13.7 (asserting that pseudorandom generators can be constructed based on any one-way function) is due to Håstad, Impagliazzo, Levin and Luby [40], who build on [30, 31].

The fact that pseudorandom generators yield significantly better derandomization than the straightforward one was first exploited by Yao [81]. The fact that for purpose of derandomization one may use pseudorandom generators that run in exponential time was first observed by Nisan and Wigderson [66], who presented a general framework for such constructions. All improved derandomization results build on the latter framework. In Particular, Theorem 13.11 is due to Impagliazzo and Wigderson [45], who build on [66, 7, 43].

Theorems 13.12 and 13.13 (regarding derandomization of space-bounded randomized classes) are due to Nisan [64, 65] and Nisan and Zuckerman [67], respectively.

Average-Case Complexity

The theory of average-case complexity was initiated by Levin [59]. Levin's laconic presentation [59] hides the fact that important choices have been made in the development of the average-case complexity theory. These choices were discussed in [24], and our presentation follows the latter text.