Introduction to Complexity Theory

Notes for a One–Semester Course

Oded Goldreich
Department of Computer Science and Applied Mathematics
Weizmann Institute of Science, Israel.
Email: oded@wisdom.weizmann.ac.il

Spring 2002 (revised: October 8, 2002)

©Copyright 2002 by Oded Goldreich.
Permission to make copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that new copies bear this notice and the full citation on the first page. Abstracting with credit is permitted.

Preface

Complexity Theory is a central field of Theoretical Computer Science, with a remarkable list of celebrated achievements as well as a very vibrant present research activity. The field is concerned with the study of the *intrinsic* complexity of computational tasks, and this study tend to aim at *generality*: It focuses on natural computational resources, and the effect of limiting those on the *class of problems* that can be solved. Put in other words, Complexity Theory aims at understanding the *nature of efficient computation*.

Topics: In my opinion, a introductory course in complexity theory should aim at exposing the students to the basic results and research directions in the field. The focus should be on concepts and ideas, and complex technical proofs should be avoided. Specific topics may include

- Revisiting NP and NPC (with emphasis on search vs decision);
- Complexity classes defined by one resource-bound – hierarchies, gaps, etc;
- Non-deterministic Space complexity (with emphasis on NL);
- Randomized Computations (e.g., ZPP, RP and BPP);
- Non-uniform complexity (e.g., P/poly, and lower bounds on restricted circuit classes);
- The Polynomial-time Hierarchy;
- The counting class #P, approximate-#P and uniqueSAT;
- Probabilistic proof systems (i.e., IP, PCP and ZK);
- Pseudorandomness (generators and derandomization);
- Time versus Space (in Turing Machines);
- Circuit-depth versus TM-space (e.g., AC, NC, SC);
- Communication complexity;
- Average-case complexity;

Of course, it would be hard (if not impossible) to cover all the above topics (even briefly) in a single-semester course (of two hours a week). Thus, a choice of topics has to be made, and the rest may be merely mentioned in a relevant lecture or in the concluding lecture. The choice may depend on other courses given in the institute; in fact, my own choice was strongly effected by this aspect.

Prerequisites: It is assumed that students have taken a course in computability, and hence are familiar with Turing Machines.

Model of Computation: Most of the presented material is quite independent of the specific (reasonable) model of computation, but some material does depend heavily on the locality of computation of Turing machines.

The partition of material to lectures: The partition of the material to lectures reflects only the logical organization of the material, and does not reflect the amount of time to be spent on each topic. Indeed, some lectures are much longer than other.

State of these notes: These notes provide an outline of an introductory course on complexity theory, including discussions and sketches of the various notions, definitions and proofs. The latter are presented in varying level of detail, where the level of detail does not reflect anything (except the amount of time spent in writing). Furthermore, the notes are neither complete nor fully proofread.

Related text: The current single-semester introductory course on complexity theory is a proper subset of a two-semester course that I gave in 1998–99 at the Weizmann Institute of Science. Lectures notes for that course are availalbe from the webpage

`http://www.wisdom.weizmann.ac.il/~oded/cc.html`

Contents

Lecture Series I

Things that should have been taught in previous courses

The first three lectures focus on material that should have been taught in the basic course on computability. Unfortunately, in many cases this material is covered but from a wrong perspective or without any (proper) perspective. Thus, although in a technical sense most of the material (e.g., the class $\mathcal{NP}$ and the notion of NP-completeness) may be known to the students, its conceptual meaning may not have been appreciated (and our aim is to try to correct this damage).

In addition, we cover some topics that may be new to most students. These topics include self-reducibility (of search problems), the existence of NP-sets that are neither in P nor NP-complete, the effect of having coNP-sets that are NP-complete, and the existence of optimal search algorithms for NP-relations.

Lecture 1

P versus NP

We assume that all students have heard of P and NP, but we suspect that many have not obtained a good explanation of what the *P vs NP question* actually represents. This unfortunate situation is due to using the standard technical definition of NP (which refers to non-deterministic polynomial-time) rather than more cumbersome definitions that clearly capture the fundamental nature of NP. Below, we take the alternative approach. In fact, we present two fundamental formulations of the *P vs NP question*, one in terms of search problems and the other in terms of decision problems.

Efficient computation. Discuss the association of efficiency with polynomial-time. (Polynomials are merely a "closed" set of moderately growing functions, where "closure" means closure under addition, multiplication and composition.)

1.1 The search version

We focus on polynomially-bounded relations. The relation $R \subseteq \{0,1\}^* \times \{0,1\}^*$ is polynomially-bounded if there exists a polynomial p such that for every $(x,y) \in R$ it holds that $|y| \leq p(|x|)$. For such a relation it makes sense to ask whether, given an "instance" x, one can efficiently find a "solution" y such that $(x,y) \in R$. The polynomially-bounded condition guarantees that intrinsic intractability may not be due to the length (or mere typing) of the required solution.

P as a natural class of search problems. With each polynomially-bounded relation R, we associate the following search problem: *given x find y such that $(x,y) \in R$ or state that no such y exists.* The class $\mathcal{P}$ corresponds to the class of search problems that are solvable in polynomial-time (i.e., there exists a polynomial-time algorithm that given x find y such that $(x,y) \in R$ or state that no such y exists).

NP as another natural class of search problems. A polynomially-bounded relation R is called an NP-relation if given an alleged instance-solution pair one can efficiently verify whether the pair is valid; that is, there exists a polynomial-time algorithm that given x and y determines whether or not $(x,y) \in R$. It is reasonable to focus on search problems for NP-relations, because the ability to recognize a valid solution seems to be a natural prerequisite for a discussion regarding finding such solutions. (Indeed, formally speaking, one can introduce non-NP-relations for which the search problem is solvable in polynomial-time; but still the restriction to NP-relations is very natural.)

The P versus NP question in terms of search problems: *Is it the case that the search problem of every NP-relation can be solved in polynomial-time?* In other words, if it is easy to test whether a solution for an instance is correct then is it also easy to find solutions to given instances? If $\mathcal{P} = \mathcal{NP}$ then this would mean that if solutions to given instances can be efficiently verified for correctness then they can also be efficiently found (when given only the instance). This would mean that all reasonable search problems (i.e., all NP-relations) are easy to solve. On the other hand, if $\mathcal{P} \neq \mathcal{NP}$ then there exist reasonable search problems (i.e., some NP-relations) that are hard to solve. In such a case, the world is more interesting: some reasonable problems are easy to solve whereas others are hard to solve.

1.2 The decision version

For an NP-relation R, we denote the set of instances having solution by L_R; that is, $L_R = \{x : \exists y\ (x,y) \in R\}$. Such a set is called an NP-set. Intuitively, an NP-set is a set of valid statements (i.e., statements of membership of a given x in L_R) that can be efficiently verified given adequate proofs (i.e., a corresponding NP-witness y such that $(x,y) \in R$).

NP-proof systems. Proof systems are defined in terms of their verification procedures. Here we focus on the natural class of efficient verification procedures, where efficiency is represented by polynomial-time computations. (We should either require that the time is polynomial in terms of the statement or confine ourselves to "short proofs" – that is, proofs of length that is bounded by a polynomial in the length of the statement.) An NP-relation R yields a natural verification procedure, which amounts to checking whether the alleged statement-proof pair is in R. This proof system satisfies the natural completeness and soundness conditions: every true statement (i.e., $x \in L_R$) has a valid proof (i.e., an NP-witness y such that $(x,y) \in R$), whereas false statements (i.e., $x \notin L_R$) have no valid proofs (i.e., $(x,y) \notin R$ for all y's).

The P versus NP question in terms of decision problems: *Is it the case that NP-proofs are useless?* That is, is it the case that for every efficiently verifiable proof system one can easily determine the validity of assertions without given suitable proofs. If that were the case, then proofs would be meaningless, because they would have no fundamental advantage over directly determining the validity of the assertion. Recall that $\mathcal{P}$ is the class of sets that can be decided efficiently (i.e., by a polynomial-time algorithm). Then the conjecture $\mathcal{P} \neq \mathcal{NP}$ asserts that proofs are useful: there exists NP-sets that cannot be decided by a polynomial-time algorithm, and so for these sets obtaining a proof of membership (for some instances) is useful (because we cannot determine membership by ourselves).

1.3 Conclusions

Verify that $\mathcal{P} \neq \mathcal{NP}$ *in terms of search problems* if and only if $\mathcal{P} \neq \mathcal{NP}$ *in terms of decision problems*. Thus, it suffices to focus on the latter (simpler) formulation.

Note that $\mathcal{NP}$ is typically defined as the class of sets that can be decided by a fictitious device called a non-deterministic polynomial-time machine. The reason that this class of fictitious devices is important is because it captures (indirectly) the definition of NP-proofs. Verify that indeed the "standard" definition of $\mathcal{NP}$ (in terms of non-deterministic polynomial-time machine) equals our definition of $\mathcal{NP}$ (in terms of the class of sets having NP-proofs).

Lecture 2

Reductions and Self-reducibility

We assume that all students have heard of reductions, but again we fear that most have obtained a conceptually-poor view of their nature. We present first the general notion of (polynomial-time) reduction among computational problems, and view the notion of a Karp-reduction as an important special case that suffices (and is more convenient) in some cases.

2.1 The general notion of a reduction

Reductions are procedures that use functionally-specified subroutines. That is, the functionality of the subroutine is specified, but its operation remains unspecified and its running-time is counted at unit cost. Analogously to algorithms, which are modeled by Turing machines, reductions can be modeled as *oracle* (Turing) machines. A reduction solves one computational problem (which may be either a search or decision problem) by using oracle (or subroutine) calls to another computational problem (which again may be either a search or decision problem). We focus on efficient (i.e., polynomial-time) reductions, which are often called Cook reductions.

The standard case is of reducing decision problems to decision problems, but we will also consider reducing search problems to search problems or reducing search problems to decision problems.

A Karp-reduction is a special case of a reduction (from a decision problem to a decision problem). Specifically, for decision problems L and L', we say that L is Karp-reducible to L' if there is a reduction of L to L' *that operates as follows*: On input x (an instance for L), the reduction computes x', makes query x' to the oracle L' (i.e., invokes the subroutine for L' on input x'), and answers whatever the latter returns.

Indeed, a Karp-reduction is a syntactically restricted notion of a reduction. This restricted case suffices for many cases (e.g., most importantly for the theory of NP-completeness), but not in case we want to reduce a search problem to a decision problem. Furthermore, whereas each decision problem is reducible to its complement, some decision problems are not Karp-reducible to their complement (e.g., the trivial decision problem). Likewise, each decision problem in $\mathcal{P}$ is (trivially) reducible to any computational problem (i.e., by a reduction that does not use the subroutine at all), whereas such a trivial reduction is disallowed by the syntax of Karp-reductions.

We comment that Karp-reductions may and should be augmented also in order to handle reductions of search problems to search problems. Such an augmented Karp-reduction of the search problem of R to the search problem of R' operates as follows: On input x (an instance for R), the reduction computes x', makes query x' to the oracle R' (i.e., invokes the subroutine for R' on input x') obtaining y' such that $(x', y') \in R'$, and uses y' to compute a solution y to x (i.e.,

$(x,y) \in R$). (Indeed, unlike in case of decision problems, the reduction cannot just return y' as an answer to x.)

2.2 Self-reducibility of search problems

The search problem for R is called self-reducible if it can be reduced to the decision problem of $L_R = \{x : \exists y\ (x,y) \in R\}$. Note that the decision problem of L_R is always reducible to the search problem for R (e.g., invoke the search subroutine and answer YES if and only if it returns some string (rather than the "no solution" symbol)).

We will see that all NP-relations that correspond to NP-complete sets are self-reducible, mostly via "natural reductions". We start with SAT, the set of satisfiable Boolean formulae. Let R_{SAT} be the set of pairs (ϕ, τ) such that τ is a satisfying assignment to the formulae ϕ. Note that R_{SAT} is an NP-relation (i.e., it is polynomially-bounded and easy to decide (by evaluating a Boolean expression)).

Proposition 2.1 (R_{SAT} is self-reducible): *The search problem R_{SAT} is reducible to SAT.*

Proof: Given a formula ϕ, we use a subroutine for SAT in order to find a satisfying assignment to ϕ (in case such exists). First, we query SAT on ϕ itself, and return "no solution" if the answer we get is 'false'. Otherwise, we let τ, initiated to the empty string, denote a prefix of a satisfying assignment of ϕ. We proceed in iterations, where in each iteration we extend τ by one bit. This is done as follows: First we derive a formula, denoted ϕ', by setting the first $|\tau| + 1$ variables of ϕ according to the values $\tau 0$. Next we query SAT on ϕ' (which means that we ask whether or not $\tau 0$ is a prefix of a satisfying assignment of ϕ). If the answer is positive then we set $\tau \leftarrow \tau 0$ else we set $\tau \leftarrow \tau 1$ (because if τ is a prefix of a satisfying assignment of ϕ and $\tau 0$ is not a prefix of a satisfying assignment of ϕ then $\tau 1$ must be a prefix of a satisfying assignment of ϕ).

A key point is that the formulae ϕ' can be simplified to contain no constants such that they fit the canonical definition of SAT. That is, after replacing some variables by constants, we should simplify clauses according to the straightforward boolean rules (e.g., a false literal can be omitted from a clause and a true literal appearing in a clause yields omitting the entire clause). ■

A similar reduction can be presented also for other NP-complete problems. Consider, for example, 3-Colorability. Note that, in this case, the process of getting rid of constants (representing partial solutions) is more involved. Details are left as an exercise. In general, if you don't see a "natural" self-reducibility process for some NP-complete relation, you should still know that a self-reduction process (alas maybe not a natural one) does exist.

Theorem 2.2 *Every NP-relation of an NP-complete set is self-reducible.*

Proof: Let R be an NP-relation of the NP-complete set L_R. Then, we combine the following sequence of reductions:

1. The search problem of R is reducible to the search problem of R_{SAT} (by the NP-completeness of the latter).
2. The search problem of R_{SAT} is reducible to SAT (by Proposition 2.1).
3. The decision problem SAT is reducible to the decision problem L_R (by the NP-completeness of the latter).

The theorem follows. ■

Lecture 3

NP-completeness

This is the third (and last) lecture devoted to material that the students have heard. Again, most students did see an exposition of the technical material in some undergraduate class, but they might have missed important conceptual points. Specifically, we stress that the mere existence of NP-complete sets (regardless if this is SAT or some other set) is amazing.

3.1 Definitions

The standard definition is that a set is NP-complete if it is in $\mathcal{NP}$ and every set in $\mathcal{NP}$ is reducible to it via a Karp-reduction. Indeed, there is no reason to insist on Karp-reductions (rather than use arbitrary reductions), except that the restricted notion suffices for all positive results and is easier to work with.

We say that a polynomially-bounded relation is NP-complete if it is an NP-relation and every NP-relation is reducible to it.

The mere fact that we have defined something (i.e., NP-completeness) does not mean that this thing exists. It is indeed remarkable that NP-complete problems do exist!

3.2 The existence of NP-complete problems

Theorem 3.1 *There exist NP-complete relations and sets.*

Proof: The proof (as well as all NP-completeness) is based on the observation that some NP-relations are "rich enough" to encode all NP-relations. This is most obvious for the "universal" NP-relation, denoted R_U (and defined below), which is used to derive the simplest proof of the current theorem.

The relation R_U consists of pairs $(\langle M, x, 1^t\rangle, y)$ such that M is a description of a (deterministic) Turing machine that accepts the pair (x, y) within t steps, where $|y| \leq t$. (Instead of requiring that $|y| \leq t$, we may require that M is canonical in the sense that it reads its entire input before halting.) It is easy to see that R_U is an NP-relation, and indeed $L_U \stackrel{\text{def}}{=} \{z : \exists y\ (z, y) \in R_U\}$ is an NP-set.

We now turn to showing that any NP-relation is reducible to R_U. As a warm-up, let us first show that any NP-set is Karp-reducible to L_U. Let R be an NP-relation, and $L_R = \{x : \exists y\ (z, y) \in R\}$ be the corresponding NP-set. Let p_R be a polynomial bounding the length of solutions in R (i.e., $|y| \leq p_R(|x|)$ for every $(x, y) \in$), let M_R be a polynomial-time machine deciding membership (of alleged (x, y) pairs) in R, and let t_R a polynomial bounding its running-time. Then the Karp-reduction maps an instance x (for L) to the instance $\langle M_R, x, 1^{t_R(|x|+p_R(|y|))}\rangle$.

Note that this mapping can be computed in polynomial-time, and that $x \in L$ if and only if $\langle M_R, x, 1^{t_R(|x|+p_R(|y|))}\rangle \in L_U$.

To reduce the search problem of R to the search problem of R_U, we use essentially the same reduction. On input an instance x (for R), we make the query $\langle M_R, x, 1^{t_R(|x|+p_R(|y|))}\rangle$ to the search R_U and return whatever the latter returns. (Note that if $x \notin L_R$ then the answer will be "no solution", whereas for every x and y it holds that $(x, y) \in R$ if and only if $(\langle M_R, x, 1^{t_R(|x|+p_R(|y|))}\rangle, y) \in R_U$. ■

3.3 CSAT and SAT

Define Boolean circuits (directed acyclic graphs with vertices labeled by Boolean operation). Prove the NP-completeness of the circuit satisfaction problem (CSAT). The proof boils down to encoding possible computations of a Turing machine by a corresponding layered circuit, where each layer represents a configuration of the machine, and the conditions of consecutive configurations are captured by uniform local gadgets in the circuit.

Define Boolean formulae (i.e., a circuit with tree structure). Prove the NP-completeness of the formula satisfaction problem (SAT), even when the formula is given in a nice form (i.e., CNF). The proof is by reduction from CSAT, which in turn boils down to introducing auxiliary variables in order to cut the computation of a deep circuit into a conjunction of related computations of shallow (i.e., depth-2) circuits (which may be presented as CNFs).

3.4 NP sets that are neither in P nor NP-complete

Many (to say the least) other NP-sets have been shown to be NP-complete. Things reach a situation in which people seem to expect any NP-set to be either NP-complete or in $\mathcal{P}$. This naive view is wrong:

Theorem 3.2 *Assuming $\mathcal{NP} \neq \mathcal{P}$, there exist NP-sets that are neither NP-complete nor in $\mathcal{P}$.*

The proof is by modifying a set in $\mathcal{NP} \setminus \mathcal{P}$ such that to fail all possible reductions (to this set) and all possible polynomial-time decision procedures (for this set). Specifically, we start with some $L \in \mathcal{NP} \setminus \mathcal{P}$ and derive $L' \subset L$ (which is also in $\mathcal{NP} \setminus \mathcal{P}$) by making each reduction (say of L) to L' fail by dropping finitely many elements from L (until the reduction fails), whereas all possible polynomial-time fail to decide L' (which differ from L only on a finite number of inputs). We use the fact that any reduction (of some set in $\mathcal{NP} \setminus \mathcal{P}$) to a finite set (i.e., a finite subset of L) must fail (while making only a finite number of possible queries), whereas any efficient decision procedure for L (or L modified on finitely many inputs) must fail on some finite portion of all possible inputs (of L). The process of modifying L into L' proceeds in iterations, alternatively failing a reduction (by dropping sufficiently many strings from the rest of L) and failing a decision procedure (by including sufficiently many strings from the rest of L). This can be done efficiently because it is inessential to determine the first location where we have enough strings as long as we determine some location where we have enough.

We mention that some natural problems (e.g., factoring) are conjecture to be neither solvable in polynomial-time nor NP-hard. See discussion following Theorem 3.3.

3.5 NP, coNP and NP-completeness

By prepending the name of a complexity class (of decision problems) with the prefix "co" we mean the class of complement sets; that is,

$$\mathrm{co}\mathcal{C} \stackrel{\mathrm{def}}{=} \{\{0,1\}^* \setminus L : L \in \mathcal{C}\}$$

Specifically, $\mathrm{co}\mathcal{NP} = \{\{0,1\}^* \setminus L : L \in \mathcal{NP}\}$ is the class of sets that are complements of NP-sets. That is, if R is an NP-relation and $L_R = \{x : \exists y\ (x,y) \in R\}$ is the associated NP-set then $\{0,1\}^* \setminus L_R = \{x : \forall y\ (x,y) \notin R\}$ is the corresponding coNP-set.

It is widely believed that $\mathcal{NP}$ is not closed under complementation (i.e., $\mathcal{NP} \neq \mathrm{co}\mathcal{NP}$). Indeed, this conjecture implies $\mathcal{P} \neq \mathcal{NP}$ (because $\mathcal{P}$ is closed under complementation), and is implied by the conjecture that $\mathcal{NP} \cap \mathrm{co}\mathcal{NP}$ is a proper superset of $\mathcal{P}$. The conjecture $\mathcal{NP} \neq \mathrm{co}\mathcal{NP}$ means that some coNP-sets (e.g., the complements of NP-complete sets) do not have NP-proof systems; that is, there is no NP-proof system for proving that a given formula is not satisfiable.

If indeed $\mathcal{P} \neq \mathcal{NP}$ then some (non-trivial) NP-sets cannot be Karp-reducible to coNP-sets (exercise: why). (Recall that the empty set cannot be Karp-reducible to $\{0,1\}^*$.) In contrast, all NP-sets are reducible to coNP-sets (by a straightforward general reduction that just flips the answer). A less obvious fact is that $\mathcal{NP} \neq \mathrm{co}\mathcal{NP}$ implies that some NP-sets cannot be reduced to sets in $\mathcal{NP} \cap \mathrm{co}\mathcal{NP}$ (even under general reductions). Specifically,

Theorem 3.3 *If $\mathcal{NP} \cap \mathrm{co}\mathcal{NP}$ contains an NP-hard set then $\mathcal{NP} = \mathrm{co}\mathcal{NP}$.*

Recall that a set is NP-hard if every NP-set is reducible to it (possibly via a general reduction). Since $\mathcal{NP} \cap \mathrm{co}\mathcal{NP}$ is conjectured to be a proper superset of $\mathcal{P}$, it follows (using the conjecture $\mathcal{NP} \neq \mathrm{co}\mathcal{NP}$) that there are NP-sets are neither in $\mathcal{P}$ nor NP-hard. Notable examples are sets related to the integer factorization problem (e.g., the set of pairs (N, s) such that s has a square root modulo N that is a quadratic residue modulo N and the least significant bit of s equals 1).

Proof: Suppose that $L \in \mathcal{NP} \cap \mathrm{co}\mathcal{NP}$ is NP-hard. Given any $L' \in \mathrm{co}\mathcal{NP}$, we will show that $L' \in \mathcal{NP}$. We will merely use the fact that L' reduces to L (which is in $\mathcal{NP} \cap \mathrm{co}\mathcal{NP}$). Such a reduction exists because L' is reducible $\overline{L}' \stackrel{\mathrm{def}}{=} \{0,1\}^* \setminus L'$ (via a general reduction), whereas $\overline{L}' \in \mathcal{NP}$ and thus is reducible to L (which is NP-hard).

To show that $L' \in \mathcal{NP}$, we will present an NP-relation, R', that characterizes L' (i.e., $L' = \{x : \exists y\ (x,y) \in R'\}$). The relation R' consists of pairs of the form $(x, ((z_1, \sigma_1, w_1), ..., (z_t, \sigma_t, w_t)))$, where on input x the reduction of L' to L accepts after making the queries $z_1, ..., z_t$, obtaining the corresponding answers $\sigma_1, ..., \sigma_t$, and for every i it holds that if $\sigma_i = 1$ then w_i is an NP-witness for $z_i \in L$, whereas if $\sigma_i = 0$ then w_i is an NP-witness for $z_i \in \{0,1\}^* \setminus L$.

We stress that we use the fact that both L and $\overline{L} \stackrel{\mathrm{def}}{=} \{0,1\}^* \setminus L$ are NP-sets, and refer to the corresponding NP-witnesses. Note that R' is indeed an NP-relation: The length of solutions is bounded by the running-time of the reduction (and the corresponding NP-witnesses). Membership in R' is decided by checking that the sequence of (z_i, σ_i)'s matches a possible query-answer sequence in an execution of the reduction[1] (regardless of the correctness of the answers), and that all answers (i.e., σ_i's) are correct. The latter condition is easily verified by use of the corresponding NP-witness. ■

[1]That is, we need to verify that on input x, after obtaining the answers $\sigma_1, ..., \sigma_{i-1}$ to the first $i-1$ queries, the i^{th} query made by the reduction equals z_i.

3.6 Optimal search algorithms for NP-relations

Actually, this section does not relate to NP-completeness, but rather to NP-relations.

The title sounds very promising, but our guess is that the students will be less excited once they see the proof. We claim the existence of an *optimal search algorithm for any NP-relation.* Furthermore, we will explicitly present such an algorithm, and prove that it is optimal in a very strong sense: for any algorithm correctly solving the same search problem, it holds that up-to some fixed additive polynomial term (which may be disregarded in case the NP-problem is not solvable in polynomial-time), our algorithm is at most a constant factor slower than the other algorithm. That is:

Theorem 3.4 *For every NP-relation R there exists an algorithm A that satisfies the following:*

1. *A correctly solves the search problem of R.*
2. *There exists a polynomial p such that for every algorithm A' that correctly solves the search problem of R and for every $x \in L_R$ it holds that $t_A(x) = O(t_{A'}(x) + p(|x|))$, where t_A* (resp., *$t_{A'}$*) *denotes the number of steps taken by A* (resp., *A'*) *on input x.*

Interestingly, we establish the optimality of A without knowing what its (optimal) running-time is. We stress that the hidden constant in the O-notation depends only on A', but in the following proof the dependence is exponential in the length of the description of algorithm A' (and it is not known whether a better dependence can be achieved).

Proof sketch: Fixing R, we let M be a polynomial-time algorithm that decides membership in R, and let p be a polynomial bounding the running-time of M. We present the following algorithm A that merely runs all possible search algorithms "in parallel" and checks the results provided by each of them (using M), halting whenever it obtains a correct solution.

Since there are infinitely many possible algorithms, we should clarify what we mean by "running them all in parallel". What we mean is to run them at different rates such that the infinite sum of rates converges to 1 (or any other constant). Viewed in different terms, for any *unbounded* function $\alpha : \mathsf{N} \to \mathsf{N}$, we proceed in iterations such that in the i^{th} iteration we let each of the first $\alpha(i)$ algorithms run for at most 2^i steps. In case some of these algorithms halts with output y, algorithm A invokes M on input (x, y) and output y if and only if $M(x, y) = 1$. The verification of a solution provided by an algorithm is also emulated at the expense of its step-count. (Put in other words, we augment each algorithm with a canonical procedure (i.e., M) that checks the validity of the solution offered by the algorithm.)

(In case we want to guarantee that A also stops on $x \notin L_R$, we may let it run an exhaustive search for a solution, in parallel to all searches, and halt with output $\perp$ in case this exhaustive search fails.)

Clearly, whenever $A(x)$ outputs y (i.e., $y \neq \perp$) it must hold that $(x, y) \in R$. Now suppose A' is an algorithm that solves R. Fixing A', for every x, let us denote by $t'(x)$ the number of steps taken by A' on input x, where $t'(x)$ also accounts for the running time of $M(x)$. Then, the $t(x)$-step execution of A on input x is "covered" by the $i(x)$th iteration of A, provided that $i(x) = \max(|A'|, \log_2 t'(x))$, where $|A'|$ denotes the length of the description of A'. Thus, the running time of A on input x, denoted $t(x)$, is at most $\sum_{j=1}^{i(x)} \alpha(j) \cdot 2^j$, and for sufficintly large x it holds that $t'(x) \geq |A'|$. Using (say) $\alpha(j) = j$, it follows that $t(x) = O(t'(x) \cdot \log t'(x))$, which almost establishes the theorem (while we don't care about establishing it as stated). ■

Historical Notes

Many sources provide historical accounts of the developments that led to the formulation of the *P vs NP Problem* and the development of the theory of NP-completeness. We thus refrain from attempting to provide such an account.

One technical point that we mention is that the three "founding papers" of the theory of NP-completeness use the three different terms of reductions used above. Specifically, Cook uses the general notion of polynomial-time reduction [17], often referred to as Cook-reductions. The notion of Karp-reductions originates from Karp's paper [47], whereas its augmentation to search problems originates from Levin's paper [57]. It is worth noting that unlike Cook and Karp's works, which treat decision problems, Levin's work is stated in terms of search problems.

The existence of NP-sets that are neither in P nor NP-complete (i.e., Theorem 3.2) was proven by Ladner [55], and the existence of optimal search algorithms for NP-relations (i.e., Theorem 3.4) was proven by Levin [57].

Lecture Series II

The most traditional material

The partition of the rest of the lectures into two lecture series is only due to historical reasons. We start with the more traditional material, most of it is due to the 1970's and the early 1980's.

Notation: We will try to always use n to denote the length of the (main) input.

Lecture 4

Complexity classes defined by a sharp threshold

There is something appealing in defining complexity classes according to a sharp threshold like the class of problems that can be solved within time t, for some function t (e.g., $t(n) = n^3$). Contrast this definition with the class of problems that can be solved within some time t that belongs to a class of functions T (e.g., polynomials). The problem with classes defined according to a (single) sharp threshold is that they are very sensitive to the specific model of computation and may not be closed under natural algorithmic operations. Typically, these problems do not occur when defining complexity classes that correspond to a resource bounded by a class of functions, provided this class has some desirable closure properties.

4.1 Definitions

Focusing on two natural complexity measures (i.e., time and space), we may define for each function $f : \mathsf{N} \to \mathsf{N}$ classes such as $\textsc{Dtime}(f)$ and $\textsc{Dspace}(f)$ corresponding to the class of decision problems that are solvable within time and space complexity f, respectively. (That is, on input x the deciding algorithm runs for at most $f(|x|)$ steps or uses at most $f(|x|)$ bits of storage.)

We stress that when measuring the space complexity of the algorithm, we don't allow it to use its input and/or output device (i.e., tape in case of Turing machines) as temporary storage. (This is done by postulating that the input and output devices are read-only and write-only respectively.)

Note that classes as above are very sensitive to the specific model of computation. For example, the time complexity of multiple-tape Turning machines may be quadratic (but not more) in the time complexity of single-tape Turning machines (e.g., consider the set $\{xx : x \in \{0,1\}\}$).

4.2 Hierarchies and Gaps

A natural property that we may expect from complexity measures is that more resources allow for more computations. That is, if g is *sufficiently greater* than f then the class of problems solvable in time (or space) g should be *strictly larger* than the class of problems solvable in time (or space) f. This property (corresponding to a time or space hierarchy) does hold in the natural cases, where the key question is what is *sufficiently greater*. The answer will be clarified from the way such hierarchy theorems are proved, which is by using *diagonalization*.

Specifically, suppose we want to prove that $\textsc{Dtime}(g)$ is a strict superset of $\textsc{Dtime}(f)$. This is done by "diagonalizing against all f-time machines". That is, we construct a set L *along with a decision algorithm for it*, such that no f-time machine can correctly decide L. In order to "effectively" define L, we should be able to emulate the execution of each f-time machine. Since we cannot effectively enumerate all f-time machines, what we do instead is emulate each possible machine while using a time-up mechanism that stops the emulation at time f. In order to do this we need, in particular, to be able to compute f (relatively fast). Note that the running-time of our decision procedure for L is determined by the time it takes to compute f and the time it takes to emulate a given number of steps. Thus, *time constructible functions*, play a central rule in such proofs, where f is time constructible if on input n the value $f(n)$ can be computed within time $f(n)$. As for the emulation overhead, it depends on the specific model of computation (typically, t steps can be emulated within time $t \log t$). Similar considerations apply to space hierarchies, but here one talks about *space constructible function* and the emulation overhead is typically only linear. For simplicity, in case of multiple-tape Turing machines, we get:

Theorem 4.1 (sketch): *For any time-constructible function t, the class* $\textsc{Dtime}(t)$ *is strictly contained in* $\textsc{Dtime}(\omega(t \log t))$. *For any space-constructible function s, the class* $\textsc{Dspace}(s)$ *is strictly contained in* $\textsc{Dspace}(\omega(s))$.

The existence of functions that are not time (or space) constructible is the reason for so-called gap theorems. Typically, such theorems say that there exist functions f (which are certainly not time constructible) such that $\textsc{Dtime}(f) = \textsc{Dtime}(f^3)$ (or $\textsc{Dtime}(f) = \textsc{Dtime}(2^{2^f})$). The reason for this phenomena is that (for such a function f) there are no machines that run more than time f but less than time f^3 (or 2^{2^f}).

Lecture 5

Space Complexity

Space complexity is aimed to measure the amount of temporary storage required for a computational task. On one hand, we don't want to count the input and output themselves within the space of computation, but on the other hand we have to make sure that the input and output device cannot be abused to provide work space (which is uncounted for). This leads to the convention of postulating that the input device (e.g., a designated input-tape of a multi-tape Turing machine) is read-only, whereas the output device (e.g., a designated output-tape of a such machine) is write-only. Space complexity accounts for the amount of space on other (storage) devices (e.g., the work-tapes of a multi-tape Turing machine) that is used throughout the computation.

5.1 Deterministic space complexity

Only regular languages can be decided in constant space. This follows by combining two facts. Firstly, constant-space Turing machines are equivalent to a generalization of finite automata that can scan (parts of the) input back and forth (in both directions and for several times). Second, the latter "sweeping automata" can be simulated by ordinary finite automata (which scan the input only once, from left to right).

At first glance one may think that sub-logarithmic (deterministic) space is not more useful than constant space, because it *seems* impossible to allocate a sub-logarithmic amount of space (since measuring the input length requires logarithmic space). However, this intuition is wrong, because the input itself (in case it is of the proper form) can be used to determine its length, whereas in case the input is not of the proper form this fact may be detectable (within sub-logarithmic space). In fact:

Theorem 5.1 $\text{DSPACE}(o(\log n))$ *is a proper superset of* $\text{DSPACE}(O(1))$.

One proof consists of presenting a *double-logarithmic space* algorithm for recognizing the *non-regular* set $L = \{x_k : k \in \mathsf{N}\} \subset \{0, 1, *\}$, where x_k equals the concatenation of all k-bit long strings (in lexicographic order) separated by $*$'s (i.e., $x_k = 0^{k-2}00 * 0^{k-2}01 * 0^{k-2}10 * 0^{k-2}11 * \cdots * 1^k$). Note that $|x_k| > 2^k$, and we claim that x_k can be recognized in space $O(\log k) = O(\log\log|x_k|)$. Furthermore, the membership of any x in L can be determined in space $O(\log\log|x|)$, by iteratively checking (in space $O(\log i)$) whether $x = x_i$, for $i = 1, 2, \ldots$. (Details are left as an exercise.) In contrast to Theorem 5.1, double-logarithmic space is indeed the smallest amount of space that is more useful than constant space; that is:

Theorem 5.2 $\text{DSPACE}(o(\log\log n)) = \text{DSPACE}(O(1))$.

The proof proceeds by considering, for each input location, the sequence of (storage) configurations of the machine at all times that it crosses this input location. For starters, the length of this "crossing sequence" is upper-bounded by the number of possible storage configurations (i.e., in case of Turing machines, we consider the contents of the tape and the head location), which is at most $t \stackrel{\text{def}}{=} 2^{s(n)} \cdot s(n)$, where s is the machine's space complexity. Thus, the number of such sequences is bounded above by t^t. But if the latter is smaller than $n/2$ then there exist three input locations that have the same sequence of configurations. Using cut-and-paste, we get a shorter input on which the machine used space $s' \stackrel{\text{def}}{=} s(n)$, which is not possible in case the original n-bit long input was the shortest one on which the machine uses space at least s'. We conclude that $t^t \geq n/2$ must hold, and $s(n) = \Omega(\log t) = \Omega(\log \log n)$ follows.

Logarithmic Space. Although Theorem 5.1 asserts that "there is life under log-space", logarithmic space will be the lowest space-complexity class that we will care about. The class of sets recognizable by deterministic machines that use logarithmic space is denoted $\mathcal{L}$; that is, $\mathcal{L} \stackrel{\text{def}}{=} \cup_c \text{DSPACE}(c \log_2 n)$.

Theorem 5.3 *$\mathcal{L} \subseteq \mathcal{P}$.*

In general, if s is at least logarithmic and is computable within time 2^s then $\text{DSPACE}(s) \subseteq \text{DTIME}(2^s)$. This follows as a special case from Theorem 5.6. (The phenomena that time relates exponentially to space occurs also in other settings.)

Another class of important log-space computations is the class of logarithmic space reductions; that is, reductions (or oracle machines) that use only logarithmic space (and as usual polynomial-time). In accordance with our conventions regarding input and outputs, we stress that the queries (resp., answers) are written on (resp., read from) a special device/tape that is write-only (resp., read-only) for the calling algorithm and read-only (resp., write-only) for the invoked oracle. We observe that all known (Karp-)reductions establishing NP-completeness results are in fact logarithmic space. Observe that if L' is log-space reducible to L'' and $L'' \in \mathcal{L}$ then so is L'. (See Section 5.2.3.)

Polynomial Space. As stated above, we will rarely treat computational problems that require less than logarithmic space. On the other hand, we will rarely treat computational problems that require more than polynomial space. The class of decision problems that are solvable in polynomial-space is denoted $\mathcal{PSPACE} \stackrel{\text{def}}{=} \cup_c \text{DSPACE}(n^c)$. A complete problem for $\mathcal{PSPACE}$ is presented in Section 6.5.

5.2 Non-deterministic space complexity

5.2.1 Two models of non-determinism

We discuss two models of non-deterministic machines. In the standard model, called the on-line model, the machine makes non-deterministic "on the fly" (or alternatively reads a non-deterministic input from a read-only tape *that can be read only in a uni-directional way*). Thus, if the machines wants to refer to such a non-deterministic choice at a latter stage then it must store the choice on its storage device (and be charged for it). In the so-called off-line model, the non-deterministic choices (or the bits of the non-deterministic input) are read from a special read-only record (or tape) *that can be scanned in both directions like the main input.* Although the off-line model fits better the motivations to $\mathcal{NP}$ (as presented in the first lecture), the on-line model seems more adequate for the study of non-deterministic in the context of space complexity. The latter thesis is based on observing that an off-line non-deterministic tape can be used to code computations, and in a sense allows to "cheat" with respect to the "real" space complexity of the computation. This is reflected in the fact that the off-line model can simulate the on-line model while using space that is logarithmic in the space used by the on-line model.[1] This result is tight: the on-line model can simulate the off-line model using (only) exponentially more space.

Theorem 5.4 (relating the two models, loosely stated:) *For* $s : \mathbb{N} \to \mathbb{N}$ *that is nice and at least logarithmic,* $\text{NSPACE}_{\text{on-line}}(s) = \text{NSPACE}_{\text{off-line}}(\log s)$.

To simulate the on-line model on the off-line model, use the non-deterministic input tape of the latter to encode an accepting computation of the former (i.e., a sequence of consecutive configurations leading from the initial configuration to an accepting configuration). The simulating machine (which verifies the legitimacy of the sequence of configurations recorded on the non-deterministic input tape) needs only store its *location within the current pair of configurations* that it examines, which requires space logarithmic in the length of a single configuration. On the other hand, the simulation of the off-line model by the on-line model uses a crossing-sequence argument. For starters, one shows that the length of such sequences is at most double-exponential in the space complexity of the off-line machine. Then the (on-line) non-deterministic input tape is used to encode the sequence of crossing-sequences, and the on-line machine checks that each consecutive pair is consistent. This requires holding one (or two) crossing-sequences in storage, which require space logarithmic in the number of such sequences (which, in turn, is exponential in the space complexity of the off-line machine).

5.2.2 Some basic facts about NSPACE

We let $\text{NSPACE}(s) \stackrel{\text{def}}{=} \text{NSPACE}_{\text{on-line}}(s)$, and focus on $\mathcal{NL} \stackrel{\text{def}}{=} \text{NSPACE}(O(\log n))$. Suitable *upwards-translation lemmas* can be used to translate simulation results concerning $\mathcal{NL}$ (resp., $\mathcal{L}$) into general simulation results concerning non-deterministic (resp., deterministic) space. Typically, the input is padded till the concrete space allowance becomes logarithmic in the padded input (i.e., n-bit long inputs are padded to length N such that $s(n) = \log N$). Next the simulation result is applied, and finally the complexity of the obtained simulation is stated in terms of the original input length. A notable property of $\mathcal{NL}$ is that this class has a very natural complete problem:

Theorem 5.5 (Directed Connectivity is NL-complete:) *Directed Connectivity is in* $\mathcal{NL}$*, and every problem in* $\mathcal{NL}$ *is reducible to Directed Connectivity by a log-space reduction.*

Proof Sketch: A non-deterministic log-space machine may decide Directed Connectivity by guessing (and verifying) the directed path "on-the-fly". To reduce $L \in \mathcal{NL}$ to Directed Connectivity, we consider the non-deterministic log-space machine that decides L. We observe that on input x, this machine uses $\ell \stackrel{\text{def}}{=} O(\log |x|)$ space, and it may be in one out of $2^\ell \cdot |x| \cdot \ell$ possible configurations (accounting for the possible contents of its work-tape and its head locations (on the input tape and work tapes)). Consider a directed graph with these configuration as vertices, and directed edges connecting ordered pairs of possibly-consecutive configurations (relating to a possible non-deterministic move). Indeed, unlike the vertices, the edges depend on the input x. Observe that

[1] A related phenomenon is that $\text{NSPACE}_{\text{off-line}}(s)$ is only known to be contained in $\text{DTIME}(2^{2^s})$, whereas (as stated in Theorem 5.6) $\text{NSPACE}_{\text{on-line}}(s) \subseteq \text{DTIME}(2^s)$. In fact, the power of the off-line model emerges from the fact that its running time is not bounded (even not "without loss of generality") by an exponent in the space-complexity.

$x \in L$ if and only if there exists a directed path in this graph leading from the initial configuration to an accepting configuration. Furthermore, this graph can be constructed in logarithmic space (from the input x). Thus, L is log-space reducible to Directed Connectivity. ■

Theorem 5.6 (non-deterministic space versus deterministic time): *If s is at least logarithmic and is computable within time 2^s then* $\text{NSPACE}(s) \subseteq \text{DTIME}(2^s)$.

Proof Sketch: By a suitable upwards-translation lemma, it suffices to prove the result for logarithmic s; that is, we need to show that $\mathcal{NL} \subseteq \mathcal{P}$. Using Theorem 5.5, we just need to show that directed connectivity can be solved in polynomial-time. This fact is well known (e.g., by the directed-DFS algorithm). ■

Theorem 5.7 (Non-deterministic versus deterministic space): $\text{NSPACE}(s) \subseteq \text{DSPACE}(s^2)$ *provided that $s : \mathsf{N} \rightarrow \mathsf{N}$ is space-constructible and at least logarithmic.*

In particular, for any polynomial p, it holds that $\text{NSPACE}(p) \subset \mathcal{PSPACE}$, where the strict inclusion is due to the space hierarchy theorem (e.g., $\text{DSPACE}(n^c) \subset \text{DSPACE}(n^{c+1})$). Contrast Theorem 5.7 with the (trivial) fact that $\text{NTIME}(t) \subseteq \text{DTIME}(2^t)$, provided that $t : \mathsf{N} \rightarrow \mathsf{N}$ is time-constructible (and at least logarithmic).

Proof Sketch: Again, it suffices to show that directed connectivity can be solved in deterministic $O(\log n)^2$ space. The basic idea is that checking whether or not there is a path of length at most ℓ from u to v in G, reduces (in log-space) to checking whether there is an intermediate vertex w such that there is a path of length at most $\lceil \ell/2 \rceil$ from u to w and a path of length at most $\lfloor \ell/2 \rfloor$ from w to v. Let $p_G(u,v,\ell) \stackrel{\text{def}}{=} 1$ if there is a path of length at most ℓ from u to v in G and $p_G(u,v,\ell) \stackrel{\text{def}}{=} 0$ otherwise. Thus, $p_G(u,v,\ell)$ can be decided recursively by scanning all vertices w in G, and checking for each w whether both $p_G(u,w,\lceil \ell/2 \rceil) = 1$ both $p_G(w,v,\lfloor \ell/2 \rfloor) = 1$ hold.

Thus, suppose we are given a directed graph G and a pair of vertices (s,t), and should decide whether or not there is a path from s to t in G. Let n denote the number of vertices in G, then we need to compute $p_G(s,t,n)$. This is done by invoking a recursive procedure that computes $p_G(u,v,\ell)$ by scanning all vertices in G, and computing for each vertex w the value of $p_G(u,w,\lceil \ell/2 \rceil) \cdot p_G(w,v,\lfloor \ell/2 \rfloor)$. The amount of space taken by each level of the recursion is $\log n$ (for storing the current value of w), and the number of levels is $\log n$. The theorem follows.

We stress that when computing $p_G(u,v,\ell)$, we make polynomially many recursive calls, but all these calls re-use the same work space. That is, when we compute $p_G(u,w,\lceil \ell/2 \rceil) \cdot p_G(w,v,\lfloor \ell/2 \rfloor)$ we re-use the space that was used for computing $p_G(u,w',\lceil \ell/2 \rceil) \cdot p_G(w',v,\lfloor \ell/2 \rfloor)$ (for the previous w'). Furthermore, when we compute $p_G(w,v,\lfloor \ell/2 \rfloor)$ we re-use the space that was used for computing $p_G(u,w,\lceil \ell/2 \rceil)$. ■

5.2.3 Composition Lemmas

Indeed, as indicated by the proof of Theorem 5.7, space (unlike time!) can be re-used. In particular, if one machine makes many recursive calls to another machine then the cost in space of these calls is the *maximum space used by a single call* (whereas the cost in terms of time of these calls is the *sum of the time taken by all calls*). Put in other words: *Suppose that L_1 is s_1-space reducible to L_2 and that L_2 is in $XSPACE(s_2)$, where $X \in \{D,N\}$. Then L_1 is in $XSPACE(s_1 + s_2')$*, where $s_2'(n) = s_2(2^{s_1(n)})$ (because $2^{s_1(n)}$ is an obvious bound on the length of queries made to L_2). Proving this claim is less trivial than it seems (even in case of a single call to L_2) because we cannot afford to store the query and the answer (which may have lengths 2^{s_1} and $2^{s_2 \circ s_1}$, respectively) in the working space of the resulting machine.

For simplicity, we focus on the single-query case.[2] Let M_1 be the reduction of L_1 to L_2 and M_2 a machine solving L_2. We emulate them both as follows. We allocate each of the machines a separate work-tape, and begin by emulating M_2 without specifying its input. When M_2 wishes to read the i^{th} (e.g., first) bit of its input (which is the i^{th} bit of the query of M_1), we run machine M_1 until it produces the i^{th} bit of its query, which we hand to M_2. We stress that we do not store all previous bits of this query, but rather discard them. Thus, we run a new emulation of M_1, per each time that M_2 wishes to read a bit of its input (i.e., the query directed to it by M_1). When M_2 outputs its decision, we store it, and emulate M_1 for the last time. In this run we discard all the query bits produced by M_1, feed it with M_2's answer, and output whatever M_2 does.

The treatment of reductions to search problems is more complex, because (unless postulated differently) the calling algorithm may scan the answer provided by the oracle back and forth (rather than read it once from left to right). To treat this case we may keep two emulations of M_1, one for producing bits of the query and the other for using the bits of the answer. (Note that the second emulation corresponds to the last emulation of M_1 in the description above.) Handling of many oracle calls is performed in a query by query manner, relying of the fact that the i^{th} answer is not available after the $i+1^{\text{st}}$ query is made. For $i = 1, 2, ...$, we handle the $i+1^{\text{st}}$ query-answer by keeping a record of the temporary configurations of M_1 before it started making the i^{th} and $i+1^{\text{st}}$ queries. We maintain four emulations of M_1, the first (resp., third) for producing bits of the i^{th} (resp., $i+1^{\text{st}}$) query, and the second (resp., fourth) for using the bits of the i^{th} (resp., $i+1^{\text{st}}$) answer. Each time we need to emulate the first or second (resp., third or fourth) copy, we *start the emulation from the recorded configuration* of M_1 before making the i^{th} (resp., $i+1^{\text{st}}$) query. Once the fourth copy starts to produce the $i+2^{\text{th}}$ query, we refresh all configurations and move to the next iteration. Specifically, the configuration of the fourth copy will be used as the second temporary configuration (as it corresponds to the configuration before making the $i+2^{\text{th}}$ query), and the current second configuration (which corresponds to the configuration before making the $i+1^{\text{st}}$ query) will be used as the first temporary configuration (for iteration $i+1$).

Teaching Note. In the next subsection we will (implicitly) use a composition result, but for that specific composition we do not need the power of the above strong composition lemma. Specifically, the reduction will make queries that are very related to its input (and thus the invoked subroutine can form the query by itself from the input). Furthermore, the answers will be of logarithmic length and thus can be stored by the reduction (as in case of invoking a decision subroutine).

5.2.4 NSPACE is closed under complementation

People tend to be discouraged by the impression that "decades of research have failed to answer any of the famous open problems of complexity theory". In addition to the fact that substantial progress towards the understanding of some open problems has been achieved, people tend to forget that some famous open problems were indeed resolved. The following result relates to a famous question that was open for three decades.[3]

[2] We leave the extension to the general multiple-query case as an exercise.

[3] In particular, using the fact that the class of sets recognized by linear-space non-deterministic machines equals the set of context-sensitive languages, Theorem 5.8 resolves the question of whether the latter set is closed under complementation. This question has been puzzling researchers since the early days of research in the area of formal languages (i.e., the 1950's). We mention that Theorem 5.8 was proven in the late 1980's.

Theorem 5.8 *$\mathcal{NL} = \text{co}\mathcal{NL}$, where* $\text{co}\mathcal{NL} \stackrel{\text{def}}{=} \{\{0,1\}^* \setminus L : L \in \mathcal{NL}\}$.

Again, using an adequate upwards-translation lemma, one can derive the closure under complementation of $\text{NSPACE}(s)$.

Proof Sketch: It suffices to show that *directed unconnectivity* (the complementation of directed connectivity) can be decided in $\mathcal{NL}$. That is, we will present a non-deterministic log-space machine M such that

- If there is *no directed path* from s to t in G then there exists a computation of M that accepts the input (G, s, t).
- If *there is a directed path* from s to t in G then all possible computations of M reject (G, s, t).

The above decision problem is log-space reducible to determining the number of nodes that are reachable from a given vertex in a given graph.[4] Thus, we focus on providing a non-deterministic log-space machine that compute the said quantity, where we say that a non-deterministic M computes the function $f : \{0,1\}^* \to \{0,1\}^*$ if the following two conditions hold:

1. For every x, there exists a computation of M that halts with output $f(x)$.
2. For every x, all possible computation of M either halt with output $f(x)$ or halt with a special "dont know" symbol, denoted $\perp$.

Fixing an n-vertex graph $G = (V, E)$ and a vertex v, we consider the set of vertices that are reachable from v by a path of length at most i. We denote this set by R_i, and observe that $R_0 = \{v\}$ and that for every $i = 1, 2, ...$, it holds that

$$R_i = R_{i-1} \cup \{u : \exists w \in R_{i-1} \text{ s.t. } (w, u) \in E\} \tag{5.1}$$

Our aim is to compute $|R_n|$. This will be done in n iterations such that at the i^{th} iteration we compute $|R_i|$. When computing $|R_i|$ we rely on the fact that $|R_{i-1}|$ is known to us, which means that we'll store $|R_{i-1}|$ (but not previous $|R_j|$'s) in memory. Our non-deterministic guess, denoted g, for $|R_i|$ will be verified as follows:

- $|R_i| \geq g$ is verified in the straightforward manner. That is, scanning V, we guess for g vertices paths of length at most i from v to them, and verify these "on-the-fly". (Indeed, we also guess for which g vertices to verify this fact.)

  (We use $\log_2 n$ bits to store the currently scanned vertex, another $\log_2 n$ bits to store an intermediate vertex on a path from v, and another $\log_2 i \leq \log_2 n$ bits to store the distance traveled so far.)

- The verification of $|R_i| \leq g$ is the interesting part of the procedure. Here we rely on the fact that we know $|R_{i-1}|$. Scanning V (again), we verify for $n - g$ (guessed) vertices that they are **not** reachable from v by paths of length at most i. Verifying that $u \notin R_i$ is done as follows.

  - We scan V guessing $|R_{i-1}|$ vertices that are in R_{i-1}, and verify each such guess in the straightforward manner. (Implicit here is a procedure that given G, v, i and $|R_{i-1}|$, produces R_{i-1} itself.)

[4] Exercise: provide such a reduction.

- For each $w \in R_{i-1}$, which was guessed and verified above, we verify that both $u \neq w$ and $(w,u) \notin E$.

By Eq. (5.1), if u passes the above verification then indeed $u \notin R_i$.

(We use $\log_2 n$ bits to store u, another $\log_2 n$ bits to count the number of vertices verified to be in R_{i-1}, another $\log_2 n$ bits to store such w, and another $2\log_2 n$ bits for verifying that $w \in R_{i-1}$.)

If any of the verifications fails, the machine halts outputting the "dont know" symbol. Exercise: assuming that the correct value of $|R_{i-1}|$ is used, prove that the above non-deterministic log-space procedure computes the value of $|R_i|$.

Observing that when computing $|R_i|$ we only need to know $|R_{i-1}|$ (and do not need $|R_j|$ for any $j < i-1$), the above yields a non-deterministic log-space machine for computing $|R_n|$. The theorem follows. ■

Lecture 6

The Polynomial-Time Hierarchy

The Polynomial-Time Hierarchy (PH) is a hierarchy of complexity classes that extends $\mathcal{NP}$. We will present two equivalent ways of defining this hierarchy, and discuss some of its properties.

6.1 Defining PH via quantifiers

Recall that $L \in \mathcal{NP}$ if there exists a *binary* (polynomially-bounded) relation R such that R is polynomial-time recognizable and

$$x \in L \text{ if and only if } \exists y \text{ s.t. } (x, y) \in R \tag{6.1}$$

Identifying $\mathcal{NP}$ with Σ_1, we define Σ_2 as containing sets L such that there exists a *3-ary* (polynomially-bounded)[1] relation R such that R is polynomial-time recognizable and

$$x \in L \text{ if and only if } \exists y_1 \forall y_2 \text{ s.t. } (x, y_1, y_2) \in R \tag{6.2}$$

(Above and below, it is important to stress that the universal quantifiers range only over strings of the adequate length.)

In general, Σ_i is defined as the class consisting of sets L such that there exists a $(i+1)$-*ary* (polynomially-bounded)[2] relation R such that R is polynomial-time recognizable and

$$x \in L \text{ if and only if } \exists y_1 \forall y_2 \cdots Q_i y_i \text{ s.t. } (x, y_1, ..., y_i) \in R \tag{6.3}$$

where Q_i is an existential (resp., universal) quantifier in case i is odd (resp., even). That is, we have i alternating quantifiers, starting with an existential one, where each quantifier ranges over strings of length polynomial in the length of x. (Note that indeed $\Sigma_1 = \mathcal{NP}$.)

Similarly, we can define classes referring to alternating sequences of quantifiers starting with a universal quantifier. Specifically, $L \in \Pi_i$ if there exists a $(i+1)$-*ary* (polynomially-bounded) relation R such that R is polynomial-time recognizable and

$$x \in L \text{ if and only if } \forall y_1 \exists y_2 \cdots Q_i y_i \text{ s.t. } (x, y_1, ..., y_i) \in R \tag{6.4}$$

where Q_i is an existential (resp., universal) quantifier in case i is even (resp., odd).

The polynomial-time hierarchy, denoted $\mathcal{PH}$, is defined as $\cup_i \Sigma_i$. That is, $L \in \mathcal{PH}$ means that there exists an i such that $L \in \mathcal{PH}$.

[1] We say that R is polynomially-bounded if there exists a polynomial p such that for every $(x, y_1, y_2) \in R$ it holds that $|y_1| + |y_2| \leq p(|x|)$.

[2] Indeed, we say that R is polynomially-bounded if there exists a polynomial p such that for every $(x, y_1, ..., y_i) \in R$ it holds that $\sum_{j=1}^{i} |y_j| \leq p(|x|)$.

Exercises: the following facts can be verified by purely syntactic considerations:

1. For every $i \geq 1$, it holds that $\Pi_i = \text{co}\Sigma_i$ (and, in particular, $\Pi_1 = \text{co}\mathcal{NP}$).
2. For every $i \geq 1$, it holds that $\Sigma_i \subseteq \Pi_{i+1}$ and $\Pi_i \subseteq \Sigma_{i+1}$. Thus, $\mathcal{PH} = \cup_i \Pi_i$.
3. For every $i \geq 1$, it holds that $\Sigma_i \subseteq \Sigma_{i+1}$.

It is widely believed that Σ_i is a strict subset of Σ_{i+1}. See further discussion in Section 6.4.

Another Exercise: Prove that $\mathcal{PH}$ is contained in $\mathcal{PSPACE}$. See further discussion in Section 6.5.

Complete sets in PH: The above definition of Σ_i (and Π_i) gives rise to ("semi-natural") complete sets for Σ_i (and Π_i). For example, consider the set of Boolean circuits of the form C such that C takes as input i equal-length strings, denoted $x_1, ..., x_i$, and it holds that $\exists x_1 \forall x_2 \cdots Q_i x_i \, C(x_1, ..., x_i) = 1$, where Q_i is an existential (resp., universal) quantifier in case i is odd (resp., even). Clearly, this set is in Σ_i, and every set in Σ_i is Karp-reducible to this set.[3] (Hint: the x_i's correspond to the y_i's in the definition of Σ_i, whereas C corresponds to x.)

Natural Examples of sets in PH: Recall that natural NP-optimization problems are captured by NP-sets that refer only to a (one-sided) bound on the value of the optimum. For example, whereas the optimization version of `maxClique` requires to find the largest clique in a given graph, the decision problem is to tell whether or not the largest clique has size *greater than or equal to* a given number.[4] Clearly, the latter decision problem is in $\mathcal{NP}$, whereas its complement (i.e., determining whether the largest clique has size *smaller than* a given number) is in co$\mathcal{NP}$. But what about determining whether the largest clique has size *equal to* a given number? (That is, the set we refer to is the set of pairs (G, K) such that the size of the largest clique in G equals K.) Note that this problem is unlikely to be in either $\mathcal{NP}$ or co$\mathcal{NP}$ (because this will imply $\mathcal{NP} = \text{co}\mathcal{NP}$),[5] but it is certainly in Σ_2 (and in Π_2). (Exercise: Present adequate 3-ary relations for the above set.) See further discussion in the next section.

6.2 Defining PH via oracles

Recall that the general notion of a reduction is based on augmenting a (deterministic) polynomial-time machine with oracle access. A natural question is what languages can be recognized by such machines when the oracle is an arbitrary NP-set (or, equivalently an NP-complete set like SAT).[6] We denote this class by $\mathcal{P}^{\mathcal{NP}}$, standing for an arbitrary "P-machine" given oracle access to some NP-set. (As indicated below, $\mathcal{P}^{\mathcal{NP}}$ is likely to be a proper superset of $\mathcal{NP}$, whereas the class of languages that are Karp-reducible to $\mathcal{NP}$ equals $\mathcal{NP}$.)

[3] The reason that we insist on Karp-reductions here will become clear below.

[4] Actually, the decision problem is typically phrased as determining whether there exists a clique of size greater than or equal to a given number.

[5] If we could have given an NP-proof that the max-clique has size equal to a given number, then we could prove that it is strictly smaller than a given number, which is a coNP-complete problem (and $\mathcal{NP} = \text{co}\mathcal{NP}$ would follow).

[6] Exercise: Show that these two formulations are indeed equivalent.

Comment: The notation $\mathcal{P}^{\mathcal{NP}}$ is consistent with the standard notation for oracle machines. That is, for an oracle machine M, (oracle) set L and string x, we let $M^L(x)$ denote the output of M on input x and oracle access to L. (Thus, when we said that L' is reduced to L we meant that there exists a polynomial-time oracle machine M such that for every x it holds that $M^L(x) = 1$ if and only if $x \in L'$.) Thus,

$$\mathcal{P}^{\mathcal{NP}} = \{L(M^{SAT}) : M \text{ is a "P-machine"}\}$$

where $L(M^{SAT})$ denotes the set of inputs that are accepted by M when given oracle access to SAT.

Exercises:

1. Show that both $\mathcal{NP}$ and co$\mathcal{NP}$ are subsets of $\mathcal{P}^{\mathcal{NP}}$.
2. In contrast, prove that the class of languages that are Karp-reducible to $\mathcal{NP}$ equals $\mathcal{NP}$.
3. Following the above discussion, define $\mathcal{P}^{\text{co}\mathcal{NP}}$ and show that it equals $\mathcal{P}^{\mathcal{NP}}$.
4. Referring to the set of pairs (G, K) such that the size of the largest clique in G equals K, show that this set is in $\mathcal{P}^{\mathcal{NP}}$.

The definition of $\mathcal{P}^{\mathcal{NP}}$ suggests that we may define also classes such as $\mathcal{NP}^{\mathcal{NP}}$. Note that such a definition does not yield a natural notion of a reduction (to NP-sets), because the "reduction" is non-deterministic. Still, a well-defined class does emerge. Specifically, $\mathcal{NP}^{\mathcal{NP}}$ is the class of sets that are accepted by a non-deterministic polynomial-time oracle machine that is given access to some NP-set. Observe that indeed $\mathcal{P}^{\mathcal{NP}} \subseteq \mathcal{NP}^{\mathcal{NP}}$.

Defining Σ_i by oracles: As before, we let $\Sigma_1 \stackrel{\text{def}}{=} \mathcal{NP}$. For $i \geq 1$, we define

$$\Sigma_{i+1} \stackrel{\text{def}}{=} \mathcal{NP}^{\Sigma_i} . \tag{6.5}$$

Indeed, Σ_2 so defined equals $\mathcal{NP}^{\mathcal{NP}}$. As we will show in the next section, the Σ_i's as defined here coincide with the classes defined in the previous section.

A general perspective – what does ${\mathcal{C}_1}^{\mathcal{C}_2}$ mean? By the above discussion it should be clear that the class ${\mathcal{C}_1}^{\mathcal{C}_2}$ can be defined for any two complexity classes $\mathcal{C}_1$ and $\mathcal{C}_2$, *provided that $\mathcal{C}_1$ is associated with a class of machines that extends naturally to access oracles.* Actually, the class ${\mathcal{C}_1}^{\mathcal{C}_2}$ *is not defined based on the class $\mathcal{C}_1$ but rather by analogy to it.* Specifically, suppose that $\mathcal{C}_1$ is the class of sets recognizable by machines of certain type (e.g., deterministic or non-deterministic) with certain resource bounds (e.g., time and/or space bounds). Then we consider analogous oracle machines (i.e., of the same type and with the same resource bounds), and say that $L \in {\mathcal{C}_1}^{\mathcal{C}_2}$ if there exists such an oracle machine M_1 and a set $L_2 \in \mathcal{C}_2$ such that $M_1^{L_2}$ accepts the set L.

Exercise: For $\mathcal{C}_1$ and $\mathcal{C}_2$ as above, prove that ${\mathcal{C}_1}^{\mathcal{C}_2} = {\mathcal{C}_1}^{\text{co}\mathcal{C}_2}$. Note that, in particular, $\mathcal{NP}^{\Sigma_i} = \mathcal{NP}^{\Pi_i}$.

6.3 Equivalence of the two definitions of PH

To avoid confusion, let use denote by Σ'_i the class defined via quantifiers (i.e., in Eq. (6.3)), and by Σ''_i the class defined by oracle machines (i.e., in Eq. (6.5)).

Theorem 6.1 *For every $i \geq 1$, it holds that $\Sigma'_i = \Sigma''_i$.*

Proof Sketch: The claim holds trivially for $i = 1$, Assuming that equality holds for $i \geq 1$, we show that it holds also for $i+1$. (Each of the two inclusions uses only the induction hypothesis of the same direction.)

Assuming that $\Sigma'_i \subseteq \Sigma''_i$, we prove that $\Sigma'_{i+1} \subseteq \Sigma''_{i+1}$, by looking at an $(i+2)$-ary relation, R, for a set $L \in \Sigma'_{i+1}$. Recall that $x \in L$ iff $\exists y_1 \forall y_2 \cdots Q_{i+1} y_{i+1}$ such that $(x, y_1, ..., y_{i+1}) \in R$. Define L' as the set of pairs (x, y) such that $\forall y_2 \cdots Q_{i+1} y_{i+1}$ it holds that $(x, y, y_2, ..., y_{i+1}) \in R$. Then, $L' \in \mathrm{co}\Sigma'_i$ and $x \in L$ iff there exists a y such that $(x, y) \in L'$. By using a straightforward non-deterministic oracle machine, we obtain that $L \in \mathcal{NP}^{\mathrm{co}\Sigma'_i} = \mathcal{NP}^{\Sigma'_i}$. Using the induction hypothesis, it follows that $L \in \mathcal{NP}^{\Sigma''_i} = \Sigma''_{i+1}$.

Assuming that $\Sigma''_i \subseteq \Sigma'_i$, we prove that $\Sigma''_{i+1} \subseteq \Sigma'_{i+1}$, by looking at a non-deterministic oracle machine M that accepts a set $L \in \Sigma''_{i+1}$ when using an oracle $L' \in \Sigma''_i$. By the definition of non-uniform acceptance, it follows that $x \in L$ iff there exists a computation of M on input x that accepts, when the queries are answered according to L'. Let use denote by $M^{L'}(x, y)$ the output of M on input x and non-deterministic choices y, when its queries are answered by L'. Then, $x \in L$ iff there exists a y such that $M^{L'}(x, y) = 1$. We may assume, without loss of generality, that M starts its computation by non-deterministically guessing all oracle answers (and acting according to these guesses), and that it accept only if these guesses turned out to be correct. In other words, there exists a polynomial-time computable predicate P, such that $M^{L'}(x, y) = 1$ iff $P(x, y) = 1$ and the j^{th} answer provided by the oracle in the computation $M^{L'}(x, y)$ equals the j^{th} bit of y, denoted $y^{(j)}$. Furthermore, since M acts according to the guessed answers that are part of y, the j^{th} query of M is determined (in polynomial-time) by (x, y), and is denoted $q^{(j)}(x, y)$. We conclude that $x \in L$ iff there exists a y such that $P(x, y) = 1$ and $y^{(j)} = 1$ iff $q^{(j)} \in L'$ for every j. Using the induction hypothesis, it holds that $L' \in \Sigma''_i = \Sigma'_i$, and we let R' denote the corresponding $(i+1)$-ary relation. Thus, $x \in L$ iff

$$\exists y \left((P(x,y) = 1) \wedge \bigwedge_j \left(((y^{(j)} = 1) \Leftrightarrow \exists y_2^{(j)} \forall y_3^{(j)} \cdots Q_{i+1} y_{i+1}^{(j)} \, (q^{(j)}(x,y), y_2^{(j)}, ..., y_{i+1}^{(j)}) \in R' \right) \right)$$

The proof is completed by observing that the above expression can be rearranged to fit the definition of Σ'_{i+1}. (Hint: we may incorporate the computation of all the $q^{(j)}(x, y)$'s into the relation R', and pull all quantifiers outside.)[7] ∎

6.4 Collapses

As stated before, it is widely believed that $\mathcal{PH}$ is a strict hierarchy; that is, that Σ_i is strictly contained in Σ_{i+1} for every $i \geq 1$. We note that if a collapse occurs at some level (i.e., $\Sigma_i = \Sigma_{i+1}$

[7] Note that, for predicates P_1 and P_2, the expression $\exists y\,(P_1(y) \Leftrightarrow \exists z P_2(y, z))$ is equivalent to the expression $\exists y\,((\neg P_1(y) \vee \exists z P_2(y, z)) \wedge ((P_1(y) \vee \neg \exists z P_2(y, z)))$, which in turn is equivalent to the expression $\exists y \exists z' \forall z''\,((\neg P_1(y) \vee P_2(y, z')) \wedge ((P_1(y) \vee \neg P_2(y, z'')))$. Note that pulling the quantifiers outside in $\wedge_{j=1}^{t} \exists y^{(j)} \forall z^{(j)} P(y^{(j)}, z^{(j)})$ yields an expression of the type $\exists y^{(1)}, ..., y^{(t)} \forall z^{(1)}, ..., z^{(t)} \wedge_{j=1}^{t} P(y^{(j)}, z^{(j)})$.

for some $i \geq 1$) then the entire hierarchy collapses to that level (i.e., $\Sigma_i = \mathcal{PH}$). (This fact is best verified from the oracle-based definition, and the verification is left as an exercise.)[8] In fact, a stronger statement can be proven.

Theorem 6.2 *If $\Sigma_i = \Pi_i$ holds for some $i \geq 1$ then $\mathcal{PH} = \Sigma_i$.*

In particular, $\mathcal{NP} = \text{co}\mathcal{NP}$ implies a total collapse (i.e., $\mathcal{PH} = \mathcal{NP}$). In light of the above discussion, it suffices to show that $\Sigma_i = \Pi_i$ implies $\Sigma_i = \Sigma_{i+1}$. This is easiest to prove using the quantifier-based definition, while relying on ideas used in the previous section. Specifically, for $L \in \Sigma_{i+1}$, we first derive a set $L' \in \Pi_i$ such that $x \in L$ if and only if there exists y such that $(x, y) \in L'$. By the hypothesis $L' \in \Sigma_i$, and so $x \in L$ iff

$$\exists y \exists y_1 \forall y_2 \cdots Q_i y_i \text{ s.t. } ((x, y), y_1, y_2, ..., y_i) \in R'$$

where R' is the $(i+1)$-ary relation guaranteed for L' (w.r.t the definition of Σ_i). By joining the two leftmost existential quantifiers and slightly modifying R' (into $R'' \stackrel{\text{def}}{=} \{(x, (y, y_1), y_2, ..., y_i) : ((x, y), y_1, y_2, ..., y_i) \in R'\}$), we conclude that $L \in \Sigma_i$. ■

6.5 Comment: a PSPACE-complete problem

Recall that the complete problem of Σ_i referred to circuits that take i input strings (and to an alternating existential and universal quantification over these inputs). A natural question that arises is what happens if we drop the restriction on the number of such inputs. That is, consider the set of circuits that take a sequence of input strings, which is (of course) bounded in length by the size of the circuit. Such a circuit, denoted C, having $t = t(G)$ input strings, denoted $x_1, ..., x_t$, is in the set QC (standing for Quantified Circuits) if and only if $\exists x_1 \forall x_2 \cdots Q_t x_t \, C(x_1, ..., x_t) = 1$.

It is easy to see that $QC \in \mathcal{PSPACE}$. To show that any problem in $\mathcal{PSPACE}$ is reducible (in fact, Karp-reducible) to QC, we follow the underlying idea of the proof of Theorem 5.7. That is, let $L \in \mathcal{PSPACE}$, let M be the corresponding polynomial-space machine, and p be the corresponding polynomial space-bound. For any $x \in \{0,1\}^n$, it holds that $x \in L$ iff M passes in at most $2^{p(n)}$ steps from the initial configuration with input x, denoted $\texttt{init}(x)$, to an accepting configuration, denoted $\texttt{ACC}$. Define a Boolean predicate p_M such that $p_M(\alpha, \beta, t) = \texttt{true}$ iff M passes in at most t steps from the configuration α to configuration β. Then, we are interested in the value $p_M(\texttt{init}(x), \texttt{ACC}, 2^{p(n)})$. On the other hand, for every α and β (and $i \in \mathsf{N}$), it holds that

$$p_M(\alpha, \beta, 2^i) = \exists \gamma \; [p_M(\alpha, \gamma, 2^{i-1}) \wedge p_M(\gamma, \beta, 2^{i-1})] \tag{6.6}$$

If we were to iterate Eq. (6.6) then the length of the formula will double in each iteration, and after $\log_2 t$ iterations we'll just get a straightforward conjunction of t formulae capturing single steps of M. Our aim is to moderate the growth of the formula size during the iterations. Towards this end, we replace Eq. (6.6) by

$$\begin{aligned} p_M(\alpha, \beta, 2^i) \;\; = \;\; & \exists \gamma \forall \sigma \exists \alpha' \beta' \; [(\sigma = 0 \rightarrow (\alpha' = \alpha \wedge \beta' = \gamma)) \wedge (\sigma = 1 \rightarrow (\alpha' = \gamma \wedge \beta' = \beta)) \\ & \wedge \, p_M(\alpha', \beta', 2^{i-1}))] \end{aligned} \tag{6.7}$$

where $\sigma \in \{0,1\}$. Observe that Eq. (6.6) is equivalent to Eq. (6.7), whereas in the latter the size of the formula grows by an additive term (rather than by a factor of 2). Thus, $p_M(\texttt{init}(x), \texttt{ACC}, 2^{p(n)})$

[8] Assuming that $\Sigma_i = \Sigma_{i+1}$, we prove by induction on $j > i$ that $\Sigma_j = \Sigma_i$. In the induction step, we have $\Sigma_{j+1} = \mathcal{NP}^{\Sigma_j} = \mathcal{NP}^{\Sigma_i} = \Sigma_{i+1} = \Sigma_i$.

can be written as a *quantified boolean formula* with $O(\log t)$ (alternating) quantifiers. The formula being quantified over will be a conjunction of ($O(\log t)$) simple logical conditions (of the type introduced in Eq. (6.7)) as well as (a single occurrence of) the formula $p_M(\cdot, \cdot, 1)$. Hence, we have actually established the PSPACE-hardness of a special case of QC corresponding to Quantified Boolean Formulae, denoted QBF.

Lecture 7

Randomized Complexity Classes

So far, our approach to computing devises was somewhat conservative: we thought of them as (repeatedly) executing a deterministic rule. A more liberal and quite realistic approach pursued in this lecture considers computing devices that use a probabilistic (or randomized) rule. Specifically, we allow probabilistic rules that choose uniformly among two predetermined possibilities, and observe that the effect of more general probabilistic rules can be efficiently approximated by a rule of the former type. We still focus on polynomial-time computations, but these are *probabilistic* polynomial-time computations. Indeed, we extend our notion of *efficient computations* from *deterministic* polynomial-time computations to *probabilistic* polynomial-time computations.

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

1. The most liberal notion is of machines with *two-sided error probability*. In case of search problems, it is required that the correct answer is output with probability that is significantly greater than 1/2 (e.g., probability at least 2/3). When this approach is applied to decision problems (solvable by probabilistic polynomial-time machines), we get the class $\mathcal{BPP}$, standing for Bounded-error Probabilistic Polynomial-time.

2. Machines with *one-sided error probability*: In case of search problems, a natural notion is of machines that output a (correct) solution (in case such exists) with probability at least 1/2, and never output a wrong solution. In case of decision problems, there are two natural cases depending on whether the machine errs on YES-instances (but not on NO-instances), or the other way around.

3. Machines that *never err*, but may output a special `don't know` symbol, say, with probability at most 1/2.

We focus on probabilistic polynomial-time machines, and on error probability that may be reduced to a "negligible" (e.g., exponentially vanishing in the input length) amount by polynomially many independent repetitions.

We comment that an alternative formulation of randomized computations is captured by (deterministic) machines that take two inputs, the first representing the actual input and the second representing the coin tosses (or the "random input"). For such machines, one considers the output distribution for any fixed first input, when the second input is uniformly distributed among the set of strings of adequate length.

7.1 Two-sided error: BPP

The standard definition of $\mathcal{BPP}$ is in terms of machines that err with probability at most 1/3. That is, $L \in \mathcal{BPP}$ if there exists a probabilistic polynomial-time machine M such that for every $x \in L$ (resp., $x \notin L$) it holds that $\Pr[M(x) = 1] \geq 2/3$ (resp., $\Pr[M(x) = 1] \leq 1/3$). In other words, letting χ_L denote the characteristic function of L, we requite that $\Pr[M(x) \neq \chi_L(x)] \leq 1/3$ for every $x \in \{0,1\}^*$. The choice of the constant 1/3 is immaterial, and any other constant smaller than 1/2 will do (and yield the very same class). In fact, a more general statement, which is proved by so-called "amplification" (see next), holds.

Error reduction (or confidence amplification). For any function $\epsilon : \mathsf{N} \rightarrow (0, 0.5)$, consider the class BPP_ϵ of sets L such that there exists a probabilistic polynomial-time machine M for which $\Pr[M(x) \neq \chi_L(x)] \leq \epsilon(|x|)$ holds. Clearly, $\mathcal{BPP} = BPP_{1/3}$. However, a wide range of other classes also equal $\mathcal{BPP}$. In particular,

1. For every positive polynomial p, the class BPP_ϵ, where $\epsilon(n) = (1/2) - (1/p(n))$, equals $\mathcal{BPP}$. That is, any error that is ("noticeably") bounded away from 1/2 (i.e., error $(1/2) - (1/\mathrm{poly}(n))$) can be reduced to an error of 1/3.
2. For every positive polynomial p, the class BPP_ϵ, where $\epsilon(n) = 2^{-p(n)}$, equals $\mathcal{BPP}$. That is, an error of 1/3 can be further reduced to an exponentially vanishing error.

Both facts are proven by applying an adequate Law of Large Numbers. That is, consider independent copies of a random variable that represents the output of the weaker machine (i.e., the machine having larger error probability). Use the adequate Law of Large Numbers to bound the probability that the average of these independent outcomes deviates from the expected value of the original random variable. Indeed, the resulting machine will invoke the original machine sufficiently many times, and rule by majority. We stress that invoking a randomized machine several times means that the random choices made in the various invocations are independent of one another.

BPP is in the Polynomial-Time Hierarchy: Clearly $\mathcal{P} \subseteq \mathcal{BPP}$, and it is commonly conjectured that equality holds (although a polynomial slow-down may occur when transforming, according to these conjectures, a probabilistic polynomial-time algorithm into a deterministic one). However, it is not known whether or not $\mathcal{BPP}$ is contained in $\mathcal{NP}$. In view of this ignorance, the following result is of interest:

Theorem 7.1 $\mathcal{BPP} \subseteq \Sigma_2$.

Proof: Suppose that $L \in \mathcal{BPP}$, and consider (by suitable error-reduction) a probabilistic polynomial-time algorithm A such that $\Pr[A(x) \neq \chi_L(x)] < 1/3\ell(|x|)$ for all $x \in \{0,1\}^*$, where $\ell(|x|)$ denotes the number of coins tossed by $A(x)$. Let us consider the residual deterministic two-input algorithm A' such that $A'(x, r)$ equals the output of A on input x and random choices $r \in \{0,1\}^{\ell(|x|)}$. We claim that $x \in L$ if and only if

$$\exists s_1, s_2, ..., s_{\ell(|x|)} \in \{0,1\}^{\ell(|x|)} \forall r \in \{0,1\}^{\ell(|x|)} \bigvee_{i=1}^{\ell(|x|)} (A'(x, s_i \oplus r) = 1) \tag{7.1}$$

Once the claim is proved, the theorem follows by observing that Eq. (7.1) fits the definition of Σ_2. In order to prove the claim, we first consider the case $x \in L$. We use the Probabilistic Method to show that an adequate sequence of s_i's exists. That is, we show that most sequences of s_i's are adequate, by upper bounding the probability that a random sequence of s_i's is not adequate:

$$
\begin{aligned}
&\mathrm{Pr}_{s_1,s_2,\ldots,s_{\ell(|x|)}}[\neg\forall r \in \{0,1\}^{\ell(|x|)} \bigvee_{i=1}^{\ell(|x|)} (A'(x, s_i \oplus r) = 1)] \\
=\ & \mathrm{Pr}_{s_1,s_2,\ldots,s_{\ell(|x|)}}[\exists r \in \{0,1\}^{\ell(|x|)} \bigwedge_{i=1}^{\ell(|x|)} (A'(x, s_i \oplus r) \neq 1)] \\
\leq\ & \sum_{r \in \{0,1\}^{\ell(|x|)}} \mathrm{Pr}_{s_1,s_2,\ldots,s_{\ell(|x|)}}[\bigwedge_{i=1}^{\ell(|x|)} (A'(x, s_i \oplus r) \neq 1)] \\
=\ & \sum_{r \in \{0,1\}^{\ell(|x|)}} \prod_{i=1}^{\ell(|x|)} \mathrm{Pr}_{s_i}[A'(x, s_i \oplus r) \neq 1] \\
<\ & 2^{\ell(|x|)} \cdot \left(\frac{1}{3\ell(|x|)}\right)^{\ell(|x|)} \quad = \quad \left(\frac{2}{3\ell(|x|)}\right)^{\ell(|x|)} \quad \ll \quad 1
\end{aligned}
$$

where the last inequality is due to the fact that, for any fixed $x \in L$ and r, it holds that $\mathrm{Pr}_{s_i}[A'(x, s_i \oplus r) \neq 1] = \mathrm{Pr}_s[A'(x,s) \neq \chi_L(x)] < 1/3\ell(|x|)$. On the other hand, for any $x \notin L$ and every sequence of s_i's, it holds that $\mathrm{Pr}_r[\bigvee_{i=1}^{\ell(|x|)} A'(x, s_i \oplus r) = 1] < 1/3 < 1$ (since $x \notin L$). Thus, Eq. (7.1) cannot possibly hold for $x \notin L$. ■

We comment that the same proof idea yields a variety of similar statements (e.g., see Section 12.2).

7.2 One-sided error: RP and coRP

The class $\mathcal{RP}$ is defined as containing any set L such that there exists a probabilistic polynomial-time machine M satisfying the following two conditions

$$
x \in L \implies \Pr[M(x) = 1] \geq \frac{1}{2} \tag{7.2}
$$

$$
x \notin L \implies \Pr[M(x) = 0] = 1 \tag{7.3}
$$

Observe that $\mathcal{RP} \subseteq \mathcal{NP}$ (e.g., note that $\mathcal{NP}$ is obtained by replacing Eq. (7.2) with the condition $\Pr[M(x) = 1] > 0$, for every $x \in L$). Again, the specific probability threshold in Eq. (7.2) is immaterial as long as it is noticeable (and sufficiently bounded from 1).[1] Thus, $\mathcal{RP} \subseteq \mathcal{BPP}$.

Exercise: Prove that L' is in the class $\mathrm{co}\mathcal{RP} = \{\{0,1\}^* \setminus L : L \in \mathcal{RP}\}$ if and only if there exists a probabilistic polynomial-time machine M' satisfying the following two conditions

$$
x \in L' \implies \Pr[M'(x) = 1] = 1 \tag{7.4}
$$

$$
x \notin L' \implies \Pr[M'(x) = 0] \geq \frac{1}{2} \tag{7.5}
$$

[1] Exercise: Let $\mathcal{RP}_\epsilon$ denote the class obtained by replacing Eq. (7.2) by the condition $\Pr[M(x) = 1] \geq 1 - \epsilon(|x|)$, for every $x \in L$. Observe that $\mathcal{RP}_{1/2} = \mathcal{RP}$, and prove that $\mathcal{RP}_{1-(1/p(n))} = \mathcal{RP}$ and $\mathcal{RP}_{2^{-p(n)}} = \mathcal{RP}$, for any positive polynomial p. (Note that amplification is easier in this case (of one-sided error).)

The well-known randomized primality testing algorithms always accept prime numbers and rejects composite number with high probability. Thus, these algorithms establish that the set of prime numbers is in co$\mathcal{RP}$.

7.3 No error: ZPP

Whereas in case of $\mathcal{BPP}$ we have allowed two-sided errors, and in case of $\mathcal{RP}$ (and co$\mathcal{RP}$) we have allowed one-sided errors, we now allow no errors at all. Instead, we allow the algorithm to output a special `don't know` symbol, denoted $\perp$, with some bounded (away from 1) probability. The resulting class is denoted $\mathcal{ZPP}$, standing for Zero-error Probabilistic Polynomial-time. The standard definition of $\mathcal{ZPP}$ is in terms of machines that output $\perp$ with probability at most 1/2. That is, $L \in \mathcal{ZPP}$ if there exists a probabilistic polynomial-time machine M such that $\Pr[M(x) \in \{\chi_L(x), \perp\}] = 1$ and $\Pr[M(x) = \chi_L(x)] \geq 1/2$ for every $x \in \{0,1\}^*$. Again, the choice of the constant (i.e., 1/2) is immaterial, and "amplification" can be conducted as in case of $\mathcal{RP}$ (and yield the very same class). In fact, as in case of $\mathcal{RP}$, a more general statement holds.

Exercise: Prove that $\mathcal{ZPP} = \mathcal{RP} \cap \text{co}\mathcal{RP}$. (Indeed, $\mathcal{ZPP} \subseteq \mathcal{RP}$ (as well as $\mathcal{ZPP} \subseteq \text{co}\mathcal{RP}$) follows by a trivial transformation of the ZPP-machine. On the other hand, $\mathcal{RP} \cap \text{co}\mathcal{RP} \subseteq \mathcal{ZPP}$ can be proved by combining the two machines guaranteed for a set in $\mathcal{RP} \cap \text{co}\mathcal{RP}$.)

7.4 Randomized space complexity

The class $\mathcal{RL}$ (Random LogSpace) is defined analogously to the class $\mathcal{NL}$, and is indeed contained in the latter. Specifically, the syntax of Random LogSpace machines is identical to the one of Non-deterministic LogSpace machines, but the acceptance condition is probabilistic as in the case of $\mathcal{RP}$. In addition, we need to require explicitly that the machine runs in polynomial-time (or else $\mathcal{RL}$ extends up to $\mathcal{NL}$).[2]

Recall that Directed Connectivity is complete for $\mathcal{NL}$ (under log-space reductions). Below we show that undirected connectivity is solvable in $\mathcal{RL}$. Specifically, consider the set of triples (G, s, t) such that the vertices s and t are connected in the (undirected) graph G. On input (G, s, t), the randomized (log-space) algorithm starts a poly($|G|$)-long random walk at vertex s, and accepts the triplet if and only if the walk passed through vertex t. By a random walk we mean that at each step we select uniformly one of the neighbors of the current vertex and move to it. Observe that the algorithm can be implemented in logarithmic space (because we only need to store the current vertex as well as the number of steps taken so far), and that we never accept (G, s, t) in case s and t are not connected. We claim that if s and t are connected in $G = (V, E)$ then a random walk of length $O(|V| \cdot |E|)$ starting at s passes through t with probability at least 1/2. It follows that undirected connectivity is indeed in $\mathcal{RL}$.

[2] Recall that, w.l.o.g, a non-deterministic log-space machine need only run for polynomial-time. Such a computation can be simulated by a randomized log-space machine that repeatedly guesses non-deterministic moves and simulates the original machine on it. Note that we expect at most 2^t tries before we guess an accepting t-time computation, where t is polynomial in the input length. But what if there are no accepting t-time computations? To halt with a probabilistic rejecting verdict we should implement a counter that counts till 2^t, but we need to do so within space $O(\log t)$ (rather than t which is easy). In fact it suffices to have a randomized counter that with high probability counts to approximately 2^t. This can be implemented by tossing t coins until all show us HEADS. The expected number of times we need to repeat the experiment is 2^t, and we can implement this by a counter that counts till t (using space $\log_2 t$).

On proving the Random Walk Claim: (Indeed, this has little to do with the current course.) Consider the connected component of vertex s, denoted $G' = (V', E')$. For any pair, (u, v), let $T_{u,v}$ be a random variable representing the number of steps taken in a random walk starting at u until v is first encountered. First verify that $\mathrm{E}[T_{u,v}] \leq 2|E'|$, for any (u, v) such that $\{u, v\} \in E'$.[3] Next, letting $\mathrm{cover}(G')$ be the expected number of steps in a random walk starting at s and ending when the last of the vertices of V' is encountered, and C be any directed cyclic tour that visits all vertices in G', we have $\mathrm{cover}(G') \leq \sum_{(u,v)\in C} \mathrm{E}[T_{u,v}] \leq |C| \cdot 2|E'|$. Letting C be a traversal of some spanning tree of G', we conclude that $\mathrm{cover}(G') < 4 \cdot |E'| \cdot |V'|$. Thus, with probability at least $1/2$, a random walk of length $8 \cdot |E'| \cdot |V'|$ starting at s visits all vertices of G'.

[3]For example, let $C_{u,v}(n)$ be a random variable counting the number of minimal u-to-v sub-paths within a random walk of length n, where the walk starts at the stationary vertex distribution (assuming the graph is not bipartite or is sligtly modified otherwise). On one hand, $\mathrm{E}[T_{u,v}] = \lim_{n\to\infty}(n/\mathrm{E}[C_{u,v}(n)])$ (due to the memoryless property of the walk). On the other hand, $\mathrm{E}[C_{u,v}(n)] + 1$ is lower bounded by the expected number of times that the edge (v, u) was travesed (from v to u) in such a (n-step) walk, where the latter expected number equals $n/2|E'|$ (because each directed edge appears (in each step) on the walk with equal probability). It follows that $\mathrm{E}[T_{u,v}] \leq \lim_{n\to\infty}(n/((n/2|E'|) - 1)) = 2|E'|$.

Lecture 8

Non-Uniform Complexity

All complexity classes considered so far are "uniform" in the sense that each set in each of these classes was defined via one finite machine (or finite expression), which applied to all input lengths. This is indeed in agreement with the basic algorithmic paradigm of designing algorithms that can handle all inputs.

In contrast, non-uniform complexity investigates what happens when we allow to use a different algorithm for each input length. Indeed, in such a case, we must bound the description size of the algorithm (otherwise any problem can be solved by incorporating in the algorithm the answers to all finitely many inputs of the adequate length). By considering non-uniform complexity, we are placing an upper-bound on what can be done by the corresponding uniform-complexity class. The hope is that by abstracting away the ("evasive") uniformity condition, we will get a finite combinatorial structure that we may be able to understand.

8.1 Circuits and advice

Focusing on non-uniform polynomial-time, we mention two standard ways of defining non-uniform complexity classes. The first way is by considering (families of) Boolean circuits (as in Section 3.3). Specifically, L is said to be in non-uniform polynomial-time, denoted $\mathcal{P}/\text{poly}$, if there exists an infinite sequence of Boolean circuits $C_1, C_2, ...$ such that for some polynomial p the following three conditions hold:

1. The circuit C_n has n inputs and one output.
2. The size (e.g., number of edges) of the circuit C_n is at most $p(n)$.
3. For every $x \in \{0,1\}^n$, it holds that $C_n(x) = 1$ if and only if $x \in L$.

That is, C_n is a non-trivial algorithm (i.e., it cannot explicitly encode all 2^n answers) for deciding the membership in L of n-bit long strings. However, although C_n has size at most $p(n)$, it is not clear whether one can construct C_n in poly(n)-time (or at any time; see below).

An alternative way of defining $\mathcal{P}/\text{poly}$ proceeds by considering "machines that take advice". That is, we consider deterministic polynomial-time machines that get two inputs, where the second input (i.e., the advice) has length that is at most polynomial in the first input. The advice may only depend on the input length, and thus it cannot explicitly encode the answers to all inputs (of that length). Specifically, $L \in \mathcal{P}/\text{poly}$ if there exists a deterministic polynomial-time machine M and an infinite sequence of advice strings $a_1, a_2, ...$ such that for some polynomial p the following conditions hold:

1. The length of a_n is at most $p(n)$.
2. For every $x \in \{0,1\}^n$, it holds that $M(x, a_n) = 1$ if and only if $x \in L$.

Exercise: Prove that the two formulations of $\mathcal{P}$/poly are indeed equivalent. Furthermore, prove that without loss of generality, the machine M (as above) may be a universal machine.

8.2 The power of non-uniformity

Waiving the "uniformity condition" allows non-uniform classes to contain non-recursive sets. This is true for $\mathcal{P}$/poly as well as for most reasonable non-uniform classes, and is due to the obvious reason that there exists non-recursive unary sets. Specifically, any unary set $L \subseteq \{1\}^*$ (possibly non-recursive), can be decided by a linear-time algorithm that uses 1-bit long advice (i.e., $a_n \stackrel{\text{def}}{=} \chi_L(1^n)$ and $M(x, a_{|x|}) = 1$ if and only if both $x = 1^{|x|}$ and $a_{|x|} = 1$).

On the other hand, the existence of sets that are not in $\mathcal{P}$/poly can be proven in a more "concrete" way than the corresponding statement for $\mathcal{P}$. Fixing any super-polynomial and sub-exponential function f, we observe that the number of possible $f(n)$-bit long advice is much smaller than the number of possible subsets of $\{0,1\}^n$, whereas these advice account for all the sets that $\mathcal{P}$/poly may recognize (using a universal machine).

We took it for granted that $\mathcal{P} \subseteq \mathcal{P}$/poly, which is indeed true (e.g., by using empty advice strings). The fact that $\mathcal{P}$/poly also contains $\mathcal{BPP}$ is less obvious. Before proving this fact, let use mention that it is widely believed that $\mathcal{P}$/poly does not contain $\mathcal{NP}$, and indeed proving the latter conjecture was suggested as a good way for establishing that $\mathcal{P} \neq \mathcal{NP}$. (Whether or not this way is a good one is controversial.)

Theorem 8.1 $\mathcal{BPP} \subset \mathcal{P}$/poly

Proof: As in the proof of Theorem 7.1, we consider an adequate amplification of $\mathcal{BPP}$. Here, for $L \in \mathcal{BPP}$, we consider (by suitable error-reduction) a probabilistic polynomial-time algorithm A such that $\Pr[A(x) \neq \chi_L(x)] < 2^{-|x|}$. Again, let us consider the residual deterministic two-input algorithm A' such that $A'(x,r)$ equals the output of A on input x and random choices $r \in \{0,1\}^{\ell(|x|)}$. Then, by a trivial counting argument, there exists a string $r \in \{0,1\}^{\ell(n)}$ such that $A'(x,r) = \chi_L(x)$ for all x's of length n. Using this string r as the advice for n-bit long inputs, we are done. ■

8.3 Uniformity

The non-uniform aspect of the definition of $\mathcal{P}$/poly is the lack of requirements regarding the constructibility of the circuits (resp., advice). As a sanity check, we note that requiring that these objects be polynomial-time constructible results in a cumbersome definition of $\mathcal{P}$. That is, suppose that we require that there is a polynomial-time algorithm A that given 1^n outputs the circuit C_n (resp., the advice a_n) for deciding $L \in \mathcal{P}$/poly (as per the definition above). Then, combining A with the standard circuit-evaluation algorithm (resp., the advice-taking machine M), we obtain an ordinary polynomial-time algorithm for deciding L.

8.4 Evidence that P/poly does not contain NP

Recall that a major motivation towards studying $\mathcal{P}$/poly is the desire to prove that $\mathcal{P}$/poly does not contain $\mathcal{NP}$ (and thus also $\mathcal{BPP} \supseteq \mathcal{P}$ does not contain $\mathcal{NP}$). In view of the fact that $\mathcal{P}$/poly contains non-recursive sets, one may wonder how feasible is the conjecture that $\mathcal{P}$/poly does not contain $\mathcal{NP}$. It would have been best if we knew that $\mathcal{NP} \subset \mathcal{P}$/poly if and only if $\mathcal{P} = \mathcal{NP}$. But we only know that $\mathcal{NP} \subset \mathcal{P}$/poly implies a collapse of the Polynomial-time Hierarchy. That is:

Theorem 8.2 $\mathcal{NP} \subseteq \mathcal{P}$/poly *implies that* $\mathcal{PH} = \Sigma_2$.

Proof sketch: We show that $\Pi_2 = \Sigma_2$, and the claim follows by Theorem 6.2. Suppose that $L \in \Pi_2$, and let us consider the corresponding quantified expression (for $x \in L$): $\forall y \exists z R(x, y, z) = 1$, where $y, z \in \{0,1\}^{\text{poly}(|x|)}$. Let $L' \stackrel{\text{def}}{=} \{(x, y) : \exists z R(x, y, z) = 1\}$, and observe that L' is in $\mathcal{NP}$, and thus in $\mathcal{P}$/poly. Thus, $x \in L$ if and only if for $m = \text{poly}(|x|)$ there exists a poly(m)-size circuit C_m for deciding $L' \cap \{0,1\}^m$ such that for all y's it holds that $C_m(x, y) = 1$. The above expression is almost of the adequate (i.e., Σ_2) form, except that we need to check that C is indeed correct on all inputs of length m. Suppose that L' was *downwards self-reducible*; that is, that deciding whether $w \in L'$ could be reduced to deciding membership in L' of shorter (than w) strings. Then, we could have revised the above expression and assert that $x \in L$ if and only if there exists a sequence of polynomial-size circuits $C_1, ..., C_m$ such that

1. for all y's it holds that $C_m(x, y) = 1$;
2. for $i = 1, ..., m$, the circuit C_i correctly determines membership in L', where correctness of C_i is expressed by saying that for all $w \in \{0,1\}^i$ the value of $C_i(w)$ is consistent with the values obtained by the downwards self-reduction (as answered by the already verified circuits $C_1, ..., C_{i-1}$).

However, we have no reason to assume that L' is self-reducible. What we do instead is reduce L' to SAT and apply the argument to SAT (using its polynomial-size circuits (which exist by the hypothesis) and its downwards self-reducibility (which is a very natural procedure)). Specifically, let f be a Karp-reduction of L' to SAT. Thus, $x \in L$ if and only if $\forall y\, f(x, y) \in SAT$. Using the hypothesis, we have $SAT \in \mathcal{P}$/poly, and thus there exists a sequence of polynomial-size circuits $C_1, C_2, ...$ for SAT. Now, we assert that $x \in L$ if and only if there exists a sequence of polynomial-size circuits $C_1, ..., C_m$, where $m = |f(x, y)|$, such that the following two conditions hold:

1. For all y's (of adequate length), $C_m(f(x, y)) = 1$.
2. For $i = 1, ..., m$, the circuit C_i correctly decides membership of i-bit long strings in SAT. Note that the correctness condition for C_i can be expressed as follows: For every i-long formula ϕ it holds that $C_i(\phi) = 1$ if and only if either $C_{i'}(\phi') = 1$ or $C_{i''}(\phi'') = 1$, where ϕ' (resp., ϕ'') is the formula obtained from ϕ by replacing its first variable with 0 (resp., 1), and i' (resp., i'') is the length of the resulting formula after straightforward simplifications (which necessarily occurs after instantiating a variable).

Observe that the expression obtained for membership in L is indeed of the Σ_2-form. The theorem follows. ■

8.5 Reductions to sparse sets

Another way of looking at $\mathcal{P}$/poly is as the class of sets that are Cook-reducible to a sparse set, where a sparse set is a set that contains at most polynomially many strings of each length. (The reason for stressing the fact that we refer to Cook-reductions will be explained below.) Let us first establish the validity of the above claim.

Proposition 8.3 *$L \in \mathcal{P}/\mathrm{poly}$ if and only if L is reducible to some sparse set.*

Proof sketch: Suppose that $L \in \mathcal{P}/\mathrm{poly}$ and suppose that n is sufficiently large. Then we can encode the nth advice string (i.e., a_n) in (the first $|a_n|$ strings of) the n-bit slice of a set S (i.e., by placing the i^{th} (n-bit) string in $S \cap \{0,1\}^n$ if and only if the i^{th} bit of a_n equals 1). Observe that S is indeed sparse (because $|a_n| = \mathrm{poly}(n)$). On input x, the reduction first retrieves the advice string $a_{|x|}$ (by making polynomially-many n-bit long queries to S), and decides according to the advice-taking $M(x, a_{|x|})$.

In case L is reducible to a sparse set S, we let the nth advice encode the list of all the strings in S that have length at most $q(n)$, where q is the polynomial bounding the running-time of the reduction. Given this advice (which is of length $\sum_{i=1}^{q(n)} |S \cap \{0,1\}^i| \cdot i = \mathrm{poly}(n)$), the advice-taking machine can emulate the answers of the oracle machine (of the reduction), and thus decide L. ■

As a direct corollary to Proposition 8.3, we obtain:

Corollary 8.4 *SAT is Cook-reducible to a sparse set if and only if $\mathcal{NP} \subseteq \mathcal{P}/\mathrm{poly}$.*

Combining Corollary 8.4 and Theorem 8.2, it follows that SAT cannot be Cook-reducible to a sparse set, unless the Polynomial-time hierarchy collapses.

Perspective: Karp-reductions to sparse sets

We have stressed the fact that we refer to Cook-reductions, because (by Corollary 8.4) SAT is Cook-reducible to a sparse set if and only if $\mathcal{NP} \subseteq \mathcal{P}/\mathrm{poly}$. In contrast, it is known that SAT is Karp-reducible to a sparse set if and only if $\mathcal{NP} = \mathcal{P}$. Thus, the difference between Cook and Karp reductions is "reflected" in the difference between $\mathcal{NP} \subseteq \mathcal{P}/\mathrm{poly}$ and $\mathcal{NP} = \mathcal{P}$.

Theorem 8.5 *SAT is Karp-reducible to a sparse set if and only if $\mathcal{NP} = \mathcal{P}$.*

Proof of a special case: Clearly, if $\mathcal{NP} = \mathcal{P}$ then SAT is Karp-reducible to any non-trivial set (e.g., to the set $\{1\}$). We establish the opposite direction only for the special case that SAT is Karp-reducible to some set S such that S is a subset of a *sparse* set $G \in \mathcal{P}$. (Such a set S is called guarded, and $S \subseteq \{1\}^*$ is indeed a special case). Specifically, using the Karp-reduction of SAT to S, we present a (deterministic) polynomial-time decision procedure for SAT. The procedure conducts a DFS on the tree of all possible partial truth assignment to the input formula, while truncating the search at nodes that are roots of sub-trees that contain no satisfying assignment (at the leaves).[1] The key observation is that each internal node (which yields a formula derived from the initial formulae by instantiating the corresponding partial truth assignment) is mapped by the reduction either to a string not in G (in which case we conclude that the sub-tree contains no satisfying assignments) or to a string in G (in which case we don't know what to do). However, once we backtrack from this internal node, we know that the corresponding element of G is not in S, and we will never extend a node mapped to this element again. Specifically, let ϕ be the input formula, and ϕ_τ denote the formula resulting from ϕ by setting its first $|\tau|$ variables according to the partial truth assignment τ. Then, the procedure proceeds as follows, using the Karp-reduction f of SAT to S:

[1]For an n-variable formulae, the leaves of the tree correspond to all possible n-bit long strings, and an internal node corresponding to τ is the parent of nodes corresponding to $\tau 0$ and $\tau 1$.

Initialization: $\tau \leftarrow \lambda$ and $B \leftarrow \emptyset$, where τ is a partial truth assignment for which we wish to determined whether or not $\phi_\tau \in SAT$, and $B \subset G \setminus S$ is a set of strings that were already "proved" not to be in S.

The following steps are recursive and return a Boolean value, representing the whether or not $\phi_\tau \in SAT$.

Internal node: Determine whether or not $\phi_\tau \in SAT$, according to the following three cases.

1. If $f(\phi_\tau) \notin G$ then return the value `false`.
   (Since $S \subseteq G$, we have $f(\phi_\tau) \notin S$, and by the validity of the reduction, $\phi_\tau \notin SAT$.)
2. If $f(\phi_\tau) \in B$ then return the value `false`.
   (Since $B \subseteq G \setminus S$, we have $f(\phi_\tau) \notin S$, and by the validity of the reduction, $\phi_\tau \notin SAT$.)
3. Otherwise (i.e., $f(\phi_\tau) \in G \setminus B$), invoke two recursive calls, for $\phi_{\tau 0}$ and $\phi_{\tau 1}$, respectively.[2] If both calls have returned `false` and $f(\phi_\tau) \in G \setminus B$ then add $f(\phi_\tau)$ to B (since $\phi_\tau \notin SAT$ holds). Actually, if the first call returns `true` then the second call does not take place.[3] In any case, return the OR-value of the two values returned by the recursive calls.

(We stress that only the third case invokes recursive calls.)

Bottom Level: If the constant "formula" ϕ_τ is `false` and $f(\phi_\tau) \in G \setminus B$ then add $f(\phi_\tau)$ to B. In any case, return the value of ϕ_τ.

It is easy to verify that the procedure returns the correct answer. The running-time analysis is based on the observation that if τ' and τ'' are not prefixes of one another and $f(\phi_{\tau'}) = f(\phi_{\tau''})$ then it cannot be that Case 3 was applied to both of them. Thus, the number of internal nodes for which Case 3 was applied is at most the depth of the tree times $|\cup_{i=1}^{m} G_i \setminus S| \leq \sum_{i=1}^{m} |G_i| = \mathrm{poly}(m)$, where $G_i \stackrel{\text{def}}{=} G \cap \{0,1\}^i$ and $m = |f(\phi)| = \mathrm{poly}(|\phi|)$. ■

[2] We may re-evaluate the condition $f(\phi_\tau) \in B$ after obtaining the answer of the first call, but this is not really necessary.

[3] Otherwise, the procedure will visit all satisfying assignments, and consequently may run for exponential time.

Lecture 9

Counting Classes

9.1 The definition of #P

A natural computational problem associated with an NP-relation R is to determine the number of solutions for a given instance; that is, given x, determine the cardinality of $R(x) \stackrel{\text{def}}{=} \{y : (x,y) \in R\}$. This problem is the counting problem associated with R. Certainly, the counting problem associated with R is not easier than the problem of deciding membership in $L_R = \{x : \exists y \text{ s.t. } (x,y) \in R\}$ (which can be casted as determining, for a given x, whether $|R(x)|$ is positive or zero).

The class $\#\mathcal{P}$ can be defined as a class of functions that count the number of solutions in NP-relations. That is, $f \in \#\mathcal{P}$ if there exists an NP-relation R such that $f(x) = |R(x)|$ for all x's. Alternatively, we can define $\#\mathcal{P}$ as a class of sets, where for every NP-relation R the set $\#R \stackrel{\text{def}}{=} \{(x,k) : |R(x)| \geq k\}$ is in $\#\mathcal{P}$. (Exercise: Formulate and show the "equivalence" between the two definitions.)

Relation to PP. The class $\#\mathcal{P}$ is related to a probabilistic class, denoted $\mathcal{PP}$, that was not defined in Lecture 7. We say that $L \in \mathcal{PP}$ if there exists a probabilistic polynomial-time algorithm A such that, for every x, it holds that $\Pr[A(x) = 1] > 1/2$ if and only if $x \in L$ (or, alternatively, $\Pr[A(x) = \chi_L(x)] > 1/2$ for every x).[1] (Recall that, in contrast, $L \in \mathcal{BPP}$ requires that $\Pr[A(x) = \chi_L(x)] > 2/3$ for every x.) Notice that any $L \in \mathcal{PP}$ can be decided by a polynomial-time oracle machine that is given oracle access to $\#R$, where R describes the actions of the PP-algorithm (i.e., $(x,r) \in R$ iff $A(x)$ accepts when using coins r). On the other hand, $\#\mathcal{P} \subseteq \mathcal{PP}$, by virtue of (a minor modification to) the following algorithm that refers to $\#R$, where $R \subseteq \cup_{n \in \mathbb{N}} \{0,1\}^n \times \{0,1\}^{m(n)}$: on input (x,k), with probability one half select y uniformly in $\{0,1\}^{m(|x|)}$ and accept iff $(x,y) \in R$, and otherwise (i.e., with probability $1/2$) accept with probability exactly $1 - (k - 0.5) \cdot 2^{-m(|x|)}$. (Exercise: Provide the missing details for all the above claims.)

9.2 #P-complete problems

We say that a computational problem is $\#\mathcal{P}$-complete if it is in $\#\mathcal{P}$ and every problem in $\#\mathcal{P}$ is reducible to it. Thus, for an NP-relation R, the problem $\#R$ (which is always in $\#\mathcal{P}$) is $\#\mathcal{P}$-complete if for any NP-relation R' it holds that $\#R'$ is reducible to $\#R$. Using the standard Karp-reductions, it is easy to show that for any known NP-complete relation R the set $\#R$ is

[1] Exercise: show the equivalence of the two formulations.

$\#\mathcal{P}$-complete. This is the case because the standard reductions (or minor modifications of them) are "parsimonious" (i.e., preserve the number of solutions). In particular:

Proposition 9.1 *$\#SAT$ is $\#\mathcal{P}$-complete, where $(\phi, k) \in \#SAT$ if and only if ϕ has at least k different satisfying assignment.*

Exercise: Verify that the standard reduction of any NP-relation to SAT is parsimonious; that is, for any NP-relation R, the standard reduction of R to SAT maps each x to a formula having exactly $|R(x)|$ satisfying assignments.

As stated above, Proposition 9.1 is merely a consequence of the nature of the reductions used in the standard context of NP-completeness results. Specifically, it is the case that the same reductions used to demonstrate NP-completeness of search problems can be used to show $\#\mathcal{P}$-completeness of the corresponding counting problems. Consequently, "hard" (i.e., NP-complete) search problems give rise to "hard" (i.e., $\#\mathcal{P}$-complete) counting problems. Interestingly, there are "hard" counting problems (i.e., $\#\mathcal{P}$-complete problems) for which the corresponding search problem is easy. For example, whereas the problem of finding a maximum matching in a given graph is "easy" (i.e., solvable in polynomial-time), the corresponding counting problem is "hard" (i.e., $\#\mathcal{P}$-complete):

Theorem 9.2 *The problem of counting the number of perfect matching in a bipartite graph is $\#\mathcal{P}$-complete. Equivalently, the problem of computing the permanent of integer matrices with 0/1-entries is $\#\mathcal{P}$-complete.*

Needless to say, the reduction used in proving Theorem 9.2 is not parsimonious (or else we could have used it to reduce $\mathcal{NP}$ to the problem of deciding whether a given graph has a perfect matching). For the same reason, the recent polynomial-time algorithm for approximating the permanent (of non-negative matrices)[2] does not yield polynomial-time approximation algorithms for all $\#\mathcal{P}$.

9.3 A randomized reduction of Approximate-#P to NP

By an approximation for a counting problem $\#R$ in $\#\mathcal{P}$, we mean a procedure that on input x outputs a "good" approximation, denoted $A(x)$, of $|R(x)|$. Specifically, we require that with high probability, the ratio $A(x)/|R(x)|$ will be bounded. For many natural NP-relations (and in particular for SAT), the following notions are all equivalent:

1. With probability at least 2/3, it holds that $A(x)$ is within a factor of 2 of $|R(x)|$ (i.e., $1 \leq A(x)/|R(x)| \leq 2$).[3]
2. With probability at least $1 - \exp(-|x|)$, it holds that $1 \leq A(x)/|R(x)| \leq 2$.
3. With probability at least $1 - \exp(-|x|)$, it holds that $1 \leq A(x)/|R(x)| \leq 1 + |x|^{-c}$, where $c > 0$ is any fixed constant.
4. With probability at least 2/3, it holds that $1 < A(x)/|R(x)| < 2^{|x|^c}$, where $c < 1$ is any fixed constant.[4]

[2]See Jerrum, Sinclair and Vigoda: A Polynomial-Time Approximation Algorithm for the Permanent of a Matrix with Non-Negative Entries, in *Proc. of the 33rd STOC*, pages 712–721, 2001.

[3]Show that this is equivalent to ability to get $A(x)$ such that $1/\sqrt{2} \leq A(x)/|R(x)| \leq \sqrt{2}$.

[4]Note that for some constant c that depends on R, the ability to approximate $|R(x)|$ to within a factor of $2^{|x|^c}$ merely requires the ability to distinguish the case $|R(x)| = 0$ from $|R(x)| > 0$ (since $|R(x)| \leq 2^{|x|^c}$ always holds). Exercise: Show that ability to approximate every $|R(x')|$ to within a factor of $2^{|x'|}$ implies ability to approximate $|R(x)|$ to within a factor of $2^{|x|^c}$.

Item 1 implies Item 2 by using straightforward error-reduction (as in case of $\mathcal{BPP}$). To show that Item 4 implies Item 1 (resp., Item 2 implies Item 3), we use the fact that for many natural NP-relations it is the case that many instances can be encoded in one (i.e., $R(\langle x_1,...,x_t\rangle) = \{\langle y_1,...,y_t\rangle : \forall i\, y_i \in R(x_i)\}$).[5] Thus, suppose that (for every x) we know how to approximate $|R(x)|$ to within a factor of $2^{|x|^{2/3}}$, and we want to approximate $|R(x)|$ to within a factor of 2 (for every x). Then, we form x' as a sequence of $t \stackrel{\text{def}}{=} |x|^2$ copies of x, and obtain a $2^{|x'|^{2/3}}$-factor approximation of $|R(x')| = |R(x)|^t$. Taking the t^{th} root of this approximation, we obtain $|R(x)|$ up-to a factor of $(2^{|x'|^{2/3}})^{1/t} = 2^{(t\cdot|x|)^{2/3}/t} = 2$.

In view of the above, we focus on providing any good approximation to the problem of counting the number of satisfying assignments to a boolean formula. The same techniques apply to any NP-complete problem.

Theorem 9.3 *The counting problem #SAT can be approximated up to a constant factor by a probabilistic polynomial-time oracle machine with oracle access to SAT.*

Proof Sketch: Given a formula ϕ on n variables, we approximate $|SAT(\phi)|$ by trying all possible powers of 2 as candidate approximations. That is, for $i = 1,...,n$, we check whether 2^i is a good approximation of $|SAT(\phi)|$. This is done by uniformly selecting a "good" hashing function $h : \{0,1\}^n \to \{0,1\}^i$, and checking whether there exists a truth assignment τ for ϕ such that the following two conditions hold:

1. the truth assignment τ satisfies ϕ (i.e., $\phi(\tau) = \texttt{true}$), and
2. h hashes τ to the all-zero string (i.e., $h(\tau) = 0^i$).

These two conditions can be encoded in a new formula (e.g., by reducing the above NP-condition to SAT).[6] The new formula ϕ' is satisfiable if and only if there exists an assignment τ (to ϕ) that satisfies the above conditions. Thus, the answer to the above question (i.e., whether such a τ exists) is obtained by making a corresponding query (i.e., ϕ') to the SAT oracle.

In the analysis, we assume that the hashing function is good in the sense that for any $S \subseteq \{0,1\}^n$, with high probability, a randomly selected hashing function h satisfies $|\{e \in S : h(e) = 0^i\}| \approx |S|/2^i$. In particular, a randomly selected hashing function h maps each string to 0^i with probability 2^{-i}, and the mapping of different strings is pairwise independent. For further details, see [28, Lect. 4].

Note that if $|SAT(\phi)| < 2^{i-2}$ then a randomly selected hashing function is unlikely to map any of the (fewer than 2^{i-2}) satisfying assignment of ϕ to 0^i. Specifically, the probability that any specific assignment is mapped to 0^i equals 2^{-i}, and so the bad event occurs with probability less than 1/4, which can be further reduced by repeating the random experiment.

On the other hand, if $|SAT(\phi)| > 2^{i+2}$ then a randomly selected hashing function is likely to map some of the (more than 2^{i+2}) satisfying assignment of ϕ to 0^i. This can be proven using the pairwise independent property of the mapping induced by a random hashing function.

Thus, with high probability, the above procedure outputs a value $v = 2^i$ such that $i - 2 < \log_2 |SAT(\phi)| < i + 3$. We stress that the entire argument can be adapted to any NP-complete problem. Furthermore, smaller approximation factors can be obtained (directly) by using tricks as in the proof of Theorem 9.4. ■

---

[5]For example, the number of satisfying assignments to a formula consisting of t formulae over distinct variables is the product of the number of satisfying assignments to each of these formulae.

[6]Alternatively, for some popular hashing functions, the condition $h(\tau) = 0^i$ is easily transformed to CNF. Thus, we obtain the formula $\phi'(z_1,...,z_n) = \phi(z_1,...,z_n) \wedge (h(z_1 \cdots z_n) = 0^i)$.

9.4 A randomized reduction of SAT to Unique-SAT

The widely believed intractability of SAT cannot be due to instances that have "very many" satisfying assignments. For example, satisfying assignments for n-variable formula having at least $2^n/n$ satisfying assignments can be found in probabilistic polynomial-time by selecting n^2 assignments at random. Going to the other extreme, one may ask whether SAT instances having very few satisfying assignments (e.g., a unique satisfying assignment) can be hard. As shown below, the answer is positive. We show that ability to solve such instances yields ability to solve arbitrary instances.

In order to formulate the above discussion, we need to introduce the notion of a promise problem, which extends (or relaxes) the notion of a decision problem. A promise problem Π is a pair of disjoint subsets, denoted $(\Pi_{\rm yes}, \Pi_{\rm no})$. A (deterministic) machine M is said to solve such a problem if $M(x) = 1$ for every $x \in \Pi_{\rm yes}$ and $M(x) = 0$ for every $x \in \Pi_{\rm no}$, whereas nothing is required in case $x \notin \Pi_{\rm yes} \cup \Pi_{\rm no}$ (i.e., x "violates the promise"). (The notion extends naturally to probabilistic machines, oracle machines, and so on.) When we say that some problem reduces to the promise problem $\Pi = (\Pi_{\rm yes}, \Pi_{\rm no})$, we mean that the reduction yields the correct output regardless of the way in which queries outside of $\Pi_{\rm yes} \cup \Pi_{\rm no}$ are answered. (This is consistent with requiring nothing from a machine that solves Π in case the input is not in $\Pi_{\rm yes} \cup \Pi_{\rm no}$.)

The computational problem of distinguishing instances with a unique solution from instances with no solution yields a natural promise problem. For example, $uniqueSAT$ (or $uSAT$) is the promise problem with yes-instances being formulae having a unique satisfying assignment and no-instances being formulae having no satisfying assignment.

Theorem 9.4 *SAT is randomly reducible to $uSAT$.*

Proof Sketch: We present a probabilistic polynomial-time oracle machine that solves SAT using an oracle to $uSAT$. Actually, it is easier to first (randomly) reduce SAT to $fewSAT$, where $fewSAT$ is the promise problem with yes-instances being formulae having between 1 and 100 satisfying assignments and no-instances being formulae having no satisfying assignment.

Observe that the procedure described in the proof of Theorem 9.3 can be easily adapted to do the work. Specifically, we accept the given SAT instance ϕ if and only if any of the oracle invocations returns the value `true`. Note that the latter event may occur only if ϕ is satisfiable (because when ϕ is unsatisfiable all queries ϕ' are unsatisfiable). On the other hand, if ϕ has $k > 8$ satisfying assignments then in iteration $i = \lfloor \log_2 k \rfloor - 2$, with high probability, the query ϕ' is satisfiable and has at most $k/2^{i-2} < 32$ satisfying assignments (i.e., ϕ' is a yes-instance of $fewSAT$).[7]

To finish-up the proof we reduce $fewSAT$ to $uSAT$. Given a formula ϕ, for $i = 1, ..., 100$, we construct a formula $\phi^{(i)}$ that has a unique satisfying assignment if and only if ϕ has exactly i satisfying assignments. For example, $\phi^{(i)}$ may consist of the conjunction of i copies of ϕ over distinct variables and a condition imposing a lexicography order between the corresponding assignments.[8]
■

[7]In order to take care of the case $k \leq 8 < 100$, we also query the $fewSAT$ oracle about ϕ itself.

[8]E.g., $\phi^{(2)}(x_1, ..., x_n, y_1, ..., y_n) \stackrel{\rm def}{=} \phi(x_1, ..., x_n) \wedge \phi(y_1, ..., y_n) \wedge (\vee_{j=0}^{n-1} ((x_{j+1} < y_{j+1}) \wedge (\wedge_{k=1}^{j} x_k = y_k)))$.

Lecture 10

Space is more valuable than time

This lecture was not given. The intention was to prove the following result, which asserts that any computation requires strictly less space than time.

Theorem 10.1 $\text{DTIME}(t) \subseteq \text{DSPACE}(t/\log t)$

That is, any given deterministic multi-tape Turing Machine (TM) of time complexity t, can be simulated using a deterministic TM of space complexity $t/\log t$. A main ingredient in the simulation is the analysis of a pebble game on directed bounded-degree graphs.

Lecture 11

Circuit Depth and Space Complexity

This lecture was not given. The intention was to explore some of the relations between Boolean circuits and Turing machines. Specifically:

- Define the complexity classes $\mathcal{NC}_i$ and $\mathcal{AC}_i$ (i.e., bounded versus unbounded fan-in circuits of polynomial-size and $O(\log^i)$-depth), and compare their computational power. Point out the connection between uniform-$\mathcal{NC}$ and "efficient" parallel computation.
- Establish a connection between the space complexity of a problem and the depth of circuits (with bounded fan-in) for the problem.

Historical Notes

For historical discussion of the material presented in Lecture 4, the reader is referred to the textbook of Hopcroft and Ullman [41]. Needless to say, the latter provides accurate statements and proofs of hierarchy and gap theorems.

Space Complexity: The emulation of non-deterministic space-bounded machines by deterministic space-bounded machines (i.e., Theorem 5.7) is due to Savitch [68]. Theorem 5.8 (i.e., $\mathcal{NL} = \text{co}\mathcal{NL}$) was proved independently by Immerman [42] and Szelepcsenyi [77].

The Polynomial-Time Hierarchy: The Polynomial-Time Hierarchy was introduced by Stockmeyer [75]. The third equivalent formulation via "alternating machines" can be found in [16].

Randomized Time Complexity: Probabilistic Turing Machines and corresponding complexity classes (including $\mathcal{BPP}, \mathcal{RP}, \mathcal{ZPP}$ and $\mathcal{PP}$) were first defined by Gill [23]. The random-walk (log-space) algorithm for deciding undirected connectivity is due to Aleliunas *et. al.* [2]. Additional examples of randomized algorithms and procedures can be found in [62] and [26, Apdx. B].

The robustness of the various classes under various error thresholds was established using straightforward amplifications (i.e., running the algorithm several times using independent random choices). Randomness-efficient amplification methods (which use related random choices in the various runs) have been studied extensively since the mid 1980's (cf. [26, Sec. 3.6]).

The fact that $\mathcal{BPP}$ is in the Polynomial-time hierarchy was proven independently by Lautemann [56] and Sipser [72]. We have followed Lautemann's proof. The ideas underlying Sipser's proof found many applications in complexity theory; in particular, they are used in the approximation procedure for $\#\mathcal{P}$ (as well as in the emulation of general interactive proofs by public-coin ones).

Non-Uniform Complexity: The class P/poly was defined by Karp and Lipton as part of a general formulation of "machines which take advise" [49]. They have noted the equivalence to the traditional formulation of polynomial-size circuits, the effect of uniformity, as well as the effect of $\mathcal{NP} \subseteq \mathcal{P}/\text{poly}$ on the Polynomial-time hierarchy (i.e., Theorem 8.2). Theorem 8.5 is due to Fortune [20].

Theorem 8.1 is attributed to Adleman [1], who actually proved that $\mathcal{RP} \subseteq \mathcal{P}/\text{poly}$ using a more involved argument.

Counting Classes: The counting class $\#\mathcal{P}$ was introduced by Valiant [79], who proved that computing the permanent of 0-1 matrices is $\#\mathcal{P}$-complete (cf. Theorem 9.2). Valiant's proof first establishes the $\#\mathcal{P}$-hardness of computing the permanent of integer matrices (the entries are actually restricted to $\{-1, 0, 1, 2, 3\}$), and next reduces the computation of the permanent of integer matrices to the the permanent of 0-1 matrices. A de-constructed version of Valinat's proof can be found in [11].

The approximation procedure for $\#\mathcal{P}$ is due to Stockmeyer [76], following an idea of Sipser [72]. Our exposition follows further developments in the area. The randomized reduction of SAT to uniqueSAT is due to Valiant and Vazirani [80]. Again, our exposition is a bit different.

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 $v_0 \stackrel{\rm def}{=} 0$). 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., $\mathcal{NP} \subseteq \mathrm{D{\scriptsize TIME}}(\exp(\mathrm{poly}(\log n)))$).

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 $\mathsf{pcp}(\log, \text{poly}(\log))$ proof system for $\mathcal{NP}$. Furthermore, this proof system has additional properties which enable proof composition as in Item (3) below.
2. A $\mathsf{pcp}(\text{poly}, O(1))$ 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 $\text{poly}(\ell)$-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 $s(n) \stackrel{\text{def}}{=} \text{poly}(\ell(n))$. [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 $r = r' = \log$, $\ell = q = \text{poly}(\log)$, and $s(n) = \text{poly}(\log(n))$] yielding a $\mathsf{pcp}(\log, \text{poly}(\log\log))$ system for $\mathcal{NP}$, which is then composed with the system of Item (2) [using $r = \log$, $\ell = \text{poly}(\log\log)$, $r' = \text{poly}$, $q = O(1)$, and $s(n) = \text{poly}(\log\log(n))$] yielding the desired $\mathsf{pcp}(\log, O(1))$ system for $\mathcal{NP}$.

The $\mathsf{pcp}(\log, \text{poly}(\log))$ 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 $t(n) \stackrel{\text{def}}{=} O(\log(n) \cdot \log\log(n))$. [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, $\mathcal{NP} \subset \mathcal{PCP}[\mathrm{poly}(n), O(1)]$. 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

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

   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

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

   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 $\mathrm{Pr}_{r,s}[rCs = 0] \leq 3/4$.

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, $\ell : \mathsf{N} \to \mathsf{N}$, *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 $\cap_{\epsilon>0}\mathrm{DTIME}(t_\epsilon)$, 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 $\mathcal{E} \stackrel{\rm def}{=} \cup_c \text{DTIME}(t_c)$, 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

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

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 $t_G(\ell^{-1}(n)) = 2^{O(\ell^{-1}(n))} = \mathrm{poly}(n)$. 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 $\ell(k) = \mathrm{time}_A(n)$.)

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 $|T_i \cap T_j| \le k'' = k'/O(1)$ 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 $j \le i$, 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$\stackrel{\text{def}}{=}\langle\mathcal{NP}, \text{P-computable}\rangle$, 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* $f : \{0,1\}^* \rightarrow \mathbf{N}$ *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,

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

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 $P(n) \stackrel{\rm def}{=} P_D(n) + P_C(n) + n$, $P_D(n)$ 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.

Bibliography

[1] L. Adleman. Two theorems on random polynomial-time. In *19th FOCS*, pages 75–83, 1978.

[2] R. Aleliunas, R.M. Karp, R.J. Lipton, L. Lovász and C. Rackoff. Random walks, universal traversal sequences, and the complexity of maze problems. In *20th FOCS*, pages 218–223, 1979.

[3] S. Arora, C. Lund, R. Motwani, M. Sudan and M. Szegedy. Proof Verification and Intractability of Approximation Problems. *JACM*, Vol. 45, pages 501–555, 1998. Preliminary version in *33rd FOCS*, 1992.

[4] S. Arora and S. Safra. Probabilistic Checkable Proofs: A New Characterization of NP. *JACM*, Vol. 45, pages 70–122, 1998. Preliminary version in *33rd FOCS*, 1992.

[5] L. Babai. Trading Group Theory for Randomness. In *17th STOC*, pages 421–429, 1985.

[6] L. Babai, L. Fortnow, L. Levin, and M. Szegedy. Checking Computations in Polylogarithmic Time. In *23rd STOC*, pages 21–31, 1991.

[7] L. Babai, L. Fortnow, N. Nisan and A. Wigderson. BPP has Subexponential Time Simulations unless EXPTIME has Publishable Proofs. *Complexity Theory*, Vol. 3, pages 307–318, 1993.

[8] L. Babai and S. Moran. Arthur-Merlin Games: A Randomized Proof System and a Hierarchy of Complexity Classes. *JCSS*, Vol. 36, pages 254–276, 1988.

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

[10] S. Ben-David, B. Chor, O. Goldreich, and M. Luby. On the Theory of Average Case Complexity. *JCSS*, Vol. 44, No. 2, April 1992, pages 193–219.

[11] A. Ben-Dor and S. Halevi. In *2nd Israel Symp. on Theory of Computing and Systems* (*ISTCS93*), IEEE Computer Society Press, 1993.

[12] M. Blum and S. Micali. How to Generate Cryptographically Strong Sequences of Pseudo-Random Bits. *SICOMP*, Vol. 13, pages 850–864, 1984. Preliminary version in *23rd FOCS*, 1982.

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

[14] L. Carter and M. Wegman. Universal Hash Functions. *JCSS*, Vol. 18, 1979, pages 143–154.

[15] G.J. Chaitin. On the Length of Programs for Computing Finite Binary Sequences. *JACM*, Vol. 13, pages 547–570, 1966.

[16] A.K. Chandra, D.C. Kozen and L.J. Stockmeyer. Alternation. *JACM*, Vol. 28, pages 114–133, 1981.

[17] S.A. Cook. The Complexity of Theorem Proving Procedures. In *. 3rd STOC*, pages 151–158, 1971.

[18] T.M. Cover and G.A. Thomas. *Elements of Information Theory.* John Wiley & Sons, Inc., New-York, 1991.

[19] U. Feige, S. Goldwasser, L. Lovász, S. Safra, and M. Szegedy. Approximating Clique is almost NP-complete. *JACM*, Vol. 43, pages 268–292, 1996. Preliminary version in *32nd FOCS*, 1991.

[20] S. Fortune. A Note on Sparse Complete Sets. *SIAM J. on Computing*, Vol. 8, pages 431–433, 1979.

[21] M. Fürer, O. Goldreich, Y. Mansour, M. Sipser, and S. Zachos. On Completeness and Soundness in Interactive Proof Systems. *Advances in Computing Research: a research annual*, Vol. 5 (Randomness and Computation, S. Micali, ed.), pages 429–442, 1989.

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

[23] J. Gill. Computational complexity of probabilistic Turing machines. *SIAM Journal on Computing*, Vol. 6(4), pages 675–695, 1977.

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

[25] O. Goldreich. *Secure Multi-Party Computation.* Unpublished amnuscript, 1998. Available from `http://www.wisdom.weizmann.ac.il/~oded/gmw.html`

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

[27] O. Goldreich. *Foundation of Cryptography – Basic Tools.* Cambridge University Press, 2001.

[28] O. Goldreich. *Randomized Methods in Computation*, Lecture Notes, Spring 2001. Available from `http://www.wisdom.weizmann.ac.il/~oded/rnd.html`

[29] O. Goldreich, S. Goldwasser, and S. Micali. How to Construct Random Functions. *JACM*, Vol. 33, No. 4, pages 792–807, 1986.

[30] O. Goldreich, H. Krawcyzk and M. Luby. On the Existence of Pseudorandom Generators. *SICOMP*, Vol. 22-6, pages 1163–1175, 1993.

[31] O. Goldreich and L.A. Levin. Hard-core Predicates for any One-Way Function. In *21st STOC*, pages 25–32, 1989.

[32] O. Goldreich, S. Micali and A. Wigderson. Proofs that Yield Nothing but their Validity or All Languages in NP Have Zero-Knowledge Proof Systems. *JACM*, Vol. 38, No. 1, pages 691–729, 1991. Preliminary version in *27th FOCS*, 1986.

[33] O. Goldreich, S. Micali and A. Wigderson. How to Play any Mental Game – A Completeness Theorem for Protocols with Honest Majority. In *19th STOC*, pages 218–229, 1987. For details see [25].

[34] S. Goldwasser and S. Micali. Probabilistic Encryption. *JCSS*, Vol. 28, No. 2, pages 270–299, 1984. Preliminary version in *14th STOC*, 1982.

[35] S. Goldwasser, S. Micali and C. Rackoff. The Knowledge Complexity of Interactive Proof Systems. *SICOMP*, Vol. 18, pages 186–208, 1989. Preliminary version in *17th STOC*, 1985. Earlier versions date to 1982.

[36] S. Goldwasser and M. Sipser. Private Coins versus Public Coins in Interactive Proof Systems. *Advances in Computing Research: a research annual*, Vol. 5 (Randomness and Computation, S. Micali, ed.), pages 73–90, 1989. Extended abstract in *18th STOC*, pages 59–68, 1986.

[37] Y. Gurevich. Complete and Incomplete Randomized NP Problems. In *Proc. of the 28th FOCS*, 1987, pages 111–117.

[38] J. Håstad. Clique is hard to approximate within $n^{1-\epsilon}$. *Acta Mathematica*, Vol. 182, pages 105–142, 1999. Combines preliminary versions in *28th STOC* (1996) and *37th FOCS* (1996).

[39] J. Håstad. Getting optimal in-approximability results. In *29th STOC*, pages 1–10, 1997.

[40] J. Håstad, R. Impagliazzo, L.A. Levin and M. Luby. A Pseudorandom Generator from any One-way Function. *SICOMP*, Volume 28, Number 4, pages 1364–1396, 1999. Combines preliminary versions by Impagliazzo et. al. in *21st STOC* (1989) and Håstad in *22nd STOC* (1990).

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

[42] N. Immerman. Nondeterministic Space is Closed Under Complementation. *SIAM Jour. on Computing*, Vol. 17, pages 760–778, 1988.

[43] R. Impagliazzo. Hard-core Distributions for Somewhat Hard Problems. In *36th FOCS*, pages 538–545, 1995.

[44] R. Impagliazzo and L.A. Levin. No Better Ways to Generate Hard NP Instances than Picking Uniformly at Random. In *Proc. of the 31st FOCS*, 1990, pages 812–821.

[45] R. Impagliazzo and A. Wigderson. P=BPP if E requires exponential circuits: Derandomizing the XOR Lemma. In *29th STOC*, pages 220–229, 1997.

[46] D.S. Johnson. The NP-Complete Column – an ongoing guide. *Jour. of Algorithms*, 1984, Vol. 4, pages 284–299.

[47] R.M. Karp. Reducibility among Combinatorial Problems. In *Complexity of Computer Computations*, R.E. Miller and J.W. Thatcher (eds.), Plenum Press, pages 85–103, 1972.

[48] R.M. Karp. Probabilistic Analysis of Algorithms. Manuscript, 1986.

[49] R.M. Karp and R.J. Lipton. Some connections between nonuniform and uniform complexity classes. In *12th STOC*, pages 302-309, 1980.

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

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

[52] D.E. Knuth. *The Art of Computer Programming*, Vol. 2 (*Seminumerical Algorithms*). Addison-Wesley Publishing Company, Inc., 1969 (first edition) and 1981 (second edition).

[53] A. Kolmogorov. Three Approaches to the Concept of "The Amount Of Information". *Probl. of Inform. Transm.*, Vol. 1/1, 1965.

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

[55] R.E. Ladner. On the Structure of Polynomial Time Reducibility. *Jour. of the ACM*, 22, 1975, pages 155–171.

[56] C. Lautemann. BPP and the Polynomial Hierarchy. *IPL*, 17, pages 215–217, 1983.

[57] L.A. Levin. Universal Search Problems. *Problemy Peredaci Informacii 9*, pages 115–116, 1973. Translated in *problems of Information Transmission 9*, pages 265–266.

[58] L.A. Levin. Randomness Conservation Inequalities: Information and Independence in Mathematical Theories. *Inform. and Control*, Vol. 61, pages 15–37, 1984.

[59] L.A. Levin. Average Case Complete Problems. *SIAM Jour. on Computing*, Vol. 15, pages 285–286, 1986.

[60] M. Li and P. Vitanyi. *An Introduction to Kolmogorov Complexity and its Applications.* Springer Verlag, August 1993.

[61] C. Lund, L. Fortnow, H. Karloff, and N. Nisan. Algebraic Methods for Interactive Proof Systems. *JACM*, Vol. 39, No. 4, pages 859–868, 1992. Preliminary version in *31st FOCS*, 1990.

[62] R. Motwani and P. Raghavan. *Randomized Algorithms*, Cambridge University Press, 1995.

[63] M. Naor. Bit Commitment using Pseudorandom Generators. *Jour. of Crypto.*, Vol. 4, pages 151–158, 1991.

[64] N. Nisan. Pseudorandom Generators for Space Bounded Computation. *Combinatorica*, Vol. 12 (4), pages 449–461, 1992.

[65] N. Nisan. $\mathcal{RL} \subseteq \mathcal{SC}$. *Journal of Computational Complexity*, Vol. 4, pages 1-11, 1994.

[66] N. Nisan and A. Wigderson. Hardness vs Randomness. *JCSS*, Vol. 49, No. 2, pages 149–167, 1994.

[67] N. Nisan and D. Zuckerman. Randomness is Linear in Space. *JCSS*, Vol. 52 (1), pages 43–52, 1996.

[68] W.J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *JCSS*, Vol. 4 (2), pages 177-192, 1970.

[69] R. Shaltiel Recent developments in explicit constructions of extractors. In the *Bulletin of the European Association for Theoretical Computer Science*, Vol. 77, June 2002, pages 67–95.

[70] A. Shamir. IP = PSPACE. *JACM*, Vol. 39, No. 4, pages 869–877, 1992. Preliminary version in *31st FOCS*, 1990.

[71] C.E. Shannon. A mathematical theory of communication. *Bell Sys. Tech. Jour.*, Vol. 27, pages 623–656, 1948.

[72] M. Sipser. A Complexity Theoretic Approach to Randomness. In *15th STOC*, pages 330–335, 1983.

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

[74] R.J. Solomonoff. A Formal Theory of Inductive Inference. *Inform. and Control*, Vol. 7/1, pages 1–22, 1964.

[75] L.J. Stockmeyer. The Polynomial-Time Hierarchy. *Theoretical Computer Science*, Vol. 3, pages 1–22, 1977.

[76] L. Stockmeyer. The Complexity of Approximate Counting. In *15th STOC*, pages 118–126, 1983.

[77] R. Szelepcsenyi. A Method of Forced Enumeration for Nondeterministic Automata. *Acta Informatica*, Vol. 26, pages 279–284, 1988.

[78] S. Vadhan. A Study of Statistical Zero-Knowledge Proofs. PhD Thesis, Department of Mathematics, MIT, 1999.

[79] L.G. Valiant. The Complexity of Computing the Permanent. *Theoretical Computer Science*, Vol. 8, pages 189–201, 1979.

[80] L.G. Valiant and V.V. Vazirani. NP Is as Easy as Detecting Unique Solutions. *Theoretical Computer Science*, Vol. 47 (1), pages 85–93, 1986.

[81] A.C. Yao. Theory and Application of Trapdoor Functions. In *23rd FOCS*, pages 80–91, 1982.