Chapter 6

Randomness and Counting

I owe this almost atrocious variety to an institution which other republics do not know or which operates in them in an imperfect and secret manner: the lottery.

Jorge Luis Borges, The Lottery In Babylon

So far, our approach to computing devices was somewhat conservative: we thought of them as executing a deterministic rule. A more liberal and quite realistic approach, which is pursued in this chapter, considers computing devices that use a probabilistic rule. This relaxation has an immediate impact on the notion of efficient computation, which is consequently associated with *probabilistic* polynomial-time computations rather than with deterministic (polynomial-time) ones. We stress that the association of efficient computation with probabilistic polynomial-time computation makes sense provided that the failure probability of the latter is negligible (which means that it may be safely ignored).

The quantitative nature of the failure probability of probabilistic algorithm provides one connection between probabilistic algorithms and counting problems. The latter are indeed a new type of computational problems, and our focus is on counting efficiently recognizable objects (e.g., NP-witnesses for a given instance of set in $\mathcal{NP}$). Randomized procedures turn out to play an important role in the study of such counting problems.

Summary: Focusing on probabilistic polynomial-time algorithms, we consider various types of probabilistic failure of such algorithms (e.g., actual error versus failure to produce output). This leads to the formulation of complexity classes such as $\mathcal{BPP}$, $\mathcal{RP}$, and $\mathcal{ZPP}$. The results presented include the existence of (non-uniform) families of polynomial-size circuits that emulate probabilistic polynomial-time algorithms (i.e., $\mathcal{BPP} \subset \mathcal{P}/\text{poly}$) and the fact that $\mathcal{BPP}$ resides in the (second level of the) Polynomial-time Hierarchy (i.e., $\mathcal{BPP} \subseteq \Sigma_2$).

We then turn to counting problems; specifically, counting the number of solutions for an instance of a search problem in $\mathcal{PC}$ (or, equivalently,

counting the number of NP-witnesses for an instance of a decision problem in $\mathcal{NP}$). We distinguish between exact counting and approximate counting (in the sense of relative approximation). In particular, while any problem in $\mathcal{PH}$ is reducible to the exact counting class $\#\mathcal{P}$, approximate counting (for $\#\mathcal{P}$) is (probabilisticly) reducible to $\mathcal{NP}$.

In general, counting problems exhibit a "richer structure" than the corresponding search (and decision) problems, even when considering only natural problems. For example, some counting problems are hard in the exact version (e.g., are $\#\mathcal{P}$-complete) but easy to approximate, while others are NP-hard to approximate. In some cases $\#\mathcal{P}$-completeness is due to the very same reduction that establishes the $\mathcal{NP}$-completeness of the corresponding decision problem, whereas in other cases new reductions are required (often because the corresponding decision problem is not $\mathcal{NP}$-complete but is rather in $\mathcal{P}$).

We also consider two other types of computational problems that are related to approximate counting. The first type refers to promise problems, called unique solution problems, in which the solver is guaranteed that the instance has at most one solution. Many NP-complete problems are randomly reducible to the corresponding unique solution problems. Lastly, we consider the problem of generating almost uniformly distributed solutions, and show that in many cases this problem is computationally equivalent to approximately counting the number of solutions.

Prerequisites: We assume basic familiarity with elementary probability theory (see Appendix D.1). In Section 6.2 we will rely extensively on formulations presented in Section 2.1 (i.e., the "NP search problem" class $\mathcal{PC}$ as well as the sets $R(x) \stackrel{\text{def}}{=} \{y : (x,y) \in R\}$, and $S_R \stackrel{\text{def}}{=} \{x : R(x) \neq \emptyset\}$ defined for every $R \in \mathcal{PC}$). In Sections 6.2.2–6.2.4 we shall extensively use various hashing functions and their properties, as presented in Appendix D.2.

6.1 Probabilistic Polynomial-Time

Considering algorithms that utilize random choices, we extend our notion of *efficient algorithms* from *deterministic* polynomial-time algorithms to *probabilistic* polynomial-time algorithms. Two conflicting questions that arise are whether it is reasonable to allow randomized computational steps and whether adding such steps buys us anything.

We first note that random events are an important part of our modeling of the world. We stress that this does not necessarily mean that we assert that the world *per se* includes genuine random choices, but rather that it is beneficial to model the world as including random choices (i.e., some phenomena appear to us as if they are random in some sense). Furthermore, it seems feasible to generate random-looking events (e.g., the outcome of a toss coin).[1] Thus, postulating that seemingly random choices can be generated by a computer is quite natural (and is in fact common practice). At the very least, this postulate yields an intuitive model of computation and the study of such a model is of natural concern.

This leads to the question of whether augmenting the computational model with the ability to make random choices buys us anything. Although randomization is known to be essential in several computational settings (e.g., cryptography (cf., Appendix C) and sampling (cf., Appendix D.3)), the question is whether randomization is useful in the context of solving decision (and search) problems. This is indeed a very good question, which is further discussed in §6.1.2.1. In fact, one of the main goals of the current section is putting this question forward. To demonstrate the potential benefit of randomized algorithms, we provide a few examples (cf., §6.1.2.2, §6.1.3.1 and §6.1.5.2).

6.1.1 Basic modeling issues

Rigorous models of probabilistic (or randomized) algorithms are defined by natural extensions of the basic machine model. We will exemplify this approach by describing the model of probabilistic Turing machines, but we stress that (again) the specific choice of the model is immaterial (as long as it is "reasonable"). A probabilistic Turing machine is defined exactly as a non-deterministic machine (see the first item of Definition 2.7), *but the definition of its computation is fundamentally different*. Specifically, whereas Definition 2.7 refers to the question of whether or not there exists a computation of the machine that (started on a specific input) reaches a certain configuration, in the case of probabilistic Turing machines we refer to *the probability that this event occurs, when at each step a choice is selected uniformly among the relevant possible choices available at this step*. That is, if the transition function of the machine maps the current state-symbol pair to several possible triples, then in the corresponding probabilistic computation one of these triples is selected at random (with equal probability) and the next configuration is determined accordingly. These random choices may be viewed as the internal coin tosses of the machine. (Indeed, as in the case of non-deterministic machines, we may assume without loss of generality that the transition function of the machine maps each state-symbol pair to *exactly* two possible triples; see Exercise 2.4.)

We stress the fundamental difference between the fictitious model of a non-deterministic machine and the realistic model of a probabilistic machine. In the case of a non-deterministic machine we consider the *existence* of an adequate sequence of choices (leading to a desired outcome), and ignore the question of how these choices are actually made. In fact, the selection of such a sequence of choices is merely a mental experiment. In contrast, in the case of a probabilistic machine, at each step a real random choice is actually made (uniformly among a set of predetermined

[1] Different perspectives on the question of the feasibility of randomized computation are offered in Chapter 8 and Appendix D.4. The pivot of Chapter 8 is the distinction between being actually random and looking random (to computationally restricted observers). In contrast, Appendix D.4 refers to various notions of randomness and to the feasibility of transforming weak forms of randomness into almost perfect forms.

possibilities), and we consider the *probability* of reaching a desired outcome.

In view of the foregoing, we consider the output distribution of such a probabilistic machine on fixed inputs; that is, for a probabilistic machine M and string $x \in \{0,1\}^*$, we denote by $M(x)$ the output distribution of M when invoked on input x, where the probability is taken uniformly over the machine's internal coin tosses. Needless to say, we will consider the probability that $M(x)$ is a "correct" answer; that is, in the case of a search problem (resp., decision problem) we will be interested in the probability that $M(x)$ is a valid solution for the instance x (resp., represents the correct decision regarding x).

The foregoing description views the internal coin tosses of the machine as taking place on-the-fly; that is, these coin tosses are performed *on-line* by the machine itself. An alternative model is one in which the sequence of coin tosses is provided by an external device, on a special "random input" tape. In such a case, we view these coin tosses as performed *off-line*. Specifically, we denote by $M'(x,r)$ the (uniquely defined) output of the residual deterministic machine M', when given the (primary) input x and random input r. Indeed, M' is a deterministic machine that takes two inputs (the first representing the actual input and the second representing the "random input"), but we consider the random variable $M(x) \stackrel{\rm def}{=} M'(x, U_{\ell(|x|)})$, where $\ell(|x|)$ denotes the number of coin tosses "expected" by $M'(x,\cdot)$.

These two perspectives on probabilistic algorithms are closely related: Clearly, the aforementioned residual deterministic machine M' yields the on-line machine M that on input x selects at random a string r of adequate length, and invokes $M'(x,r)$. On the other hand, the computation of any on-line machine M is captured by the residual machine M' that emulates the actions of $M(x)$ based on an auxiliary input r (obtained by M' and representing a possible outcome of the internal coin tosses of M). (Indeed, there is no harm in supplying more coin tosses than are actually used by M, and so the length of the aforementioned auxiliary input may be set to equal the time complexity of M.) For sake of clarity and future reference, we summarize the foregoing discussion in the following definition.

Definition 6.1 (on-line and off-line formulations of probabilistic polynomial-time):

- *We say that M is a* on-line probabilistic polynomial-time machine *if there exists a polynomial p such that when invoked on any input $x \in \{0,1\}^*$, machine M always halts within at most $p(|x|)$ steps* (regardless of the outcome of its internal coin tosses). *In such a case $M(x)$ is a random variable.*

- *We say that M' is a* off-line probabilistic polynomial-time machine *if there exists a polynomial p such that, for every $x \in \{0,1\}^*$ and $r \in \{0,1\}^{p(|x|)}$, when invoked on the* primary input *x and the* random-input *sequence r, machine M' halts within at most $p(|x|)$ steps. In such a case, we will consider the random variable $M'(x, U_{p(|x|)})$, where U_m denotes a random variable uniformly distributed over $\{0,1\}^m$.*

Clearly, in the context of time-complexity, the on-line and off-line formulations are equivalent (i.e., given an on-line probabilistic polynomial-time machine we can derive a functionally equivalent off-line (probabilistic polynomial-time) machine, and vice versa). Thus, in the sequel, we will freely use whichever is more convenient.

Failure probability. A major aspect of randomized algorithms (probabilistic machines) is that they may fail (see Exercise 6.1). That is, with some specified ("failure") probability, these algorithms may fail to produce the desired output. We discuss two aspects of this failure: its *type* and its *magnitude*.

1. The type of failure is a qualitative notion. One aspect of this type is whether, in case of failure, the algorithm produces a wrong answer or merely an indication that it failed to find a correct answer. Another aspect is whether failure may occur on all instances or merely on certain types of instances. Let us clarify these aspects by considering three natural types of failure, giving rise to three different types of algorithms.

   (a) The most liberal notion of failure is the one of two-sided error. This term originates from the setting of decision problems, where it means that (in case of failure) the algorithm may err in both directions (i.e., it may rule that a yes-instance is a no-instance, and vice versa). In the case of search problems two-sided error means that, when failing, the algorithm may output a wrong answer on any input. That is, the algorithm may falsely rule that the input has no solution and it may also output a wrong solution (both in case the input has a solution and in case it has no solution).

   (b) An intermediate notion of failure is the one of one-sided error. Again, the term originates from the setting of decision problems, where it means that the algorithm may err only in one direction (i.e., either on yes-instances or on no-instances). Indeed, there are two natural cases depending on whether the algorithm errs on yes-instances but not on no-instances, or the other way around. Analogous cases occur also in the setting of search problems. In one case the algorithm never outputs a wrong solution but may falsely rule that the input has no solution. In the other case the indication that an input has no solution is never wrong, but the algorithm may output a wrong solution.

   (c) The most conservative notion of failure is the one of zero-sided error. In this case, the algorithm's failure amounts to indicating its failure to find an answer (by outputting a special `don't know` symbol). We stress that in this case the algorithm *never provides a wrong answer*.

   Indeed, the forgoing discussion ignores the probability of failure, which is the subject of the next item.

2. The magnitude of failure is a quantitative notion. It refer to the probability that the algorithm fails, where the type of failure is fixed (e.g., as in the forgoing discussion).

   When actually using a randomized algorithm we typically wish its failure probability to be negligible, which intuitively means that the failure event is so rare that it can be ignored in practice. Formally, we say that a quantity is negligible if, as a function of the relevant parameter (e.g., the input length), this quantity vanishes faster than the reciprocal of any positive polynomial.

For ease of presentation, we sometimes consider alternative upper-bounds on the probability of failure. These bounds are selected in a way that allows (and in fact facilitates) "error reduction" (i.e., converting a probabilistic polynomial-time algorithm that satisfies such an upper-bound into one in which the failure probability is negligible). For example, in the case of two-sided error we need to be able to distinguish the correct answer from wrong answers by sampling, and in the other types of failure "hitting" a correct answer suffices.

In the following three sections (i.e., Sections 6.1.2–6.1.4), we will discuss complexity classes corresponding to the aforementioned three *types* of failure. For sake of simplicity, the failure probability itself will be set to a constant that allows error reduction.

Randomized reductions. Before turning to the more detailed discussion, we mention that randomized reductions play an important role in complexity theory. Such reductions can be defined analogously to the standard Cook-Reductions (resp., Karp-reductions), and again a discussion of the type and magnitude of the failure probability is in place. For clarity, we spell-out the two-sided error versions.

- In analogy to Definition 2.9, we say that a problem Π is probabilistic polynomial-time reducible to a problem Π' if there exists a probabilistic polynomial-time oracle machine M such that, for every function f that solves Π' and for every x, with probability at least $1-\mu(|x|)$, the output $M^f(x)$ is a correct solution to the instance x, where μ is a negligible function.

- In analogy to Definition 2.11, we say that a decision problem S is reducible to a decision problem S' via a randomized Karp-reduction if there exists a probabilistic polynomial-time algorithm A such that, for every x, it holds that $\Pr[\chi_{S'}(A(x)) = \chi_S(x)] \geq 1-\mu(|x|)$, where χ_S (resp., $\chi_{S'}$) is the characteristic function of S (resp., S') and μ is a negligible function.

These reductions preserve efficient solvability and are transitive: see Exercise 6.2.

6.1.2 Two-sided error: The complexity class BPP

In this section we consider the most liberal notion of probabilistic polynomial-time algorithms that is still meaningful. We allow the algorithm to err on each input, but require the error probability to be *negligible*. The latter requirement guarantees the usefulness of such algorithms, because in reality we may ignore the negligible error probability.

Before focusing on the decision problem setting, let us say a few words on the search problem setting (see Definition 1.1). Following the previous paragraph, we say that a probabilistic (polynomial-time) algorithm A solves the search problem of the relation R if for every $x \in S_R$ (i.e., $R(x) \stackrel{\text{def}}{=} \{y : (x,y) \in R\} \neq \emptyset$) it holds that $\Pr[A(x) \in R(x)] > 1 - \mu(|x|)$ and for every $x \notin S_R$ it holds that $\Pr[A(x) = \perp] > 1-\mu(|x|)$, where μ is a negligible function. Note that we did not require that, when invoked on input x that has a solution (i.e., $R(x) \neq \emptyset$), the algorithm always outputs the same solution. Indeed, a stronger requirement is that for every such x there exists $y \in R(x)$ such that $\Pr[A(x) = y] > 1 - \mu(|x|)$. The latter version and quantitative relaxations of it allow for error-reduction (see Exercise 6.3).

Turning to decision problems, we consider probabilistic polynomial-time algorithms that err with negligible probability. That is, we say that a probabilistic (polynomial-time) algorithm A decides membership in S if for every x it holds that $\Pr[A(x) = \chi_S(x)] > 1 - \mu(|x|)$, where χ_S is the characteristic function of S (i.e., $\chi_S(x) = 1$ if $x \in S$ and $\chi_S(x) = 0$ otherwise) and μ is a negligible function. The class of decision problems that are solvable by probabilistic polynomial-time algorithms is denoted $\mathcal{BPP}$, standing for Bounded-error Probabilistic Polynomial-time. Actually, the standard definition refers to machines that err with probability at most $1/3$.

Definition 6.2 (the class $\mathcal{BPP}$): *A decision problem S is in $\mathcal{BPP}$ if there exists a probabilistic polynomial-time algorithm A such that for every $x \in S$ it holds that* $\Pr[A(x) = 1] \geq 2/3$ *and for every $x \notin S$ it holds that* $\Pr[A(x) = 0] \geq 2/3$.

The choice of the constant $2/3$ is immaterial, and any other constant greater than $1/2$ will do (and yields the very same class). Similarly, the complementary constant $1/3$ can be replaced by various negligible functions (while preserving the class). Both facts are special cases of the robustness of the class, discussed next, which is established using the process of error reduction.

Error reduction (or confidence amplification). For $\varepsilon : \mathbb{N} \to (0, 0.5)$, let $\mathcal{BPP}_\varepsilon$ denote the class of decision problems that can be solved in probabilistic polynomial-time with error probability upper-bounded by ε; that is, $S \in \mathcal{BPP}_\varepsilon$ if there exists a probabilistic polynomial-time algorithm A such that for every x it holds that $\Pr[A(x) \neq \chi_S(x)] \leq \varepsilon(|x|)$. By definition, $\mathcal{BPP} = \mathcal{BPP}_{1/3}$. However, a wide range of other classes also equal $\mathcal{BPP}$. In particular, we mention two extreme cases:

1. For every positive polynomial p and $\varepsilon(n) = (1/2) - (1/p(n))$, the class $\mathcal{BPP}_\varepsilon$ equals $\mathcal{BPP}$. That is, any error that is ("noticeably") bounded away from $1/2$ (i.e., error $(1/2) - (1/\text{poly}(n))$) can be reduced to an error of $1/3$.

2. For every positive polynomial p and $\varepsilon(n) = 2^{-p(n)}$, the class $\mathcal{BPP}_\varepsilon$ equals $\mathcal{BPP}$. That is, an error of $1/3$ can be further reduced to an exponentially vanishing error.

Both facts are proved by invoking the weaker algorithm (i.e., the one having a larger error probability bound) for an adequate number of times, and ruling 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. The success probability of such a process is analyzed by applying an adequate Law of Large Numbers (see Exercise 6.4).

6.1.2.1 On the power of randomization

Let us turn back to the natural question raised at the beginning of Section 6.1; that is, *was anything gained by extending the definition of efficient computation to include also probabilistic polynomial-time ones.*

This phrasing seems too generic. We certainly gained the ability to toss coins (and generate various distributions). More concretely, randomized algorithms are essential in many settings (see, e.g., Chapter 9, Section 10.1.2, Appendix C, and Appendix D.3) and seem essential in others (see, e.g., Sections 6.2.2–6.2.4). What we mean to ask here is *whether allowing randomization increases the power of polynomial-time algorithms also in the restricted context of solving decision and search problems?*

The question is whether $\mathcal{BPP}$ extends beyond $\mathcal{P}$ (where clearly $\mathcal{P} \subseteq \mathcal{BPP}$). It is commonly conjectured that the answer is negative. Specifically, under some reasonable assumptions, it holds that $\mathcal{BPP} = \mathcal{P}$ (see Part 1 of Theorem 8.19). We note, however, that a polynomial slow-down occurs in the proof of the latter result; that is, randomized algorithms that run in time $t(\cdot)$ are emulated by deterministic algorithms that run in time $\mathrm{poly}(t(\cdot))$. This slow-down seems inherent to the aforementioned approach (see §8.3.3.2). Furthermore, for some concrete problems (most notably primality testing (cf. §6.1.2.2)), the known probabilistic polynomial-time algorithm is significantly faster (and conceptually simpler) than the known deterministic polynomial-time algorithm. Thus, we believe that even in the context of decision problems, the notion of probabilistic polynomial-time algorithms is advantageous.

We note that the fundamental nature of $\mathcal{BPP}$ will remain intact even in the (rather unlikely) case that it turns out that randomization offers no computational advantage (i.e., even if every problem that can be decided in probabilistic polynomial-time can be decided by a deterministic algorithm of essentially the same complexity). Such a result would address a fundamental question regarding the power of randomness.[2] We now turn from the foregoing philosophical (and partially hypothetical) discussion to a concrete discussion of what is known about $\mathcal{BPP}$.

BPP is in the Polynomial-Time Hierarchy: While it may be that $\mathcal{BPP} = \mathcal{P}$, it is not known whether or not $\mathcal{BPP}$ is contained in $\mathcal{NP}$. The source of trouble is the two-sided error probability of $\mathcal{BPP}$, which is incompatible with the absolute rejection of no-instances required in the definition of $\mathcal{NP}$ (see Exercise 6.8). In view of this ignorance, it is interesting to note that $\mathcal{BPP}$ resides in the second level of the Polynomial-Time Hierarchy (i.e., $\mathcal{BPP} \subseteq \Sigma_2$). This is a corollary of Theorem 6.9.

Trivial derandomization. A straightforward way of eliminating randomness from an algorithm is trying all possible outcomes of its internal coin tosses, collecting the relevant statistics and deciding accordingly. This yields $\mathcal{BPP} \subseteq \mathcal{PSPACE} \subseteq$

[2] By analogy, establishing that $\mathcal{IP} = \mathcal{PSPACE}$ (cf. Theorem 9.4) does not diminish the importance of any of these classes, because each class models something fundamentally different.

$\mathcal{EXP}$, which is considered the trivial derandomization of $\mathcal{BPP}$. In Section 8.3 we will consider various non-trivial derandomizations of $\mathcal{BPP}$, which are known under various intractability assumptions. The interested reader, who may be puzzled by the connection between derandomization and computational difficulty, is referred to Chapter 8.

Non-uniform derandomization. In many settings (and specifically in the context of solving search and decision problems), the power of randomization is superseded by the power of non-uniform advice. Intuitively, the non-uniform advice may specify a sequence of coin tosses that is good for all (primary) inputs of a specific length. In the context of solving search and decision problems, such an advice must be good for *each* of these inputs[3], and thus its existence is guaranteed only if the error probability is low enough (so as to support a union bound). The latter condition can be guaranteed by error-reduction, and thus we get the following result.

Theorem 6.3 *$\mathcal{BPP}$ is* (strictly) *contained in* $\mathcal{P}$/poly.

Proof: Recall that $\mathcal{P}$/poly contains undecidable problems (Theorem 3.7), which are certainly not in $\mathcal{BPP}$. Thus, we focus on showing that $\mathcal{BPP} \subseteq \mathcal{P}$/poly. By the discussion regarding error-reduction, for every $S \in \mathcal{BPP}$ there exists a (deterministic) polynomial-time algorithm A and a polynomial p such that for every x it holds that $\Pr[A(x, U_{p(|x|)}) \neq \chi_S(x)] < 2^{-|x|}$. Using a union bound, it follows that $\Pr_{r \in \{0,1\}^{p(n)}}[\exists x \in \{0,1\}^n \text{ s.t. } A(x,r) \neq \chi_S(x)] < 1$. Thus, for every $n \in \mathbb{N}$, there exists a string $r_n \in \{0,1\}^{p(n)}$ such that for every $x \in \{0,1\}^n$ it holds that $A(x, r_n) = \chi_S(x)$. Using such a sequence of r_n's as advice, we obtain the desired non-uniform machine (establishing $S \in \mathcal{P}$/poly). ■

Digest. The proof of Theorem 6.3 combines error-reduction with a simple application of the Probabilistic Method (cf. [10]), where the latter refers to proving the existence of an object by analyzing the probability that a random object is adequate. In this case, we sought a non-uniform advice, and proved it existence by analyzing the probability that a random advice is good. The latter event was analyzed by identifying the space of possible advice with the set of possible sequences of internal coin tosses of a randomized algorithm.

6.1.2.2 A probabilistic polynomial-time primality test

Teaching note: Although primality has been recently shown to be in $\mathcal{P}$, we believe that the following example provides a nice illustration to the power of randomized algorithms.

[3] In other contexts (see, e.g., Chapters 7 and 8), it suffices to have an advice that is good on the average, where the average is taken over all relevant (primary) inputs.

We present a simple probabilistic polynomial-time algorithm for deciding whether or not a given number is a prime. The only Number Theoretic facts that we use are:

Fact 1: For every prime $p > 2$, each quadratic residue mod p has exactly two square roots mod p (and they sum-up to p).[4]

Fact 2: For every (odd and non-integer-power) composite number N, each quadratic residue mod N has at least four square roots mod N.

Our algorithm uses as a black-box an algorithm, denoted `sqrt`, that given a prime p and a quadratic residue mod p, denoted s, returns the smallest among the two modular square roots of s. There is no guarantee as to what the output is in the case that the input is not of the aforementioned form (and in particular in the case that p is not a prime). Thus, we actually present a probabilistic polynomial-time reduction of testing primality to extracting square roots modulo a prime (which is a search problem with a promise; see Section 2.4.1).

Construction 6.4 (the reduction): *On input a natural number $N > 2$ do*

1. *If N is either even or an integer-power*[5] *then* `reject`.
2. *Uniformly select $r \in \{1, ..., N-1\}$, and set $s \leftarrow r^2 \bmod N$.*
3. *Let $r' \leftarrow$ `sqrt`(s, N). If $r' \equiv \pm r \pmod N$ then* `accept` *else* `reject`.

Indeed, in the case that N is composite, the reduction invokes `sqrt` on an illegitimate input (i.e., it makes a query that violates the promise of the problem at the target of the reduction). In such a case, there is not guarantee as to what `sqrt` answers, but actually a bluntly wrong answer only plays in our favor. In general, we will show that if N is composite, then the reduction rejects with probability at least $1/2$, regardless of how `sqrt` answers. We mention that there exists a probabilistic polynomial-time algorithm for implementing `sqrt` (see Exercise 6.16).

Proposition 6.5 *Construction 6.4 constitutes a probabilistic polynomial-time reduction of testing primality to extracting square roots module a prime. Furthermore, if the input is a prime then the reduction always accepts, and otherwise it rejects with probability at least* $1/2$.

We stress that Proposition 6.5 refers to the reduction itself; that is, `sqrt` is viewed as a ("perfect") oracle that, for every prime P and quadratic residue $s \pmod P$, returns $r < s/2$ such that $r^2 \equiv s \pmod P$. Combining Proposition 6.5 with a probabilistic polynomial-time algorithm that computes `sqrt` with negligible error probability, we obtain that testing primality is in $\mathcal{BPP}$.

[4] That is, for every $r \in \{1, ..., p-1\}$, the equation $x^2 \equiv r^2 \pmod p$ has two solutions modulo p (i.e., r and $p - r$).

[5] This can be checked by scanning all possible powers $e \in \{2, ..., \log_2 N\}$, and (approximately) solving the equation $x^e = N$ for each value of e (i.e., finding the smallest integer i such that $i^e \geq N$). Such a solution can be found by binary search.

Proof: By Fact 1, on input a prime number N, Construction 6.4 always accepts (because in this case, for every $r \in \{1, ..., N-1\}$, it holds that $\texttt{sqrt}(r^2 \bmod N, N) \in \{r, N-r\}$). On the other hand, suppose that N is an odd composite that is not an integer-power. Then, by Fact 2, each quadratic residue s has at least four square roots, and each of these square roots is equally likely to be chosen at Step 2 (in other words, s yields no information regarding which of its modular square roots was selected in Step 2). Thus, for every such s, the probability that either $\texttt{sqrt}(s, N)$ or $N - \texttt{sqrt}(s, N)$ equal the root chosen in Step 2 is at most 2/4. It follows that, on input a composite number, the reduction rejects with probability at least 1/2. ■

Reflection: Construction 6.4 illustrates an interesting aspect of randomized algorithms (or rather reductions); that is, their ability to take advantage of information that is unknown to the invoked subroutine. Specifically, Construction 6.4 generates a problem instance (N, s), which hides crucial information (regarding how s was generated). Any subroutine that answers correctly in the case that N is prime provides probabilistic evidence that N is a prime, where the probability space refers to the missing information (regarding how s was generated in the case that N is composite).

Comment. Testing primality is actually in $\mathcal{P}$. However, the deterministic algorithm demonstrating this fact is more complex than Construction 6.4 (and its analysis is even more complicated).

6.1.3 One-sided error: The complexity classes RP and coRP

In this section we consider notions of probabilistic polynomial-time algorithms having one-sided error. The notion of one-sided error refers to a natural partition of the set of instances; that is, yes-instances versus no-instances in the case of decision problems, and instances having solution versus instances having no solution in the case of search problems. We focus on decision problems, and comment that an analogous treatment can be provided for search problems (see Exercise 6.3).

Definition 6.6 (the class $\mathcal{RP}$)[6]: *A decision problem S is in $\mathcal{RP}$ if there exists a probabilistic polynomial-time algorithm A such that for every $x \in S$ it holds that* $\Pr[A(x)=1] \geq 1/2$ *and for every $x \notin S$ it holds that* $\Pr[A(x)=0] = 1$.

The choice of the constant 1/2 is immaterial, and any other constant greater than zero will do (and yields the very same class). Similarly, this constant can be replaced by $1 - \mu(|x|)$ for various negligible functions μ (while preserving the class). Both facts are special cases of the robustness of the class (see Exercise 6.5).

Observe that $\mathcal{RP} \subseteq \mathcal{NP}$ (see Exercise 6.8) and that $\mathcal{RP} \subseteq \mathcal{BPP}$ (by the aforementioned error-reduction). Defining $\text{co}\mathcal{RP} = \{\{0,1\}^* \setminus S : S \in \mathcal{RP}\}$, note

[6] The initials RP stands for Random Polynomial-time, which fails to convey the restricted type of error allowed in this class. The only nice feature of this notation is that it is reminiscent of NP, thus reflecting the fact that $\mathcal{RP}$ is a randomized polynomial-time class that is contained in $\mathcal{NP}$.

that co$\mathcal{RP}$ corresponds to the opposite direction of one-sided error probability. That is, *a decision problem S is in* co$\mathcal{RP}$ *if there exists a probabilistic polynomial-time algorithm A such that for every $x \in S$ it holds that* $\Pr[A(x)=1] = 1$ *and for every $x \notin S$ it holds that* $\Pr[A(x)=0] \geq 1/2$.

6.1.3.1 Testing polynomial identity

An appealing example of a one-sided error randomized algorithm refers to the problem of determining whether two polynomials are identical. For simplicity, we assume that we are given an oracle for the evaluation of each of the two polynomials. An alternative presentation that refers to polynomials that are represented by arithmetic circuits (cf. Appendix B.3) yields a standard decision problem in co$\mathcal{RP}$ (see Exercise 6.17). Either way, we refer to multi-variant polynomials and to the question of whether they are identical over any field (or, equivalently, whether they are identical over a sufficiently large finite field). Note that it suffices to consider finite fields that are larger than the degree of the two polynomials.

Construction 6.7 (Polynomial-Identity Test): *Let n be an integer and F be a finite field. Given black-box access to $p, q : \mathrm{F}^n \to \mathrm{F}$, uniformly select $r_1, ..., r_n \in \mathrm{F}$, and accept if and only if $p(r_1, ..., r_n) = q(r_1, ..., r_n)$.*

Clearly, if $p \equiv q$ then Construction 6.7 always accepts. The following lemma implies that if p and q are different polynomials, each of total degree at most d over the finite field F, then Construction 6.7 accepts with probability at most $d/|\mathrm{F}|$.

Lemma 6.8 *Let $p : \mathrm{F}^n \to \mathrm{F}$ be a non-zero polynomial of total degree d over the finite field F. Then*

$$\Pr_{r_1,...,r_n \in \mathrm{F}}[p(r_1, ..., r_n) = 0] \;\leq\; \frac{d}{|\mathrm{F}|}.$$

Proof: The lemma is proven by induction on n. The base case of $n = 1$ follows immediately by the Fundamental Theorem of Algebra (i.e., any non-zero univariate polynomial of degree d has at most d distinct roots). In the induction step, we write p as a polynomial in its first variable with coefficients that are polynomials in the other variables. That is,

$$p(x_1, x_2, ..., x_n) = \sum_{i=0}^{d} p_i(x_2, ..., x_n) \cdot x_1^i$$

where p_i is a polynomial of total degree at most $d-i$. Let i be the largest integer for which p_i is not identically zero. Dismissing the case $i = 0$ and using the induction hypothesis, we have

$$\begin{aligned}
&\Pr_{r_1,r_2,...,r_n}[p(r_1, r_2, ..., r_n) = 0] \\
&\quad\leq\; \Pr_{r_2,...,r_n}[p_i(r_2, ..., r_n) = 0] \\
&\qquad + \Pr_{r_1,r_2,...,r_n}[p(r_1, r_2, ..., r_n) = 0 \,|\, p_i(r_2, ..., r_n) \neq 0] \\
&\quad\leq\; \frac{d-i}{|\mathrm{F}|} + \frac{i}{|\mathrm{F}|}
\end{aligned}$$

where the second term is bounded by fixing any sequence $r_2, ..., r_n$ for which $p_i(r_2, ..., r_n) \neq 0$ and considering the univariate polynomial $p'(x) \stackrel{\rm def}{=} p(x, r_2, ..., r_n)$ (which by hypothesis is a non-zero polynomial of degree i). ■

Reflection: Lemma 6.8 may be viewed as asserting that for every non-zero polynomial of degree d over F at least a $1-(d/|\mathrm{F}|)$ fraction of its domain does not evaluate to zero. Thus, if $d \ll |\mathrm{F}|$ then most of the evaluation points constitute a witness for the fact that the polynomial is non-zero. We know of no efficient deterministic algorithm that, given a representation of the polynomial via an arithmetic circuit, finds such a witness. Indeed, Construction 6.7 attempts to find a witness by merely selecting it at random.

6.1.3.2 Relating BPP to RP

A natural question regarding probabilistic polynomial-time algorithms refers to the relation between two-sided and one-sided error probability. For example, *is $\mathcal{BPP}$ contained in $\mathcal{RP}$?* Loosely speaking, we show that $\mathcal{BPP}$ is reducible to co$\mathcal{RP}$ by *one-sided error* randomized Karp-reductions, where the actual statement refers to the promise problem versions of both classes (briefly defined in the following paragraph). Note that $\mathcal{BPP}$ is trivially reducible to co$\mathcal{RP}$ by *two-sided error* randomized Karp-reductions, whereas a deterministic Karp-reduction of $\mathcal{BPP}$ to co$\mathcal{RP}$ would imply $\mathcal{BPP} = \mathrm{co}\mathcal{RP} = \mathcal{RP}$ (see Exercise 6.9).

First, we refer the reader to the general discussion of promise problems in Section 2.4.1. Analogously to Definition 2.31, we say that the promise problem $\Pi = (S_{\rm yes}, S_{\rm no})$ is in (the promise problem extension of) $\mathcal{BPP}$ *if there exists a probabilistic polynomial-time algorithm A such that for every $x \in S_{\rm yes}$ it holds that* $\mathsf{Pr}[A(x)=1] \geq 2/3$ *and for every $x \in S_{\rm no}$ it holds that* $\mathsf{Pr}[A(x)=0] \geq 2/3$. Similarly, Π is in co$\mathcal{RP}$ if for every $x \in S_{\rm yes}$ it holds that $\mathsf{Pr}[A(x) = 1] = 1$ and for every $x \in S_{\rm no}$ it holds that $\mathsf{Pr}[A(x)=0] \geq 1/2$. Probabilistic reductions among promise problems are defined by adapting the conventions of Section 2.4.1; specifically, queries that violate the promise at the target of the reduction may be answered arbitrarily.

Theorem 6.9 *Any problem in $\mathcal{BPP}$ is reducible by a one-sided error randomized Karp-reduction to* co$\mathcal{RP}$, where co$\mathcal{RP}$ (and possibly also $\mathcal{BPP}$) denotes the corresponding class of promise problems. *Specifically, the reduction always maps a* no*-instance to a* no*-instance.*

It follows that $\mathcal{BPP}$ is reducible by a one-sided error randomized Cook-reduction to $\mathcal{RP}$. Thus, using the conventions of Section 3.2.2 and referring to classes of promise problems, we may write $\mathcal{BPP} \subseteq \mathcal{RP}^{\mathcal{RP}}$. In fact, since $\mathcal{RP}^{\mathcal{RP}} \subseteq \mathcal{BPP}^{\mathcal{BPP}} = \mathcal{BPP}$, we have $\mathcal{BPP} = \mathcal{RP}^{\mathcal{RP}}$. Theorem 6.9 may be paraphrased as saying that the combination of the one-sided error probability of the reduction and the one-sided error probability of co$\mathcal{RP}$ can account for the two-sided error probability of $\mathcal{BPP}$. We warn that this statement is not a triviality like $1 + 1 = 2$, and in particular we do not know whether it holds for classes of standard decision problems (rather than for the classes of promise problems considered in Theorem 6.9).

Proof: Recall that we can easily reduce the error probability of BPP-algorithms, and derive probabilistic polynomial-time algorithms of exponentially vanishing error probability. But this does not eliminate the error altogether (not even on "one side"). In general, there seems to be no hope to eliminate the error, unless we (either do something earth-shaking or) *change the setting as done when allowing a one-sided error randomized reduction to a problem in* co$\mathcal{RP}$. The latter setting can be viewed as a two-move randomized game (i.e., a random move by the reduction followed by a random move by the decision procedure of co$\mathcal{RP}$), and it enables applying different quantifiers to the two moves (i.e., allowing error in one direction in the first quantifier and error in the other direction in the second quantifier). In the next paragraph, which is inessential to the actual proof, we illustrate the potential power of this setting.

> **Teaching note:** The following illustration represents an alternative way of proving Theorem 6.9. This way seems conceptual simpler but it requires a starting point (or rather an assumption) that is much harder to establish, where both comparisons are with respect to the actual proof of Theorem 6.9 (which follows the illustration).

An illustration. Suppose that for some set $S \in \mathcal{BPP}$ there exists a polynomial p' and an off-line BPP-algorithm A' such that for every x it holds that $\Pr_{r\in\{0,1\}^{2p'(|x|)}}[A'(x,r)\neq\chi_S(x)] < 2^{-(p'(|x|)+1)}$; that is, the algorithm uses $2p'(|x|)$ bits of randomness and has error probability smaller than $2^{-p'(|x|)}/2$. Note that such an algorithm cannot be obtained by standard error-reduction (see Exercise 6.10). Anyhow, such a small error probability allows a partition of the string r such that one part accounts for the entire error probability on yes-instances while the other part accounts for the error probability on no-instances. Specifically, for every $x \in S$, it holds that $\Pr_{r'\in\{0,1\}^{p'(|x|)}}[(\forall r'' \in \{0,1\}^{p'(|x|)})\, A'(x,r'r'') = 1] > 1/2$, whereas for every $x \notin S$ and every $r' \in \{0,1\}^{p'(|x|)}$ it holds that $\Pr_{r''\in\{0,1\}^{p'(|x|)}}[A'(x,r'r'') = 1] < 1/2$. Thus, the error on yes-instances is "pushed" to the selection of r', whereas the error on no-instances is pushed to the selection of r''. This yields a one-sided error randomized Karp-reduction that maps x to (x,r'), where r' is uniformly selected in $\{0,1\}^{p'(|x|)}$, such that deciding S is reduced to the coRP problem (regarding pairs (x,r')) that is decided by the (on-line) randomized algorithm A'' defined by $A''(x,r') \stackrel{\text{def}}{=} A'(x,r'U_{p'(|x|)})$. For details, see Exercise 6.11. The actual proof, which avoids the aforementioned hypothesis, follows.

The actual starting point. Consider any BPP-problem with a characteristic function χ (which, in case of a promise problem, is a partial function, defined only over the promise). By standard error-reduction, there exists a probabilistic polynomial-time algorithm A such that for every x on which χ is defined it holds that $\Pr[A(x)\neq\chi(x)] < \mu(|x|)$, where μ is a negligible function. Looking at the corresponding off-line algorithm A' and denoting by p the polynomial that bounds the running time of A, we have

$$\mathsf{Pr}_{r\in\{0,1\}^{p(|x|)}}[A'(x,r)\neq\chi(x)] \;<\; \mu(|x|) \;<\; \frac{1}{2p(|x|)} \tag{6.1}$$

for all sufficiently long x's on which χ is defined. We show a randomized one-sided error Karp-reduction of χ to a promise problem in co$\mathcal{RP}$.

Teaching note: Some readers may prefer skipping the following two paragraphs and proceeding directly to the formal description of the randomized mapping (which follows). To such readers, we recommend returning to the two skipped paragraphs after reading the formal analysis.

The main idea. As in the illustrating paragraph, the basic idea is "pushing" the error probability on yes-instances (of χ) to the reduction, while pushing the error probability on no-instances to the coRP-problem. Focusing on the case that $\chi(x)=1$, this is achieved by augmenting the input x with a random sequence of "modifiers" that act on the random-input of algorithm A' such that for a good choice of modifiers it holds that for every $r\in\{0,1\}^{p(|x|)}$ there exists a modifier in this sequence that when applied to r yields r' that satisfies $A'(x,r')=1$. Indeed, not all sequences of modifiers are good, but a random sequence will be good with high probability and bad sequences will be accounted for in the error probability of the reduction. On the other hand, using only modifiers that are permutations guarantees that the error probability on no-instances only increase by a factor that equals the number of modifiers that we use, and this error probability will be accounted for by the error probability of the coRP-problem. Details follow.

The aforementioned modifiers are implemented by shifts (of the set of all strings by fixed offsets). Thus, we augment the input x with a random sequence of shifts, denoted $s_1,...,s_m\in\{0,1\}^{p(|x|)}$, such that for a good choice of $(s_1,...,s_m)$ it holds that for every $r\in\{0,1\}^{p(|x|)}$ there exists an $i\in[m]$ such that $A'(x,r\oplus s_i)=1$. We will show that, for any yes-instance x and a suitable choice of m, with very high probability, a random sequence of shifts is good. Thus, for $A''(\langle x,s_1,...,s_m\rangle,r)\stackrel{\rm def}{=}\vee_{i=1}^m A'(x,r\oplus s_i)$, it holds that, with very high probability over the choice of $s_1,...,s_m$, a yes-instance x is mapped to an augmented input $\langle x,s_1,...,s_m\rangle$ that is accepted by A'' with probability 1. On the other hand, the acceptance probability of augmented no-instances (for any choice of shifts) only increases by a factor of m. In further detailing the foregoing idea, we start by explicitly stating the simple randomized mapping (to be used as a randomized Karp-reduction), and next define the target promise problem.

The randomized mapping. On input $x\in\{0,1\}^n$, we set $m=p(|x|)$, uniformly select $s_1,...,s_m\in\{0,1\}^m$, and output the pair $(x,\overline{s})$, where $\overline{s}=(s_1,...,s_m)$. Note that this mapping, denoted M, is easily computable by a probabilistic polynomial-time algorithm.

The promise problem. We define the following promise problem, denoted $\Pi=(\Pi_{\rm yes},\Pi_{\rm no})$, having instances of the form $(x,\overline{s})$ such that $|\overline{s}|=p(|x|)^2$.

- The yes-instances are pairs $(x,\overline{s})$, where $\overline{s}=(s_1,...,s_m)$ and $m=p(|x|)$, such that for every $r\in\{0,1\}^m$ there exists an i satisfying $A'(x,r\oplus s_i)=1$.

- The no-instances are pairs $(x,\overline{s})$, where again $\overline{s}=(s_1,...,s_m)$ and $m=p(|x|)$, such that for at least half of the possible $r\in\{0,1\}^m$, for every i it holds that $A'(x,r\oplus s_i)=0$.

To see that Π is indeed a co$\mathcal{RP}$ promise problem, we consider the following randomized algorithm. On input $(x,(s_1,...,s_m))$, where $m=p(|x|)=|s_1|=\cdots=|s_m|$, the algorithm uniformly selects $r\in\{0,1\}^m$, and accepts if and only if $A'(x,r\oplus s_i)=1$ for some $i\in\{1,...,m\}$. Indeed, yes-instances of Π are accepted with probability 1, whereas no-instances of Π are rejected with probability at least 1/2.

Analyzing the reduction: We claim that the randomized mapping M reduces χ to Π with one-sided error. Specifically, we will prove two claims.

Claim 1: If x is a yes-instance (i.e., $\chi(x)=1$) then $\Pr[M(x)\in\Pi_{\rm yes}]>1/2$.

Claim 2: If x is a no-instance (i.e., $\chi(x)=0$) then $\Pr[M(x)\in\Pi_{\rm no}]=1$.

We start with Claim 2, which is easier to establish. Recall that $M(x)=(x,(s_1,...,s_m))$, where $s_1,...,s_m$ are uniformly and independently distributed in $\{0,1\}^m$. We note that (by Eq. (6.1) and $\chi(x)=0$), for every possible choice of $s_1,...,s_m\in\{0,1\}^m$ and every $i\in\{1,...,m\}$, the fraction of r's that satisfy $A'(x,r\oplus s_i)=1$ is at most $\frac{1}{2m}$. Thus, for every possible choice of $s_1,...,s_m\in\{0,1\}^m$, for at most half of the possible $r\in\{0,1\}^m$ there exists an i such that $A'(x,r\oplus s_i)=1$ holds. Hence, the reduction M *always* maps the no-instance x (i.e., $\chi(x)=0$) to a no-instance of Π (i.e., an element of $\Pi_{\rm no}$).

Turning to Claim 1 (which refers to $\chi(x)=1$), we will show shortly that in this case, with very high probability, the reduction M maps x to a yes-instance of Π. We upper-bound the probability that the reduction fails (in case $\chi(x)=1$) as follows:

$$\begin{aligned}
\Pr[M(x)\notin\Pi_{\rm yes}] &= \Pr_{s_1,\ldots,s_m}[\exists r\in\{0,1\}^m \text{ s.t. } (\forall i)\ A'(x,r\oplus s_i)=0] \\
&\leq \sum_{r\in\{0,1\}^m}\Pr_{s_1,\ldots,s_m}[(\forall i)\ A'(x,r\oplus s_i)=0] \\
&= \sum_{r\in\{0,1\}^m}\prod_{i=1}^{m}\Pr_{s_i}[A'(x,r\oplus s_i)=0] \\
&< 2^m\cdot\left(\frac{1}{2m}\right)^m
\end{aligned}$$

where the last inequality is due to Eq. (6.1). It follows that if $\chi(x)=1$ then $\Pr[M(x)\in\Pi_{\rm yes}]\gg 1/2$.

Combining both claims, it follows that the randomized mapping M reduces χ to Π, with one-sided error on yes-instances. Recalling that $\Pi\in$ co$\mathcal{RP}$, the theorem follows. ■

BPP is in PH. The traditional presentation of the ideas underlying the proof of Theorem 6.9 uses them for showing that $\mathcal{BPP}$ *is in the Polynomial-time Hierarchy* (where both classes refer to standard decision problems). Specifically, to prove that $\mathcal{BPP} \subseteq \Sigma_2$ (see Definition 3.8), define the polynomial-time computable predicate $\varphi(x, \overline{s}, r) \stackrel{\text{def}}{=} \bigvee_{i=1}^{m}(A'(x, s_i \oplus r) = 1)$, and observe that

$$\chi(x) = 1 \quad \Rightarrow \quad \exists \overline{s}\, \forall r \;\; \varphi(x, \overline{s}, r) \tag{6.2}$$

$$\chi(x) = 0 \quad \Rightarrow \quad \forall \overline{s}\, \exists r \;\; \neg\varphi(x, \overline{s}, r) \tag{6.3}$$

(where Eq. (6.3) is equivalent to $\neg\exists \overline{s}\, \forall r \;\; \varphi(x, \overline{s}, r)$). Note that Claim 1 (in the proof of Theorem 6.9) establishes that *most* sequences $\overline{s}$ satisfy $\forall r\, \varphi(x, \overline{s}, r)$, whereas Eq. (6.2) only requires the existence of *at least one* such $\overline{s}$. Similarly, Claim 2 establishes that for every $\overline{s}$ *most* choices of r violate $\varphi(x, \overline{s}, r)$, whereas Eq. (6.3) only requires that for every $\overline{s}$ there exists *at least one* such r. We comment that the same proof idea yields a variety of similar statements (e.g., $\mathcal{BPP} \subseteq \mathcal{MA}$, where $\mathcal{MA}$ is a randomized version of $\mathcal{NP}$ defined in Section 9.1).[7]

6.1.4 Zero-sided error: The complexity class ZPP

We now consider probabilistic polynomial-time algorithms that never err, but may fail to provide an answer. Focusing on decision problems, the corresponding 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 $\perp$ (indicating failure) with probability at most 1/2. That is, $S \in \mathcal{ZPP}$ *if there exists a probabilistic polynomial-time algorithm* A *such that for every* $x \in \{0,1\}^*$ *it holds that* $\Pr[A(x) \in \{\chi_S(x), \perp\}] = 1$ *and* $\Pr[A(x) = \chi_S(x)] \geq 1/2$*, where* $\chi_S(x) = 1$ *if* $x \in S$ *and* $\chi_S(x) = 0$ *otherwise*. Again, the choice of the constant (i.e., 1/2) is immaterial, and "error-reduction" can be performed showing that algorithms that yield a meaningful answer with noticeable probability can be amplified to algorithms that fail with negligible probability (see Exercise 6.6).

Theorem 6.10 $\mathcal{ZPP} = \mathcal{RP} \cap \mathrm{co}\mathcal{RP}$.

Proof Sketch: The fact that $\mathcal{ZPP} \subseteq \mathcal{RP}$ (as well as $\mathcal{ZPP} \subseteq \mathrm{co}\mathcal{RP}$) follows by a trivial transformation of the ZPP-algorithm; that is, replacing the failure indicator $\perp$ by a "no" verdict (resp., "yes" verdict). Note that the choice of what to say in case the ZPP-algorithm fails is determined by the type of error that we are allowed.

In order to prove that $\mathcal{RP} \cap \mathrm{co}\mathcal{RP} \subseteq \mathcal{ZPP}$ we combine the two algorithm guaranteed for a set in $\mathcal{RP} \cap \mathrm{co}\mathcal{RP}$. The point is that we can trust the RP-algorithm (resp., coNP-algorithm) in the case that it says "yes" (resp., "no"), but not in the case that it says "no" (resp., "yes"). Thus, we invoke both algorithms,

[7] Specifically, the class $\mathcal{MA}$ is defined by allowing the verification algorithm V in Definition 2.5 to be probabilistic and err on no-instances; that is, for every $x \in S$ there exists $y \in \{0,1\}^{\mathrm{poly}(|x|)}$ such that $\Pr[V(x,y) = 1] = 1$, whereas for every $x \notin S$ and every y it holds that $\Pr[V(x,y) = 0] \geq 1/2$. We note that $\mathcal{MA}$ can be viewed as a hybrid of the two aforementioned pairs of conditions; specifically, each problem in $\mathcal{MA}$ satisfy the conjunction of Eq. (6.2) and Claim 2. Other randomized versions of $\mathcal{NP}$ (i.e., variants of $\mathcal{MA}$) are considered in Exercise 6.12.

and output a definite answer only if we obtain an answer that we can trust (which happen with high probability). Otherwise, we output $\perp$. $\square$

Expected polynomial-time. In some sources $\mathcal{ZPP}$ is defined in terms of randomized algorithms that run in expected polynomial-time and always output the correct answer. This definition is equivalent to the one we used (see Exercise 6.7).

6.1.5 Randomized Log-Space

In this section we discuss probabilistic polynomial-time algorithms that are further restricted such that they are allowed to use only a logarithmic amount of space.

6.1.5.1 Definitional issues

When defining space-bounded randomized algorithms, we face a problem analogous to the one discussed in the context of non-deterministic space-bounded computation (see Section 5.3). Specifically, the on-line and the off-line versions (formulated in Definition 6.1) are no longer equivalent, unless we restrict the off-line machine to access its random-input tape in a uni-directional manner. The issue is that, in the context of space-bounded computation (and unlike in the case that we only care about time-bounds), the outcome of the internal coin tosses (in the on-line model) cannot be recorded for free. Bearing in mind that, *in the current context*, we wish to model real algorithms (rather than present a fictitious model that captures a fundamental phenomena as in Section 5.3), it is clear that *using the on-line version is the natural choice.*

An additional issue that arises is the need to explicitly bound the running-time of space-bounded randomized algorithms. Recall that, without loss of generality, the number of steps taken by a space-bounded non-deterministic machine is at most exponential in its space complexity, because the shortest path between two configurations in the (directed) graph of possible configurations is upper-bounded by its size (which in turn is exponential in the space-bound). This reasoning fails in the case of randomized algorithms, because the shortest path between two configurations does not bound the expected number of random steps required for going from the first configuration to the second one. In fact, as we shall shortly see, failing to upper-bound the running time of log-space randomized algorithms seems to allow them too much power; that is, such (unrestricted) log-space randomized algorithms can emulate non-deterministic log-space computations (in exponential time). The emulation consists of repeatedly invoking the NL-machine, while using random choices in the role of the non-deterministic moves. If the input is a yes-instance then, in each attempt, with probability at least 2^{-t}, we "hit" an accepting t-step (non-deterministic) computation, where t is polynomial in the input length. Thus, the randomized machine accepts such a yes-instance after an expected number of 2^t trials. To allow for the rejection of no-instances (rather than looping infinitely in vain), we wish to implement a counter that counts till 2^t (or so) and reject the input if 2^t trials were made and have all failed (to hit an accepting computation of the NL-machine). We need to implement such a counter 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. The implementation of such a counter is left to Exercise 6.18, and using it we may obtain a randomized algorithm that halts with high probability (on every input), always rejects a no-instance, and accepts each yes-instance with probability at least 1/2.

In light of the foregoing discussion, when defining randomized log-space algorithms we explicitly require that the algorithms halt in polynomial-time. Modulo this convention, the relation between classes $\mathcal{RL}$ (resp., $\mathcal{BPL}$) and $\mathcal{NL}$ is analogous to the relation between $\mathcal{RP}$ (resp., $\mathcal{BPP}$) and $\mathcal{NP}$. Specifically, the probabilistic acceptance condition of $\mathcal{RL}$ (resp., $\mathcal{BPL}$) is as in the case of $\mathcal{RP}$ (resp., $\mathcal{BPP}$).

Definition 6.11 (the classes $\mathcal{RL}$ and $\mathcal{BPL}$): *We say that a randomized log-space algorithm is* admissible *if it always halts in a polynomial number of steps.*

- *A decision problem S is in $\mathcal{RL}$ if there exists an admissible* (on-line) *randomized log-space algorithm A such that for every $x \in S$ it holds that* $\Pr[A(x) = 1] \geq 1/2$ *and for every $x \notin S$ it holds that* $\Pr[A(x) = 0] = 1$.
- *A decision problem S is in $\mathcal{BPL}$ if there exists an admissible* (on-line) *randomized log-space algorithm A such that for every $x \in S$ it holds that* $\Pr[A(x) = 1] \geq 2/3$ *and for every $x \notin S$ it holds that* $\Pr[A(x) = 0] \geq 2/3$.

Clearly, $\mathcal{RL} \subseteq \mathcal{NL} \subseteq \mathcal{P}$ and $\mathcal{BPL} \subseteq \mathcal{P}$. Note that the classes $\mathcal{RL}$ and $\mathcal{BPL}$ remain unchanged even if we allow the algorithms to run for *expected* polynomial-time and have non-halting computations. Such algorithms can be easily transformed into admissible algorithms by truncating long computations, while using a (standard) counter (which can be implemented in logarithmic-space). Also note that error-reduction is applicable in the current setting (while essentially preserving both the time and space bounds).

6.1.5.2 The accidental tourist sees it all

An appealing example of a randomized log-space algorithm is presented next. It refers to the problem of deciding undirected connectivity, and demonstrates that this problem is in $\mathcal{RL}$. (Recall that in Section 5.2.4 we proved that this problem is actually in $\mathcal{L}$, but the algorithm and its analysis were more complicated.) In contrast, recall that Directed Connectivity is complete for $\mathcal{NL}$ (under log-space reductions).

For sake of simplicity, we consider the following computational problem: *given an undirected graph G and a pair of vertices (s,t), determine whether or not s and t are connected in G.* Note that deciding undirected connectivity (of a given undirected graph) is log-space reducible to the foregoing problem (e.g., just check the connectivity of all pairs of vertices).

Construction 6.12 *On input (G,s,t), the randomized algorithm starts a* $\text{poly}(|G|)$-*long random walk at vertex s, and accepts the triplet if and only if the walk passed* *through the vertex t. By a random walk we mean that at each step the algorithm selects uniformly one of the neighbors of the current vertex and moves 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). Obviously, if s and t are not connected in G then the algorithm always rejects (G, s, t). Proposition 6.13 implies that if s and t are connected (in G) then the algorithm accepts with probability at least $1/2$. It follows that undirected connectivity is in $\mathcal{RL}$.

Proposition 6.13 *With probability at least $1/2$, a random walk of length $O(|V| \cdot |E|)$ starting at any vertex of the graph $G = (V, E)$ passes through all the vertices that reside in the same connected component as the start vertex.*

Thus, such a random walk may be used to explore the relevant connected component (in any graph). Following this walk one is likely to see all that there is to see in that component.

Proof Sketch: We will actually show that if G is connected then, with probability at least $1/2$, a random walk starting at s visits all the vertices of G. For any pair of vertices (u, v), let $X_{u,v}$ be a random variable representing the number of steps taken in a random walk starting at u until v is *first encountered*. The reader may verify that for every edge $\{u, v\} \in E$ it holds that $\mathrm{E}[X_{u,v}] \leq 2|E|$; see Exercise 6.19. Next, we let $\mathrm{cover}(G)$ denote the expected number of steps in a random walk starting at s and ending when the last of the vertices of V is encountered. Our goal is to upper-bound $\mathrm{cover}(G)$. Towards this end, we consider an arbitrary directed cyclic-tour C that visits all vertices in G, and note that

$$\mathrm{cover}(G) \;\leq\; \sum_{(u,v)\in C} \mathrm{E}[X_{u,v}] \;\leq\; |C| \cdot 2|E|.$$

In particular, selecting C as a traversal of some spanning tree of G, we conclude that $\mathrm{cover}(G) \;<\; 4 \cdot |V| \cdot |E|$. Thus, with probability at least $1/2$, a random walk of length $8 \cdot |V| \cdot |E|$ starting at s visits all vertices of G. $\square$

6.2 Counting

We now turn to a new type of computational problems, which vastly generalize decision problems of the NP-type. We refer to counting problems, and more specifically to counting objects that can be efficiently recognized. The search and decision versions of NP provide suitable definitions of efficiently recognized objects, which in turn yield corresponding counting problems:

1. For each search problem having efficiently checkable solutions (i.e., a relation $R \subseteq \{0,1\}^* \times \{0,1\}^*$ in $\mathcal{PC}$ (see Definition 2.3)), we consider the problem of counting the number of solutions for a given instance. That is, on input x, we are required to output $|\{y : (x,y) \in R\}|$.

2. For each decision problem S in $\mathcal{NP}$, and each corresponding verification procedure V (as in Definition 2.5), we consider the problem of counting the number of NP-witnesses for a given instance. That is, on input x, we are required to output $|\{y : V(x,y)=1\}|$.

We shall consider these types of counting problems as well as relaxations (of these counting problems) that refer to approximating the said quantities (see Sections 6.2.1 and 6.2.2, respectively). Other related topics include "problems with unique solutions" (see Section 6.2.3) and "uniform generation of solutions" (see Section 6.2.4). Interestingly, randomized procedures will play an important role in many of the results regarding the aforementioned types of problems.

6.2.1 Exact Counting

In continuation to the foregoing discussion, we define the class of problems concerned with counting efficiently recognized objects. (Recall that $\mathcal{PC}$ denotes the class of search problems having polynomially long solutions that are efficiently checkable; see Definition 2.3.)

Definition 6.14 (counting efficiently recognized objects – $\#\mathcal{P}$): *The class $\#\mathcal{P}$ consists of all functions that count solutions to a search problem in $\mathcal{PC}$. That is, $f : \{0,1\}^* \to \mathbb{N}$ is in $\#\mathcal{P}$ if there exists $R \in \mathcal{PC}$ such that, for every x, it holds that $f(x) = |R(x)|$, where $R(x) = \{y : (x,y) \in R\}$. In this case we say that f is the* counting problem associated with *R, and denote the latter by $\#R$* (i.e., $\#R = f$).

Every decision problem in $\mathcal{NP}$ is Cook-reducible to $\#\mathcal{P}$, because every such problem can be cast as deciding membership in $S_R = \{x : |R(x)| > 0\}$ for some $R \in \mathcal{PC}$ (see Section 2.1.2). It also holds that $\mathcal{BPP}$ is Cook-reducible to $\#\mathcal{P}$ (see Exercise 6.20). The class $\#\mathcal{P}$ is sometimes defined in terms of decision problems, as is implicit in the following proposition.

Proposition 6.15 (a decisional version of $\#\mathcal{P}$): *For any $f \in \#\mathcal{P}$, deciding membership in $S_f \stackrel{\text{def}}{=} \{(x,N) : f(x) \geq N\}$ is computationally equivalent to computing f.*

Actually, the claim holds for any function $f : \{0,1\}^* \to \mathbb{N}$ for which there exists a polynomial p such that for every $x \in \{0,1\}^*$ it holds that $f(x) \leq 2^{p(|x|)}$.

Proof: Since the relation R vouching for $f \in \#\mathcal{P}$ (i.e., $f(x) = |R(x)|$) is polynomially bounded, there exists a polynomial p such that for every x it holds that $f(x) \leq 2^{p(|x|)}$. Deciding membership in S_f is easily reduced to computing f (i.e., we accept the input (x,N) if and only if $f(x) \geq N$). Computing f is reducible to deciding S_f by using a binary search (see Exercise 2.9). This relies on the fact that, on input x and oracle access to S_f, we can determine whether or not $f(x) \geq N$ by making the query (x,N). Note that we know a priori that $f(x) \in [0, 2^{p(|x|)}]$. ■

The counting class $\#\mathcal{P}$ is also related to the problem of enumerating all possible solutions to a given instance (see Exercise 6.21).

6.2.1.1 On the power of $\#\mathcal{P}$

As indicated, $\mathcal{NP} \cup \mathcal{BPP}$ is (easily) reducible to $\#\mathcal{P}$. Furthermore, as stated in Theorem 6.16, the entire Polynomial-Time Hierarchy (as defined in Section 3.2) is Cook-reducible to $\#\mathcal{P}$ (i.e., $\mathcal{PH} \subseteq \mathcal{P}^{\#\mathcal{P}}$). On the other hand, any problem in $\#\mathcal{P}$ is solvable in polynomial space, and so $\mathcal{P}^{\#\mathcal{P}} \subseteq \mathcal{PSPACE}$.

Theorem 6.16 *Every set in $\mathcal{PH}$ is Cook-reducible to $\#\mathcal{P}$.*

We do not present a proof of Theorem 6.16 here, because the known proofs are rather technical. Furthermore, one main idea underlying these proofs appears in a more clear form in the proof of Theorem 6.29. Nevertheless, in Section F.1 we present a proof of a related result, which implies that $\mathcal{PH}$ is reducible to $\#\mathcal{P}$ via *randomized* Karp-reductions.

6.2.1.2 Completeness in $\#\mathcal{P}$

The definition of $\#\mathcal{P}$-completeness is analogous to the definition of $\mathcal{NP}$-completeness. That is, *a counting problem f is $\#\mathcal{P}$*-complete *if $f \in \#\mathcal{P}$ and every problem in $\#\mathcal{P}$ is Cook-reducible to f.*

We claim that the counting problems associated with the NP-complete problems presented in Section 2.3.3 are all $\#\mathcal{P}$-complete. We warn that this fact is not due to the mere NP-completeness of these problems, but rather to an additional property of the reductions establishing their NP-completeness. Specifically, the Karp-reductions that were used (or variants of them) have the extra property of preserving the number of NP-witnesses (as captured by the following definition).

Definition 6.17 (parsimonious reductions): *Let $R, R' \in \mathcal{PC}$ and let g be a Karp-reduction of $S_R = \{x : R(x) \neq \emptyset\}$ to $S_{R'} = \{x : R'(x) \neq \emptyset\}$, where $R(x) = \{y : (x,y) \in R\}$ and $R'(x) = \{y : (x,y) \in R'\}$. We say that g is* parsimonious (with respect to R and R') *if for every x it holds that $|R(x)| = |R'(g(x))|$. In such a case we say that g is a* parsimonious reduction of R to R'.

We stress that the condition of being parsimonious refers to the two underlying relations R and R' (and not merely to the sets S_R and $S_{R'}$). The requirement that g is a Karp-reduction is partially redundant, because if g is polynomial-time computable and for every x it holds that $|R(x)| = |R'(g(x))|$, then g constitutes a Karp-reduction of S_R to $S_{R'}$. Specifically, $|R(x)| = |R'(g(x))|$ implies that $|R(x)| > 0$ (i.e., $x \in S_R$) if and only if $|R'(g(x))| > 0$ (i.e., $g(x) \in S_{R'}$). The reader may easily verify that the Karp-reduction underlying the proof of Theorem 2.19 as well as many of the reductions used in Section 2.3.3 are parsimonious (see Exercise 2.29).

Theorem 6.18 *Let $R \in \mathcal{PC}$ and suppose that every search problem in $\mathcal{PC}$ is parsimoniously reducible to R. Then the counting problem associated with R is $\#\mathcal{P}$-complete.*

Proof: Clearly, the counting problem associated with R, denoted $\#R$, is in $\#\mathcal{P}$. To show that every $f' \in \#\mathcal{P}$ is reducible to f, we consider the relation $R' \in \mathcal{PC}$

that is counted by f'; that is, $\#R' = f'$. Then, by the hypothesis, there exists a parsimonious reduction g of R' to R. This reduction also reduces $\#R'$ to $\#R$; specifically, $\#R'(x) = \#R(g(x))$ for every x. ■

Corollaries. As an immediate corollary of Theorem 6.18, we get that counting the number of satisfying assignments to a given CNF formula is $\#\mathcal{P}$-complete (because $R_{\mathtt{SAT}}$ is $\mathcal{PC}$-complete via parsimonious reductions). Similar statements hold for all the other NP-complete problems mentioned in Section 2.3.3 and in fact for all NP-complete problems listed in [82]. These corollaries follow from the fact that all known reductions among natural NP-complete problems are either parsimonious or can be easily modified to be so.

We conclude that many counting problems associated with NP-complete search problems are $\#\mathcal{P}$-complete. It turns out that also counting problems associated with efficiently solvable search problems may be $\#\mathcal{P}$-complete.

Theorem 6.19 *There exist $\#\mathcal{P}$-complete counting problems that are associated with efficiently solvable search problems. That is, there exists $R \in \mathcal{PF}$* (see Definition 2.2) *such that $\#R$ is $\#\mathcal{P}$-complete.*

Theorem 6.19 can be established by presenting artificial $\#\mathcal{P}$-complete problems (see Exercise 6.22). The following proof uses a natural counting problem.

Proof: Consider the relation $R_{\mathtt{dnf}}$ consisting of pairs (ϕ, τ) such that ϕ is a DNF formula and τ is an assignment satisfying it. Note that the search problem of $R_{\mathtt{dnf}}$ is easy to solve (e.g., by picking an arbitrary truth assignment that satisfies the first term in the input formula). To see that $\#R_{\mathtt{dnf}}$ is $\#\mathcal{P}$-complete consider the following reduction from $\#R_{\mathtt{SAT}}$ (which is $\#\mathcal{P}$-complete by Theorem 6.18). Given a CNF formula ϕ, transform $\neg\phi$ into a DNF formula ϕ' by applying de-Morgan's Law, query $\#R_{\mathtt{dnf}}$ on ϕ', and return $2^n - \#R_{\mathtt{dnf}}(\phi')$, where n denotes the number of variables in ϕ (resp., ϕ'). ■

Reflections: We note that Theorem 6.19 is not established by a parsimonious reduction. This fact should not come as a surprise because a parsimonious reduction of $\#R'$ to $\#R$ implies that $S_{R'} = \{x : \exists y \text{ s.t. } (x,y) \in R'\}$ is reducible to $S_R = \{x : \exists y \text{ s.t. } (x,y) \in R\}$, where in our case $S_{R'}$ is NP-Complete while $S_R \in \mathcal{P}$ (since $R \in \mathcal{PF}$). Nevertheless, the proof of Theorem 6.19 is related to the hardness of some underlying decision problem (i.e., the problem of deciding whether a given DNF formula is a tautology (i.e., whether $\#R_{\mathtt{dnf}}(\phi') = 2^n$)). But does there exist a $\#\mathcal{P}$-complete problem that is "not based on some underlying NP-complete decision problem"? Amazingly enough, the answer is positive.

Theorem 6.20 *Counting the number of perfect matchings in a bipartite graph is $\#\mathcal{P}$-complete.*[8]

[8]See Appendix G.1 for basic terminology regarding graphs.

Equivalently (see Exercise 6.23), the problem of computing the permanent of matrices with 0/1-entries is $\#\mathcal{P}$-complete. Recall that the permanent of an n-by-n matrix $M = (m_{i,j})$, denoted $\mathtt{perm}(M)$, equals the sum over all permutations π of $[n]$ of the products $\prod_{i=1}^{n} m_{i,\pi(i)}$. Theorem 6.20 is proven by composing the following two (many-to-one) reductions (asserted in Propositions 6.21 and 6.22, respectively) and using the fact that $\#R_{\mathtt{3SAT}}$ is $\#\mathcal{P}$-complete (see Theorem 6.18 and Exercise 2.29). Needless to say, the resulting reduction is not parsimonious.

Proposition 6.21 *The counting problem of* 3SAT (i.e., $\#R_{\mathtt{3SAT}}$) *is reducible to computing the permanent of integer matrices. Furthermore, there exists an* even *integer $c > 0$ and a finite set of integers I such that, on input a 3CNF formula ϕ, the reduction produces an integer matrix M_ϕ with entries in I such that* $\mathtt{perm}(M_\phi) = c^m \cdot \#R_{\mathtt{3SAT}}(\phi)$ *where m denotes the number of clauses in ϕ.*

The original proof of Proposition 6.21 uses $c = 2^{10}$ and $I = \{-1, 0, 1, 2, 3\}$. It can be shown (see Exercise 6.24 (which relies on Theorem 6.29)) that, for every integer $n > 1$ that is relatively prime to c, computing the permanent modulo n is NP-hard (under randomized reductions). Thus, using the case of $c = 2^{10}$, this means that computing the permanent modulo n is NP-hard for any odd $n > 1$. In contrast, computing the permanent modulo 2 (which is equivalent to computing the determinant modulo 2) is easy (i.e., can be done in polynomial-time and even in $\mathcal{NC}$). Thus, assuming $\mathcal{NP} \not\subseteq \mathcal{BPP}$, Proposition 6.21 cannot hold for an odd c (because by Exercise 6.24 it would follow that computing the permanent modulo 2 is NP-Hard). We also note that, assuming $\mathcal{P} \neq \mathcal{NP}$, Proposition 6.21 cannot possibly hold for a set I containing only non-negative integers (see Exercise 6.25).

Proposition 6.22 *Computing the permanent of integer matrices is reducible to computing the permanent of 0/1-matrices. Furthermore, the reduction maps any integer matrix A into a 0/1-matrix A'' such that the permanent of A can be easily computed from A and the permanent of A''.*

Teaching note: We do not recommend presenting the proofs of Propositions 6.21 and 6.22 in class. The high-level structure of the proof of Proposition 6.21 has the flavor of some sophisticated reductions among NP-problems, but the crucial point is the existence of adequate gadgets. We do not know of a high-level argument establishing the existence of such gadgets nor of any intuition as to why such gadgets exist.[9] Instead, the existence of such gadgets is proved by a design that is both highly non-trivial and *ad hoc* in nature. Thus, the proof of Proposition 6.21 boils down to a complicated design problem that is solved in a way that has little pedagogical value. In contrast, the proof of Proposition 6.22 uses two simple ideas that can be useful in other settings. With suitable hints, this proof can be used as a good exercise.

Proof of Proposition 6.21: We will use the correspondence between the permanent of a matrix A and the sum of the weights of the cycle covers of the weighted directed graph represented by the matrix A. A cycle cover of a graph is

[9]Indeed, the conjecture that such gadgets exist can only be attributed to ingenuity.

a collection of simple[10] *vertex-disjoint* directed cycles that covers all the graph's vertices, and its weight is the product of the weights of the corresponding edges. The SWCC of a weighted directed graph is the sum of the weights of all its cycle covers.

Given a 3CNF formula ϕ, we construct a directed weighted graph G_ϕ such that the SWCC of G_ϕ equals equals $c^m \cdot \#R_{\mathtt{3SAT}}(\phi)$, where c is a universal constant and m denotes the number of clauses in ϕ. We may assume, without loss of generality, that each clause of ϕ has exactly three literals (which are not necessarily distinct).

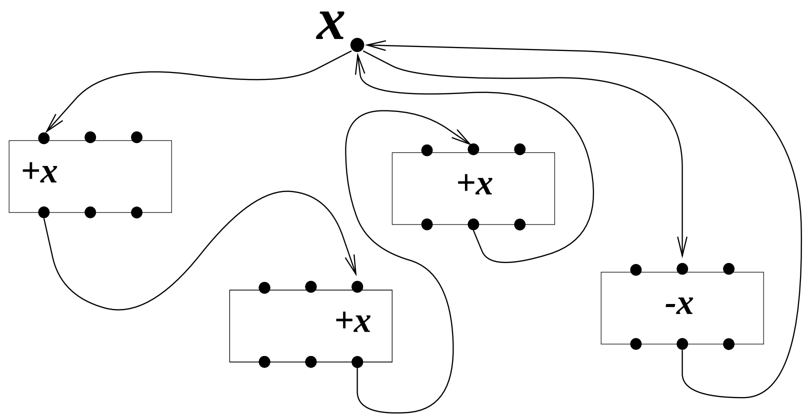
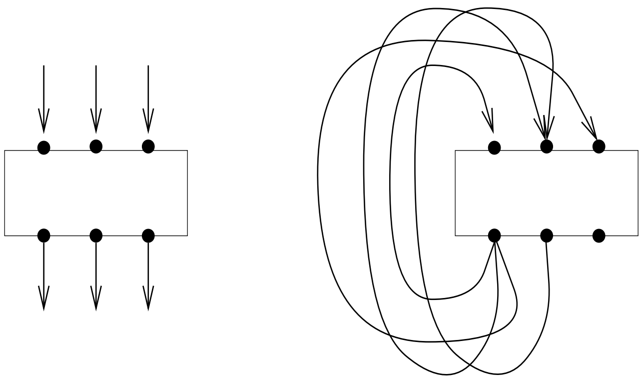


Figure 6.1: Tracks connecting gadgets in the reduction to cycle cover.

We start with a high-level description (of the construction) that refers to (clause) gadgets, each containing some internal vertices and internal (weighted) edges, which are *unspecified at this point.* In addition, each gadget has three pairs of designated vertices, one pair per each literal appearing in the clause, where one vertex in the pair is designated as an entry vertex and the other as an exit vertex. The graph G_ϕ consists of m such gadgets, one per each clause (of ϕ), and n auxiliary vertices, one per each variable (of ϕ), as well as some *additional directed edges*, each having weight 1. Specifically, for each variable, we introduce two tracks, one per each of the possible literals of this variable. The track associated with a literal consists of directed edges (each having weight 1) that form a simple "cycle" passing through the corresponding (auxiliary) vertex as well as through the designated vertices that correspond to the occurrences of this literal in the various clauses. Specifically, for each such occurrence, the track enters the corresponding clause gadget at the entry-vertex corresponding to this literal and exits at the corresponding exit-vertex. (If a literal does not appear in ϕ then the corresponding track is a self-loop on the corresponding variable.) See Figure 6.1 showing the two tracks of a variable x that occurs positively in three clauses and negatively in one clause. The entry-vertices (resp., exit-vertices) are drawn on the top (resp., bottom) part of each gadget.

[10]Here a simple cycle is a strongly connected directed graph in which each vertex has a single incoming (resp., outgoing) edge. In particular, self-loops are allowed.



On the left is a gadget with the track edges adjacent to it (as in the real construction). On the right is a gadget and four out of the nine external edges (two of which are nice) used in the analysis.

Figure 6.2: External edges for the analysis of the clause gadget

For the purpose of stating the desired properties of the clause gadget, we augment the gadget by nine external edges (of weight 1), one per each pair of (not necessarily matching) entry and exit vertices such that the edge goes from the exit-vertex to the entry-vertex (see Figure 6.2). (We stress that this is an auxiliary construction that differs from and yet is related to the use of gadgets in the foregoing construction of G_ϕ.) The three edges that link the designated pairs of vertices that correspond to the three literals are called nice. We say that a collection of edges C (e.g., a collection of cycles in the augmented gadget) uses the external edges S if the intersection of C with the set of the (nine) external edges equals S. We postulate the following three properties of the clause gadget.

1. The sum of the weights of all cycle covers (of the gadget) that do not use any external edge (i.e., use the empty set of external edges) equals zero.

2. Let $V(S)$ denote the set of vertices incident to S, and say that S is nice if it is non-empty and the vertices in $V(S)$ can be perfectly matched using nice edges.[11] Then, there exists a constant c (indeed the one postulated in the proposition's claim) such that, for any nice set S, the sum of the weights of all cycle covers that use the external edges S equals c.

3. For any non-nice set $S \neq \emptyset$ of external edges, the sum of the weights of all cycle covers that use the external edges S equals zero.

[11] Clearly, any non-empty set of nice edges is a nice set. Thus, a singleton set is nice if and only if the corresponding edge is nice. On the other hand, any set S of three (vertex-disjoint) external edges is nice, because $V(S)$ has a perfect matching using all three nice edges. Thus, the notion of nice sets is "non-trivial" only for sets of two edges. Such a set S is nice if and only if $V(S)$ consists of two pairs of corresponding designated vertices.

Note that the foregoing three cases exhaust all the possible ones. Also note that the set of external edges used by a cycle cover (of the augmented gadget) must be a matching (i.e., these edges must be vertex disjoint).

Intuitively, there is a correspondence between nice sets of external edges (of an augmented gadget) and the pairs of edges on tracks that pass through the (unaugmented) gadget. Indeed, we now turn back to G_ϕ, which uses unaugmented gadgets. Using the foregoing properties of the (augmented) gadgets, it can be shown that each satisfying assignment of ϕ contributes exactly c^m to the SWCC of G_ϕ (see Exercise 6.26). It follows that the SWCC of G_ϕ equals $c^m \cdot \#R_{\mathtt{3SAT}}(\phi)$.

Having established the validity of the abstract reduction, we turn to the implementation of the clause gadget. The first implementation is a *Deus ex Machina*, with a corresponding adjacency matrix depicted in Figure 6.3. Its validity (for the value $c = 12$) can be verified by computing the permanent of the corresponding sub-matrices (see analogous analysis in Exercise 6.28).

The gadget uses eight vertices, where the first six are the designated (entry and exit) vertices. The entry-vertex (resp., exit-vertex) associated with the i^{th} literal is numbered i (resp., $i+3$). The corresponding adjacency matrix follows.

$$\begin{pmatrix} 1 & 0 & 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 1 & -1 & 0 & 1 & 1 \\ 0 & 0 & -1 & -1 & 2 & 0 & 1 & 1 \\ 0 & 0 & 0 & -1 & -1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 & 2 & -1 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{pmatrix}$$

Note that the edge $3 \to 6$ can be contracted, but the resulting 7-vertex graph will not be consistent with our (inessentially stringent) definition of a gadget by which the six designated vertices should be distinct.

Figure 6.3: A Deus ex Machina clause gadget for the reduction to cycle cover.

A more structured implementation of the clause gadget is depicted in Figure 6.4, which refers to a (hexagon) box to be implemented later. The box contains several vertices and weighted edges, but only two of these vertices, called terminals, are connected to the outside (and are shown in Figure 6.4). The clause gadget consists of five copies of this box, where three copies are designated for the three literals of the clause (and are marked `LB1`, `LB2`, and `LB3`), as well as additional vertices and edges shown in Figure 6.4. In particular, the clause gadget contains the six aforementioned designated vertices (i.e., a pair of entry and exit vertices per each literal), two additional vertices (shown at the two extremes of the figure), and some edges (all having weight 1). Each designated vertex has a self-loop, and is incident to a single additional edge that is outgoing (resp., incoming) in case the vertex is an entry-vertex (resp., exit-vertex) of the gadget. The two terminals of each box that is associated with some literal are connected to the corresponding pair of designated vertices (e.g., the outgoing edge of `entry1` is incident at the right terminal of the box `LB1`). Note that the five boxes reside on a directed path (going from left to right), and the only edges going in the opposite direction are those drawn below this path.

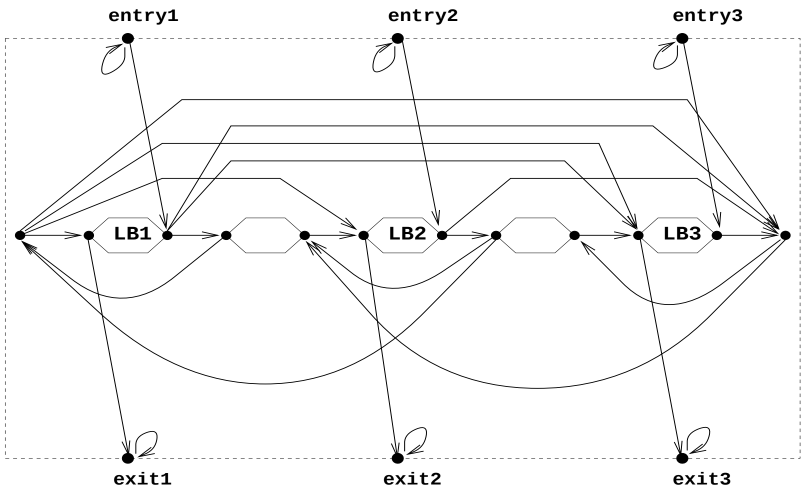
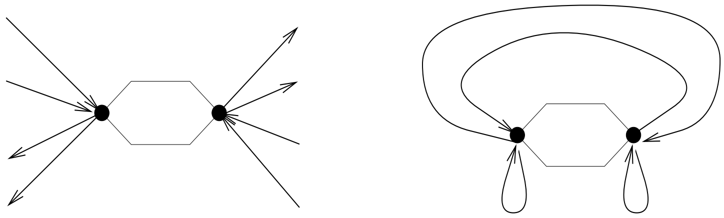


Figure 6.4: A structured clause gadget for the reduction to cycle cover.



On the left is a box with potential edges adjacent to it (as in the gadget construction). On the right is a box and the four external edges used in the analysis.

Figure 6.5: External edges for the analysis of the box

In continuation to the foregoing, we wish to state the desired properties of the box. Again, we do so by considering the augmentation of the box by external edges (of weight 1) incident at the specified vertices. In this case (see Figure 6.5), we have a pair of anti-parallel edges connecting the two terminals of the box as well as two self-loops (one on each terminal). We postulate the following three properties of the box.

1. The sum of the weights of all cycle covers (of the box) that do not use any external edge equals zero.

2. There exists a constant b (in our case $b = 4$) such that, for each of the two anti-parallel edges, the sum of the weights of all cycle covers that use this edge equals b.

3. For any (non-empty) set S of the self-loops, the sum of the weights of all cycle covers (of the box) that use S equals zero.

Note that the foregoing three cases exhaust all the possible ones. It can be shown that the conditions regarding the box imply that the construction presented in Figure 6.4 satisfies the conditions that were postulated for the clause gadget (see Exercise 6.27). Specifically, we have $c = b^5$. As for box itself, a smaller *Deus ex Machina* is provided by the following 4-by-4 adjacency matrix

$$\begin{pmatrix} 0 & 1 & -1 & -1 \\ 1 & -1 & 1 & 1 \\ 0 & 1 & 1 & 2 \\ 0 & 1 & 3 & 0 \end{pmatrix} \tag{6.4}$$

where the two terminals correspond to the first and the fourth vertices. Its validity (for the value $b = 4$) can be verified by computing the permanent of the corresponding sub-matrices (see Exercise 6.28). ■

Proof of Proposition 6.22: The proof proceeds in two steps. In the first step we show that computing the permanent of *integer matrices* is reducible to computing the permanent of *non-negative matrices*. This reduction proceeds as follows. For an n-by-n integer matrix $A = (a_{i,j})_{i,j\in[n]}$, let $\|A\|_\infty = \max_{i,j}(|a_{i,j}|)$ and $Q_A = 2(n!)\cdot\|A\|_\infty^n + 1$. We note that, given A, the value Q_A can be computed in polynomial-time, and in particular $\log_2 Q_A < n^2 \log\|A\|_\infty$. Given the matrix A, the reduction constructs the non-negative matrix $A' = (a_{i,j} \bmod Q_A)_{i,j\in[n]}$ (i.e., the entries of A' are in $\{0, 1, ..., Q_A - 1\}$), queries the oracle for the permanent of A', and outputs $v \stackrel{\text{def}}{=} \texttt{perm}(A') \bmod Q_A$ if $v < Q_A/2$ and $-(Q_A - v)$ otherwise. The key observation is that

$$\texttt{perm}(A) \equiv \texttt{perm}(A') \pmod{Q_A}, \text{ while } |\texttt{perm}(A)| \leq (n!)\cdot\|A\|_\infty^n < Q_A/2.$$

Thus, $\texttt{perm}(A') \bmod Q_A$ (which is in $\{0, 1, ..., Q_A - 1\}$) determines $\texttt{perm}(A)$. We note that $\texttt{perm}(A')$ is likely to be much larger than $Q_A > |\texttt{perm}(A)|$; it is merely that $\texttt{perm}(A')$ and $\texttt{perm}(A)$ are equivalent modulo Q_A.

In the second step we show that computing the permanent of non-negative matrices is reducible to computing the permanent of 0/1-matrices. In this reduction, we view the computation of the permanent as the computation of the sum of the weights of all the cycle covers (SWCC) of the corresponding weighted directed graph (see proof of Proposition 6.21). Thus, we reduce the computation of the SWCC of directed graphs *with non-negative weights* to the computation of the SWCC of *unweighted directed graphs with no parallel edges* (which correspond to 0/1-matrices). The reduction is via local replacements that preserve the value of the SWCC. These local replacements combine the following two local replacements (which preserve the SWCC):

1. Replacing an edge of weight $w = \prod_{i=1}^{t} w_i$ by a path of length t (i.e., $t-1$ internal nodes) with the corresponding weights $w_1, ..., w_t$, and self-loops (with weight 1) on all internal nodes.

   Note that a cycle-cover that uses the original edge corresponds to a cycle-cover that uses the entire path, whereas a cycle-cover that does not use the original edge corresponds to a cycle-cover that uses all the self-loops.

2. Replacing an edge of weight $w = \sum_{i=1}^{t} w_i$ by t parallel 2-edge paths such that the first edge on the i^{th} path has weight w_i, the second edge has weight 1, and the intermediate node has a self-loop (with weight 1). (Paths of length two are used because parallel edges are not allowed.)

   Note that a cycle-cover that uses the original edge corresponds to a collection of cycle-covers that use one out of the t paths (and the self-loops of all other intermediate nodes), whereas a cycle-cover that does not use the original edge corresponds to a cycle-cover that uses all the self-loops.

In particular, we may write each positive edge-weight w, having binary expansion $\sigma_{|w|-1} \cdots \sigma_0$, as $\sum_{i:\sigma_i=1}(1+1)^i$, and apply the adequate replacements (i.e., first apply the additive replacement to the outer sum (over $\{i : \sigma_i = 1\}$), next apply the product replacement to each power 2^i, and finally apply the additive replacement to each $1+1$). Applying this process to the matrix A' obtained in the first step, we *efficiently* obtain a matrix A'' with 0/1-entries such that $\mathtt{perm}(A') = \mathtt{perm}(A'')$. (In particular, the dimension of A'' is polynomial in the length of the binary representation of A', which in turn is polynomial in the length of the binary representation of A.) Combining the two reductions (steps), the proposition follows. ■

6.2.2 Approximate Counting

Having seen that exact counting (for relations in $\mathcal{PC}$) seems even harder than solving the corresponding search problems, we turn to relaxations of the counting problem. Before focusing on relative approximation, we briefly consider approximation with (large) additive deviation.

Let us consider the counting problem associated with an arbitrary $R \in \mathcal{PC}$. Without loss of generality, we assume that all solutions to n-bit instances have the same length $\ell(n)$, where indeed ℓ is a polynomial. We first note that, while it may be hard to compute $\#R$, given x it is easy to approximate $\#R(x)$ up to an additive error of $0.01 \cdot 2^{\ell(|x|)}$ (by randomly sampling potential solutions for x). Indeed, such an approximation is very rough, but it is not trivial (and in fact we do not know how to obtain it deterministically). In general, we can efficiently produce at random an estimate of $\#R(x)$ that, with high probability, deviates from the correct value by at most an additive term that is related to the absolute upper-bound on the number of solutions (i.e., $2^{\ell(|x|)}$).

Proposition 6.23 (approximation with large additive deviation): *Let $R \in \mathcal{PC}$ and ℓ be a polynomial such that $R \subseteq \cup_{n\in\mathbb{N}}\{0,1\}^n \times \{0,1\}^{\ell(n)}$. Then, for every polynomial p, there exists a probabilistic polynomial-time algorithm A such that for every $x \in \{0,1\}^*$ and $\delta \in (0,1)$ it holds that*

$$\mathsf{Pr}[|A(x,\delta) - \#R(x)| > (1/p(|x|)) \cdot 2^{\ell(|x|)}] < \delta. \tag{6.5}$$

As usual, δ is presented to A in binary, and hence the running time of $A(x,\delta)$ is upper-bounded by $\mathrm{poly}(|x| \cdot \log(1/\delta))$.

Proof Sketch: On input x and δ, algorithm A sets $t = \Theta(p(|x|)^2 \cdot \log(1/\delta))$, selects uniformly $y_1, ..., y_t$ and outputs $2^{\ell(|x|)} \cdot |\{i : (x, y_i) \in R\}|/t$. $\square$

Discussion. Proposition 6.23 is meaningful in the case that $\#R(x) > (1/p(|x|)) \cdot 2^{\ell(|x|)}$ holds for some x's. But otherwise, a trivial approximation (i.e., outputting the constant value zero) meets the bound of Eq. (6.5). In contrast to this notion of *additive approximation*, a *relative factor approximation* is typically more meaningful. Specifically, we will be interested in approximating $\#R(x)$ up-to a constant factor (or some other reasonable factor). In §6.2.2.1, we consider a natural $\#\mathcal{P}$-complete problem for which such a relative approximation can be obtained in probabilistic polynomial-time. We do not expect this to happen for every counting problem in $\#\mathcal{P}$, because a relative approximation allows for distinguishing instances having no solution from instances that do have solutions (i.e.,, deciding membership in S_R is reducible to a relative approximation of $\#R$). Thus, relative approximation for all $\#\mathcal{P}$ is at least as hard as deciding all problems in $\mathcal{NP}$. However, in §6.2.2.2 we show that the former is not harder than the latter; that is, relative approximation for any problem in $\#\mathcal{P}$ can be obtained by a randomized Cook-reduction to $\mathcal{NP}$. Before turning to these results, let us state the underlying definition (and actually strengthen it by requiring approximation to within a factor of $1 \pm \varepsilon$, for $\varepsilon \in (0,1)$).[12]

Definition 6.24 (approximation with relative deviation): *Let $f : \{0,1\}^* \to \mathbb{N}$ and $\varepsilon, \delta : \mathbb{N} \to [0,1]$. A randomized process Π is called an (ε,δ)-*approximator of *f if for every x it holds that*

$$\mathsf{Pr}\left[|\Pi(x) - f(x)| > \varepsilon(|x|) \cdot f(x)\right] \;<\; \delta(|x|). \tag{6.6}$$

We say that f is efficiently $(1-\varepsilon)$-approximable (or just $(1-\varepsilon)$-approximable) *if there exists a probabilistic polynomial-time algorithm A that constitute an $(\varepsilon, 1/3)$-approximator of f.*

[12] We refrain from formally defining an F-factor approximation, for an arbitrary F, although we shall refer to this notion in several informal discussions. There are several ways of defining the aforementioned term (and they are all equivalent when applied to our informal discussions). For example, an F-factor approximation of $\#R$ may mean that, with high probability, the output $A(x)$ satisfies $\#R(x)/F(|x|) \leq A(x) \leq F(|x|) \cdot \#R(x)$. Alternatively, we may require that $\#R(x) \leq A(x) \leq F(|x|) \cdot \#R(x)$ (or, alternatively, that $\#R(x)/F(|x|) \leq A(x) \leq \#R(x)$.

The error probability of the latter algorithm A (which has error probability 1/3) can be reduced to δ by $O(\log(1/\delta))$ repetitions (see Exercise 6.29). Typically, the running time of A will be polynomial in $1/\varepsilon$, and ε is called the deviation parameter.

6.2.2.1 Relative approximation for $\#R_{\texttt{dnf}}$

In this subsection we present a natural $\#\mathcal{P}$-complete problem for which constant factor approximation can be found in probabilistic polynomial-time. Stronger results regarding unnatural $\#\mathcal{P}$-complete problems appear in Exercise 6.30.

Consider the relation $R_{\texttt{dnf}}$ consisting of pairs (ϕ, τ) such that ϕ is a DNF formula and τ is an assignment satisfying it. Recall that the search problem of $R_{\texttt{dnf}}$ is easy to solve and that the proof of Theorem 6.19 establishes that $\#R_{\texttt{dnf}}$ is $\#\mathcal{P}$-complete (via a non-parsimonious reduction). Still, as we shall see, there exists a probabilistic polynomial-time algorithm that provides a constant factor approximation of $\#R_{\texttt{dnf}}$. We warn that the fact that $\#R_{\texttt{dnf}}$ is $\#\mathcal{P}$-complete *via a non-parsimonious reduction* means that the constant factor approximation for $\#R_{\texttt{dnf}}$ does not seem to imply a similar approximation for all problems in $\#\mathcal{P}$. In fact, we should not expect each problem in $\#\mathcal{P}$ to have a (probabilistic) polynomial-time constant-factor approximation algorithm because this would imply $\mathcal{NP} \subseteq \mathcal{BPP}$ (since a constant factor approximation allows for distinguishing the case in which the instance has no solution from the case in which the instance has a solution).

The approximation algorithm for $\#R_{\texttt{dnf}}$ is obtained by a deterministic reduction of the task of $(\varepsilon, 1/3)$-approximating $\#R_{\texttt{dnf}}$ to an (additive deviation) approximation of the type provided in Proposition 6.23. Consider a DNF formula $\phi = \bigvee_{i=1}^{m} C_i$, where each $C_i : \{0,1\}^n \to \{0,1\}$ is a conjunction. Our task is to approximate the number of assignments that satisfy at least one of the conjunctions. Actually, we will deal with the more general problem in which we are (implicitly) given m subsets $S_1, ..., S_m \subseteq \{0,1\}^n$ and wish to approximate $|\bigcup_i S_i|$. In our case, each S_i is the set of assignments that satisfy the conjunction C_i. In general, we make two computational assumptions regarding these sets (while letting "efficient" mean *implementable in time polynomial in* $n \cdot m$):

1. Given $i \in [m]$, one can efficiently determine $|S_i|$.

2. Given $i \in [m]$ and $J \subseteq [m]$, one can efficiently approximate $\mathsf{Pr}_{s \in S_i}\left[s \in \bigcup_{j \in J} S_j\right]$ up to an *additive deviation of* $1/\mathrm{poly}(n+m)$.

These assumptions are satisfied in our setting (where $S_i = C_i^{-1}(1)$, see Exercise 6.31). Now, the key observation towards approximating $|\bigcup_{i=1}^{m} S_i|$ is that

$$\left|\bigcup_{i=1}^{m} S_i\right| = \sum_{i=1}^{m}\left|S_i \setminus \bigcup_{j<i} S_j\right| = \sum_{i=1}^{m} \mathsf{Pr}_{s \in S_i}\left[s \notin \bigcup_{j<i} S_j\right] \cdot |S_i| \qquad (6.7)$$

and that the probabilities in Eq. (6.7) can be approximated by the second assumption. This leads to the following algorithm, where ε denotes the desired deviation parameter (i.e., we wish to obtain $(1 \pm \varepsilon) \cdot |\bigcup_{i=1}^{m} S_i|$).

Construction 6.25 *Let $\varepsilon' = \varepsilon/m$. For $i = 1$ to m do:*

1. *Using the first assumption, compute $|S_i|$.*
2. *Using the second assumption, obtain an approximation $\widetilde{p}_i = p_i \pm \varepsilon'$, where $p_i \stackrel{\text{def}}{=} \mathsf{Pr}_{s \in S_i}[s \notin \bigcup_{j<i} S_j]$. Set $a_i \stackrel{\text{def}}{=} \widetilde{p}_i \cdot |S_i|$.*

Output the sum of the a_i's.

Let $N_i = p_i \cdot |S_i|$, and note that by Eq. (6.7) it holds that $|\bigcup_i S_i| = \sum_i N_i$. We are interested in the quality of the approximation to $\sum_i N_i$ provided by $\sum_i a_i$. Using $a_i = (p_i \pm \varepsilon') \cdot |S_i| = N_i \pm \varepsilon' \cdot |S_i|$ (for each i), we have $\sum_i a_i = \sum_i N_i \pm \varepsilon' \cdot \sum_i |S_i|$. Using $\sum_i |S_i| \le m \cdot |\bigcup_i S_i| = m \cdot \sum_i N_i$ (and $\varepsilon = m\varepsilon'$), we get $\sum_i a_i = (1 \pm \varepsilon) \cdot \sum_i N_i$. Thus, we obtain the following result (see Exercise 6.31).

Proposition 6.26 *For every positive polynomial p, the counting problem of $R_{\mathtt{dnf}}$ is efficiently $(1-(1/p))$-approximable.*

Using the reduction presented in the proof of Theorem 6.19, we conclude that the number of *un*satisfying assignments to a given CNF formula is efficiently $(1-(1/p))$-approximable. We warn, however, that the number of satisfying assignments to such a formula is *not* efficiently approximable. This concurs with the general phenomenon by which *relative approximation may be possible for one quantity, but not for the complementary quantity.* Needless to say, such a phenomenon does not occur in the context of additive-deviation approximation.

6.2.2.2 Relative approximation for $\#\mathcal{P}$

Recall that we cannot expect to efficiently approximate every $\#\mathcal{P}$ problem, where throughout the rest of this section "approximation" is used as a shorthand for "relative approximation" (as in Definition 6.24). Specifically, efficiently approximating $\#R$ yields an efficient algorithm for deciding membership in $S_R = \{x : R(x) \neq \emptyset\}$. Thus, at best we can hope that approximating $\#R$ is not harder than deciding S_R (i.e., that approximating $\#R$ is reducible in polynomial-time to S_R). This is indeed the case for every NP-complete problem (i.e., if S_R is NP-complete). More generally, we show that approximating any problem in $\#\mathcal{P}$ is reducible in probabilistic polynomial-time to $\mathcal{NP}$.

Theorem 6.27 *For every $R \in \mathcal{PC}$ and every positive polynomial p, there exists a probabilistic polynomial-time oracle machine that when given oracle access to $\mathcal{NP}$ constitutes a $(1/p, \mu)$-approximator of $\#R$, where μ is a negligible function* (e.g., $\mu(n) = 2^{-n}$).

Recall that it suffices to provide a $(1/p, \delta)$-approximator of $\#R$, for any constant $\delta < 0.5$, because error reduction is applicable in this context (see Exercise 6.29). Furthermore, it suffices to provide a $(1/2, \delta)$-approximator for every problem in $\#\mathcal{P}$ (see Exercise 6.32).

Teaching note: The following proof relies on the notion of hashing functions, presented in Appendix D.2. Specifically, we shall assume familiarity with the basic definition (see Appendix D.2.1), at least one construction (see Appendix D.2.2), and Lemma D.4 (of Appendix D.2.3). The more advanced material of Appendix D.2.3 (which follows Lemma D.4) will not be used in the current section (but part of it will be used in §6.2.4.2).

Proof: Given x, we show how to approximate $|R(x)|$ to within some constant factor. The desired $(1-(1/p))$-approximation can be obtained as in Exercise 6.32. We may also assume that $R(x) \neq \emptyset$, by starting with the query "is x in S_R" and halting (with output 0) if the answer is negative. Without loss of generality, we assume that $R(x) \subseteq \{0,1\}^\ell$, where $\ell = \text{poly}(|x|)$. We focus on finding some $i \in \{1, ..., \ell\}$ such that $2^{i-4} \leq |R(x)| \leq 2^{i+4}$.

We proceed in iterations. For $i = 1, ..., \ell+1$, we find out whether or not $|R(x)| < 2^i$. If the answer is positive then we halt with output 2^i, and otherwise we proceed to the next iteration. (Indeed, if we were able to obtain correct answers to all these queries then the output 2^i would satisfy $2^{i-1} \leq |R(x)| < 2^i$.)

Needless to say, the key issue is how to check whether $|R(x)| < 2^i$. The main idea is to use a "random sieve" on the set $R(x)$ such that each element passes the sieve with probability 2^{-i}. Thus, we expect $|R(x)|/2^i$ elements of $R(x)$ to pass the sieve. Assuming that the number of elements in $R(x)$ that pass the random sieve is indeed $\lfloor |R(x)|/2^i \rfloor$, it holds that $|R(x)| \geq 2^i$ *if and only if some element of* $R(x)$ *passes the sieve*. Assuming that the sieve can be implemented efficiently, the question of whether or not some element in $R(x)$ passed the sieve is of an "NP-type" (and thus can be referred to our NP-oracle). Combining both assumptions, we may implement the foregoing process by proceeding to the next iteration as long as some element of $R(x)$ passes the sieve. Furthermore, this implementation will provide a reasonably good approximation even if the number of elements in $R(x)$ that pass the random sieve is only approximately equal to $|R(x)|/2^i$. In fact, the level of approximation that this implementation provides is closely related to the level of approximation that is provided by the random sieve. Details follow.

Implementing a random sieve. The random sieve is implemented by using a family of hashing functions (see Appendix D.2). Specifically, in the i^{th} iteration we use a family H_ℓ^i such that each $h \in H_\ell^i$ has a poly(ℓ)-bit long description and maps ℓ-bit long strings to i-bit long strings. Furthermore, the family is accompanied with an efficient evaluation algorithm (i.e., mapping adequate pairs (h, x) to $h(x)$) and satisfies (for every $S \subseteq \{0,1\}^\ell$)

$$\mathsf{Pr}_{h \in H_\ell^i}[|\{y \in S : h(y) = 0^i\}| \notin (1-\varepsilon, 1+\varepsilon) \cdot 2^{-i}|S|] \;<\; \frac{2^i}{\varepsilon^2 |S|} \qquad (6.8)$$

(see Lemma D.4). *The random sieve will let y pass if and only if $h(y) = 0^i$*. Indeed, this random sieve is not as perfect as we assumed in the foregoing discussion, but Eq. (6.8) suggests that in some sense this sieve is good enough. In particular, Eq. (6.8) implies that if $i \le \log_2 |S| - O(1)$ then some string in S is likely to pass the sieve, whereas if $i \ge \log_2 |S| + O(1)$ then no string in S is likely to pass the sieve.

Implementing the queries. Recall that for some x, i and $h \in H_\ell^i$, we need to determine whether $\{y \in R(x) : h(y) = 0^i\} = \emptyset$. This type of question can be cast as membership in the set

$$S_{R,H} \stackrel{\text{def}}{=} \{(x,i,h) : \exists y \text{ s.t. } (x,y) \in R \wedge h(y) = 0^i\}. \tag{6.9}$$

Using the hypotheses that $R \in \mathcal{PC}$ and that the family of hashing functions has an efficient evaluation algorithm, it follows that $S_{R,H}$ is in $\mathcal{NP}$.

The actual procedure. On input $x \in S_R$ and oracle access to $S_{R,H}$, we proceed in iterations, starting with $i = 1$ and halting at $i = \ell$ (if not before), where ℓ denotes the length of the potential solutions for x. In the i^{th} iteration (where $i < \ell$), we uniformly select $h \in H_\ell^i$ and query the oracle on whether or not $(x, i, h) \in S_{R,H}$. If the answer is negative then we halt with output 2^i, and otherwise we proceed to the next iteration (using $i \leftarrow i + 1$). Needless to say, if we reach the last iteration (i.e., $i = \ell$) then we just halt with output 2^ℓ.

Indeed, we have ignored the case that $x \notin S_R$, which can be easily handled by a minor modification of the foregoing procedure. Specifically, on input x, we first query S_R on x and halt with output 0 if the answer is negative. Otherwise we proceed as in the foregoing procedure.

The analysis. We upper-bound separately the probability that the procedure outputs a value that is too small and the probability that it outputs a value that is too big. In light of the foregoing discussion, we may assume that $|R(x)| > 0$, and let $i_x = \lfloor \log_2 |R(x)| \rfloor \ge 0$. Intuitively, at any iteration $i < i_x$, we expect (at least) $2^{i_x - i}$ elements of $R(x)$ to pass the sieve and thus we are unlikely to halt before iteration $i_x - O(1)$. Similarly, we are unlikely to reach iteration $i_x + O(1)$ because at this stage we expect no elements of $R(x)$ to pass the sieve (since the actual expectation is $2^{-O(1)}$). A more rigorous analysis (of both cases) follows.

1. The probability that the procedure *halts in a specific iteration $i < i_x$* equals $\mathsf{Pr}_{h \in H_\ell^i}[|\{y \in R(x) : h(y) = 0^i\}| = 0]$, which in turn is upper-bounded by $2^i/|R(x)|$ (using Eq. (6.8) with $\varepsilon = 1$).[13] Thus, the probability that the procedure halts *before* iteration $i_x - 3$ is upper-bounded by $\sum_{i=0}^{i_x-4} 2^i/|R(x)|$, which in turn is less than $1/8$ (because $i_x \le \log_2 |R(x)|$). It follows that, with probability at least $7/8$, the output is at least $2^{i_x - 3} > |R(x)|/16$ (because $i_x > (\log_2 |R(x)|) - 1$).

2. The probability that the procedure *does not halt in iteration $i > i_x$* equals $\mathsf{Pr}_{h \in H_\ell^i}[|\{y \in R(x) : h(y) = 0^i\}| \ge 1]$, which in turn is upper-bounded by

[13] Note that 0 does not reside in the open interval $(0, 2\rho)$, where $\rho = |R(x)|/2^i > 0$.

$\alpha/(\alpha-1)^2$, where $\alpha = 2^i/|R(x)| > 1$ (using Eq. (6.8) with $\varepsilon = \alpha - 1$).[14] Thus, the probability that the procedure does not halt by iteration $i_x + 4$ is upper-bounded by $8/49 < 1/6$ (because $i_x > (\log_2 |R(x)|) - 1$). Thus, with probability at least 5/6, the output is at most $2^{i_x+4} \leq 16 \cdot |R(x)|$ (because $i_x \leq \log_2 |R(x)|$).

Thus, with probability at least $(7/8)-(1/6) > 2/3$, the foregoing procedure outputs a value v such that $v/16 \leq |R(x)| < 16v$. Reducing the deviation by using the ideas presented in Exercise 6.32 (and reducing the error probability as in Exercise 6.29), the theorem follows. ■

Digest. The key observation underlying the proof Theorem 6.27 is that, while (even with the help of an NP-oracle) we cannot directly test whether the number of solutions is greater than a given number, we can test (with the help of an NP-oracle) whether the number of solutions that "survive a random sieve" is greater than zero. Since the number of solutions that survive a random sieve reflects the total number of solutions (normalized by the sieve's density), this offers a way of approximating the total number of solutions.

We mention that one can also test whether the number of solutions that "survive a random sieve" is greater than a small number, where small means polynomial in the length of the input (see Exercise 6.34). Specifically, the complexity of this test is linear in the size of the threshold, and not in the length of its binary description. Indeed, in many settings it is more advantageous to use a threshold that is polynomial in some efficiency parameter (rather than using the threshold zero); examples appear in §6.2.4.2 and in [103].

6.2.3 Searching for unique solutions

A natural computational problem (regarding search problems), which arises when discussing the number of solutions, is the problem of distinguishing instances having a single solution from instances having no solution (or finding the unique solution whenever such exists). We mention that instances having a single solution facilitate numerous arguments (see, for example, Exercise 6.24 and §10.2.2.1). Formally, searching for and deciding the existence of unique solutions are defined within the framework of promise problems (see Section 2.4.1).

Definition 6.28 (search and decision problems for unique solution instances): *The* set of instances having unique solutions *with respect to the binary relation R is defined as* $\mathsf{US}_R \stackrel{\rm def}{=} \{x : |R(x)| = 1\}$, *where* $R(x) \stackrel{\rm def}{=} \{y : (x,y) \in R\}$. *As usual, we denote* $S_R = \{x : |R(x)| \geq 1\}$, *and* $\overline{S}_R \stackrel{\rm def}{=} \{0,1\}^* \setminus S_R = \{x : |R(x)| = 0\}$.

[14] Here we use the fact that $1 \notin (2\alpha^{-1} - 1, 1)$. A better bound can be obtained by using the hypothesis that, for every y, when h is uniformly selected in H_ℓ^i, the value of $h(y)$ is uniformly distributed in $\{0,1\}^i$. In this case, $\mathsf{Pr}_{h \in H_\ell^i}[|\{y \in R(x) : h(y) = 0^i\}| \geq 1]$ is upper-bounded by $\mathsf{E}_{h \in H_\ell^i}[|\{y \in R(x) : h(y) = 0^i\}|] = |R(x)|/2^i$.

- *The problem of* finding unique solutions for R *is defined as the* search problem R with promise $\mathtt{US}_R \cup \overline{S}_R$ (see Definition 2.29).

  In continuation to Definition 2.30, candid searching for unique solutions for R *is defined as the* search problem R with promise $\mathtt{US}_R$.

- *The problem of* deciding unique solution for R *is defined as the* promise problem $(\mathtt{US}_R, \overline{S}_R)$ (see Definition 2.31).

Interestingly, in many natural cases, the promise does not make any of these problems any easier than the original problem. That is, for all known NP-complete problems, the original problem is reducible in probabilistic polynomial-time to the corresponding unique instances problem.

Theorem 6.29 *Let $R \in \mathcal{PC}$ and suppose that every search problem in $\mathcal{PC}$ is parsimoniously reducible to R. Then solving the search problem of R* (resp., deciding membership in S_R) *is reducible in probabilistic polynomial-time to finding unique solutions for R* (resp., to the promise problem $(\mathtt{US}_R, \overline{S}_R)$). *Furthermore, there exists a probabilistic polynomial-time computable mapping M such that for every $x \in \overline{S}_R$ it holds that $\Pr[M(x) \in \overline{S}_R] = 1$, whereas for every $x \in S_R$ it holds that $\Pr[M(x) \in \mathtt{US}_R] \geq 1/\mathrm{poly}(|x|)$.*

We highlight the fact that the hypothesis asserts that R is $\mathcal{PC}$-complete *via parsimonious reductions*; this hypothesis is crucial to Theorem 6.29 (see Exercise 6.35). The large (but bounded-away from 1) error probability of the randomized Karp-reduction M can be reduced by repetitions, yielding a randomized Cook-reduction with exponentially vanishing error probability. Note that the resulting reduction may make many queries that violate the promise, and still yields the correct answer (with high probability) by relying on queries that satisfy the promise. (Specifically, in the case of search problems, we avoid wrong solutions by checking each solution obtained, while in the case of decision problems we rely on the fact that for every $x \in \overline{S}_R$ it always holds that $M(x) \in \overline{S}_R$.)

Proof: We focus on establishing the furthermore clause (and the main claim follows). The proof uses many of the ideas of the proof of Theorem 6.27, and we refer to the latter for motivation. We shall again make essential use of hashing functions, and rely on the material presented in Appendix D.2.1–D.2.2.

As in the proof of Theorem 6.27, the idea is to apply a "random sieve" on $R(x)$, this time with the hope that a single element survives. Specifically, if we let each element pass the sieve with probability approximately $1/|R(x)|$ then with constant probability a single element survives. In such a case, we shall obtain an instance with a unique solution (i.e., an instance of $S_{R,H}$ having a single NP-witness), which will (essentially) fulfill our quest. Sieving will be performed by a random function selected in an adequate hashing family (see Appendix D.2). A couple of questions arise:

1. *How do we get an approximation to $|R(x)|$?* Note that we need such an approximation in order to determine the adequate hashing family. Note that

invoking Theorem 6.27 will not do, because the said oracle machine uses an oracle to $\mathcal{NP}$ (which puts us back to square one, let alone that the said reduction makes many queries).[15] Instead, we just select $m \in \{0, ..., \text{poly}(|x|)\}$ uniformly and note that (if $|R(x)| > 0$ then) $\mathsf{Pr}[m = \lceil \log_2 |R(x)| \rceil] = 1/\text{poly}(|x|)$. Next, we randomly map x to (x, m, h), where h is uniformly selected in an adequate hashing family.

2. *How does the question of whether a single element of $R(x)$ pass the random sieve translate to an instance of the unique-solution problem for R?* Recall that in the proof of Theorem 6.27 the non-emptiness of the set of element of $R(x)$ that pass the sieve (defined by h) was determined by checking membership (of (x, m, h)) in $S_{R,H} \in \mathcal{NP}$ (defined in Eq. (6.9)). Furthermore, the number of NP-witnesses for $(x, m, h) \in S_{R,H}$ equals the number of elements of $R(x)$ that pass the sieve. Thus, a single element of $R(x)$ passes the sieve (defined by h) if and only if $(x, m, h) \in S_{R,H}$ has a single NP-witness. Using the parsimonious reduction of $S_{R,H}$ to S_R (which is guaranteed by the theorem's hypothesis), we obtained the desired instance.

Note that in case $R(x) = \emptyset$ the aforementioned mapping always generates a no-instance (of $S_{R,H}$ and thus of S_R). Details follow.

Implementation (i.e., the mapping M). As in the proof of Theorem 6.27, we assume, without loss of generality, that $R(x) \subseteq \{0,1\}^\ell$, where $\ell = \text{poly}(|x|)$. We start by uniformly selecting $m \in \{1, ..., \ell + 1\}$ and $h \in H_\ell^m$, where H_ℓ^m is a family of efficiently computable and pairwise-independent hashing functions (see Definition D.1) mapping ℓ-bit long strings to m-bit long strings. Thus, we obtain an instance (x, m, h) of $S_{R,H} \in \mathcal{NP}$ such that the set of valid solutions for (x, m, h) equals $\{y \in R(x) : h(y) = 0^m\}$. Using the parsimonious reduction g of the NP-witness relation of $S_{R,H}$ to R (i.e., the NP-witness relation of S_R), we map (x, m, h) to $g(x, m, h)$, and it holds that $|\{y \in R(x) : h(y) = 0^m\}|$ equals $|R(g(x, m, h))|$. To summarize, on input x the randomized mapping M outputs the instance $M(x) \stackrel{\text{def}}{=} g(x, m, h)$, where $m \in \{1, ..., \ell + 1\}$ and $h \in H_\ell^m$ are uniformly selected.

The analysis. Note that for any $x \in \overline{S}_R$ it holds that $\mathsf{Pr}[M(x) \in \overline{S}_R] = 1$. Assuming that $x \in S_R$, with probability exactly $1/(\ell + 1)$ it holds that $m = m_x$, where $m_x \stackrel{\text{def}}{=} \lceil \log_2 |R(x)| \rceil + 1$. Focusing on the case that $m = m_x$, for a uniformly selected $h \in H_\ell^m$, we shall lower-bound the probability that the set $R_h(x) \stackrel{\text{def}}{=} \{y \in R(x) : h(y) = 0^m\}$ is a singleton. First, using the Inclusion-Exclusion Principle, we lower-bound $\mathsf{Pr}_{h \in H_\ell^{m_x}}[|R_h(x)| > 0]$ by

$$\sum_{y \in R(x)} \mathsf{Pr}_{h \in H_\ell^{m_x}}[h(y) = 0^{m_x}] \; - \sum_{y_1 < y_2 \in R(x)} \mathsf{Pr}_{h \in H_\ell^{m_x}}[h(y_1) = h(y_2) = 0^{m_x}].$$

[15] Needless to say, both problems can be resolved by using a reduction to unique-solution instances, but we still do not have such a reduction – we are currently designing it.

Next, we upper-bound $\mathsf{Pr}_{h\in H_\ell^{m_x}}[|R_h(x)|>1]$ by

$$\sum_{y_1<y_2\in R(x)} \mathsf{Pr}_{h\in H_\ell^{m_x}}[h(y_1)=h(y_2)=0^{m_x}].$$

Combining these two bounds, we get

$$\begin{aligned}
&\mathsf{Pr}_{h\in H_\ell^{m_x}}[|R_h(x)|=1] \\
&\quad= \mathsf{Pr}_{h\in H_\ell^{m_x}}[|R_h(x)|>0] \;-\; \mathsf{Pr}_{h\in H_\ell^{m_x}}[|R_h(x)|>1] \\
&\quad\geq \sum_{y\in R(x)} \mathsf{Pr}_{h\in H_\ell^{m_x}}[h(y)=0^{m_x}] \;-\; 2\cdot \sum_{y_1<y_2\in R(x)} \mathsf{Pr}_{h\in H_\ell^{m_x}}[h(y_1)=h(y_2)=0^{m_x}] \\
&\quad= |R(x)|\cdot 2^{-m_x} - 2\cdot \binom{|R(x)|}{2}\cdot 2^{-2m_x}
\end{aligned}$$

where the last equality is due to the pairwise independence property. Using $2^{m_x-2}<|R(x)|\leq 2^{m_x-1}$, it follows that

$$\mathsf{Pr}_{h\in H_\ell^{m_x}}[|R_h(x)|=1] \geq \min_{1/4<\rho\leq 1/2}\{\rho-\rho^2\} > \frac{1}{8}.$$

Thus, $\mathsf{Pr}[M(x)\in \mathtt{US}_R]\geq 1/(8(\ell+1))$, and the theorem follows. ∎

Comment. Theorem 6.29 is sometimes stated as referring to the unique solution problem of SAT. In this case and when using a specific family of pairwise independent hashing functions, the use of the parsimonious reduction can be avoided. For details see Exercise 6.37.

Digest. The proof of Theorem 6.29 combines two reduction steps, which refer to the NP-witness relation of $S_{R,H}$, herein denoted R'. The main step is a many-to-one randomized reduction of the search problem of R (resp., of S_R) to the problem of finding unique solutions for R' (resp., to $(\mathtt{US}_{R'},\overline{S}_{R'})$). The second step is a deterministic many-to-one reduction of the latter problem to the problem of finding unique solutions for R. Indeed, the proof of Theorem 6.29 focuses on the first step, while the second step is provided by the parsimonious reduction of R' to R (which is guaranteed by the hypothesis). As stated in the previous comment, in the case of SAT there is a direct way of performing the second step.

6.2.4 Uniform generation of solutions

Recall that approximately counting the number of solutions for a relation R is a straining of the decision problem S_R (which asks for distinguishing the case that some solutions exist from the case that no solutions exist). We now turn to a new type of computational problems, which may be viewed as a straining of search problems. We refer to the task of generating a uniformly distributed solution for a given instance, rather than merely finding an adequate solution. Nevertheless, as we shall see, for many natural problems (and all NP-complete ones) generating a uniformly distributed solution is randomly reducible to finding a solution.

Needless to say, by definition, algorithms solving this ("uniform generation") task must be randomized. Focusing on relations in $\mathcal{PC}$ we consider two versions of the problem, which differ by the level of approximation provided for the desired (uniform) distribution.[16]

Definition 6.30 (uniform generation): *Let $R \in \mathcal{PC}$ and $S_R = \{x : |R(x)| \geq 1\}$, and let Π be a probabilistic process.*

1. *We say that Π* solves the uniform generation problem of *R if, on input $x \in S_R$, the process Π outputs either an element of $R(x)$ or a special symbol, denoted $\perp$, such that $\mathsf{Pr}[\Pi(x) \in R(x)] \geq 1/2$ and for every $y \in R(x)$ it holds that $\mathsf{Pr}[\Pi(x) = y \mid \Pi(x) \in R(x)] = 1/|R(x)|$.*

2. *For $\varepsilon : \mathbb{N} \to [0,1]$, we say that Π* solves the $(1-\varepsilon)$-approximate uniform generation problem of *R if, on input $x \in S_R$, the distribution $\Pi(x)$ is $\varepsilon(|x|)$-close[17] to the uniform distribution on $R(x)$.*

In both cases, without loss of generality, we may require that if $x \notin S_R$ then $\mathsf{Pr}[\Pi(x) = \perp] = 1$. More generally, we may require that Π never outputs a string not in $R(x)$.

Note that the error probability of uniform generation (as in Item 1) can be made exponentially vanishing (in $|x|$) by employing error-reduction. In contrast, we are not aware of any general way of reducing the deviation of an approximate uniform generation procedure (as in Item 2).[18]

In §6.2.4.1 we show that, for many search problems, approximate uniform generation is computationally equivalent to approximate counting. In §6.2.4.2 we present a direct approach for solving the uniform generation problem of any search problem in $\mathcal{PC}$ by using an oracle to $\mathcal{NP}$. Thus, the uniform generation problem of any NP-complete problem is randomly reducible to the problem itself (either in its search or decision version).

6.2.4.1 Relation to approximate counting

We show that, for many natural search problems in $\mathcal{PC}$, the approximate counting problem associated with R is computationally equivalent to approximate uniform generation with respect to R. Specifically, we refer to search problems $R \in \mathcal{PC}$ such that $R'(x; y') \stackrel{\text{def}}{=} \{y'' : (x, y'y'') \in R\}$ is *strongly parsimoniously reducible to* R, where a strongly parsimonious reduction of R' to R is a parsimonious reduction g

[16] Note that a probabilistic algorithm running in strict polynomial-time is not able to output a perfectly uniform distribution on sets of certain sizes. Specifically, referring to the standard model that allows only for uniformly selected binary values, such algorithms cannot output a perfectly uniform distribution on sets having cardinality that is not a power of two.

[17] See Appendix D.1.1.

[18] We note that in some cases, the deviation of an approximate uniform generation procedure can be reduced. See discussion following Theorem 6.31.

that is coupled with an efficiently computable 1-1 mapping of pairs $(g(x), y) \in R$ to pairs $(x, h(x, y)) \in R'$ (i.e., h is efficiently computable and $h(x, \cdot)$ is a 1-1 mapping of $R(g(x))$ to $R'(x)$). For technical reasons, we also assume that $|g(x)| \geq |x|$ for every x.[19] Note that, for many natural search problems R, the corresponding R' is strongly parsimoniously reducible to R, where the additional technical condition may be enforced by adequate padding (cf., Exercise 2.30). This holds, in particular, for the search problems of `SAT` and `Perfect Matching`.

Recalling that both types of approximation problems are parameterized by the level of precision, we obtain the following quantitative form of the aforementioned equivalence.

Theorem 6.31 *Let $R \in \mathcal{PC}$ and let ℓ be a polynomial such that for every $(x,y) \in R$ it holds that $|y| \leq \ell(|x|)$. Suppose that R' is strongly parsimoniously reducible to R, where $R'(x; y') \stackrel{\rm def}{=} \{y'' : (x, y'y'') \in R\}$.*

1. From approximate counting to approximate uniform generation: *Let $\varepsilon(n) = 1/5\ell(n)$ and let $\mu : \mathbb{N} \to (0,1)$ be a function satisfying $\mu(n) \geq \exp(-\mathrm{poly}(n))$. Then, $(1-\mu)$-approximate uniform generation for R is reducible in probabilistic polynomial-time to $(1-\varepsilon)$-approximating $\#R$.*

2. From approximate uniform generation to approximate counting: *For every non-increasing and noticeable $\varepsilon : \mathbb{N} \to (0,1)$* (i.e., $\varepsilon(n) \geq 1/\mathrm{poly}(n)$ for every n), *the problem of $(1-\varepsilon)$-approximating $\#R$ is reducible in probabilistic polynomial-time to $(1-\varepsilon')$-approximate uniform generation problem of R, where $\varepsilon'(n) = \varepsilon(n)/7\ell(n)$.*

In fact, Part 1 holds also in case R' is just parsimoniously reducible to R.

Note that the quality of the approximate uniform generation asserted in Part 1 (i.e., μ) is independent of the quality of the approximate counting procedure (i.e., ε) to which the former is reduced, provided that the approximate counter performs better than some threshold. On the other hand, the quality of the approximate counting asserted in Part 2 (i.e., ε) does depend on the quality of the approximate uniform generation (i.e., ε'), but cannot reach beyond a certain bound (i.e., noticeable relative deviation). Recall, that for problems that are NP-complete under parsimonious reductions the quality of approximate counting procedures can be improved (see Exercise 6.33). However, Theorem 6.31 is most useful when applied to problems that are not NP-complete, because for problems that are NP-complete both approximate counting and uniform generation are randomly reducible to the corresponding search problem (see Exercise 6.39).

Proof: Throughout the proof, we assume for simplicity (and in fact without loss of generality) that $R(x) \neq \emptyset$ and $R(x) \subseteq \{0,1\}^{\ell(|x|)}$.

Towards Part 1, let us first reduce the uniform generation problem of R to $\#R$ (rather than to approximating $\#R$). On input $x \in S_R$, we shall generate

[19] This technical condition allows us to replace deviation bounds expressed in terms of $|g(x)|$ by bounds expressed in terms of $|x|$, while relying on the fact that $\varepsilon(|g(x)|) \leq \varepsilon(|x|)$ holds for any non-increasing $\varepsilon : \mathbb{N} \to (0,1)$.

a uniformly distributed $y \in R(x)$ by randomly generating its bits one after the other. We proceed in iterations, entering the i^{th} iteration with an $(i-1)$-bit long string y' such that $R'(x;y') \stackrel{\text{def}}{=} \{y'' : (x, y'y'') \in R\}$ is not empty. With probability $|R'(x;y'1)|/|R'(x;y')|$ we set the i^{th} bit to equal 1, and otherwise we set it to equal 0. We obtain both $|R'(x;y'1)|$ and $|R'(x;y')|$ by using a parsimonious reduction g of $R' = \{((x;y'),y'') : (x,y'y'') \in R\} \in \mathcal{PC}$ to R. That is, we obtain $|R'(x;y')|$ by querying for the value of $|R(g(x;y'))|$. Ignoring integrality issues, all this works perfectly (i.e., we generate an $\ell(n)$-bit string uniformly distributed in $R(x)$) as long as we have oracle access to $\#R$. Since we only have oracle access to an approximation of $\#R$, a careful implementation of the foregoing idea is in place.

Let us denote the approximation oracle by A. Firstly, by adequate error reduction, we may assume that, for every z, it holds that $\Pr[A(z) \in (1\pm\varepsilon(n))\cdot\#R(z)] > 1-\mu'(|z|)$, where $\mu'(n) = \mu(n)/\ell(n)$. In the rest of the analysis we ignore the probability that the estimate of $\#R(z)$ provided by the randomized oracle A (on query z) deviates from the aforementioned interval. (We note that these rare events are the only source of the possible deviation of the output distribution from the uniform distribution on $R(x)$.)[20] Next, let us assume for a moment that A is *deterministic* and that for every x and y' it holds that

$$A(g(x;y'0)) + A(g(x;y'1)) \;\le\; A(g(x;y')). \tag{6.10}$$

We also assume that the approximation is correct at the "trivial level" (where one may just check whether or not (x,y) is in R); that is, for every $y \in \{0,1\}^{\ell(|x|)}$, it holds that

$$A(g(x;y)) = 1 \text{ if } (x,y) \in R \text{ and } A(g(x;y)) = 0 \text{ otherwise.} \tag{6.11}$$

We modify the i^{th} iteration of the foregoing procedure such that, when entering with the $(i-1)$-bit long prefix y', we set the i^{th} bit to $\sigma \in \{0,1\}$ with probability $A(g(x;y'\sigma))/A(g(x;y'))$ and halt (with output $\perp$) with the residual probability (i.e., $1-(A(g(x;y'0))/A(g(x;y'))) - (A(g(x;y'1))/A(g(x;y')))$). Indeed, Eq. (6.10) guarantees that the latter instruction is sound, since the two main probabilities sum-up to at most 1. If we completed the last (i.e., $\ell(|x|)^{\text{th}}$) iteration, then we output the $\ell(|x|)$-bit long string that was generated. Thus, as long as Eq. (6.10) holds (but regardless of other aspects of the quality of the approximation), every $y = \sigma_1\cdots\sigma_{\ell(|x|)} \in R(x)$, is output with probability

$$\frac{A(g(x;\sigma_1))}{A(g(x;\lambda))}\cdot\frac{A(g(x;\sigma_1\sigma_2))}{A(g(x;\sigma_1))}\cdots\frac{A(g(x;\sigma_1\sigma_2\cdots\sigma_{\ell(|x|)}))}{A(g(x;\sigma_1\sigma_2\cdots\sigma_{\ell(|x|)-1}))} \tag{6.12}$$

which, by Eq. (6.11), equals $1/A(g(x;\lambda))$. Thus, the procedure outputs each element of $R(x)$ with equal probability, and never outputs a non-$\perp$ value that is outside $R(x)$. It follows that the quality of approximation only effects the probability

[20] Note that the (negligible) effect of these rare events may not be easy to correct. For starters, we do not necessarily get an indication when these rare events occur. Furthermore, these rare events may occur with different probability in the different invocations of algorithm A (i.e., on different queries).

that the procedure outputs a non-$\perp$ value (which in turn equals $|R(x)|/A(g(x;\lambda))$). The key point is that, as long as Eq. (6.11) holds, the specific approximate values obtained by the procedure are immaterial – with the exception of $A(g(x;\lambda))$, all these values "cancel out".

We now turn to enforcing Eq. (6.10) and Eq. (6.11). We may enforce Eq. (6.11) by performing the straightforward check (of whether or not $(x,y)\in R$) rather than invoking $A(g(x,y))$.[21] As for Eq. (6.10), we enforce it artificially by using $A'(x,y') \stackrel{\rm def}{=} (1+\varepsilon(|x|))^{3(\ell(|x|)-|y'|)}\cdot A(g(x;y'))$ instead of $A(g(x;y'))$. Recalling that $A(g(x;y'))=(1\pm\varepsilon(|x|))\cdot|R'(x;y')|$, we have

$$\begin{array}{rcl} A'(x,y') & > & (1+\varepsilon(|x|))^{3(\ell(|x|)-|y'|)}\cdot(1-\varepsilon(|x|))\cdot|R'(x;y')| \\ A'(x,y'\sigma) & < & (1+\varepsilon(|x|))^{3(\ell(|x|)-|y'|-1)}\cdot(1+\varepsilon(|x|))\cdot|R'(x;y'\sigma)| \end{array}$$

and the claim (that Eq. (6.10) holds) follows by using $(1-\varepsilon(|x|))\cdot(1+\varepsilon(|x|))^3 > (1+\varepsilon(|x|))$. Note that the foregoing modification only effects the probability of outputting a non-$\perp$ value; this good event now occurs with probability $|R'(x;\lambda)|/A'(x,\lambda)$, which is lower-bounded by $(1+\varepsilon(|x|))^{-(3\ell(|x|)+1)} > 1/2$, where the inequality is due to the setting of ε (i.e., $\varepsilon(n)=1/5\ell(n)$). Finally, we refer to our assumption that A is deterministic. This assumption was only used in order to identify the value of $A(g(x,y'))$ obtained and used in the $(|y'|-1)^{\rm st}$ iteration with the value of $A(g(x,y'))$ obtained and used in the $|y'|^{\rm th}$ iteration. The same effect can be obtained by just re-using the former value (in the $|y'|^{\rm th}$ iteration) rather than re-invoking A in order to obtain it. Part 1 follows.

Towards Part 2, let use first reduce the task of approximating $\#R$ to the task of (exact) uniform generation for R. On input $x\in S_R$, the reduction uses the tree of possible prefixes of elements of $R(x)$ in a somewhat different manner. Again, we proceed in iterations, entering the $i^{\rm th}$ iteration with an $(i-1)$-bit long string y' such that $R'(x;y') \stackrel{\rm def}{=} \{y'' : (x,y'y'')\in R\}$ is not empty. At the $i^{\rm th}$ iteration we estimate the bigger among the two fractions $|R'(x;y'0)|/|R'(x;y')|$ and $|R'(x;y'1)|/|R'(x;y')|$, by uniformly sampling the uniform distribution over $R'(x;y')$. That is, taking ${\rm poly}(|x|/\varepsilon'(|x|))$ uniformly distributed samples in $R'(x;y')$, we obtain with overwhelmingly high probability an approximation of these fractions up to an additive deviation of at most $\varepsilon'(|x|)$. This means that we obtain a relative approximation up-to a factor of $1\pm3\varepsilon'(|x|)$ for the fraction (or fractions) that is (resp., are) bigger than $1/3$. Indeed, we may not be able to obtain such a good relative approximation of the other fraction (in the case that the other fraction is very small), but this does not matter. It also does not matter that we cannot tell which is the bigger fraction among the two; it only matter that we use an approximation that indicates a quantity that is, say, bigger than $1/3$. We proceed to the next iteration by augmenting y' using the bit that corresponds to such a quantity. Specifically, suppose that we obtained the approximations $a_0(y')\approx|R'(x;y'0)|/|R'(x;y')|$ and $a_1(y')\approx|R'(x;y'1)|/|R'(x;y')|$. Then we ex-

[21] Alternatively, we note that since A is a $(1-\varepsilon)$-approximator for $\varepsilon<1$ it must hold that $\#R'(z)=0$ implies $A(z)=0$. Also, since $\varepsilon<1/3$, if $\#R'(z)=1$ then $A(z)\in(2/3,4/3)$, which may be rounded to 1.

tend y' by the bit 1 if $a_1(y') > a_0(y')$ and extend y' by the bit 0 otherwise. Finally, when we reach $y = \sigma_1 \cdots \sigma_{\ell(|x|)}$ such that $(x, y) \in R$, we output

$$a_{\sigma_1}(\lambda)^{-1} \cdot a_{\sigma_2}(\sigma_1)^{-1} \cdots a_{\sigma_{\ell(|x|)}}(\sigma_1\sigma_2 \cdots \sigma_{\ell(|x|)-1})^{-1} \tag{6.13}$$

where for each i it holds that $a_{\sigma_i}(\sigma_1\sigma_2 \cdots \sigma_{i-1})$ is $(1 \pm 3\varepsilon'(|x|)) \cdot \frac{|R'(x;\sigma_1\sigma_2\cdots\sigma_i)|}{|R'(x;\sigma_1\sigma_2\cdots\sigma_{i-1})|}$.

As in Part 1, actions regarding R' (in this case uniform generation in R') are conducted via the parsimonious reduction g to R. That is, whenever we need to sample uniformly in the set $R'(x; y')$, we sample the set $R(g(x; y'))$ and recover the corresponding element of $R'(x; y')$ by using the mapping guaranteed by the hypothesis that g is strongly parsimonious. Finally, note that so far we assumed a uniform generation procedure for R, but using an $(1 - \varepsilon')$-approximate uniform generation merely means that all our approximations deviate by another additive term of ε'. Thus, with overwhelmingly high probability, for each i it holds that $a_{\sigma_i}(\sigma_1\sigma_2 \cdots \sigma_{i-1})$ is $(1 \pm 6\varepsilon'(|x|)) \cdot |R'(x; \sigma_1\sigma_2 \cdots \sigma_i)|/|R'(x; \sigma_1\sigma_2 \cdots \sigma_{i-1})|$. It follows that, on input x, when using an oracle that provides a $(1 - \varepsilon')$-approximate uniform generation for R, with overwhelmingly high probability, the output (as defined in Eq. (6.13)) is in

$$\prod_{i=1}^{\ell(|x|)} \left((1 \pm 6\varepsilon'(|x|))^{-1} \cdot \frac{|R'(x; \sigma_1 \cdots \sigma_{i-1})|}{|R'(x; \sigma_1 \cdots \sigma_i)|} \right) \tag{6.14}$$

where the error probability is due to the unlikely case that in one of the iterations our approximations deviates from the correct value by more than an additive deviation term of $2\varepsilon'(n)$. Noting that Eq. (6.14) equals $(1 \pm 6\varepsilon'(|x|))^{-\ell(|x|)} \cdot |R(x)|$ and using $(1 \pm 6\varepsilon'(|x|))^{-\ell(|x|)} \subset (1 \pm \varepsilon(|x|))$ (which holds for $\varepsilon' = \varepsilon/7\ell$), Part 2 follows.
■

6.2.4.2 A direct procedure for uniform generation

We conclude the current chapter by presenting a direct procedure for solving the uniform generation problem of any $R \in \mathcal{PC}$. This procedure uses an oracle to $\mathcal{NP}$ (or to S_R itself in case it is NP-complete), which is unavoidable because solving the uniform generation problem of R implies solving the corresponding search problem (which in turn implies deciding membership in S_R). One advantage of this procedure, over the reduction presented in §6.2.4.1, is that it solves the uniform generation problem rather than the *approximate* uniform generation problem.

We are going to use hashing again, but this time we use a family of hashing functions having a stronger "uniformity property" (see Appendix D.2.3). Specifically, we will use a family of ℓ-wise independent hashing functions mapping ℓ-bit strings to m-bit strings, where ℓ bounds the length of solutions in R, and rely on the fact that such a family satisfies Lemma D.6. Intuitively, such functions partition $\{0,1\}^\ell$ into 2^m cells and Lemma D.6 asserts that these partitions "uniformly shatter" all sufficiently large sets. That is, for every set $S \subseteq \{0,1\}^\ell$ of size $\Omega(\ell \cdot 2^m)$, the partition induced by almost every function in this family is such that each cell contains approximately $|S|/2^m$ elements of S. In particular, if $|S| = \Theta(\ell \cdot 2^m)$ then each cell contains $\Theta(\ell)$ elements of S. We denote this family of functions by H_ℓ^m, and rely on the fact that its elements have succinct and effective representation (as defined in Appendix D.2.1).

Loosely speaking, the following procedure (for uniform generation) first selects a random hashing function and tests whether it "uniformly shatters" the target set $S = R(x)$. If this condition holds then the procedure selects a cell at random and retrieve all the elements of S residing in the chosen cell. Finally, the procedure either outputs one of the retrieved elements or halts with no output, where each retrieved element is output with a fixed probability p (which is independent of the actual number of elements of S that reside in the chosen cell). This guarantees that each element $e \in S$ is output with the same probability (i.e., $2^{-m} \cdot p$), regardless of the number of elements of S that resides with e in the same cell.

In the following construction, we assume that on input x we also obtain a good approximation to the size of $R(x)$. This assumption can be enforced by using an approximate counting procedure as a preprocessing stage. Alternatively, the ideas presented in the following construction yield such an approximate counting procedure.

Construction 6.32 (uniform generation): *On input x and $m'_x \in \{m_x, m_x + 1\}$, where $m_x \stackrel{\text{def}}{=} \lfloor \log_2 |R(x)| \rfloor$ and $R(x) \subseteq \{0,1\}^\ell$, the oracle machine proceeds as follows.*

1. Selecting a partition that "uniformly shatters" $R(x)$. *The machine sets $m = \max(0, m'_x - \log_2 40\ell)$ and selects uniformly $h \in H_\ell^m$. Such a function defines a partition of $\{0,1\}^\ell$ into 2^m cells*[22]*, and the hope is that each cell contains approximately the same number of elements of $R(x)$. Next, the machine checks that this is indeed the case or rather than no cell contains more that 120ℓ elements of $R(x)$* (i.e., more than twice the expected number)*. This is done by checking whether or not $(x, h, 1^{120\ell+1})$ is in the set $S_{R,H}^{(1)}$ defined as follows*

$$\begin{aligned} S_{R,H}^{(1)} &\stackrel{\text{def}}{=} \{(x', h', 1^t) : \exists v \text{ s.t. } |\{y : (x', y) \in R \wedge h'(y) = v\}| \geq t\} \quad (6.15) \\ &= \{(x', h', 1^t) : \exists v, y_1, ..., y_t \text{ s.t. } \psi^{(1)}(x', h', v, y_1, ..., y_t)\}, \end{aligned}$$

where $\psi^{(1)}(x', h', v, y_1, ..., y_t)$ holds if and only if $y_1 < y_2 \cdots < y_t$ and for every $j \in [t]$ it holds that $(x', y_j) \in R \wedge h'(y_j) = v$. Note that $S_{R,H}^{(1)} \in \mathcal{NP}$.

If the answer is positive (i.e., there exists a cell that contains more that 120ℓ elements of $R(x)$) *then the machine halts with output $\perp$. Otherwise, the machine continues with this choice of h. In this case, no cell contains more that 120ℓ elements of $R(x)$* (i.e., for every $v \in \{0,1\}^m$, it holds that $|\{y : (x, y) \in R \wedge h(y) = v\}| \leq 120\ell$)*. We stress that this is an absolute guarantee that follows from $(x, h, 1^{120\ell+1}) \notin S_{R,H}^{(1)}$.*

[22]For sake of uniformity, we allow also the case of $m = 0$, which is rather artificial. In this case all hashing functions in H_ℓ^0 map $\{0,1\}^\ell$ to the empty string, which is viewed as 0^0, and thus define a trivial partition of $\{0,1\}^\ell$ (i.e., into a single cell).

2. Selecting a cell and determining the number of elements of $R(x)$ that are contained in it. *The machine selects uniformly $v \in \{0,1\}^m$ and determines $s_v \stackrel{\text{def}}{=} |\{y : (x,y) \in R \wedge h(y) = v\}|$ by making queries to the following NP-set*

$$S_{R,H}^{(2)} \stackrel{\text{def}}{=} \{(x', h', v', 1^t) : \exists y_1, ..., y_t \text{ s.t. } \psi^{(1)}(x', h', v', y_1, ..., y_t)\}. \tag{6.16}$$

Specifically, for $i = 1, ..., 120\ell$, it checks whether $(x, h, v, 1^i)$ is in $S_{R,H}^{(2)}$, and sets s_v to be the largest value of i for which the answer is positive.

3. Obtaining all the elements of $R(x)$ that are contained in the selected cell, and outputting one of them at random. *Using s_v, the procedure reconstructs the set $S_v \stackrel{\text{def}}{=} \{y : (x,y) \in R \wedge h(y) = v\}$, by making queries to the following NP-set*

$$S_{R,H}^{(3)} \stackrel{\text{def}}{=} \{(x', h', v', 1^t, j) : \exists y_1, ..., y_t \text{ s.t. } \psi^{(3)}(x', h', v', y_1, ..., y_t, j)\}, \tag{6.17}$$

where $\psi^{(3)}(x', h', v', y_1, ..., y_t, j)$ holds if and only if $\psi^{(1)}(x', h', v', y_1, ..., y_t)$ holds and the j^{th} bit of $y_1 \cdots y_t$ equals 1. Specifically, for $j_1 = 1, ..., s_v$ and $j_2 = 1, ..., \ell$, we make the query $(x, h, v, 1^{s_v}, (j_1 - 1) \cdot \ell + j_2)$ in order to determine the j_2^{th} bit of y_{j_1}. Finally, having recovered S_v, the procedure outputs each $y \in S_v$ with probability $1/120\ell$, and outputs $\perp$ otherwise (i.e., with probability $1 - (s_v/120\ell)$).

Recall that for $|R(x)| = \Omega(\ell)$ and $m = m'_x - \log_2 40\ell$, Lemma D.6 implies that, with overwhelmingly high probability (over the choice of $h \in H_\ell^m$), each set $\{y : (x,y) \in R \wedge h(y) = v\}$ has cardinality $(1 \pm 0.5)|R(x)|/2^m$. Thus, ignoring the case of $|R(x)| = O(\ell)$, Step 1 can be easily adapted to yield an approximate counting procedure for $\#R$; see Exercise 6.38, which also handles the case of $|R(x)| = O(\ell)$ by using ideas as in Step 2. However, our aim is to establish the following result.

Proposition 6.33 *Construction 6.32 solves the uniform generation problem of R.*

Proof: Intuitively, by Lemma D.6 (and the setting of m), with overwhelmingly high probability, a uniformly selected $h \in H_\ell^m$ partitions $R(x)$ into 2^m cells, each containing at most 120ℓ elements. Following is the tedious proof of this fact. Since $m = \max(0, m'_x - \log_2 40\ell)$, we may focus on the case that $m'_x > \log_2 40\ell$ (as in the other case $|R(x)| \leq 2^{m'_x+1} \leq 80\ell$). In this case, by Lemma D.6 (using $\varepsilon = 0.5$ and $m = m'_x - \log_2 40\ell \leq \log_2 |R(x)| - \log_2 20\ell$ (which implies $m \leq \log_2 |R(x)| - \log_2(5\ell/\varepsilon^2)$)), with overwhelmingly high probability, each set $\{y : (x,y) \in R \wedge h(y) = v\}$ has cardinality $(1 \pm 0.5)|R(x)|/2^m$. Using $m'_x > (\log_2 |R(x)|) - 1$ (and $m = m'_x - \log_2 40\ell$), it follows that $|R(x)|/2^m < 80\ell$ and hence each cell contains at most 120ℓ elements of $R(x)$. We also note that, using $m'_x \leq (\log_2 |R(x)|) + 1$, it follows that $|R(x)|/2^m \geq 20\ell$ and hence each cell contains at least 10ℓ elements of $R(x)$.

The key observation, stated in Step 1, is that if the procedure does not halt in Step 1 then it is indeed the case that h induces a partition in which each cell contains at most 120ℓ elements of $R(x)$. The fact that these cells may contain a different number of elements is immaterial, because each element is output with the same probability (i.e., $1/120\ell$). What matters is that the average number of elements in the various cells is sufficiently large, because this average number determines the probability that the procedure outputs an element of $R(x)$ (rather than $\perp$). Specifically, conditioned on not halting in Step 1, the probability that Step 3 outputs some element of $R(x)$ equals the average number of elements per cell (i.e., $|R(x)|/2^m$) divided by 120ℓ. Recalling that for $m > 0$ (resp., $m = 0$) it holds that $|R(x)|/2^m \geq 20\ell$ (resp., $|R(x)| \geq 1$), we conclude that in this case some element of $R(x)$ is output with probability at least $1/6$ (resp., $|R(x)|/120\ell$). Recalling that Step 1 halts with negligible probability, it follows that the procedure outputs some element of $R(x)$ with probability at least $0.99 \cdot \min((|R(x)|/120\ell), (1/6))$. ■

Comments. We can easily improve the performance of Construction 6.32 by dealing separately with the case $m = 0$. In such a case, Step 3 can be simplified and improved by uniformly selecting and outputting an element of S_λ (which equals $R(x)$). Under this modification, the procedure outputs some element of $R(x)$ with probability at least $1/6$. In any case, recall that the probability that a uniform generation procedure outputs $\perp$ can be deceased by repeated invocations.

Digest. Construction 6.32 is the culmination of the "hashing paradigm" that is aimed at allowing various manipulations of arbitrary sets. In particular, as seen in Construction 6.32, hashing can be used in order to partition a large set into an adequate number of small subsets that are of approximately the same size. We stress that hashing is performed by randomly selecting a function in an adequate family. Indeed, the use of randomization for such purposes (i.e., allowing manipulation of large sets) seems indispensable.

Chapter Notes

One key aspect of randomized procedures is their success probability, which is obviously a quantitative notion. This aspect provides a clear connection between probabilistic polynomial-time algorithms considered in Section 6.1 and the counting problems considered in Section 6.2 (see also Exercise 6.20). More appealing connections between randomized procedures and counting problems (e.g., the application of randomization in approximate counting) are presented in Section 6.2. These connections justify the presentation of these two topics in the same chapter.

Randomized algorithms

Making people take an unconventional step requires compelling reasons, and indeed the study of randomized algorithms was motivated by a few compelling examples. Ironically, the appeal of the two most famous examples (discussed next) has been somewhat diminished due to subsequent finding, but the fundamental questions that emerged remain fascinating regardless of the status of these two examples.

These questions refer to the power of randomization in various computational settings, and in particular in the context of decision and search problems. We shall return to these questions after briefly reviewing the story of the aforementioned examples.

The first example: primality testing. For more than two decades, primality testing was the archetypical example of the usefulness of randomization in the context of efficient algorithms. The celebrated algorithms of Solovay and Strassen [206] and of Rabin [179], proposed in the late 1970's, established that deciding primality is in co$\mathcal{RP}$ (i.e., these tests always recognize correctly prime numbers, but they may err on composite inputs). (The approach of Construction 6.4, which only establishes that deciding primality is in $\mathcal{BPP}$, is commonly attributed to M. Blum.) In the late 1980's, Adleman and Huang [2] proved that deciding primality is in $\mathcal{RP}$ (and thus in $\mathcal{ZPP}$). In the early 2000's, Agrawal, Kayal, and Saxena [3] showed that deciding primality is actually in $\mathcal{P}$. One should note, however, that strong evidence to the fact that deciding primality is in $\mathcal{P}$ was actually available from the start: we refer to Miller's deterministic algorithm [161], which relies on the Extended Riemann Hypothesis.

The second example: undirected connectivity. Another celebrated example to the power of randomization, specifically in the context of log-space computations, was provided by testing undirected connectivity. The random-walk algorithm presented in Construction 6.12 is due to Aleliunas, Karp, Lipton, Lovász, and Rackoff [5]. Recall that a deterministic log-space algorithm was found twenty-five years later (see Section 5.2.4 or [185]).

Another famous example: polynomial identity testing. A third famous example, which dates back to about the same period, is the polynomial identity tester of [62, 194, 235]. This tester, presented in §6.1.3.1, has found many applications in complexity theory (some are implicit in subsequent chapters). Needless to say, in the abstract setting of Construction 6.7, randomization is indispensable. Interestingly, the computational version mentioned in Exercise 6.17 has so far resisted de-randomization attempts (cf. [130]).

Other randomized algorithms. In addition to the three foregoing examples, several other appealing randomized algorithms are known. Confining ourselves to the context of search and decision problems, we mention the algorithms for finding perfect matchings and minimum cuts in graphs (see, e.g., [87, Apdx. B.1] or [163, Sec. 12.4&10.2]), and note the prominent role of randomization in computational number theory (see, e.g., [22] or [163, Chap. 14]). We mention that randomized algorithms are more abundant in the context of approximation problems (let alone in other computational settings (cf., e.g., Chapter 9, Appendix C, and Appendix D.3). For a general textbook on randomized algorithms, we refer the interested reader to [163].

While it can be shown that randomization is essential in several important computational settings (cf., e.g., Chapter 9, Section 10.1.2, Appendix C, and Appendix D.3), a fundamental question is whether randomization is essential in the context of search and decision problems. The prevailing conjecture is that randomization is of *limited help* in the context of time-bounded and space-bounded algorithms. For example, it is conjectured that $\mathcal{BPP} = \mathcal{P}$ and $\mathcal{BPL} = \mathcal{L}$. Note that such conjectures do not rule out the possibility that randomization is helpful also in these contexts, they merely says that this help is limited. For example, it may be the case that any quadratic-time randomized algorithm can be emulated by a cubic-time deterministic algorithm, but not by a quadratic-time deterministic algorithm.

On the study of $\mathcal{BPP}$. The conjecture $\mathcal{BPP} = \mathcal{P}$ is referred to as a full derandomization of $\mathcal{BPP}$, and can be shown to hold under some reasonable intractability assumptions. This result (and related ones) will be presented in Section 8.3. In the current chapter, we only presented uncoditional results regarding $\mathcal{BPP}$ like $\mathcal{BPP} \subset \mathcal{P}/\text{poly}$ and $\mathcal{BPP} \subseteq \mathcal{PH}$. Our presentation of Theorem 6.9 follows the proof idea of Lautemann [146]. A different proof technique, which yields a weaker result but found more applications (see, e.g., Theorems 6.27 and F.2), was presented (independently) by Sipser [202].

On the role of promise problems. In addition to their use in the formulation of Theorem 6.9, promise problems allow for establishing complete problems and hierarchy theorems for randomized computation (see Exercises 6.14 and 6.15, respectively). We mention that such results are not known for the corresponding classes of standard decision problems. The technical difficulty is that we do not know how to enumerate and/or recognize probabilistic machines that utilize a non-trivial probabilistic decision rule.

On the feasibility of randomized computation. Different perspectives on this question are offered by Chapter 8 and Appendix D.4. Specifically, as advocated in Chapter 8, generating uniformly distributed bit sequences is not really necessary for implementing randomized algorithms; it suffices to generate sequences that look (to their user) as if they are uniformly distributed. In many cases this leads to reducing the number of coin tosses in such implementations, and at times even to a full (efficient) derandomization (see Sections 8.3 and 8.4). A less radical approach is presented in Appendix D.4, which deals with the task of extracting almost uniformly distributed bit sequences from sources of weak randomness. Needless to say, these two approaches are complimentary and can be combined.

Counting problems

The counting class $\#\mathcal{P}$ was introduced by Valiant [223], who proved that computing the permanent of 0/1-matrices is $\#\mathcal{P}$-complete (i.e., Theorem 6.20). Interestingly, like in the case of Cook's introduction of NP-completeness [55], Valiant's motivation was determining the complexity of a specific problem (i.e., the permanent).

Our presentation of Theorem 6.20 is based both on Valiant's paper [223] and on subsequent studies (most notably [29]). Specifically, the high-level structure of the reduction presented in Proposition 6.21 as well as the "structured" design of the clause gadget is taken from [223], whereas the Deus Ex Machina gadget presented in Figure 6.3 is based on [29]. The proof of Proposition 6.22 is also based on [29] (with some variants). Turning back to the design of clause gadgets we regret not being able to cite and/or use a systematic study of this design problem.

As noted in the main text, we decided not to present a proof of Toda's Theorem [215], which asserts that every set in $\mathcal{PH}$ is Cook-reducible to $\#\mathcal{P}$ (i.e., Theorem 6.16). Appendix F.1 contains a proof of a related result, which implies that $\mathcal{PH}$ is reducible to $\#\mathcal{P}$ via probabilistic polynomial-time reductions. Alternative proofs can be found in [132, 207, 215].

Approximate counting and related problems. The approximation procedure for $\#\mathcal{P}$ is due to Stockmeyer [209], following an idea of Sipser [202]. Our exposition, however, follows further developments in the area. The randomized reduction of $\mathcal{NP}$ to problems of unique solutions was discovered by Valiant and Vazirani [225]. Again, our exposition is a bit different.

The connection between approximate counting and uniform generation (presented in §6.2.4.1) was discovered by Jerrum, Valiant, and Vazirani [129], and turned out to be very useful in the design of algorithms (e.g., in the "Markov Chain approach" (see [163, Sec. 11.3.1])). The direct procedure for uniform generation (presented in §6.2.4.2) is taken from [26].

In continuation to §6.2.2.1, which is based on [135], we refer the interested reader to [128], which presents a probabilistic polynomial-time algorithm for approximating the permanent of non-negative matrices. This fascinating algorithm is based on the fact that knowing (approximately) certain parameters of a non-negative matrix M allows to approximate the same parameters for a matrix M', provided that M and M' are sufficiently similar. Specifically, M and M' may differ only on a single entry, and the ratio of the corresponding values must be sufficiently close to one. Needless to say, the actual observation (is not generic but rather) refers to specific parameters of the matrix, which include its permanent. Thus, given a matrix M for which we need to approximate the permanent, we consider a sequence of matrices $M_0, ..., M_t \approx M$ such that M_0 is the all 1's matrix (for which it is easy to evaluate the said parameters), and each M_{i+1} is obtained from M_i by reducing some adequate entry by a factor sufficiently close to one. This process of (polynomially many) gradual changes, allows to transform the dummy matrix M_0 into a matrix M_t that is very close to M (and hence has a permanent that is very close to the permanent of M). Thus, approximately obtaining the parameters of M_t allows to approximate the permanent of M.

Finally, we mention that Section 10.1.1 provides a treatment of a different type of approximation problems. Specifically, when given an instance x (for a search problem R), rather than seeking an approximation of the number of solutions (i.e.,

$\#R(x)$), one seeks an approximation of the value of the best solution (i.e., best $y \in R(x)$), where the value of a solution is defined by an auxiliary function.

Exercises

Exercise 6.1 Show that if a search (resp., decision) problem can be solved by a probabilistic polynomial-time algorithm having zero failure probability, then the problem can be solve by a deterministic polynomial-time algorithm.
(Hint: replace the internal coin tosses by a fixed outcome that is easy to generate deterministically (e.g., the all-zero sequence).)

Exercise 6.2 (randomized reductions) In continuation to the definitions presented in Section 6.1.1, prove the following:

1. If a problem Π is probabilistic polynomial-time reducible to a problem that is solvable in probabilistic polynomial-time then Π is solvable in probabilistic polynomial-time, where by solving we mean solving correctly except with negligible probability.

   Warning: Recall that in the case that Π' is a search problem, we required that on input x the solver provides a correct solution with probability at least $1 - \mu(|x|)$, but we did not require that it always returns the same solution.

   (Hint: without loss of generality, the reduction does not make the same query twice.)

2. Prove that probabilistic polynomial-time reductions are transitive.

3. Prove that randomized Karp-reductions are transitive and that they yield a special case of probabilistic polynomial-time reductions.

Define one-sided error and zero-sided error randomized (Karp- and Cook-) reductions, and consider the foregoing items when applied to them. Note that the implications for the case of one-sided error are somewhat subtle.

Exercise 6.3 (on the definition of probabilistically solving a search problem) In continuation to the discussion at the beginning of Section 6.1.2, suppose that for some probabilistic polynomial-time algorithm A and a positive polynomial p the following holds: for every $x \in S_R \stackrel{\text{def}}{=} \{z : R(z) \neq \emptyset\}$ there exists $y \in R(x)$ such that $\Pr[A(x) = y] > 0.5 + (1/p(|x|))$, whereas for every $x \notin S_R$ it holds that $\Pr[A(x) = \perp] > 0.5 + (1/p(|x|))$.

1. Show that there exists a probabilistic polynomial-time algorithm that solves the search problem of R with negligible error probability.

   (Hint: See Exercise 6.4 for a related procedure.)

2. Reflect on the need to require that one (correct) solution occurs with probability greater than $0.5 + (1/p(|x|))$. Specifically, what can we do if it is only guaranteed that for every $x \in S_R$ it holds that $\Pr[A(x) \in R(x)] > 0.5 + (1/p(|x|))$ (and for every $x \notin S_R$ it holds that $\Pr[A(x) = \perp] > 0.5 + (1/p(|x|))$)?

Note that R is not necessarily in $\mathcal{PC}$. Indeed, in the case that $R \in \mathcal{PC}$ we can eliminate the error probability for every $x \notin S_R$, and perform error-reduction for $x \in S_R$ as in the case of $\mathcal{RP}$.

Exercise 6.4 (error-reduction for $\mathcal{BPP}$) For $\varepsilon : \mathbb{N} \to [0,1]$, let $\mathcal{BPP}_\varepsilon$ denote the class of decision problems that can be solved in probabilistic polynomial-time with error probability upper-bounded by ε. Prove the following two claims:

1. For every positive polynomial p and $\varepsilon(n) = (1/2) - (1/p(n))$, the class $\mathcal{BPP}_\varepsilon$ equals $\mathcal{BPP}$.

2. For every positive polynomial p and $\varepsilon(n) = 2^{-p(n)}$, the class $\mathcal{BPP}$ equals $\mathcal{BPP}_\varepsilon$.

Formulate a corresponding version for the setting of search problem. Specifically, for every input that has a solution, consider the probability that a specific solution is output.

Guideline: Given an algorithm A for the syntactically weaker class, consider an algorithm A' that on input x invokes A on x for $t(|x|)$ times, and rules by majority. For Part 1 set $t(n) = O(p(n)^2)$ and apply Chebyshev's Inequality. For Part 2 set $t(n) = O(p(n))$ and apply the Chernoff Bound.

Exercise 6.5 (error-reduction for $\mathcal{RP}$) For $\rho : \mathbb{N} \to [0,1]$, we define the class of decision problem $\mathcal{RP}_\rho$ such that it contains S if there exists a probabilistic polynomial-time algorithm A such that for every $x \in S$ it holds that $\mathsf{Pr}[A(x) = 1] \geq \rho(|x|)$ and for every $x \notin S$ it holds that $\mathsf{Pr}[A(x) = 0] = 1$. Prove the following two claims:

1. For every positive polynomial p, the class $\mathcal{RP}_{1/p}$ equals $\mathcal{RP}$.

2. For every positive polynomial p, the class $\mathcal{RP}$ equals $\mathcal{RP}_\rho$, where $\rho(n) = 1 - 2^{-p(n)}$.

(Hint: The one-sided error allows using an "or-rule" (rather than a "majority-rule") for the decision.)

Exercise 6.6 (error-reduction for $\mathcal{ZPP}$) For $\rho : \mathbb{N} \to [0,1]$, we define the class of decision problem $\mathcal{ZPP}_\rho$ such that it contains S if there exists a probabilistic polynomial-time algorithm A such that for every x it holds that $\mathsf{Pr}[A(x) = \chi_S(x)] \geq \rho(|x|)$ and $\mathsf{Pr}[A(x) \in \{\chi_S(x), \perp\}] = 1$, where $\chi_S(x) = 1$ if $x \in S$ and $\chi_S(x) = 0$ otherwise. Prove the following two claims:

1. For every positive polynomial p, the class $\mathcal{ZPP}_{1/p}$ equals $\mathcal{ZPP}$.

2. For every positive polynomial p, the class $\mathcal{ZPP}$ equals $\mathcal{ZPP}_\rho$, where $\rho(n) = 1 - 2^{-p(n)}$.

Exercise 6.7 (an alternative definition of $\mathcal{ZPP}$) We say that the decision problem S is solvable in expected probabilistic polynomial-time if there exists a randomized algorithm A and a polynomial p such that for every $x \in \{0,1\}^*$ it holds that $\Pr[A(x) = \chi_S(x)] = 1$ and the expected number of steps taken by $A(x)$ is at most $p(|x|)$. Prove that $S \in \mathcal{ZPP}$ if and only if S is solvable in expected probabilistic polynomial-time.

Guideline: Repeatedly invoking a ZPP algorithm until it yields an output other than $\perp$ yields an expected probabilistic polynomial-time solver. On the other hand, truncating runs of an expected probabilistic polynomial-time algorithm once they exceed twice the expected number of steps (and outputting $\perp$ on such runs), we obtain a ZPP algorithm.

Exercise 6.8 Prove that for every $S \in \mathcal{NP}$ there exists a probabilistic polynomial-time algorithm A such that for every $x \in S$ it holds that $\Pr[A(x) = 1] > 0$ and for every $x \notin S$ it holds that $\Pr[A(x) = 0] = 1$. That is, A has error probability at most $1 - \exp(-\text{poly}(|x|))$ on yes-instances but never errs on no-instances. Thus, $\mathcal{NP}$ may be fictitiously viewed as having a huge one-sided error probability.

Exercise 6.9 Let $\mathcal{BPP}$ and co$\mathcal{RP}$ be classes of promise problems (as in Theorem 6.9).

1. Prove that every problem in $\mathcal{BPP}$ is reducible to the set $\{1\} \in \mathcal{P}$ by a *two-sided error* randomized Karp-reduction.
2. Prove that if a set S is Karp-reducible to $\mathcal{RP}$ (resp., co$\mathcal{RP}$) via a deterministic reduction then $S \in \mathcal{RP}$ (resp., $S \in$ co$\mathcal{RP}$).

Exercise 6.10 (randomness-efficient error-reductions) Note that standard error-reduction (as in Exercise 6.4) yields error probability δ at the cost of increasing the randomness complexity by a *factor* of $O(\log(1/\delta))$. Using the randomness-efficient error-reductions outlined in §D.4.1.3, show that error probability δ can be obtained at the cost of increasing the randomness complexity from r to $O(r) + 1.5\log_2(1/\delta)$. Note that this allows satisfying the hypothesis made in the illustrative paragraph of the proof of Theorem 6.9.

Exercise 6.11 In continuation to the illustrative paragraph in the proof of Theorem 6.9, consider the promise problem $\Pi' = (\Pi'_{\text{yes}}, \Pi'_{\text{no}})$ such that $\Pi'_{\text{yes}} = \{(x, r') : |r'| = p'(|x|) \wedge (\forall r'' \in \{0,1\}^{|r'|})\, A'(x, r'r'') = 1\}$ and $\Pi'_{\text{no}} = \{(x, r') : x \notin S\}$. Recall that for every x it holds that $\Pr_{r \in \{0,1\}^{2p'(|x|)}}[A'(x,r) \neq \chi_S(x)] < 2^{-(p'(|x|)+1)}$.

1. Show that mapping x to (x, r'), where r' is uniformly distributed in $\{0,1\}^{p'(|x|)}$, constitutes a one-sided error randomized Karp-reduction of S to Π'.
2. Show that Π' is in the promise problem class co$\mathcal{RP}$.

Exercise 6.12 (randomized versions of $\mathcal{NP}$) In continuation to Footnote 7, consider the following two variants of $\mathcal{MA}$ (which we consider the main randomized version of $\mathcal{NP}$).

1. $S \in \mathcal{MA}^{(1)}$ if there exists a probabilistic polynomial-time algorithm V such that for every $x \in S$ there exists $y \in \{0,1\}^{\mathrm{poly}(|x|)}$ such that $\mathsf{Pr}[V(x,y)=1] \geq 1/2$, whereas for every $x \notin S$ and every y it holds that $\mathsf{Pr}[V(x,y)=0] = 1$.

2. $S \in \mathcal{MA}^{(2)}$ if there exists a probabilistic polynomial-time algorithm V such that for every $x \in S$ there exists $y \in \{0,1\}^{\mathrm{poly}(|x|)}$ such that $\mathsf{Pr}[V(x,y)=1] \geq 2/3$, whereas for every $x \notin S$ and every y it holds that $\mathsf{Pr}[V(x,y)=0] \geq 2/3$.

Prove that $\mathcal{MA}^{(1)} = \mathcal{NP}$ whereas $\mathcal{MA}^{(2)} = \mathcal{MA}$.

Guideline: For the first part, note that a sequence of internal coin tosses that makes V accept (x,y) can be incorporated into y itself (yielding a standard NP-witness). For the second part, apply the ideas underlying the proof of Theorem 6.9, and note that an adequate sequence of shifts (to be used by the verifier) can be incorporated in the single message sent by the prover.

Exercise 6.13 ($\mathcal{BPP} \subseteq \mathcal{ZPP}^{\mathcal{NP}}$) In continuation to the proof of Theorem 6.9, present a zero-error randomized reduction of $\mathcal{BPP}$ to $\mathcal{NP}$, where all classes are the standard classes of decision problems.

Guideline: On input x, the ZPP-machine uniformly selects $\overline{s} = (s_1, ..., s_m)$, and for each $\sigma \in \{0,1\}$ makes the query $(x, \sigma, \overline{s})$, which is answered positively by the (coNP) oracle if for every r it holds that $\vee_i (A(x, r \oplus s_i) = \sigma)$. The machine outputs σ if and only if the query $(x, \sigma, \overline{s})$ was answered positively, and outputs $\perp$ otherwise (i.e., both queries were answered negatively).

Exercise 6.14 (completeness for promise problem versions of $\mathcal{BPP}$) Referring to the promise problem version of $\mathcal{BPP}$, present a promise problem that is complete for this class under (deterministic log-space) Karp-reductions.

Guideline: The promise problem consists of yes-instances that are Boolean circuits that accept at least a 2/3 fraction of their possible inputs and no-instances that are Boolean circuits that reject at least a 2/3 fraction of their possible inputs. The reduction is essentially the one provided in the proof of Theorem 2.21, and the promise is used in an essential way in order to provide a BPP-algorithm.

Exercise 6.15 (hierarchy theorems for promise problem versions of BPTIME**)** Fixing a model of computation, let BPTIME(t) denote the class of promise problems that are solvable by a randomized algorithm of time-complexity t that has a two-sided error probability at most 1/3. (The standard definition refers only to decision problems.) Formulate and prove results analogous to Theorem 4.3 and Corollary 4.4.

Guideline (by Dieter van Melkebeek): Apply the "delayed diagonalization" method used to prove Theorem 4.6 rather than the simple diagonalization used in Theorem 4.3. Analogously to the proof of Theorem 4.6, for every $\sigma \in \{0,1\}$, define $A_M(x) = \sigma$ if $\mathsf{Pr}[M'(x) = \sigma] \geq 2/3$ and define $A_M(x) = \perp$ otherwise (i.e., if $1/3 < \mathsf{Pr}[M'(x) = 1] < 2/3$), where $M'(x)$ denotes the computation of $M(x)$ truncated after $t_1(|x|)$ steps. For $x \in [\alpha_M, \beta_M - 1]$, define $f(x) = A_M(x+1)$, where $f(x) = \perp$ means that x violates the promise.

Define $f(\beta_M) = 1$ if $A_M(\alpha_M) = 0$ and $f(\beta_M) = 0$ otherwise (i.e., if $A_M(\alpha_M) \in \{1, \perp\}$). Note that $f(x)$ is computable in randomized time $\widetilde{O}(t_1(|x| + 1))$ by emulating a single computation of $M'(x)$ if $x \in [\alpha_M, \beta_M - 1]$ and emulating all computations of $M'(\alpha_M)$ if $x = \beta_M$. Prove that the promise problem f cannot be solved in randomized time t_1, by noting that β_M satisfies the promise and that for every $x \in [\alpha_M + 1, \beta_M]$ that satisfies the promise (i.e., $f(x) \in \{0, 1\}$) it holds that if $A_M(x) = f(x)$ then $f(x-1) = A_M(x) \in \{0, 1\}$.

Exercise 6.16 (extracting square roots modulo a prime) Using the following guidelines, present a probabilistic polynomial-time algorithm that, on input a prime P and a quadratic residue $s \pmod{P}$, returns r such that $r^2 \equiv s \pmod{P}$.

1. Prove that if $P \equiv 3 \pmod{4}$ then $s^{(P+1)/4} \bmod P$ is a square root of the quadratic residue $s \pmod{P}$.

2. Note that the procedure suggested in Item 1 relies on the ability to find an *odd* integer e such that $s^e \equiv 1 \pmod{P}$. Indeed, once such e is found, we may output $s^{(e+1)/2} \bmod P$. (In Item 1, we used $e = (P-1)/2$, which is odd since $P \equiv 3 \pmod{4}$.)

   Show that it suffices to find an *odd* integer e together with a residue t and an *even* integer e' such that $s^e t^{e'} \equiv 1 \pmod{P}$, because $s \equiv s^{e+1} t^{e'} \equiv (s^{(e+1)/2} t^{e'/2})^2$.

3. Given a prime $P \equiv 1 \pmod{4}$, a quadratic residue s, and any quadratic non-residue t (i.e., residue t such that $t^{(P-1)/2} \equiv -1 \pmod{P}$), show that e and e' as in Item 2 can be efficiently found.[23]

4. Prove that, for a prime P, with probability $1/2$ a uniformly chosen $t \in \{1, ..., P\}$ satisfies $t^{(P-1)/2} \equiv -1 \pmod{P}$.

Note that randomization is used only in the last item, which in turn is used only for $P \equiv 1 \pmod{4}$.

Exercise 6.17 Referring to the definition of arithmetic circuits (cf. Appendix B.3), show that the following decision problem is in $\text{co}\mathcal{RP}$: *Given a pair of circuits (C_1, C_2) of depth d over a field that has more than 2^{d+1} elements, determine whether the circuits compute the same polynomial.*

Guideline: Note that each of these circuits computes a polynomial of degree at most 2^d.

Exercise 6.18 (small-space randomized step-counter) As defined in Exercise 4.5, a step-counter is an algorithm that halts after issuing a number of "signals" as specified in its input, where these signals are defined as entering (and leaving)

[23] Write $(P-1)/2 = (2j_0 + 1) \cdot 2^{i_0}$, and note that $s^{(2j_0+1)\cdot 2^{i_0}} \equiv 1 \pmod{P}$, which may be written as $s^{(2j_0+1)\cdot 2^{i_0}} t^{(2j_0+1)\cdot 2^{i_0+1}} \equiv 1 \pmod{P}$. Given that for some $i' > i > 0$ and j' it holds that $s^{(2j_0+1)\cdot 2^{i}} t^{(2j'+1)\cdot 2^{i'}} \equiv 1 \pmod{P}$, show how to find $i'' > i-1$ and j'' such that $s^{(2j_0+1)\cdot 2^{i-1}} t^{(2j''+1)\cdot 2^{i''}} \equiv 1 \pmod{P}$. (Extra hint: $s^{(2j_0+1)\cdot 2^{i-1}} t^{(2j'+1)\cdot 2^{i'-1}} \equiv \pm 1 \pmod{P}$ and $t^{(2j_0+1)\cdot 2^{i_0}} \equiv -1 \pmod{P}$.) Applying this reasoning for i_0 times, we get what we need.

a designated state (of the algorithm). Recall that a step-counter may be run in parallel to another procedure in order to suspend the execution after a predetermined number of steps (of the other procedure) has elapsed. Note that there exists a simple deterministic machine that, on input n, halts after issuing n signals while using $O(1)+\log_2 n$ space (and $\widetilde{O}(n)$ time). The goal of this exercise is presenting a (randomized) step-counter that allows for many more signals while using the same amount of space. Specifically, present a (randomized) algorithm that, on input n, uses $O(1)+\log_2 n$ space (and $\widetilde{O}(2^n)$ time) and halts after issuing an expected number of 2^n signals. Furthermore, prove that, with probability at least $1-2^{-k+1}$, this step-counter halts after issuing a number of signals that is between 2^{n-k} and 2^{n+k}.

Guideline: Repeat the following experiment till reaching success. Each trial consists of uniformly selecting n bits (i.e., tossing n unbiased coins), and is deemed successful if all bits turn out to equal the value 1 (i.e., all outcomes equal HEAD). Note that such a trial can be implemented by using space $O(1)+\log_2 n$ (mainly for implementing a standard counter for determining the number of bits). Thus, each trial is successful with probability 2^{-n}, and the expected number of trials is 2^n.

Exercise 6.19 (analysis of random walks on arbitrary undirected graphs) In order to complete the proof of Proposition 6.13, prove that if $\{u,v\}$ is an edge of the graph $G=(V,E)$ then $\mathrm{E}[X_{u,v}] \leq 2|E|$. Recall that, for a fixed graph, $X_{u,v}$ is a random variable representing the number of steps taken in a random walk that starts at the vertex u until the vertex v is first encountered.

Guideline: Let $Z_{u,v}(n)$ be a random variable counting the number of *minimal* paths from u to v that appear along a random walk of length n, where the walk starts at the stationary vertex distribution (which is well-defined assuming the graph is not bipartite, which in turn may be enforced by adding a self-loop). On one hand, $\mathsf{E}[X_{u,v}+X_{v,u}] = \lim_{n\to\infty}(n/\mathsf{E}[Z_{u,v}(n)])$, due to the memoryless property of the walk. On the other hand, letting $\chi_{v,u}(i) \stackrel{\rm def}{=} 1$ if the edge $\{u,v\}$ was traversed from v to u in the $i^{\rm th}$ step of such a random walk and $\chi_{v,u}(i) \stackrel{\rm def}{=} 0$ otherwise, we have $\sum_{i=1}^{n}\chi_{v,u}(i) \leq Z_{u,v}(n)+1$ and $\mathrm{E}[\chi_{v,u}(i)] = 1/2|E|$ (because, in each step, each directed edge appears on the walk with equal probability). It follows that $\mathrm{E}[X_{u,v}] < 2|E|$.

Exercise 6.20 (the class $\mathcal{PP} \supseteq \mathcal{BPP}$ and its relation to $\#\mathcal{P}$) In contrast to $\mathcal{BPP}$, which refers to useful probabilistic polynomial-time algorithms, the class $\mathcal{PP}$ does not capture such algorithms but is rather closely related to $\#\mathcal{P}$. A decision problem S is in $\mathcal{PP}$ if there exists a probabilistic polynomial-time algorithm A such that, for every x, it holds that $x \in S$ if and only if $\mathsf{Pr}[A(x)=1] > 1/2$. Note that $\mathcal{BPP} \subseteq \mathcal{PP}$. Prove that $\mathcal{PP}$ is Cook-reducible to $\#\mathcal{P}$ and vise versa.

Guideline: For $S \in \mathcal{PP}$ (by virtue of the algorithm A), consider the relation R such that $(x,r) \in R$ if and only if A accepts the input x when using the random-input $r \in \{0,1\}^{p(|x|)}$, where p is a suitable polynomial. Thus, $x \in S$ if and only if $|R(x)| > 2^{p(|x|)-1}$, which in turn can de determined by querying the counting function of R. To reduce $f \in \#\mathcal{P}$ to $\mathcal{PP}$, consider the relation $R \in \mathcal{PC}$ that is counted by f (i.e., $f(x) = |R(x)|$) and the decision problem S_f as defined in Proposition 6.15. Let p be the polynomial specifying the length of solutions for R (i.e., $(x,y) \in R$ implies $|y| = p(|x|)$), and consider the following algorithm A': On input (x,N), with probability $1/2$, algorithm A' uniformly selects $y \in \{0,1\}^{p(|x|)}$ and accepts if and only if $(x,y) \in R$, and otherwise (i.e., with the remaining probability of $1/2$) algorithm A' accepts with probability exactly $\frac{2^{p(|x|)}-N+0.5}{2^{p(|x|)}}$. Prove that $(x,N) \in S_f$ if and only if $\Pr[A'(x)=1] > 1/2$.

Exercise 6.21 (enumeration problems) For any binary relation R, define the enumeration problem of R as a function $f_R : \{0,1\}^* \times \mathbb{N} \to \{0,1\}^* \cup \{\perp\}$ such that $f_R(x,i)$ equals the i^{th} element in $|R(x)|$ if $|R(x)| \geq i$ and $f_R(x,i) = \perp$ otherwise. The above definition refers to the standard lexicographic order on strings, but any other efficient order of strings will do.[24]

1. Prove that, for any polynomially bounded R, computing $\#R$ is reducible to computing f_R.
2. Prove that, for any $R \in \mathcal{PC}$, computing f_R is reducible to some problem in $\#\mathcal{P}$.

   Guideline: Consider the binary relation $R' = \{(\langle x,b\rangle, y) : (x,y) \in R \wedge y \leq b\}$, and show that f_R is reducible to $\#R'$. (Extra hint: Note that $f_R(x,i) = y$ if and only if $|R'(\langle x,y\rangle)| = i$ and for every $y' < y$ it holds that $|R'(\langle x,y'\rangle)| < i$.)

Exercise 6.22 (artificial $\#\mathcal{P}$-complete problems) Show that there exists a relation $R \in \mathcal{PC}$ such that $\#R$ is $\#\mathcal{P}$-complete and $S_R = \{0,1\}^*$. Furthermore, prove that for every $R' \in \mathcal{PC}$ there exists $R \in \mathcal{PF} \cap \mathcal{PC}$ such that for every x it holds that $\#R(x) = \#R'(x) + 1$. Note that Theorem 6.19 follows by starting with any relation $R' \in \mathcal{PC}$ such that $\#R'$ is $\#\mathcal{P}$-complete.

Exercise 6.23 (computing the permanent of integer matrices) Prove that computing the permanent of matrices with 0/1-entries is computationally equivalent to computing the number of perfect matchings in bipartite graphs.

Guideline: Given a bipartite graph $G = ((X,Y),E)$, consider the matrix M representing the edges between X and Y (i.e., the (i,j)-entry in M is 1 if the i^{th} vertex of X is connected to the j^{th} entry of Y), and note that only perfect matchings in G contribute to the permanent of M.

Exercise 6.24 (computing the permanent modulo 3) Combining Proposition 6.21 and Theorem 6.29, prove that for every fixed $n > 1$ that does not divide any power of c, computing the permanent modulo n is NP-hard under randomized reductions. Since Proposition 6.21 holds for $c = 2^{10}$, hardness holds for every integer $n > 1$ that is not a power of 2. (We mention that, on the other hand, for any fixed $n = 2^e$, the permanent modulo n can be computed in polynomial-time [223, Thm. 3].)

[24] An order of strings is a 1-1 and onto mapping μ from the natural numbers to the set of all strings. Such order is called efficient if both μ and its inverse are efficiently computable. The standard lexicographic order satisfies $\mu(i) = y$ if the string $1y$ is the (compact) binary expansion of the integer i; that is $\mu(1) = \lambda$, $\mu(2) = 0$, $\mu(3) = 1$, $\mu(4) = 00$, etc.

Guideline: Apply the reduction of Proposition 6.21 to the promise problem of deciding whether a 3CNF formula has a unique satisfiable assignment or is unsatisfiable. Note that for any m it holds that $c^m \not\equiv 0 \pmod{n}$.

Exercise 6.25 (negative values in Proposition 6.21) Assuming $\mathcal{P} \neq \mathcal{NP}$, prove that Proposition 6.21 cannot hold for a set I containing only non-negative integers. Note that the claim holds even if the set I is not finite (and even if I is the set of all non-negative integers).

Guideline: A reduction as in Proposition 6.21 yields a Karp-reduction of `3SAT` to deciding whether the permanent of a matrix with entries in I is non-zero. Note that the permanent of a *non-negative* matrix is non-zero if and only if the corresponding bipartite graph has a perfect matching.

Exercise 6.26 (high-level analysis of the permanent reduction) Establish the correctness of the high-level reduction presented in the proof of Proposition 6.21. That is, show that if the clause gadget satisfies the three conditions postulated in the said proof, then each satisfying assignment of ϕ contributes exactly c^m to the SWCC of G_ϕ whereas unsatisfying assignments have no contribution.

Guideline: Cluster the cycle covers of G_ϕ according to the set of track edges that they use (i.e., the edges of the cycle cover that belong to the various tracks). (Note the correspondence between these edges and the external edges used in the definition of the gadget's properties.) Using the postulated conditions (regarding the clause gadget) prove that, for each such set T of track edges, if the sum of the weights of all cycle covers that use the track edges T is non-zero then the following hold:

1. The intersection of T with the set of track edges incident at each specific clause gadget is non-empty. Furthermore, if this set contains an incoming edge (resp., outgoing edge) of some entry-vertex (resp., exit-vertex) then it also contains an outgoing edge (resp., incoming edge) of the corresponding exit-vertex (resp., entry-vertex).
2. If T contains an edge that belongs to some track then it contains all edges of this track. It follows that, for each variable x, the set T contains the edges of a single track associated with x.
3. The tracks "picked" by T correspond to a single truth assignment to the variables of ϕ, and this assignment satisfies ϕ (because, for each clause, T contains an external edge that corresponds to a literal that satisfies this clause).

Note that different sets of the aforementioned type yield different satisfying assignments, and that each satisfying assignment is obtained from some set of the aforementioned type.

Exercise 6.27 (analysis of the implementation of the clause gadget) Establish the correctness of the implementation of the clause gadget presented in the proof of Proposition 6.21. That is, show that if the box satisfy the three conditions postulated in the said proof, then the clause gadget of Figure 6.4 satisfies the conditions postulated for it.

Guideline: Cluster the cycle covers of a gadget according to the set of non-box edges that they use, where non-box edges are the edges shown in Figure 6.4. Using the postulated conditions (regarding the box) prove that, for each set S of non-box edges, if the sum of the weights of all cycle covers that use the non-box edges S is non-zero then the following hold:

1. The intersection of S with the set of edges incident at each box must contain two (non-selfloop) edges, one incident at each of the box's terminals. Needless to say, one edge is incoming and the other outgoing. Referring to the six edges that connects one of the six designated vertices (of the gadget) with the corresponding box terminals as connectives, note that if S contains a connective incident at the terminal of some box then it must also contain the connective incident at the other terminal. In such a case, we say that this box is picked by S,
2. Each of the three (literal-designated) boxes that is not picked by S is "traversed" from left to right (i.e., the cycle cover contains an incoming edge of the left terminal and an outgoing edge of the right terminal). Thus, the set S must contain a connective, because otherwise no directed cycle may cover the leftmost vertex shown in Figure 6.4. That is, S must pick some box.
3. The set S is fully determined by the non-empty set of boxes that it picks.

The postulated properties of the clause gadget follow, with $c = b^5$.

Exercise 6.28 (analysis of the design of a box for the clause gadget) Prove that the 4-by-4 matrix presented in Eq. (6.4) satisfies the properties postulated for the "box" used in the second part of the proof of Proposition 6.21. In particular:

1. Show a correspondence between the conditions required of the box and conditions regarding the value of the permanent of certain sub-matrices of the adjacency matrix of the graph.

   (Hint: For example, show that the first condition correspond to requiring that the value of the permanent of the entire matrix equals zero. The second condition refers to sub-matrices obtained by omitting either the first row and fourth column or the fourth row and first column.)

2. Verify that the matrix in Eq. (6.4) satisfies the aforementioned conditions (regarding the value of the permanent of certain sub-matrices).

Prove that no 3-by-3 matrix (and thus also no 2-by-2 matrix) can satisfy the aforementioned conditions.

Exercise 6.29 (error reduction for approximate counting) Show that the error probability δ in Definition 6.24 can be reduced from $1/3$ (or even $(1/2) + (1/\mathrm{poly}(|x|))$ to $\exp(-\mathrm{poly}(|x|))$.

Guideline: Invoke the weaker procedure for an adequate number of times and take the *median* value among the values obtained in these invocations.

Exercise 6.30 (strong approximation for some $\#\mathcal{P}$-complete problems) Show that there exists $\#\mathcal{P}$-complete problems (albeit unnatural ones) for which an $(\varepsilon, 0)$-approximation can be found by a (deterministic) polynomial-time algorithm. Furthermore, the running-time depends polynomially on $1/\varepsilon$.

Guideline: Combine any $\#\mathcal{P}$-complete problem referring to some $R_1 \in \mathcal{PC}$ with a trivial counting problem (e.g., the counting problem associated with the trivial relation $R_2 = \cup_{n\in\mathbb{N}}\{(x,y) : x,y \in \{0,1\}^n\}$). Show that, without loss of generality, it holds that $\#R_1(x) \le 2^{|x|/2}$. Prove that the counting problem of $R = \{(x,1y) : (x,y) \in R_1\} \cup \{(x,0y) : (x,y) \in R_2\}$ is $\#\mathcal{P}$-complete (by reducing from $\#R_1$). Present a deterministic algorithm that, on input x and $\varepsilon > 0$, outputs an $(\varepsilon, 0)$-approximation of $\#R(x)$ in time $\text{poly}(|x|/\varepsilon)$ (Extra hint: distinguish between $\varepsilon \ge 2^{-|x|/2}$ and $\varepsilon < 2^{-|x|/2}$).

Exercise 6.31 (relative approximation for DNF satisfaction) Referring to the text of §6.2.2.1, prove the following claims.

1. Both assumptions regarding the general setting hold in case $S_i = C_i^{-1}(1)$, where $C_i^{-1}(1)$ denotes the set of truth assignments that satisfy the conjunction C_i.

   Guideline: In establishing the second assumption note that it reduces to the conjunction of the following two assumptions:

   (a) Given i, one can efficiently generate a uniformly distributed element of S_i. Actually, generating a distribution that is almost uniform over S_i suffices.

   (b) Given i and x, one can efficiently determine whether $x \in S_i$.

2. Prove Proposition 6.26, relating to details such as the error probability in an implementation of Construction 6.25.

3. Note that Construction 6.25 does not require exact computation of $|S_i|$. Analyze the output distribution in the case that we can only approximate $|S_i|$ up-to a factor of $1 \pm \varepsilon'$.

Exercise 6.32 (reducing the relative deviation in approximate counting) Prove that, for any $R \in \mathcal{PC}$ and every polynomial p and constant $\delta < 0.5$, there exists $R' \in \mathcal{PC}$ such that $(1/p, \delta)$-approximation for $\#R$ is reducible to $(1/2, \delta)$-approximation for $\#R'$. Furthermore, for any $F(n) = \exp(\text{poly}(n))$, prove that there exists $R'' \in \mathcal{PC}$ such that $(1/p, \delta)$-approximation for $\#R$ is reducible to approximating $\#R''$ to within a factor of F with error probability δ.

Guideline (for the main part): For $t(n) = \Theta(p(n))$, define R' such that $(y_1, ..., y_{t(|x|)}) \in R'(x)$ if and only if $(\forall i)\, y_i \in R(x)$. Note that $|R(x)| = |R'(x)|^{1/t(|x|)}$, and thus if $a = (1 \pm (1/2)) \cdot |R'(x)|$ then $a^{1/t(|x|)} = (1 \pm (1/2))^{1/t(|x|)} \cdot |R(x)|$.

Exercise 6.33 (deviation reduction in approximate counting, cont.) In continuation to Exercise 6.32, prove that if R is NP-complete via parsimonious reductions then, for every positive polynomial p and constant $\delta < 0.5$, the problem of $(1/p, \delta)$-approximation for $\#R$ is reducible to $(1/2, \delta)$-approximation for $\#R$. (Hint: Compose the reduction (to the problem of $(1/2, \delta)$-approximation for $\#R'$) provided in Exercise 6.32 with the parsimonious reduction of $\#R'$ to $\#R$.)
Prove that, for every function F' such that $F'(n) = \exp(n^{o(1)})$, we can also reduce the aforementioned problems to the problem of approximating $\#R$ to within a factor of F' with error probability δ.

Guideline: Using R'' as in Exercise 6.32, we encounter a technical difficulty. The issue is that the composition of the ("amplifying") reduction of $\#R$ to $\#R''$ with the parsimonious reduction of $\#R''$ to $\#R$ may increase the length of the instance. Indeed, the length of the new instance is polynomial in the length of the original instance, but this polynomial may depend on R'', which in turn depends on F'. Thus, we cannot use $F'(n) = \exp(n^{1/O(1)})$ but $F'(n) = \exp(n^{o(1)})$ is fine.

Exercise 6.34 Referring to the procedure in the proof Theorem 6.27, show how to use an NP-oracle in order to determine whether the number of solutions that "pass a random sieve" is greater than t. You are allowed queries of length polynomial in the length of x, h and in the size of t.

Guideline: Consider the set $S'_{R,H} \stackrel{\text{def}}{=} \{(x, i, h, 1^t) : \exists y_1, ..., y_t \text{ s.t. } \psi'(x, h, y_1, ..., y_t)\}$, where $\psi'(x, h, y_1, ..., y_t)$ holds if and only if the y_j are different and for every j it holds that $(x, y_j) \in R \wedge h(y_j) = 0^i$.

Exercise 6.35 (parsimonious reductions and Theorem 6.29) Demonstrate the importance of parsimonious reductions in Theorem 6.29 by proving that there exists a search problem $R \in \mathcal{PC}$ such that every problem in $\mathcal{PC}$ is reducible to R (by a non-parsimonious reduction) and still the the promise problem $(\mathtt{US}_R, \overline{S}_R)$ is decidable in polynomial-time.

Guideline: Consider the following artificial witness relation R for $\mathtt{SAT}$ in which $(\phi, \sigma\tau) \in R$ if $\sigma \in \{0, 1\}$ and τ satisfies ϕ. Note that the standard witness relation of $\mathtt{SAT}$ is reducible to R, but this reduction is not parsimonious. Also note that $\mathtt{US}_R = \emptyset$ and thus $(\mathtt{US}_R, \overline{S}_R)$ is trivial.

Exercise 6.36 In continuation to Exercise 6.35, prove that there exists a search problem $R \in \mathcal{PC}$ such that $\#R$ is $\#\mathcal{P}$-complete and still the the promise problem $(\mathtt{US}_R, \overline{S}_R)$ is decidable in polynomial-time. Provide one proof for the case that R is $\mathcal{PC}$-complete and another proof for $R \in \mathcal{PF}$.

Guideline: For the first case, the relation R suggested in the guideline to Exercise 6.35 will do. For the second case, rely on Theorem 6.20 and on the fact that it is easy to decide $(\mathtt{US}_R, \overline{S}_R)$ when R is the corresponding perfect matching relation (by computing the determinant).

Exercise 6.37 Prove that $\mathtt{SAT}$ is randomly reducible to deciding unique solution for $\mathtt{SAT}$, *without using the fact that* $\mathtt{SAT}$ *is NP-complete via parsimonious reductions.*

Guideline: Follow the proof of Theorem 6.29, while using the family of pairwise independent hashing functions provided in Construction D.3. Note that, in this case, the condition $(\tau \in R_{\mathtt{SAT}}(\phi)) \wedge (h(\tau) = 0^i)$ can be directly encoded as a CNF formula. That is, consider the formula ϕ_h such that $\phi_h(z) \stackrel{\text{def}}{=} \phi(z) \wedge (h(z) = 0^i)$, and note that $h(z) = 0^i$ can be written as the conjunction of i conditions, where each condition is a CNF that is logically equivalent to the parity of some of the bits of z (where the identity of these bits is determined by h).

Exercise 6.38 (an alternative procedure for approximate counting) Adapt Step 1 of Construction 6.32 so to obtain an approximate counting procedure for $\#R$.

Guideline: For $m = 0, 1, ...\ell$, the procedure invokes Step 1 of Construction 6.32 until a negative answer is obtained, and outputs $120\ell \cdot 2^m$ for the current value of m. For $|R(x)| > 80\ell$, this yields a constant factor approximation of $|R(x)|$. In fact, we can obtain a better estimate by making additional queries at iteration m (i.e., queries of the form $(x, h, 1^i)$ for $i = 10\ell, ..., 120\ell$). The case $|R(x)| \leq 80\ell$ can be treated by using Step 2 of Construction 6.32, in which case we obtain an exact count.

Exercise 6.39 Let R be an arbitrary $\mathcal{PC}$-complete search problem. Show that approximate counting and uniform generation for R can be randomly reduced to deciding membership in S_R, where by approximate counting we mean a $(1-(1/p)$-approximation for any polynomial p.

Guideline: Note that Construction 6.32 yields such procedures (see also Exercise 6.38), except that they make oracle calls to some other set in $\mathcal{NP}$. Using the NP-completeness of S_R, we are done.

Chapter 7

The Bright Side of Hardness

So saying she donned her beautiful, glittering golden–Ambrosial sandals, which carry her flying like the wind over the vast land and sea; she grasped the redoubtable bronze-shod spear, so stout and sturdy and strong, wherewith she quells the ranks of heroes who have displeased her, the [bright-eyed] daughter of her mighty father.

Homer, Odyssey, 1:96–101

The existence of natural computational problems that are (or seem to be) infeasible to solve is usually perceived as bad news, because it means that we cannot do things we wish to do. But these bad news have a positive side, because hard problem can be "put to work" to our benefit, most notably in cryptography.

It seems that utilizing hard problems requires the ability to efficiently generate hard instances, which is not guaranteed by the notion of worst-case hardness. In other words, we refer to the gap between "occasional" hardness (e.g., worst-case hardness or mild average-case hardness) and "typical" hardness (with respect to some tractable distribution). Much of the current chapter is devoted to bridging this gap, which is known by the term *hardness amplification*. The actual applications of typical hardness are presented in Chapter 8 and Appendix C.

Summary: We consider two conjectures that are related to $\mathcal{P} \neq \mathcal{NP}$. The first conjecture is that there are problems that are solvable in exponential-time (i.e., in $\mathcal{E}$) but are not solvable by (non-uniform) families of small (say polynomial-size) circuits. We show that this worst-case conjecture can be transformed into an average-case hardness result; specifically, we obtain predicates that are strongly "inapproximable" by small circuits. Such predicates are used towards derandomizing $\mathcal{BPP}$ in a non-trivial manner (see Section 8.3).

The second conjecture is that there are problems in NP (i.e., search problems in $\mathcal{PC}$) for which it is easy to generate (solved) instances that

are typically hard to solve (for a party that did not generate these instances). This conjecture is captured in the formulation of *one-way functions*, which are functions that are easy to evaluate but hard to invert (in an average-case sense). We show that functions that are hard to invert in a relatively mild average-case sense yield functions that are hard to invert in a strong average-case sense, and that the latter yield predicates that are very hard to approximate (called *hard-core predicates*). Such predicates are useful for the construction of general-purpose pseudorandom generators (see Section 8.2) as well as for a host of cryptographic applications (see Appendix C).

In the rest of this chapter, the actual order of presentation of the two aforementioned conjectures and their consequences is reversed: We start (in Section 7.1) with the study of one-way functions, and only later (in Section 7.2) turn to the study of problems in $\mathcal{E}$ that are hard for small circuits.

Teaching note: We list several reasons for preferring the aforementioned order of presentation. First, we mention the great conceptual appeal of one-way functions and the fact that they have very practical applications. Second, hardness amplification in the context of one-way functions is technically simpler than the amplification of hardness in the context of $\mathcal{E}$. (In fact, Section 7.2 is the most technical text in this book.) Third, some of the techniques that are shared by both treatments seem easier to understand first in the context of one-way functions. Last, the current order facilitates the possibility of teaching hardness amplification only in one incarnation, where the context of one-way functions is recommended as the incarnation of choice (for the aforementioned reasons).

If you wish to teach hardness amplification and pseudorandomness in the two aforementioned incarnations, then we suggest following the order of the current text. That is, first teach hardness amplification in its two incarnations, and only next teach pseudorandomness in the corresponding incarnations.

Prerequisites: We assume a basic familiarity with elementary probability theory (see Appendix D.1) and randomized algorithms (see Section 6.1). In particular, standard conventions regarding random variables (presented in Appendix D.1.1) and various "laws of large numbers" (presented in Appendix D.1.2) will be extensively used.

7.1 One-Way Functions

Loosely speaking, one-way functions are functions that are easy to evaluate but hard (on the average) to invert. Thus, in assuming that one-way functions exist, we are postulating the existence of *efficient processes* (i.e., the computation of the function in the forward direction) *that are hard to reverse*. Analogous phenomena in daily life are known to us in abundance (e.g., the lighting of a match). Thus, the assumption that one-way functions exist is a complexity theoretic analogue of our daily experience.

One-way functions can also be thought of as efficient ways for generating "puzzles" that are infeasible to solve; that is, the puzzle is a random image of the function and a solution is a corresponding preimage. Furthermore, the person generating the puzzle knows a solution to it and can efficiently verify the validity of (possibly other) solutions to the puzzle. In fact, as explained in Section 7.1.1, every mechanism for generating such puzzles can be converted to a one-way function.

The reader may note that when presented in terms of generating hard puzzles, one-way functions have a clear cryptographic flavor. Indeed, one-way functions are central to cryptography, but we shall not explore this aspect here (and rather refer the reader to Appendix C). Similarly, one-way functions are closely related to (general-purpose) pseudorandom generators, but this connection will be explored in Section 8.2. Instead, in the current section, we will focus on one-way functions *per se*.

Teaching note: While we recommend including a basic treatment of pseudorandomness within a course on complexity theory, we do not recommend doing so with respect to cryptography. The reason is that cryptography is far more complex than pseudorandomness (e.g., compare the definition of secure encryption to the the definition of pseudorandom generators). The extra complexity is due to conceptual richness, which is something good, except that some of these conceptual issues are central to cryptography but not to complexity theory. Thus, teaching cryptography in the context of a course on complexity theory is likely to either overload the course with material that is not central to complexity theory or cause a superficial and misleading treatment of cryptography. We are not sure as to which of these two possibilities is worse. Still, for the benefit of the interested reader, we have included an overview of the foundations of cryptography as an appendix to the main text (see Appendix C).

7.1.1 Generating hard instances and one-way functions

Let us start by examining the prophecy, made in the preface to this chapter, by which intractable problems can be used to our benefit. The basic idea is that intractable problems offer a way of generating an obstacle that stands in the way of our opponents and thus protects our interests. These opponents may be either real (e.g., in the context of cryptography) or imaginary (e.g., in the context of derandomization), but in both cases we wish to prevent them from seeing something or doing something. Hard obstacles seems useful towards this goal.

Let us assume that $\mathcal{P} \neq \mathcal{NP}$ or even that $\mathcal{NP}$ is not contained in $\mathcal{BPP}$. Can we use this assumption to our benefit? Not really: The $\mathcal{NP} \not\subseteq \mathcal{BPP}$ assumption refers to the worst-case complexity of problems, while benefiting from hard problems seems to require the ability to generate hard instances. In particular, the generated instances should be typically hard and not merely occasionally hard; that is, we seek average-case hardness and not merely worst-case hardness.

Taking a short digression, we mention that in Section 7.2 we shall see that worst-case hardness (of $\mathcal{NP}$ or even $\mathcal{E}$) can be transformed into average-case hardness of $\mathcal{E}$. Such a transformation is not known for $\mathcal{NP}$ itself, and in some applications (e.g., in cryptography) we do need the hard-on-the-average problem to be in $\mathcal{NP}$.

In this case, we currently need to assume that, for some problem in $\mathcal{NP}$, it is the case that hard instances are easy to generate (and not merely exist). That is, we assume that $\mathcal{NP}$ is "hard on the average" *with respect to a distribution that is efficiently sampleable.* This assumption will be further discussed in Section 10.2.

However, for the aforementioned applications (e.g., in cryptography) this assumption does not seem to suffice either: we know how to utilize such "hard on the average" problems *only when we can efficiently generate hard instances coupled with adequate solutions.*[1] That is, we assume that, for some search problem in $\mathcal{PC}$ (resp., decision problem in $\mathcal{NP}$), we can efficiently generate instance-solution pairs (resp., yes-instances coupled with corresponding NP-witnesses) such that the instance is hard to solve (resp., hard to verify as belonging to the set). Needless to say, the hardness assumption refers to a person that does not get the solution (resp., witness). Thus, we can efficiently generate hard "puzzles" coupled with solutions, and so we may present to others hard puzzles for which we know a solution.

Let us formulate the foregoing discussion. Referring to Definition 2.3, we consider a relation R in $\mathcal{PC}$ (i.e., R is polynomially bounded and membership in R can be determined in polynomial-time), and assume that there exists a probabilistic polynomial-time algorithm G that satisfies the following two conditions:

1. On input 1^n, algorithm G always generates a pair in R such that the first element has length n. That is, $\mathsf{Pr}[G(1^n) \in R \cap (\{0,1\}^n \times \{0,1\}^*)] = 1$.

2. It is typically infeasible to find solutions to instances that are generated by G; that is, when only given the first element of $G(1^n)$, it is infeasible to find an adequate solution. Formally, denoting the first element of $G(1^n)$ by $G_1(1^n)$, for every probabilistic polynomial-time (solver) algorithm S, it holds that $\mathsf{Pr}[(G_1(1^n), S(G_1(1^n)) \in R] = \mu(n)$, where μ vanishes faster than any polynomial fraction (i.e., for every positive polynomial p and all sufficiently large n it is the case that $\mu(n) < 1/p(n)$).

We call G a generator of solved intractable instances for R. We will show that such a generator exists if and only if one-way functions exist, where one-way functions are functions that are easy to evaluate but hard (on the average) to invert. That is, a function $f:\{0,1\}^* \to \{0,1\}^*$ is called one-way if there is an efficient algorithm that on input x outputs $f(x)$, whereas any feasible algorithm that tries to find a preimage of $f(x)$ under f may succeed only with negligible probability (where the probability is taken uniformly over the choices of x and the algorithm's coin tosses). Associating feasible computations with probabilistic polynomial-time algorithms and negligible functions with functions that vanish faster than any polynomial fraction, we obtain the following definition.

Definition 7.1 (one-way functions): *A function $f:\{0,1\}^* \to \{0,1\}^*$ is called* one-way *if the following two conditions hold:*

[1] We wish to stress the difference between the two gaps discussed here. Our feeling is that the non-usefulness of worst-case hardness (*per se*) is far more intuitive than the non-usefulness of average-case hardness that does not correspond to an efficient generation of "solved" instances.

1. Easy to evaluate: *There exist a polynomial-time algorithm A such that $A(x) = f(x)$ for every $x \in \{0,1\}^*$.*

2. Hard to invert: *For every probabilistic polynomial-time algorithm A', every polynomial p, and all sufficiently large n,*

$$\mathsf{Pr}_{x\in\{0,1\}^n}[A'(f(x),1^n) \in f^{-1}(f(x))] \ < \ \frac{1}{p(n)} \tag{7.1}$$

where the probability is taken uniformly over all the possible choices of $x \in \{0,1\}^n$ and all the possible outcomes of the internal coin tosses of algorithm A'.

Algorithm A' is given the auxiliary input 1^n so as to allow it to run in time polynomial in the length of x, which is important in case f drastically shrinks its input (e.g., $|f(x)| = O(\log|x|)$). Typically (and, in fact, without loss of generality, see Exercise 7.1), f is length preserving, in which case the auxiliary input 1^n is redundant. Note that A' is not required to output a specific preimage of $f(x)$; any preimage (i.e., element in the set $f^{-1}(f(x))$) will do. (Indeed, in case f is 1-1, the string x is the only preimage of $f(x)$ under f; but in general there may be other preimages.) It is required that algorithm A' fails (to find a preimage) with overwhelming probability, when the probability is also taken over the input distribution. That is, f is "typically" hard to invert, not merely hard to invert in some ("rare") cases.

Proposition 7.2 *The following two conditions are equivalent:*

1. *There exists a generator of solved intractable instances for some $R \in \mathcal{NP}$.*

2. *There exist one-way functions.*

Proof Sketch: Suppose that G is such a generator of solved intractable instances for some $R \in \mathcal{NP}$, and suppose that on input 1^n it tosses $\ell(n)$ coins. For simplicity, we assume that $\ell(n) = n$, and consider the function $g(r) = G_1(1^{|r|}, r)$, where $G(1^n, r)$ denotes the output of G on input 1^n when using coins r (and G_1 is as in the foregoing discussion). Then g must be one-way, because an algorithm that inverts g on input $x = g(r)$ obtains r' such that $G_1(1^n, r') = x$ and $G(1^n, r')$ must be in R (which means that the second element of $G(1^n, r')$ is a solution to x). In case $\ell(n) \neq n$ (and assuming without loss of generality that $\ell(n) \geq n$), we define $g(r) = G_1(1^n, s)$ where n is the largest integer such that $\ell(n) \leq |r|$ and s is the $\ell(n)$-bit long prefix of r.

Suppose, on the other hand, that f is a one-way function (and that f is length preserving). Consider $G(1^n)$ that uniformly selects $r \in \{0,1\}^n$ and outputs $(f(r), r)$, and let $R \stackrel{\text{def}}{=} \{(f(x), x) : x \in \{0,1\}^*\}$. Then R is in $\mathcal{PC}$ and G is a generator of solved intractable instances for R, because any solver of R (on instances generated by G) is effectively inverting f on $f(U_n)$. $\square$

Comments. Several candidates one-way functions and variation on the basic definition appear in Appendix C.2.1. Here, for the sake of future discussions, we define a stronger version of one-way functions, which refers to the infeasibility of inverting the function by non-uniform circuits of polynomial-size. We seize the opportunity and use an alternative technical formulation, which is based on the probabilistic conventions in Appendix D.1.1.[2]

Definition 7.3 (one-way functions, non-uniformly hard): *A one-way function* $f : \{0,1\}^* \to \{0,1\}^*$ *is said to be* non-uniformly hard to invert *if for every family of polynomial-size circuits* $\{C_n\}$*, every polynomial* p*, and all sufficiently large* n*,*

$$\mathsf{Pr}[C_n(f(U_n), 1^n) \in f^{-1}(f(U_n))] \;<\; \frac{1}{p(n)}$$

We note that if a function is infeasible to invert by polynomial-size circuits then it is hard to invert by probabilistic polynomial-time algorithms; that is, non-uniformity (more than) compensates for lack of randomness. See Exercise 7.2.

7.1.2 Amplification of Weak One-Way Functions

In the forgoing discussion we have interpreted "hardness on the average" in a very strong sense. Specifically, we required that any feasible algorithm fails to solve the problem (e.g., invert the one-way function) *almost always* (i.e., *except with negligible probability*). This interpretation is indeed the one that is suitable for various applications. Still, a weaker interpretation of hardness on the average, which is also appealing, only requires that any feasible algorithm fails to solve the problem *often enough* (i.e., *with noticeable probability*). The main thrust of the current section is showing that the mild form of hardness on the average can be transformed into the strong form discussed in Section 7.1.1. Let us first define the mild form of hardness on the average, using the framework of one-way functions. Specifically, we define weak one-way functions.

Definition 7.4 (weak one-way functions): *A function* $f:\{0,1\}^* \to \{0,1\}^*$ *is called* weakly one-way *if the following two conditions hold:*

1. Easy to evaluate: *As in Definition 7.1.*
2. Weakly hard to invert: *There exists a positive polynomial* p *such that for every probabilistic polynomial-time algorithm* A' *and all sufficiently large* n*,*

$$\mathsf{Pr}_{x\in\{0,1\}^n}[A'(f(x), 1^n) \notin f^{-1}(f(x))] \;>\; \frac{1}{p(n)} \tag{7.2}$$

where the probability is taken uniformly over all the possible choices of $x \in \{0,1\}^n$ *and all the possible outcomes of the internal coin tosses of algorithm* A'*. In such a case, we say that* f *is* $1/p$-one-way.

[2]Specifically, letting U_n denote a random variable uniformly distributed in $\{0,1\}^n$, we may write Eq. (7.1) as $\mathsf{Pr}[A'(f(U_n), 1^n) \in f^{-1}(f(U_n))] \;<\; 1/p(n)$, recalling that both occurrences of U_n refer to the same sample.

Here we require that algorithm A' fails (to find an f-preimage for a random f-image) with noticeable probability, rather than with overwhelmingly high probability (as in Definition 7.1). For clarity, we will occasionally refer to one-way functions as in Definition 7.1 by the term strong one-way functions.

We note that, assuming that one-way functions exist at all, there exists weak one-way functions that are not strongly one-way (see Exercise 7.3). Still, any weak one-way function can be transformed into a strong one-way function. This is indeed the main result of the current section.

Theorem 7.5 (amplification of one-way functions): *The existence of weak one-way functions implies the existence of strong one-way functions.*

Proof Sketch: The construction itself is straightforward. We just parse the argument to the new function into sufficiently many blocks, and apply the weak one-way function on the individual blocks. That is, suppose that f is $1/p$-one-way, for some polynomial p, and consider the following function

$$\begin{aligned} F(x_1, ..., x_t) &= (f(x_1), ..., f(x_t)) \\ &\quad \text{where } t \stackrel{\text{def}}{=} n \cdot p(n) \text{ and } x_1, ..., x_t \in \{0,1\}^n. \end{aligned} \tag{7.3}$$

(Indeed F should be extended to strings of length outside $\{n^2 \cdot p(n) : n \in \mathbb{N}\}$ and this extension must be hard to invert on all preimage lengths.)[3]

We warn that the hardness of inverting the resulting function F is not established by mere "combinatorics" (i.e., considering, for any $S \subset \{0,1\}^n$, the relative volume of S^t in $(\{0,1\}^n)^t$, where S represents the set of f-preimages that are mapped by f to an image that is "easy to invert"). Specifically, one may *not* assume that the potential inverting algorithm works independently on each block. Indeed this assumption seems reasonable, but we do not know if nothing is lost by this restriction. (In fact, proving that nothing is lost by this restriction is a formidable research project.) In general, we should not make assumptions regarding the class of all efficient algorithms (as underlying the definition of one-way functions), unless we can actually prove that nothing is lost by such assumptions.

The hardness of inverting the resulting function F is proved via a so called "reducibility argument" (which is used to prove all conditional results in the area). By a reducibility argument we actually mean a reduction, but one that is analyzed with respect to average case complexity. Specifically, we show that any algorithm that inverts the resulting function F with non-negligible success probability can be used to construct an algorithm that inverts the original function f with success probability that violates the hypothesis (regarding f). In other words, we reduce the task of "strongly inverting" f (i.e., violating its weak one-wayness) to the task of "weakly inverting" F (i.e., violating its strong one-wayness). In particular, on input $y = f(x)$, the reduction invokes the F-inverter (polynomially) many times, each time feeding it with a sequence of random f-images that contains y at a

[3] One simple extension is defining $F(x)$ to equal $F(x_1, ..., x_{n \cdot p(n)})$, where n is the largest integer satisfying $n^2 p(n) \leq |x|$ and x_i is the i^{th} consecutive n-bit long string in x (i.e., $x = x_1 \cdots x_{n \cdot p(n)} x'$, where $x_1, ..., x_{n \cdot p(n)} \in \{0,1\}^n$).

random location. (Indeed such a sequence corresponds to a random image of F.) Details follow.

Suppose towards the contradiction that F is not strongly one-way; that is, there exists a probabilistic polynomial-time algorithm B' and a polynomial $q(\cdot)$ so that for infinitely many m's

$$\mathsf{Pr}[B'(F(U_m)) \in F^{-1}(F(U_m))] > \frac{1}{q(m)} \tag{7.4}$$

Focusing on such a generic m and assuming (see Footnote 3) that $m = n^2p(n)$, we present the following probabilistic polynomial-time algorithm, A', for inverting f. On input y and 1^n (where supposedly $y = f(x)$ for some $x \in \{0,1\}^n$), algorithm A' proceeds by applying the following probabilistic procedure, denoted I, on input y for $t'(n)$ times, where $t'(\cdot)$ is a polynomial that depends on the polynomials p and q (specifically, we set $t'(n) \stackrel{\text{def}}{=} 2n^2 \cdot p(n) \cdot q(n^2p(n))$).

Procedure I (on input y and 1^n):

For $i = 1$ to $t(n) \stackrel{\text{def}}{=} n \cdot p(n)$ do begin
(1) Select uniformly and independently a sequence of strings $x_1, ..., x_{t(n)} \in \{0,1\}^n$.
(2) Compute $(z_1, ..., z_{t(n)}) \leftarrow B'(f(x_1), ..., f(x_{i-1}), y, f(x_{i+1}), ..., f(x_{t(n)}))$
(Note that y is placed in the i^{th} position instead of $f(x_i)$.)
(3) If $f(z_i) = y$ then halt and output z_i.
(This is considered a *success*).
end

Using Eq. (7.4), we now present a lower bound on the success probability of algorithm A', deriving a contradiction to the theorem's hypothesis. To this end we define a set, denoted S_n, that contains all n-bit strings on which the procedure I succeeds with probability greater than $n/t'(n)$. (The probability is taken only over the coin tosses of procedure I). Namely,

$$S_n \stackrel{\text{def}}{=} \left\{ x \in \{0,1\}^n : \mathsf{Pr}[I(f(x)) \in f^{-1}(f(x))] > \frac{n}{t'(n)} \right\}$$

In the next two claims we shall show that S_n contains all but at most a $1/2p(n)$ fraction of the strings of length n, and that for each string $x \in S_n$ algorithm A' inverts f on $f(x)$ with probability exponentially close to 1. It will follow that A' inverts f on $f(U_n)$ with probability greater than $1 - (1/p(n))$, in contradiction to the theorem's hypothesis.

Claim 7.5.1: For every $x \in S_n$

$$\mathsf{Pr}\left[A'(f(x)) \in f^{-1}(f(x))\right] > 1 - 2^{-n}$$

This claim follows directly from the definitions of S_n and A'.

Claim 7.5.2:

$$|S_n| > \left(1 - \frac{1}{2p(n)}\right) \cdot 2^n$$

The rest of the proof is devoted to establishing this claim, and indeed combining Claims 7.5.1 and 7.5.2, the theorem follows.

The key observation is that, for every $i \in [t(n)]$ and every $x_i \in \{0,1\}^n \setminus S_n$, it holds that

$$\begin{aligned}
&\Pr\left[B'(F(U_{n^2p(n)})) \in F^{-1}(F(U_{n^2p(n)})) \,\middle|\, U_n^{(i)} = x_i\right] \\
&\quad \leq \Pr\left[I(f(x_i)) \in f^{-1}(f(x_i))\right] \;\leq\; \frac{n}{t'(n)}
\end{aligned}$$

where $U_n^{(1)}, ..., U_n^{(n\cdot p(n))}$ denote the n-bit long blocks in the random variable $U_{n^2p(n)}$. It follows that

$$\begin{aligned}
\xi &\stackrel{\text{def}}{=} \Pr\left[B'(F(U_{n^2p(n)})) \in F^{-1}(F(U_{n^2p(n)})) \wedge \left(\exists i \text{ s.t. } U_n^{(i)} \in \{0,1\}^n \setminus S_n\right)\right] \\
&\leq \sum_{i=1}^{t(n)} \Pr\left[B'(F(U_{n^2p(n)})) \in F^{-1}(F(U_{n^2p(n)})) \wedge U_n^{(i)} \in \{0,1\}^n \setminus S_n\right] \\
&\leq t(n) \cdot \frac{n}{t'(n)} \;=\; \frac{1}{2q(n^2p(n))}
\end{aligned}$$

where the equality is due to $t'(n) = 2n^2 \cdot p(n) \cdot q(n^2p(n))$ and $t(n) = n \cdot p(n)$. On the other hand, using Eq. (7.4), we have

$$\begin{aligned}
\xi &\geq \Pr\left[B'(F(U_{n^2p(n)})) \in F^{-1}(F(U_{n^2p(n)}))\right] \;-\; \Pr\left[(\forall i)\, U_n^{(i)} \in S_n\right] \\
&\geq \frac{1}{q(n^2p(n))} - \Pr\left[U_n \in S_n\right]^{t(n)} .
\end{aligned}$$

Using $t(n) = n \cdot p(n)$, we get $\Pr[U_n \in S_n] > (1/2q(n^2p(n)))^{1/(n\cdot p(n))}$, which implies $\Pr[U_n \in S_n] > 1 - (1/2p(n))$ for sufficiently large n. Claim 7.5.2 follows, and so does the theorem. □

Digest. Let us recall the structure of the proof of Theorem 7.5. Given a weak one-way function f, we first constructed a polynomial-time computable function F with the intention of later proving that F is strongly one-way. To prove that F is strongly one-way, we used a *reducibility argument*. The argument transforms efficient algorithms that supposedly contradict the strong one-wayness of F into efficient algorithms that contradict the hypothesis that f is weakly one-way. Hence F must be strongly one-way. We stress that our algorithmic transformation, which is in fact a randomized Cook reduction, makes no implicit or explicit assumptions about the structure of the prospective algorithms for inverting F. Such assumptions (e.g., the "natural" assumption that the inverter of F works independently on each block) cannot be justified (at least not at our current state of understanding of the nature of efficient computations).

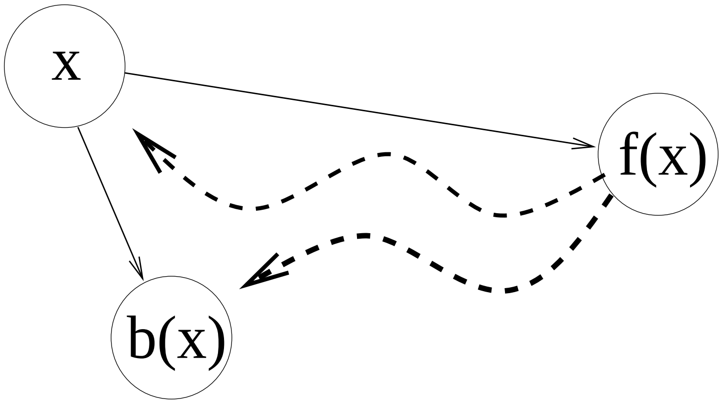
We use the term a *reducibility argument*, rather than just saying a reduction so as to emphasize that we do *not* refer here to standard (worst-case complexity) reductions. Let us clarify the distinction: In both cases we refer to *reducing* the task of solving one problem to the task of solving another problem; that is, we use a procedure solving the second task in order to construct a procedure that solves the first task. However, in standard reductions one assumes that the second task has a perfect procedure solving it on all instances (i.e., on the worst-case), and constructs such a procedure for the first task. Thus, the reduction may invoke the given procedure (for the second task) on very "non-typical" instances. This cannot be allowed in our reducibility arguments. Here, we are given a procedure that solves the second task *with certain probability with respect to a certain distribution.* Thus, in employing a reducibility argument, we cannot invoke this procedure on any instance. Instead, we must consider the probability distribution, on instances of the second task, induced by our reduction. In our case (as in many cases) the latter distribution equals the distribution to which the hypothesis (regarding solvability of the second task) refers, but in general these distributions need only be "sufficiently close" in an adequate sense (which depends on the analysis). In any case, a careful consideration of the distribution induced by the reducibility argument is due. (Indeed, the same issue arises in the context of reductions among "distributional problems" considered in Section 10.2.)

An information theoretic analogue. Theorem 7.5 (or rather its proof) has a natural information theoretic (or "probabilistic") analogue that refers to the amplification of the success probability by repeated experiments: If some event occurs with probability p in a single experiment, then the event will occur with very high probability (i.e., $1-e^{-n}$) when the experiment is repeated n/p times. The analogy is to evaluating the function F at a random input, where each block of this input may be viewed as an attempt to hit the noticeable "hard region" of f. The reader is probably convinced at this stage that the proof of Theorem 7.5 is much more complex than the proof of the information theoretic analogue. In the information theoretic context the repeated experiments are independent by definition, whereas in the computational context no such independence can be guaranteed. (Indeed, the independence assumption corresponds to the naive argument discussed at the beginning of the proof of Theorem 7.5.) Another indication to the difference between the two settings follows. In the information theoretic setting, the probability that the event did not occur in any of the repeated trials decreases exponentially with the number of repetitions. In contrast, in the computational setting we can only reach an unspecified negligible bound on the inverting probabilities of polynomial-time algorithms. Furthermore, for all we know, it may be the case that F can be efficiently inverted on $F(U_{n^2 p(n)})$ with success probability that is sub-exponentially decreasing (e.g., with probability $2^{-(\log_2 n)^3}$), whereas the analogous information theoretic bound is exponentially decreasing (i.e., e^{-n}).

7.1.3 Hard-Core Predicates

One-way functions *per se* suffice for one central application: the construction of secure signature schemes (see Appendix C.6). For other applications, one relies not merely on the infeasibility of fully recovering the preimage of a one-way function, but rather on the infeasibility of meaningfully guessing bits in the preimage. The latter notion is captured by the definition of a hard-core predicate.

Recall that saying that a function f is one-way means that given a typical y (in the range of f) it is infeasible to find a preimage of y under f. This does not mean that it is infeasible to find partial information about the preimage(s) of y under f. Specifically, it may be easy to retrieve half of the bits of the preimage (e.g., given a one-way function f consider the function f' defined by $f'(x,r) \stackrel{\text{def}}{=} (f(x),r)$, for every $|x|=|r|$). We note that hiding partial information (about the function's preimage) plays an important role in more advanced constructs (e.g., pseudorandom generators and secure encryption). With this motivation in mind, we will show that essentially any one-way function hides specific partial information about its preimage, where this partial information is easy to compute from the preimage itself. This partial information can be considered as a "hard core" of the difficulty of inverting f. Loosely speaking, a *polynomial-time computable* (Boolean) predicate b, is called a hard-core of a function f if no feasible algorithm, given $f(x)$, can guess $b(x)$ with success probability that is non-negligibly better than one half.



The solid arrows depict easily computable transformation while the dashed arrows depict infeasible transformations.

Figure 7.1: The hard-core of a one-way function – an illustration.

Definition 7.6 (hard-core predicates): *A polynomial-time computable predicate $b:\{0,1\}^* \rightarrow \{0,1\}$ 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*

$$\mathsf{Pr}\left[A'(f(x))=b(x)\right] < \frac{1}{2} + \frac{1}{p(n)}$$

where the probability is taken uniformly over all the possible choices of $x \in \{0,1\}^n$ and all the possible outcomes of the internal coin tosses of algorithm A'.

Note that for every $b : \{0,1\}^* \rightarrow \{0,1\}$ and $f : \{0,1\}^* \rightarrow \{0,1\}^*$, there exist obvious algorithms that guess $b(x)$ from $f(x)$ with success probability at least one half (e.g., the algorithm that, obliviously of its input, outputs a uniformly chosen bit). Also, if b is a hard-core predicate (of any function) then it follows that b is almost unbiased (i.e., for a uniformly chosen x, the difference $|\Pr[b(x)=0] - \Pr[b(x)=1]|$ must be a negligible function in n).

Since b itself is polynomial-time computable, the failure of efficient algorithms to approximate $b(x)$ from $f(x)$ (with success probability that is non-negligibly higher than one half) must be due either to an information loss of f (i.e., f not being one-to-one) or to the difficulty of inverting f. For example, for $\sigma \in \{0,1\}$ and $x' \in \{0,1\}^*$, the predicate $b(\sigma x') = \sigma$ is a hard-core of the function $f(\sigma x') \stackrel{\rm def}{=} 0x'$. Hence, in this case the fact that b is a hard-core of the function f is due to the fact that f loses information (specifically, the first bit: σ). On the other hand, in the case that f loses no information (i.e., f is one-to-one) a hard-core for f may exist only if f is hard to invert. In general, the interesting case is when being a hard-core is a computational phenomenon rather than an information theoretic one (which is due to "information loss" of f). It turns out that any one-way function has a modified version that possesses a hard-core predicate.

Theorem 7.7 (a generic hard-core predicate): *For any one-way function f, the inner-product mod 2 of x and r, denoted $b(x,r)$, is a hard-core of $f'(x,r) = (f(x),r)$.*

In other words, Theorem 7.7 asserts that, given $f(x)$ and a random subset $S \subseteq [|x|]$, it is infeasible to guess $\oplus_{i\in S} x_i$ significantly better than with probability $1/2$, where $x = x_1 \cdots x_n$ is uniformly distributed in $\{0,1\}^n$.

Proof Sketch: The proof is by a so-called "reducibility argument" (see Section 7.1.2). Specifically, we reduce the task of inverting f to the task of predicting the hard-core of f', while making sure that the reduction (when applied to input distributed as in the inverting task) generates a distribution as in the definition of the predicting task. Thus, a contradiction to the claim that b is a hard-core of f' yields a contradiction to the hypothesis that f is hard to invert. We stress that this argument is far more complex than analyzing the corresponding "probabilistic" situation (i.e., the distribution of $(r, b(X,r))$, where $r \in \{0,1\}^n$ is uniformly distributed and X is a random variable with super-logarithmic min-entropy (which represents the "effective" knowledge of x, when given $f(x)$)).[4]

Our starting point is a probabilistic polynomial-time algorithm B that satisfies, for some polynomial p and infinitely many n's, $\Pr[B(f(X_n),U_n) = b(X_n,U_n)] > (1/2) + (1/p(n))$, where X_n and U_n are uniformly and independently distributed over $\{0,1\}^n$. Using a simple averaging argument, we focus on a $\varepsilon \stackrel{\rm def}{=} 1/2p(n)$

[4]The min-entropy of X is defined as $\min_v\{\log_2(1/\Pr[X=v])\}$; that is, if X has min-entropy m then $\max_v\{\Pr[X=v]\} = 2^{-m}$. The Leftover Hashing Lemma (see Appendix D.2) implies that, in this case, $\Pr[b(X,U_n)=1|U_n] = \frac{1}{2} \pm 2^{-\Omega(m)}$, where U_n denotes the uniform distribution over $\{0,1\}^n$.

fraction of the x's for which $\Pr[B(f(x),U_n)=b(x,U_n)] > (1/2)+\varepsilon$ holds. We will show how to use B in order to invert f, on input $f(x)$, provided that x is in this good set (which has density ε).

As a warm-up, suppose for a moment that, for the aforementioned x's, algorithm B succeeds with probability p such that $p > \frac{3}{4} + 1/\text{poly}(|x|)$ rather than $p > \frac{1}{2} + 1/\text{poly}(|x|)$. In this case, retrieving x from $f(x)$ is quite easy: To retrieve the i^{th} bit of x, denoted x_i, we randomly select $r \in \{0,1\}^{|x|}$, and obtain $B(f(x),r)$ and $B(f(x),r\oplus e^i)$, where $e^i = 0^{i-1}10^{|x|-i}$ and $v \oplus u$ denotes the addition mod 2 of the binary vectors v and u. A key observation underlying the foregoing scheme as well as the rest of the proof is that $b(x,r\oplus s) = b(x,r) \oplus b(x,s)$, which can be readily verified by writing $b(x,y) = \sum_{i=1}^{n} x_i y_i \bmod 2$ and noting that addition modulo 2 of bits corresponds to their XOR. Now, note that if both $B(f(x),r) = b(x,r)$ and $B(f(x),r\oplus e^i) = b(x,r\oplus e^i)$ hold, then $B(f(x),r) \oplus B(f(x),r\oplus e^i)$ equals $b(x,r) \oplus b(x,r\oplus e^i) = b(x,e^i) = x_i$. The probability that both $B(f(x),r)=b(x,r)$ and $B(f(x),r\oplus e^i)=b(x,r\oplus e^i)$ hold, for a random r, is at least $1-2\cdot(1-p) > \frac{1}{2} + \frac{1}{\text{poly}(|x|)}$. Hence, repeating the foregoing procedure sufficiently many times (using independent random choices of such r's) and ruling by majority, we retrieve x_i with very high probability. Similarly, we can retrieve all the bits of x, and hence invert f on $f(x)$. However, the entire analysis was conducted under (the unjustifiable) assumption that $p > \frac{3}{4}+\frac{1}{\text{poly}(|x|)}$, whereas we only know that $p > \frac{1}{2}+\varepsilon$ for $\varepsilon = 1/\text{poly}(|x|)$.

The problem with the foregoing procedure is that it doubles the original error probability of algorithm B on inputs of the form $(f(x),\cdot)$. Under the unrealistic (foregoing) assumption that B's average error on such inputs is non-negligibly smaller than $\frac{1}{4}$, the "error-doubling" phenomenon raises no problems. However, in general (and even in the special case where B's error is exactly $\frac{1}{4}$) the foregoing procedure is unlikely to invert f. Note that the *average* error probability of B (for a fixed $f(x)$, when the average is taken over a random r) can not be decreased by repeating B several times (e.g., for every x, it may be that B always answer correctly on three quarters of the pairs $(f(x),r)$, and always err on the remaining quarter). What is required is an *alternative way of using* the algorithm B, a way that does not double the original error probability of B.

The key idea is generating the r's in a way that allows applying algorithm B only once per each r (and i), instead of twice. Specifically, we will invoke B on $(f(x),r\oplus e^i)$ in order to obtain a "guess" for $b(x,r\oplus e^i)$, and obtain $b(x,r)$ in a different way (which does not involve using B). The good news is that the error probability is no longer doubled, since we only use B to get a "guess" of $b(x,r\oplus e^i)$. The bad news is that we still need to know $b(x,r)$, and it is not clear how we can know $b(x,r)$ without applying B. The answer is that we can guess $b(x,r)$ by ourselves. This is fine if we only need to guess $b(x,r)$ for one r (or logarithmically in $|x|$ many r's), but the problem is that we need to know (and hence guess) the value of $b(x,r)$ for polynomially many r's. The obvious way of guessing these $b(x,r)$'s yields an exponentially small success probability. Instead, we generate these polynomially many r's such that, on one hand they are "sufficiently random" whereas, on the other hand, we can guess all the $b(x,r)$'s with noticeable success probability.[5] Specifically, generating the r's in a specific *pairwise independent* manner will satisfy both these (conflicting) requirements. We stress that in case we are successful (in our guesses for all the $b(x,r)$'s), we can retrieve x with high probability. Hence, we retrieve x with noticeable probability.

A word about the way in which the pairwise independent r's are generated (and the corresponding $b(x,r)$'s are guessed) is indeed in place. To generate $m = \text{poly}(|x|)$ many r's, we uniformly (and independently) select $\ell \stackrel{\text{def}}{=} \log_2(m+1)$ strings in $\{0,1\}^{|x|}$. Let us denote these strings by $s^1,...,s^\ell$. We then guess $b(x,s^1)$ through $b(x,s^\ell)$. Let us denote these guesses, which are uniformly (and independently) chosen in $\{0,1\}$, by σ^1 through σ^ℓ. Hence, the probability that all our guesses for the $b(x,s^i)$'s are correct is $2^{-\ell} = \frac{1}{\text{poly}(|x|)}$. The different r's correspond to the different *non-empty* subsets of $\{1,2,...,\ell\}$. Specifically, for every such subset J, we let $r^J \stackrel{\text{def}}{=} \oplus_{j\in J} s^j$. The reader can easily verify that the r^J's are pairwise independent and each is uniformly distributed in $\{0,1\}^{|x|}$; see Exercise 7.5. The key observation is that $b(x,r^J) = b(x,\oplus_{j\in J}s^j) = \oplus_{j\in J} b(x,s^j)$. Hence, our guess for $b(x,r^J)$ is $\oplus_{j\in J}\sigma^j$, and with noticeable probability all our guesses are correct. Wrapping-up everything, we obtain the following procedure, where $\varepsilon = 1/\text{poly}(n)$ represents a lower-bound on the advantage of B in guessing $b(x,\cdot)$ for an ε fraction of the x's (i.e., for these good x's it holds that $\Pr[B(f(x),U_n) = b(x,U_n)] > \frac{1}{2}+\varepsilon$).

Inverting procedure (on input $y = f(x)$ and parameters n and ε):
Set $\ell = \log_2(n/\varepsilon^2) + O(1)$.
(1) Select uniformly and independently $s^1,...,s^\ell \in \{0,1\}^n$.
Select uniformly and independently $\sigma^1,...,\sigma^\ell \in \{0,1\}$.
(2) For every non-empty $J \subseteq [\ell]$, compute $r^J = \oplus_{j\in J} s^j$ and $\rho^J = \oplus_{j\in J}\sigma^j$.
(3) For $i = 1,...,n$ determine the bit z_i according to the majority vote of the $(2^\ell - 1)$-long sequence of bits $(\rho^J \oplus B(f(x), r^J \oplus e^i))_{\emptyset\neq J\subseteq[\ell]}$.
(4) Output $z_1 \cdots z_n$.

Note that the "voting scheme" employed in Step 3 uses pairwise independent samples (i.e., the r^J's), but works essentially as well as it would have worked with independent samples (i.e., the independent r's).[6] That is, for every i and J, it holds that $\Pr_{s^1,...,s^\ell}[B(f(x), r^J\oplus e^i) = b(x, r^J\oplus e^i)] > (1/2)+\varepsilon$, where $r^J = \oplus_{j\in J}s^j$, and (for every fixed i) the events corresponding to different J's are pairwise independent. It follows that *if for every* $j \in [\ell]$ *it holds that* $\sigma^j = b(x,s^j)$, then for every i and J we have

$$\Pr_{s^1,...,s^\ell}[\rho^J \oplus B(f(x), r^J \oplus e^i) = b(x,e^i)] \tag{7.5}$$

[5] Alternatively, we can try all polynomially many possible guesses. In such a case, we shall output a list of candidates that, with high probability, contains x. (See Exercise 7.6.)

[6] Our focus here is on the accuracy of the approximation obtained by the sample, and not so much on the error probability. We wish to approximate $\Pr[b(x,r) \oplus B(f(x), r\oplus e^i) = 1]$ up to an additive term of ε, because such an approximation allows to correctly determine $b(x,e^i)$. A pairwise independent sample of $O(t/\varepsilon^2)$ points allows for an approximation of a value in $[0,1]$ up to an additive term of ε with error probability $1/t$, whereas a totally random sample of the same size yields error probability $\exp(-t)$. Since we can afford setting $t = \text{poly}(n)$ and having error probability $1/2n$, the difference in the error probability between the two approximation schemes is not important here. For a wider perspective see Appendix D.1.2 and D.3.

$$= \mathsf{Pr}_{s^1,\ldots,s^\ell}[B(f(x), r^J \oplus e^i) = b(x, r^J \oplus e^i)] \;>\; \frac{1}{2} + \varepsilon$$

where the equality is due to $\rho^J = \oplus_{j\in J}\sigma^j = b(x, r^J) = b(x, r^J \oplus e^i) \oplus b(x, e^i)$. Note that Eq. (7.5) refers to the correctness of a single vote for $b(x, e^i)$. Using $m = 2^\ell - 1 = O(n/\varepsilon^2)$ and noting that these (Boolean) votes are pairwise independent, we infer that the probability that the majority of these votes is wrong is upper-bounded by $1/2n$. Using a union bound on all i's, we infer that with probability at least $1/2$, all majority votes are correct and thus x is retrieved correctly. Recall that the foregoing is conditioned on $\sigma^j = b(x, s^j)$ for every $j \in [\ell]$, which in turn holds with probability $2^{-\ell} = (m+1)^{-1} = \Omega(\varepsilon^2/n) = 1/\mathrm{poly}(n)$. Thus, x is retrieved correctly with probability $1/\mathrm{poly}(n)$, and the theorem follows. $\square$

Digest. Looking at the proof of Theorem 7.7, we note that it actually refers to an arbitrary black-box $B_x(\cdot)$ that approximates $b(x, \cdot)$; specifically, in the case of Theorem 7.7 we used $B_x(r) \stackrel{\mathrm{def}}{=} B(f(x), r)$. In particular, the proof does not use the fact that we can verify the correctness of the preimage recovered by the described process. Thus, the proof actually establishes *the existence of a* $\mathrm{poly}(n/\varepsilon)$*-time oracle machine that, for every* $x \in \{0,1\}^n$*, given oracle access to any* $B_x : \{0,1\}^n \to \{0,1\}$ *satisfying*

$$\mathsf{Pr}_{r\in\{0,1\}^n}[B_x(r) = b(x, r)] \geq \frac{1}{2} + \varepsilon \tag{7.6}$$

outputs x *with probability at least* $\mathrm{poly}(\varepsilon/n)$. Specifically, x is output with probability at least $p \stackrel{\mathrm{def}}{=} \Omega(\varepsilon^2/n)$. Noting that x is merely a string for which Eq. (7.6) holds, it follows that the number of strings that satisfy Eq. (7.6) is at most $1/p$. Furthermore, by iterating the foregoing procedure for $\widetilde{O}(1/p)$ times we can obtain all these strings (see Exercise 7.7).

Theorem 7.8 (Theorem 7.7, revisited): *There exists a probabilistic oracle machine that, given parameters* n, ε *and oracle access to any function* $B : \{0,1\}^n \to \{0,1\}$*, halts after* $\mathrm{poly}(n/\varepsilon)$ *steps and with probability at least* $1/2$ *outputs a list of all strings* $x \in \{0,1\}^n$ *that satisfy*

$$\mathsf{Pr}_{r\in\{0,1\}^n}[B(r) = b(x, r)] \geq \frac{1}{2} + \varepsilon,$$

where $b(x, r)$ *denotes the inner-product mod 2 of* x *and* r.

This machine can be modified such that, with high probability, its output list does not include any string x such that $\mathsf{Pr}_{r\in\{0,1\}^n}[B(r) = b(x, r)] < \frac{1}{2} + \frac{\varepsilon}{2}$.

Theorem 7.8 means that if given some information about x it is hard to recover x, then given the same information and a random r it is hard to predict $b(x, r)$. This assertion is proved by the counter-positive (see Exercise 7.14). Indeed, the foregoing statement is in the spirit of Theorem 7.7 itself, except that it refers to any "information about x" (rather than to the value $f(x)$). To demonstrate the point, let us rephrase the foregoing statement as follows: *for every randomized process* Π*, if given* s *it is hard to obtain* $\Pi(s)$ *then given* s *and a random* r *it is hard to predict* $b(\Pi(s), r)$.[7]

A coding theory perspective. Theorem 7.8 can be viewed as a list decoding procedure for the Hadamard Code, where the Hadamard encoding of a string $x \in \{0,1\}^n$ is the 2^n-bit long string containing $b(x,r)$ for every $r \in \{0,1\}^n$. In contrast to *standard decoding* in which the task is recovering the unique information that is encoded in the codeword that is closest to the given string, in list decoding the task is recovering all strings having encoding that is at a specified distance from the given string.[8] We mention that list decoding is applicable and valuable in the case that the specified distance does not allow for unique decoding (i.e., the specified distance is greater than half the distance of the code).

Applications of hard-core predicates. Turning back to hard-core predicates, we mention that they play a central role in the construction of general-purpose pseudorandom generators (see Section 8.2), commitment schemes and zero-knowledge proofs (see Sections 9.2.2 and C.4.3), and encryption schemes (see Appendix C.5).

7.1.4 Reflections on hardness amplification

Let us take notice that something truly amazing happens in Theorems 7.5 and 7.7. We are not talking merely of using an assumption to derive some conclusion; this is common practice in Mathematics and Science (and was indeed done several times in previous chapters, starting with Theorem 2.28). The thing that is special about Theorems 7.5 and 7.7 (and we shall see more of this in Section 7.2 as well as in Sections 8.2 and 8.3) is that a relatively mild intractability assumption is shown to imply a stronger intractability result.

This strengthening of an intractability phenomenon (a.k.a hardness amplification) takes place while we admit that we do not understand the intractability phenomenon (because we do not understand the nature of efficient computation). Nevertheless, hardness amplification is enabled by the use of the counter-positive, which in this case is called a reducibility argument. At this point things look less miraculous: a reducibility argument calls for the design of a procedure (i.e., a reduction) and a probabilistic analysis of its behavior. The design and analysis of such procedures may not be easy, but it is certainly within the standard expertise of computer science. The fact that hardness amplification is achieved via this counter-positive is best represented in the statement of Theorem 7.8.

[7] Indeed, Theorem 7.7 is obtained as a special case by letting $\Pi(s)$ be uniformly distributed in $f^{-1}(s)$.

[8] Further discussion of error-correcting codes and list-decoding is provided in Appendix E.1.

7.2 Hard Problems in E

As in Section 7.1, we start with the assumption $\mathcal{P} \neq \mathcal{NP}$ and seek to use it to our benefit. Again, we shall actually use a seemingly stronger assumption; here the strengthening is in requiring *worst-case* hardness with respect to *non-uniform* models of computation (rather than average-case hardness with respect to the standard uniform model). Specifically, we shall assume that $\mathcal{NP}$ cannot be solved by (non-uniform) families of polynomial-size circuits; that is, $\mathcal{NP}$ is not contained in $\mathcal{P}/\mathrm{poly}$ (even not infinitely often).

Our goal is to transform this worst-case assumption into an average-case condition, which is useful for our applications. Since the transformation will not yield a problem in $\mathcal{NP}$ but rather one in $\mathcal{E}$, we might as well take the seemingly weaker assumption by which $\mathcal{E}$ is not contained in $\mathcal{P}/\mathrm{poly}$ (see Exercise 7.9). That is, our starting point is actually that *there exists an exponential-time solvable decision problem such that any family of polynomial-size circuit fails to solve it correctly on all but finitely many input lengths.*[9]

A different perspective on our assumption is provided by the fact that $\mathcal{E}$ contains problems that cannot be solved in polynomial-time (cf.. Section 4.2.1). The current assumption goes beyond this fact by postulating the failure of non-uniform polynomial-time machines rather than the failure of (uniform) polynomial-time machines.

Recall that our goal is to obtain a predicate (i.e., a decision problem) that is computable in exponential-time but is inapproximable by polynomial-size circuits. For sake of later developments, we formulate a general notion of inapproximability.

Definition 7.9 (inapproximability, a general formulation): *We say that $f : \{0,1\}^* \to \{0,1\}$ is (S,ρ)-*inapproximable *if for every family of S-size circuits $\{C_n\}_{n\in\mathbb{N}}$ and all sufficiently large n it holds that*

$$\mathsf{Pr}[C_n(U_n) \neq f(U_n)] \geq \frac{\rho(n)}{2} \tag{7.7}$$

*We say that f is T-*inapproximable *if it is $(T, 1-(1/T))$-inapproximable.*

We chose the specific form of Eq. (7.7) such that the "level of inapproximability" represented by the parameter ρ will range in $(0,1)$ and increase with the value of ρ. Specifically, (almost-everywhere) *worst-case* hardness for circuits of size S is represented by (S,ρ)-inapproximability with $\rho(n) = 2^{-n+1}$ (i.e., in this case $\mathsf{Pr}[C(U_n) \neq f(U_n)] \geq 2^{-n}$ for every circuit C_n of size $S(n)$). On the other hand, no predicate can be (S,ρ)-inapproximable for $\rho(n) = 1 - 2^{-n}$ even with $S(n) = O(n)$ (i.e., $\mathsf{Pr}[C(U_n) = f(U_n)] \geq 0.5 + 2^{-n-1}$ holds for some linear-size circuit; see Exercise 7.10).

We note that Eq. (7.7) can be interpreted as an upper-bound on the *correlation* of each adequate circuit with f (i.e., Eq. (7.7) is equivalent to $\mathsf{E}[\chi(C(U_n), f(U_n))] \leq$

[9]Note that our starting point is actually stronger than assuming the existence of a function f in $\mathcal{E} \setminus \mathcal{P}/\mathrm{poly}$. Such an assumption would mean that any family of polynomial-size circuit fails to compute f correctly on infinitely many input lengths, whereas our starting point postulates failures on all but finitely many lengths.

$1 - \rho(n)$, where $\chi(\sigma,\tau) = 1$ if $\sigma = \tau$ and $\chi(\sigma,\tau) = -1$ otherwise).[10] Thus, T-inapproximability means that no family of size T circuits can correlate f better than $1/T$.

We note that the existence of a non-uniformly hard one-way function (as in Definition 7.3) implies the existence of an exponential-time computable predicate that is T-inapproximable for every polynomial T. (For details see Exercise 7.21.) However, our goal in this section is to establish this conclusion under a seemingly weaker assumption.

On almost everywhere hardness. We highlight the fact that both our assumptions and conclusions refer to *almost everywhere* hardness. For example, our starting point is not merely that $\mathcal{E}$ is not contained in $\mathcal{P}/\mathrm{poly}$ (or in other circuit size classes to be discussed), but rather that this is the case almost everywhere. Note that by saying that f has circuit complexity exceeding S, we merely mean that *there are infinitely many n's* such that no circuit of size $S(n)$ can compute f correctly on all inputs of length n. In contrast, by saying that f has circuit complexity exceeding S almost everywhere, we mean that *for all but finite many n's* no circuit of size $S(n)$ can computes f correctly on all inputs of length n. (Indeed, it is not known whether an "infinitely often" type of hardness implies a corresponding "almost everywhere" hardness.)

The class $\mathcal{E}$. Recall that $\mathcal{E}$ denote the class of exponential-time solvable decision problems (equivalently, exponential-time computable Boolean predicates); that is, $\mathcal{E} = \cup_\varepsilon \mathrm{DTIME}(t_\varepsilon)$, where $t_\varepsilon(n) \stackrel{\mathrm{def}}{=} 2^{\varepsilon n}$.

The rest of this section. We start (in Section 7.2.1) with a treatment of assumptions and hardness amplification regarding polynomial-size circuits, which suffice for non-trivial derandomization of $\mathcal{BPP}$. We then turn (in Section 7.2.2) to assumptions and hardness amplification regarding exponential-size circuits, which yield a "full" derandomization of $\mathcal{BPP}$ (i.e., $\mathcal{BPP} = \mathcal{P}$). In fact, both sections contain material that is applicable to various other circuit-size bounds, but the motivational focus is as stated.

Teaching note: Section 7.2.2 is advanced material, which is best left for independent reading. Furthermore, for one of the central results (i.e., Lemma 7.23) only an outline is provided and the interested reader is referred to the original paper [125].

7.2.1 Amplification wrt polynomial-size circuits

Our goal here is to prove the following result.

Theorem 7.10 *Suppose that for every polynomial p there exists a problem in $\mathcal{E}$ having circuit complexity that is almost-everywhere greater than p. Then there exist polynomial-inapproximable Boolean functions in $\mathcal{E}$; that is, for every polynomial p there exists a p-inapproximable Boolean function in $\mathcal{E}$.*

[10] Indeed, $\mathsf{E}[\chi(X,Y)] = \mathsf{Pr}[X=Y] - \mathsf{Pr}[X\neq Y] = 1 - 2\mathsf{Pr}[X\neq Y]$.

Theorem 7.10 is used towards deriving a meaningful derandomization of $\mathcal{BPP}$ under the aforementioned assumption (see Part 2 of Theorem 8.19). We present two proofs of Theorem 7.10. The first proof proceeds in two steps:

1. Starting from the worst-case hypothesis, we first establish some mild level of average-case hardness (i.e., a mild level of inapproximability). Specifically, we show that for every polynomial p there exists a problem in $\mathcal{E}$ that is (p, ε)-inapproximable for $\varepsilon(n) = 1/n^3$.

2. Using the foregoing mild level of inapproximability, we obtain the desired strong level of inapproximability (i.e., p'-inapproximability for every polynomial p'). Specifically, for every two polynomials p_1 and p_2, we prove that *if the function f is $(p_1, 1/p_2)$-inapproximable, then the function $F(x_1, ..., x_{t(n)}) = \oplus_{i=1}^{t(n)} f(x_i)$, where $t(n) = n \cdot p_2(n)$ and $x_1, ..., x_{t(n)} \in \{0,1\}^n$, is p'-inapproximable for $p'(t(n) \cdot n) = p_1(n)^{\Omega(1)}/\text{poly}(t(n))$.* This claim is known as Yao's XOR Lemma and its proof is far more complex than the proof of its information theoretic analogue (discussed at the beginning of §7.2.1.2).

The second proof of Theorem 7.10 consists of showing that the construction employed in the first step, when composed with Theorem 7.8, actually yields the desired end result. This proof will uncover a connection between hardness amplification and coding theory. Our presentation will thus proceed in three corresponding steps (presented in §7.2.1.1-7.2.1.3, and schematically depicted in Figure 7.2).

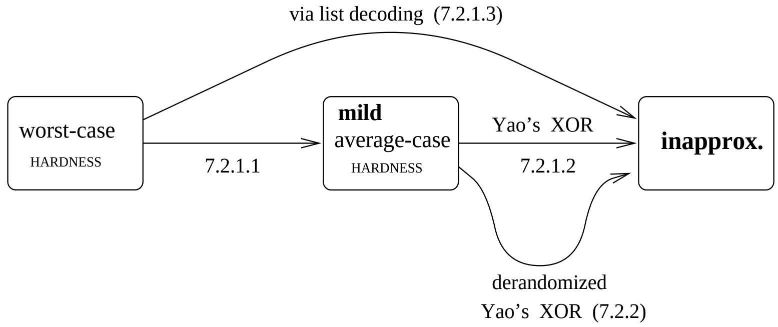


Figure 7.2: Proofs of hardness amplification: organization

7.2.1.1 From worst-case hardness to mild average-case hardness

The transformation of worst-case hardness into average-case hardness (even in a mild sense) is indeed remarkable. Note that worst-case hardness may be due to a relatively small number of instances, whereas even mild forms of average-case hardness refer to a very large number of possible instances.[11] In other words, we should transform hardness that may occur on a negligible fraction of the instances

[11] Indeed, worst-case hardness with respect to polynomial-size circuits cannot be due to a polynomial number of instances, because a polynomial number of instances can be hard-wired into

into hardness that occurs on a noticeable fraction of the instances. Intuitively, we should "spread" the hardness of few instances (of the original problem) over all (or most) instances (of the transformed problem). The counter-positive view is that computing the value of typical instances of the transformed problem should enable solving the original problem on every instance.

The aforementioned transformation is based on the *self-correction paradigm*, to be reviewed first. The paradigm refers to functions g that can be evaluated at any desired point by using the value of g at a few random points, where each of these points is uniformly distributed in the function's domain (but indeed the points are not independently distributed). The key observation is that if $g(x)$ can be reconstructed based on the value of g at t such random points, then such a reconstruction can tolerate a $1/3t$ fraction of errors (regarding the values of g). Thus, if we can correctly obtain the value of g on all but at most a $1/3t$ fraction of its domain, then we can probabilistically recover the correct value of g at any point with very high probability. It follows that if no probabilistic polynomial-time algorithm can correctly compute g *in the worst-case sense*, then every probabilistic polynomial-time algorithm must fail to correctly compute g *on more than a* $1/3t$ *fraction of its domain.*

The archetypical example of a self-correctable function is any m-variate polynomial of individual degree d over a finite field F such that $|F| > dm + 1$. The value of such a polynomial at any desired point x can be recovered based on the values of $dm + 1$ points (other than x) that reside on a random line that passes through x. Note that each of these points is uniformly distributed in F^m, which is the function's domain. (For details, see Exercise 7.11.)

Recall that we are given an arbitrary function $f \in \mathcal{E}$ that is hard to compute in the worst-case. Needless to say, this function is not necessarily self-correctable (based on relatively few points), but it can be extended into such a function. Specifically, we extend $f : [N] \to \{0,1\}$ (viewed as $f : [N^{1/m}]^m \to \{0,1\}$) to an m-variate polynomial of individual degree d over a finite field F such that $|F| > dm+1$ and $(d+1)^m = N$. Intuitively, in terms of worst-case complexity, the extended function is at least as hard as f, which means that it is hard (in the worst-case). The point is that the extended function is self-correctable and thus its worst-case hardness implies that it must be at least mildly hard in the average-case. Details follow.

Construction 7.11 (multi-variate extension)[12]: *For any function* $f_n : \{0,1\}^n \to \{0,1\}$*, a finite field* F*, a set* $H \subset F$ *and an integer* m *such that* $|H|^m = 2^n$ *and* $|F| > (|H|-1)m + 1$*, we consider the function* $\hat{f}_n : F^m \to F$ *defined as the* m*-variate polynomial of individual degree* $|H|-1$ *that extends* $f_n : H^m \to \{0,1\}$*. That*

such circuits. Still, for all we know, worst-case hardness may be due to a small super-polynomial number of instances (e.g., $n^{\log_2 n}$ instances). In contrast, even mild forms of average-case hardness must be due to an exponential number of instances (i.e., $2^n/\mathrm{poly}(n)$ instances).

[12]The algebraic fact underlying this construction is that for any function $f : H^m \to F$ there exists a unique m-variate polynomial $\hat{f} : F^m \to F$ of individual degree $|H|-1$ such that for every $x \in H^m$ it holds that $\hat{f}(x) = f(x)$. This polynomial is called a multi-variate polynomial extension of f, and it can be found in $\mathrm{poly}(|H|^m \log |F|)$-time. For details, see Exercise 7.12.

is, we identify $\{0,1\}^n$ *with* H^m*, and define* $\hat{f}_n$ *as the unique* m*-variate polynomial of individual degree* $|H|-1$ *that satisfies* $\hat{f}_n(x) = f_n(x)$ *for every* $x \in H^m$*, where we view* $\{0,1\}$ *as a subset of* F*.*

Note that $\hat{f}_n$ can be evaluated at any desired point, by evaluating f_n on its entire domain, and determining the unique m-variate polynomial of individual degree $|H|-1$ that agrees with f_n on H^m (see Exercise 7.12). Thus, for $f : \{0,1\}^* \to \{0,1\}$ in $\mathcal{E}$, the corresponding $\hat{f}$ (defined by separately extending the restriction of f to each input length) is also in $\mathcal{E}$. For the sake of preserving various complexity measures, we wish to have $|F^m| = \text{poly}(2^n)$, which leads to setting $m = n/\log_2 n$ (yielding $|H| = n$ and $|F| = \text{poly}(n)$). In particular, in this case $\hat{f}_n$ is defined over strings of length $O(n)$. The mild average-case hardness of $\hat{f}$ follows by the forgoing discussion. In fact, we state and prove a more general result.

Theorem 7.12 *Suppose that there exists a Boolean function* f *in* $\mathcal{E}$ *having circuit complexity that is almost-everywhere greater than* S*. Then, there exists an exponential-time computable function* $\hat{f} : \{0,1\}^* \to \{0,1\}^*$ *such that* $|\hat{f}(x)| \le |x|$ *and for every family of circuit* $\{C'_{n'}\}_{n' \in \mathbb{N}}$ *of size* $S'(n') = S(n'/O(1))/\text{poly}(n')$ *it holds that* $\Pr[C'_{n'}(U_{n'}) \ne \hat{f}(U_{n'})] > (1/n')^2$*. Furthermore,* $\hat{f}$ *does not depend on* S*.*

Theorem 7.12 seems to complete the first step of the proof of Theorem 7.10, except that we desire a Boolean function rather than a function that merely does not stretch its input. The extra step of obtaining a Boolean function that is $(\text{poly}(n), n^{-3})$-inapproximable is taken in Exercise 7.13.[13] Essentially, if $\hat{f}$ is hard to compute on a noticeable fraction of its inputs then the Boolean predicate that on input (x,i) returns the i^{th} bit of $\hat{f}(x)$ must be mildly inapproximable.

Proof Sketch: Given f as in the hypothesis and for every $n \in \mathbb{N}$, we consider the restriction of f to $\{0,1\}^n$, denoted f_n, and apply Construction 7.11 to it, while using $m = n/\log n$, $|H| = n$ and $n^2 < |F| = \text{poly}(n)$. Recall that the resulting function $\hat{f}_n$ maps strings of length $n' = \log_2 |F^m| = O(n)$ to strings of length $\log_2 |F| = O(\log n)$. Following the foregoing discussion, we shall show that circuits that approximate $\hat{f}_n$ too well yield circuits that compute f_n correctly on each input. Using the hypothesis regarding the size of the latter, we shall derive a lower-bound on the size of the former. The actual (reducibility) argument proceeds as follows. We fix an arbitrary circuit $C'_{n'}$ that satisfies

$$\Pr[C'_{n'}(U_{n'}) = \hat{f}_n(U_{n'})] \ge 1 - (1/n')^2 > 1 - (1/3t), \tag{7.8}$$

where $t \stackrel{\text{def}}{=} (|H|-1)m + 1 = o(n^2)$ exceeds the total degree of $\hat{f}_n$. Using the self-correction feature of $\hat{f}_n$, we observe that by making t oracle calls to $C'_{n'}$ we can probabilistically recover the value of ($\hat{f}_n$ and thus of) f_n on each input, with probability at least 2/3. Using error-reduction and (non-uniform) derandomization as in

[13] A quantitatively stronger bound can be obtained by noting that the proof of Theorem 7.12 actually establishes an error lower-bound of $\Omega((\log n')/(n')^2)$ and that $|\hat{f}(x)| = O(\log |x|)$.

the proof of Theorem 6.3,[14] we obtain a circuit of size $n^3 \cdot |C'_{n'}|$ that computes f_n. By the hypothesis $n^3 \cdot |C'_{n'}| > S(n)$, and so $|C'_{n'}| > S(n'/O(1))/\mathrm{poly}(n')$. Recalling that $C'_{n'}$ is an arbitrary circuit that satisfies Eq. (7.8), the theorem follows. □

Digest. The proof of Theorem 7.12 is actually a worst-case to average-case reduction. That is, the proof consists of a self-correction procedure that allows for the evaluation of f at any desired n-bit long point, using oracle calls to any circuit that computes $\hat{f}$ correctly on a $1-(1/n')^2$ fraction of the n'-bit long inputs. We recall that if $f \in \mathcal{E}$ then $\hat{f} \in \mathcal{E}$, but we do not know how to preserve the complexity of f in case it is in $\mathcal{NP}$. (Various indications to the difficulty of a worst-case to average-case reduction for $\mathcal{NP}$ are known; see, e.g., [40].)

We mention that the ideas underlying the proof of Theorem 7.12 have been applied in a large variety of settings. For example, we shall see applications of the self-correction paradigm in §9.3.2.1 and in §9.3.2.2. Furthermore, in §9.3.2.2 we shall re-encounter the very same multi-variate extension used in the proof of Theorem 7.12.

7.2.1.2 Yao's XOR Lemma

Having obtained a mildly inapproximable predicate, we wish to obtain a strongly inapproximable one. The information theoretic context provides an appealing suggestion: Suppose that X is a Boolean random variable (representing the mild inapproximability of the aforementioned predicate) that equals 1 with probability ε. Then XORing the outcome of n/ε independent samples of X yields a bit that equals 1 with probability $0.5 \pm \exp(-\Omega(n))$. It is tempting to think that the same should happen in the computational setting. That is, if f is hard to approximate correctly with probability exceeding $1-\varepsilon$ then XORing the output of f on n/ε non-overlapping parts of the input should yield a predicate that is hard to approximate correctly with probability that is non-negligibly higher than $1/2$. The latter assertion turns out to be correct, but (even more than in Section 7.1.2) the proof of the computational phenomenon is considerably more complex than the analysis of the information theoretic analogue.

Theorem 7.13 (Yao's XOR Lemma): *There exist a universal constant $c > 0$ such that the following holds. If, for some polynomials p_1 and p_2, the Boolean function f is $(p_1, 1/p_2)$-inapproximable, then the function $F(x_1, ..., x_{t(n)}) = \oplus_{i=1}^{t(n)} f(x_i)$, where $t(n) = n \cdot p_2(n)$ and $x_1, ..., x_{t(n)} \in \{0,1\}^n$, is p'-inapproximable for $p'(t(n) \cdot n) = p_1(n)^c / t(n)^{1/c}$. Furthermore, the claim holds also if the polynomials p_1 and p_2 are replaced by any integer functions.*

[14] First, we apply the foregoing probabilistic procedure $O(n)$ times and take a majority vote. This yields a probabilistic procedure that, on input $x \in \{0,1\}^n$, invokes $C'_{n'}$ for $o(n^3)$ times and computes $f_n(x)$ correctly with probability greater than $1-2^{-n}$. Finally, we just fix a sequence of random choices that is good for all 2^n possible inputs, and obtain a circuit of size $n^3 \cdot |C'_{n'}|$ that computes f_n correctly on every n-bit input.

Combining Theorem 7.12 (and Exercise 7.13), and Theorem 7.13, we obtain a proof of Theorem 7.10. (Recall that an alternative proof is presented in §7.2.1.3.)

We note that proving Theorem 7.13 seems more difficult than proving Theorem 7.5 (i.e., the amplification of one-way functions), due to two issues. Firstly, unlike in Theorem 7.5, the computational problems are not in $\mathcal{PC}$ and thus we cannot efficiently recognize correct solutions to them. Secondly, unlike in Theorem 7.5, solutions to instances of the transformed problem do not correspond of the concatenation of solutions for the original instances, but are rather a function of the latter that losses almost all the information about the latter. The proof of Theorem 7.13 presented next deals with each of these two difficulties separately.

Several different proofs of Theorem 7.13 are known. As just stated, the proof that we present is conceptually appealing because it deal separately with two unrelated difficulties. Furthermore, this proof benefits most from the material already presented in Section 7.1. The proof proceeds in two steps:

1. First we prove that the corresponding "direct product" function $P(x_1, ..., x_{t(n)}) = (f(x_1), ..., f(x_{t(n)}))$ is difficult to compute in a strong average-case sense.
2. Next we establish the desired result by an application of Theorem 7.8.

Thus, given Theorem 7.8, our main focus is on the first step, which is of independent interest (and is thus generalized from Boolean functions to arbitrary ones).

Theorem 7.14 (The Direct Product Lemma): *Let p_1 and p_2 be two polynomials, and suppose that $f : \{0,1\}^* \to \{0,1\}^*$ is such that for every family of p_1-size circuits, $\{C_n\}_{n\in\mathbb{N}}$, and all sufficiently large $n \in \mathbb{N}$, it holds that $\mathsf{Pr}[C_n(U_n) \neq f(U_n)] > 1/p_2(n)$. Let $P(x_1, ..., x_{t(n)}) = (f(x_1), ..., f(x_{t(n)}))$, where $x_1, ..., x_{t(n)} \in \{0,1\}^n$ and $t(n) = n \cdot p_2(n)$. Then, for any $\varepsilon' : \mathbb{N} \to [0,1]$, setting p' such that $p'(t(n) \cdot n) = p_1(n)/\mathrm{poly}(t(n)/\varepsilon'(t(n) \cdot n))$, it holds that every family of p'-size circuits, $\{C'_m\}_{m\in\mathbb{N}}$, satisfies $\mathsf{Pr}[C'_m(U_m) = P(U_m)] < \varepsilon'(m)$. Furthermore, the claim holds also if the polynomials p_1 and p_2 are replaced by any integer functions.*

In particular, for an adequate constant $c > 0$, selecting $\varepsilon'(t(n) \cdot n) = p_1(n)^{-c}$, we obtain $p'(t(n) \cdot n) = p_1(n)^c / t(n)^{1/c}$, and so $\varepsilon'(m) \leq 1/p'(m)$.

Deriving Theorem 7.13 from Theorem 7.14. Theorem 7.13 follows from Theorem 7.14 by considering the function $P'(x_1, ..., x_{t(n)}, r) = b(f(x_1) \cdots f(x_{t(n)}), r)$, where f is a Boolean function, $r \in \{0,1\}^{t(n)}$, and $b(y, r)$ is the inner-product modulo 2 of the $t(n)$-bit long strings y and r. Note that, for the corresponding P, we have $P'(x_1, ..., x_{t(n)}, r) \equiv b(P(x_1, ..., x_{t(n)}), r)$, whereas $F(x_1, ..., x_{t(n)}) = P'(x_1, ..., x_{t(n)}, 1^{t(n)})$. Intuitively, the inapproximability of P' should follow from the strong average-case hardness of P (via Theorem 7.8), whereas it should be possible to reduce the approximation of P' to the approximation of F (and thus derive the desired inapproximability of F). Indeed, this intuition does not fail, but detailing the argument seems a bit cumbersome (and so we only provide the clues here). Assuming that f is $(p_1, 1/p_2)$-inapproximable, we first apply Theorem 7.14 (with $\varepsilon'(t(n) \cdot n) = p_1(n)^{-c}$) and then apply Theorem 7.8 (see Exercise 7.14), inferring that P' is p'-inapproximable for $p'(t(n) \cdot n) = p_1(n)^{\Omega(1)}/\mathrm{poly}(t(n))$. The less obvious part of the argument is reducing the approximation of P' to the approximation of F. The key observation is that

$$P'(x_1, ..., x_{t(n)}, r) = F(z_1, ..., z_{t(n)}) \oplus \bigoplus_{i:r_i=0} f(z_i) \tag{7.9}$$

where $z_i = x_i$ if $r_i = 1$ and is an arbitrary n-bit long string otherwise. Now, if somebody provides us with samples of the distribution $(U_n, f(U_n))$, then we can use these samples in the role of the pairs $(z_i, f(z_i))$ for the indices i that satisfy $r_i = 0$. Considering a best choice of such samples (i.e., one for which we obtain the best approximation of P'), we obtain a circuit that approximates P' (by using a circuit that approximates F and the said choices of samples). (The details are left for Exercise 7.16.) Theorem 7.13 follows.

Proving Theorem 7.14. Note that Theorem 7.14 is closely related to Theorem 7.5; see Exercise 7.17 for details. This suggests employing an analogous proof strategy; that is, converting circuits that violate the theorem's conclusion into circuits that violate the theorem's hypothesis. We note, however, that things were much simpler in the context of Theorem 7.5: there we could (efficiently) check whether or not a value contained in the output of the circuit that solves the direct-product problem constitutes a correct answer for the corresponding instance of the basic problem. Lacking such an ability in the current context, we shall have to use such values more carefully. Loosely speaking, we shall take a weighted majority vote among various answers, where the weights reflect our confidence in the correctness of the various answers.

We establish Theorem 7.14 by applying the following lemma that provides quantitative bounds on the feasibility of computing the direct product of two functions. In this lemma, $\{Y_m\}_{m\in\mathbb{N}}$ and $\{Z_m\}_{m\in\mathbb{N}}$ are independent probability ensembles such that $Y_m, Z_m \in \{0,1\}^m$, and $X_n = (Y_{\ell(n)}, Z_{n-\ell(n)})$ for some function $\ell : \mathbb{N} \to \mathbb{N}$. The lemma refers to the success probability of computing the direct product function $F : \{0,1\}^* \to \{0,1\}^*$ defined by $F(yz) = (F_1(y), F_2(z))$, where $|y| = \ell(|yz|)$, when given bounds on the success probability of computing F_1 and F_2 (separately). Needless to say, these probability bounds refer to circuits of certain sizes. We stress that *the lemma is not symmetric with respect to the two functions*: *it guarantees a stronger* (and in fact lossless) *preservation of circuit sizes for one of the functions* (which is arbitrarily chosen to be F_1).

Lemma 7.15 (Direct Product, a quantitative two argument version): *For $\{Y_m\}$, $\{Z_m\}$, F_1, F_2, ℓ, $\{X_n\}$ and F as in the foregoing, let $\rho_1(\cdot)$ be an upper-bound on the success probability of $s_1(\cdot)$-size circuits in computing F_1 over $\{Y_m\}$. That is, for every such circuit family $\{C_m\}$*

$$\Pr[C_m(Y_m) = F_1(Y_m)] \le \rho_1(m).$$

Likewise, suppose that $\rho_2(\cdot)$ is an upper-bound on the probability that $s_2(\cdot)$-size circuits compute F_2 over $\{Z_m\}$. Then, for every function $\varepsilon : \mathbb{N} \to \mathbb{R}$, the function

ρ defined as

$$\rho(n) \stackrel{\rm def}{=} \rho_1(\ell(n)) \cdot \rho_2(n-\ell(n)) + \varepsilon(n)$$

is an upper-bound on the probability that families of $s(\cdot)$-size circuits correctly compute F over $\{X_n\}$, where

$$s(n) \stackrel{\rm def}{=} \min\left\{ s_1(\ell(n))\ ,\ \frac{s_2(n-\ell(n))}{\text{poly}(n/\varepsilon(n))} \right\}.$$

Theorem 7.14 is derived from Lemma 7.15 by using a *careful induction*, which capitalizes on the highly quantitative form of Lemma 7.15 and in particular on the fact that no loss is incurred for one of the two functions that are used. We first detail this argument, and next establish Lemma 7.15 itself.

Deriving Theorem 7.14 from Lemma 7.15. We write $P(x_1, x_2, ..., x_{t(n)})$ as $P^{(t(n))}(x_1, x_2, ..., x_{t(n)})$, where $P^{(i)}(x_1, ..., x_i) = (f(x_1), ..., f(x_i))$ and $P^{(i)}(x_1, ..., x_i) \equiv (P^{(i-1)}(x_1, ..., x_{i-1}), f(x_i))$. For any function ε, *we shall prove by induction on i that circuits of size $s(n) = p_1(n)/\text{poly}(t(n)/\varepsilon(n))$ cannot compute $P^{(i)}(U_{i\cdot n})$ with success probability greater than* $(1-(1/p_2(n))^i + (i-1)\cdot\varepsilon(n)$, where p_1 and p_2 are as in Theorem 7.14. Thus, no $s(n)$-size circuit can compute $P^{(t(n))}(U_{t(n)\cdot n})$ with success probability greater than $(1-(1/p_2(n))^{t(n)} + (t(n)-1)\cdot\varepsilon(n) = \exp(-n) + (t(n)-1)\cdot\varepsilon(n)$. Recalling that this is established for any function ε, Theorem 7.14 follows (by using $\varepsilon(n) = \varepsilon'(t(n)\cdot n)/t(n)$, and observing that the setting $s(n) = p'(t(n)\cdot n)$ satisfies $s(n) = p_1(n)/\text{poly}(t(n)/\varepsilon(n))$).

Turning to the induction itself, we first note that its basis (i.e., $i = 1$) is guaranteed by the theorem's hypothesis (i.e., the hypothesis of Theorem 7.14 regarding f). The induction step (i.e., from i to $i+1$) will be proved by using Lemma 7.15 with $F_1 = P^{(i)}$ and $F_2 = f$, along with the parameter setting $\rho_1^{(i)}(i\cdot n) = (1-(1/p_2(n))^i + (i-1)\cdot\varepsilon(n)$, $s_1^{(i)}(i\cdot n) = s(n)$, $\rho_2^{(i)}(n) = 1-(1/p_2(n))$ and $s_2^{(i)}(n) = \text{poly}(n/\varepsilon(n))\cdot s(n) = p_1(n)$. Details follow.

Note that the induction hypothesis (regarding $P^{(i)}$) implies that F_1 satisfies the hypothesis of Lemma 7.15 (w.r.t size $s_1^{(i)}$ and success rate $\rho_1^{(i)}$), whereas the theorem's hypothesis regarding f implies that F_2 satisfies the hypothesis of Lemma 7.15 (w.r.t size $s_2^{(i)}$ and success rate $\rho_2^{(i)}$). Thus, $F = P^{(i+1)}$ satisfies the lemma's conclusion with respect to circuits of size $\min(s_1^{(i)}(i\cdot n), s_2^{(i)}(n)/\text{poly}(n/\varepsilon(n))) = s(n)$ and success rate $\rho_1^{(i)}(i\cdot n)\cdot\rho_2^{(i)}(n) + \varepsilon(n)$ which is upper-bounded by $(1-(1/p_2(n))^{i+1} + i\cdot\varepsilon(n)$. This completes the induction step.

We stress the fact that we used induction for a non-constant number of steps, and that this was enabled by the highly quantitative form of the inductive claim and the small loss incurred by the inductive step. Specifically, the size bound did not decrease during the induction (although we could afford a small additive loss in each step, but not a constant factor loss). Likewise, the success rate suffered an additive increase of $\varepsilon(n)$ in each step, which was accommodated by the inductive claim. Thus, *assuming the correctness of Lemma 7.15, we have established Theorem 7.14.*
□

Proof of Lemma 7.15: Proceeding (as usual) by the contrapositive, we consider a family of $s(\cdot)$-size circuits $\{C_n\}_{n\in\mathbb{N}}$ that violates the lemma's conclusion; that is, $\Pr[C_n(X_n) = F(X_n)] > \rho(n)$. We will show how to use such circuits in order to obtain either circuits that violate the lemma's hypothesis regarding F_1 or circuits that violate the lemma's hypothesis regarding F_2. Towards this end, it is instructive to write the success probability of C_n in a conditional form, while denoting the i^{th} output of $C_n(x)$ by $C_n(x)_i$ (i.e., $C_n(x) = (C_n(x)_1, C_n(x)_2)$):

$$\begin{aligned}
&\Pr[C_n(Y_{\ell(n)}, Z_{n-\ell(n)}) = F(Y_{\ell(n)}, Z_{n-\ell(n)})] \\
&\quad = \Pr[C_n(Y_{\ell(n)}, Z_{n-\ell(n)})_1 = F_1(Y_{\ell(n)})] \\
&\qquad \cdot \Pr[C_n(Y_{\ell(n)}, Z_{n-\ell(n)})_2 = F_2(Z_{n-\ell(n)}) \mid C_n(Y_{\ell(n)}, Z_{n-\ell(n)})_1 = F_1(Y_{\ell(n)})].
\end{aligned}$$

The basic idea is that if the first factor is greater than $\rho_1(\ell(n))$ then we immediately derive a circuit (i.e., $C'_n(y) = C_n(y, Z_{n-\ell(n)})_1$) contradicting the lemma's hypothesis regarding F_1, whereas if the second factor is significantly greater than $\rho_2(n - \ell(n))$ then we can obtain a circuit contradicting the lemma's hypothesis regarding F_2. The treatment of the latter case is indeed not obvious. The idea is that a sufficiently large sample of $(Y_{\ell(n)}, F_1(Y_{\ell(n)}))$, which may be hard-wired into the circuit, allows using the conditional probability space (in such a circuit) towards an attempt to approximate F_2. That is, on input z, we select uniformly a string y satisfying $C_n(y,z)_1 = F_1(y)$ (from the aforementioned sample), and output $C_n(y,z)_2$. For a fixed z, sampling of the conditional space (i.e., y's satisfying $C_n(y,z)_1 = F_1(y)$) is possible provided that $\Pr[C_n(Y_{\ell(n)}, z)_1 = F_1(Y_{\ell(n)})]$ holds with noticeable probability. The last caveat motivates a separate treatment of z's having a noticeable value of $\Pr[C_n(Y_{\ell(n)}, z)_1 = F_1(Y_{\ell(n)})]$ and of the rest of z's (which are essentially ignored). Details follow.

Let us first simplify the notations by fixing a generic n and using the abbreviations $C = C_n$, $\varepsilon = \varepsilon(n)$, $\ell = \ell(n)$, $Y = Y_\ell$, and $Z = Y_{n-\ell}$. We call z good if $\Pr[C(Y,z)_1 = F_1(Y)] \geq \varepsilon/2$ and let G be the set of good z's. Next, rather than considering the event $C(Y,Z) = F(Y,Z)$, we consider the combined event $C(Y,Z) = F(Y,Z) \wedge Z \in G$, which occurs with almost the same probability (up to an additive error term of $\varepsilon/2$). This is the case because, for any $z \notin G$, it holds that

$$\Pr[C(Y,z) = F(Y,z)] \leq \Pr[C(Y,z)_1 = F_1(Y)] < \varepsilon/2$$

and thus z's that are not good do not contribute much to $\Pr[C(Y,Z) = F(Y,Z)]$; that is, $\Pr[C(Y,Z) = F(Y,Z) \wedge Z \in G]$ is lower-bounded by $\Pr[C(Y,Z) = F(Y,Z)] - \varepsilon/2$. Using $\Pr[C(Y,z) = F(Y,z)] > \rho(n) = \rho_1(\ell) \cdot \rho_2(n-\ell) + \varepsilon$, we have

$$\Pr[C(Y,Z) = F(Y,Z) \ \wedge \ Z \in G] > \rho_1(\ell) \cdot \rho_2(n-\ell) + \frac{\varepsilon}{2}. \tag{7.10}$$

We proceed according to the forgoing outline, first showing that if $\Pr[C(Y,Z)_1 = F_1(Y)] > \rho_1(\ell)$ then we immediately derive circuits violating the hypothesis concerning F_1. Actually, we prove something stronger (which we will actually need for the other case).

Claim 7.15.1: For every z, it holds that $\Pr[C(Y,z)_1 = F_1(Y)] \leq \rho_1(\ell)$.

Proof: Otherwise, using any $z \in \{0,1\}^{n-\ell}$ that satisfies $\Pr[C(Y,z)_1 = F_1(Y)] > \rho_1(\ell)$, we obtain a circuit $C'(y) \stackrel{\text{def}}{=} C(y,z)_1$ that contradicts the lemma's hypothesis concerning F_1. $\square$

Using Claim 7.15.1, we show how to obtain a circuit that violates the lemma's hypothesis concerning F_2, and doing so we complete the proof of the lemma.

Claim 7.15.2: There exists a circuit C'' of size $s_2(n-\ell)$ such that

$$\begin{aligned}\Pr[C''(Z)=F_2(Z)] &\geq \frac{\Pr[C(Y,Z)=F(Y,Z) \wedge Z\in G]}{\rho_1(\ell)} - \frac{\varepsilon}{2} \\ &> \rho_2(n-\ell)\end{aligned}$$

Proof: The second inequality is due to Eq. (7.10), and thus we focus on establishing the first inequality. We construct the circuit C'' as suggested in the foregoing outline. Specifically, we take a $\text{poly}(n/\varepsilon)$-large sample, denoted S, from the distribution $(Y, F_1(Y))$ and let $C''(z) \stackrel{\text{def}}{=} C(y,z)_2$, where (y,v) is a uniformly selected among the elements of S for which $C(y,z)_1 = v$ holds. Details follow.

Let m be a sufficiently large number that is upper-bounded by a polynomial in n/ε, and consider a random sequence of m pairs, generated by taking m independent samples from the distribution $(Y, F_1(Y))$. We stress that we do not assume here that such a sample, denoted S, can be produced by an efficient (uniform) algorithm (but, jumping ahead, we remark that such a sequence can be fixed non-uniformly). For each $z \in G \subseteq \{0,1\}^{n-\ell}$, we denote by S_z the set of pairs $(y,v) \in S$ for which $C(y,z)_1 = v$. Note that S_z is a random sample of the residual probability space defined by $(Y, F_1(Y))$ conditioned on $C(Y,z)_1 = F_1(Y)$. Also, with overwhelmingly high probability, $|S_z| = \Omega(n/\varepsilon^2)$, because $z \in G$ implies $\Pr[C(Y,z)_1 = F_1(Y)] \geq \varepsilon/2$ and $m = \Omega(n/\varepsilon^3)$.[15] Thus, for each $z \in G$, with overwhelming probability (taken over the choices of S), the sample S_z provides a good approximation to the conditional probability space.[16] In particular, with probability greater than $1-2^{-n}$, it holds that

$$\frac{|\{(y,v) \in S_z : C(y,z)_2 = F_2(z)\}|}{|S_z|} \geq \Pr[C(Y,z)_2 = F_2(z) \,|\, C(Y,z)_1 = F_1(Y)] - \frac{\varepsilon}{2}. \tag{7.11}$$

Thus, with positive probability, Eq. (7.11) holds for all $z \in G \subseteq \{0,1\}^{n-\ell}$. The circuit C'' computing F_2 is now defined as follows. The circuit will contain a set $S = \{(y_i, v_i) : i = 1, ..., m\}$ (i.e., S is "hard-wired" into the circuit C'') such that the following two conditions hold:

1. For every $i \in [m]$ it holds that $v_i = F_1(y_i)$.
2. For each good z the set $S_z = \{(y,v) \in S : C(y,z)_1 = v\}$ satisfies Eq. (7.11).

   (In particular, S_z is not empty for any good z.)

---

[15] Note that the expected size of S_z is $m \cdot \varepsilon/2 = \Omega(n/\varepsilon^2)$. Using Chernoff Bound, we get $\Pr_S[|S_z| < m\varepsilon/4] = \exp(-\Omega(n/\varepsilon^2)) < 2^{-n}$.

[16] For $T_z = \{y : C(y,z)_1 = F_1(y)\}$, we are interested in a sample S' of T_z such that $|\{y \in S' : C(y,z)_2 = F_2(z)\}|/|S'|$ approximates $\Pr[C(Y,z)_2 = F_2(z) \,|\, Y \in T_z]$ up-to an additive term of $\varepsilon/2$. Using Chernoff Bound again, we note that a random $S' \subset T_z$ of size $\Omega(n/\varepsilon^2)$ provides such an approximation with probability greater than $1-2^{-n}$.

On input z, the circuit C'' first determines the set S_z, by running C for m times and checking, for each $i = 1, ..., m$, whether or not $C(y_i, z) = v_i$. In case S_z is empty, the circuit returns an arbitrary value. Otherwise, the circuit selects uniformly a pair $(y, v) \in S_z$ and outputs $C(y, z)_2$. (The latter random choice can be eliminated by an averaging argument; see Exercise 7.15.) Using the definition of C'' and Eq. (7.11), we have:

$$\begin{aligned}
\Pr[C''(Z)=F_2(Z)] &\geq \sum_{z\in G} \Pr[Z=z] \cdot \Pr[C''(z)=F_2(z)] \\
&= \sum_{z\in G} \Pr[Z=z] \cdot \frac{|\{(y,v) \in S_z : C(y,z)_2 = F_2(z)\}|}{|S_z|} \\
&\geq \sum_{z\in G} \Pr[Z=z] \cdot \left(\Pr[C(Y,z)_2 = F_2(z) \mid C(Y,z)_1 = F_1(Y)] - \frac{\varepsilon}{2}\right) \\
&= \sum_{z\in G} \Pr[Z=z] \cdot \left(\frac{\Pr[C(Y,z)_2 = F_2(z) \wedge C(Y,z)_1 = F_1(Y)]}{\Pr[C(Y,z)_1 = F_1(Y)]} - \frac{\varepsilon}{2}\right)
\end{aligned}$$

Next, using Claim 7.15.1, we have:

$$\begin{aligned}
\Pr[C''(Z)=F_2(Z)] &\geq \left(\sum_{z\in G} \Pr[Z=z] \cdot \frac{\Pr[C(Y,z)=F(Y,z)]}{\rho_1(\ell)}\right) - \frac{\varepsilon}{2} \\
&= \frac{\Pr[C(Y,Z)=F(Y,Z) \wedge Z \in G]}{\rho_1(\ell)} - \frac{\varepsilon}{2}
\end{aligned}$$

Finally, using Eq. (7.10), the claim follows. □

This completes the proof of the lemma. ■

Comments. Firstly, we wish to call attention to the care with which an inductive argument needs to be carried out in the computational setting, especially when a non-constant number of inductive steps is concerned. Indeed, our inductive proof of Theorem 7.14 involves invoking a quantitative lemma (i.e., Lemma 7.15) that allows to keep track of the relevant quantities (e.g., success probability and circuit size) throughout the induction process. Secondly, we mention that Lemma 7.15 (as well as Theorem 7.14) has a uniform complexity version that assumes that one can efficiently sample the distribution $(Y_{\ell(n)}, F_1(Y_{\ell(n)}))$ (resp., $(U_n, f(U_n))$). For details see [99]. Indeed, a good lesson from the proof of Lemma 7.15 is that non-uniform circuits can "effectively sample" any distribution. Lastly, we mention that Theorem 7.5 (the amplification of one-way functions) and Theorem 7.13 (Yao's XOR Lemma) also have (tight) quantitative versions (see, e.g., [88, Sec. 2.3.2] and [99, Sec. 3], respectively).

7.2.1.3 List decoding and hardness amplification

Recall that Theorem 7.10 was proved in §7.2.1.1-7.2.1.2, by first constructing a mildly inapproximable predicate via Construction 7.11, and then amplifying its hardness via Yao's XOR Lemma. In this subsection we show that the construction used in the first step (i.e., Construction 7.11) actually yields a strongly inapproximable predicate. Thus, we provide an alternative proof of Theorem 7.10. Specifically, we show that a strongly inapproximable predicate (as asserted in Theorem 7.10) can be obtained by combining Construction 7.11 (with a suitable choice of parameters) and the inner-product construction (of Theorem 7.8). The main ingredient of this argument is captured by the following result.

Proposition 7.16 *Suppose that there exists a Boolean function f in $\mathcal{E}$ having circuit complexity that is almost-everywhere greater than S, and let $\varepsilon : \mathbb{N} \to [0,1]$ satisfying $\varepsilon(n) > 2^{-n}$. Let f_n be the restriction of f to $\{0,1\}^n$, and let $\hat{f}_n$ be the function obtained from f_n when applying Construction 7.11*[17] *with $|H| = n/\varepsilon(n)$ and $|F| = |H|^3$. Then, the function $\hat{f} : \{0,1\}^* \to \{0,1\}^*$, defined by $\hat{f}(x) = \hat{f}_{|x|/3}(x)$, is computable in exponential-time and for every family of circuit $\{C'_{n'}\}_{n' \in \mathbb{N}}$ of size $S'(n') = \mathrm{poly}(\varepsilon(n'/3)/n') \cdot S(n'/3)$ it holds that $\mathsf{Pr}[C'_{n'}(U_{n'}) = \hat{f}(U_{n'})] < \varepsilon'(n') \stackrel{\mathrm{def}}{=} \varepsilon(n'/3)$.*

Before turning to the proof of Proposition 7.16, let us describe how it yields an alternative proof of Theorem 7.10. Firstly, for some $\gamma > 0$, Proposition 7.16 yields an exponential-time computable function $\hat{f}$ such that $|\hat{f}(x)| \leq |x|$ and for every family of circuit $\{C'_{n'}\}_{n' \in \mathbb{N}}$ of size $S'(n') = S(n'/3)^\gamma/\mathrm{poly}(n')$ it holds that $\mathsf{Pr}[C'_{n'}(U_{n'}) = \hat{f}(U_{n'})] < 1/S'(n')$. Combining this with Theorem 7.8 (cf. Exercise 7.14), we infer that $P(x,r) = b(\hat{f}(x), r)$, where $|r| = |\hat{f}(x)| \leq |x|$, is S''-inapproximable for $S''(n'') = S'(n''/2)^{\Omega(1)}/\mathrm{poly}(n'')$. In particular, for every polynomial p, we obtain a p-inapproximable predicate in $\mathcal{E}$ by applying the foregoing with $S(n) = \mathrm{poly}(n, p(n))$. Thus, Theorem 7.10 follows.

> **Teaching note:** The following material is very advanced and is best left for independent reading. Furthermore, its understanding requires being comfortable with basic notions of error-correcting codes (as presented in Appendix E.1).

Proposition 7.16 is proven by observing that the transformation of f_n to $\hat{f}_n$ constitutes a "good" code (see §E.1.1.4) and that any such code provides a worst-case to (strongly) average-case reduction. We start by defining the class of codes that suffices for the latter reduction, while noting that the code underlying the mapping $f_n \mapsto \hat{f}_n$ is actually stronger than needed.

Definition 7.17 (efficient codes supporting implicit decoding): *For fixed functions $q, \ell : \mathbb{N} \to \mathbb{N}$ and $\alpha : \mathbb{N} \to [0,1]$, the mapping $\Gamma : \{0,1\}^* \to \{0,1\}^*$ is said to be* efficient and supports implicit decoding *with parameters q, ℓ, α if it satisfies the following two conditions:*

[17] Recall that in Construction 7.11 we have $|H|^m = 2^n$, which may yield a non-integer m if we insist on $|H| = n/\varepsilon(n)$. This problem was avoided in the proof of Theorem 7.12 (where $|H| = n$ was used), but is more acute in the current context because of ε (e.g., we may have $\varepsilon(n) = 2^{-2n/7}$). Thus, we should either relax the requirement $|H|^m = 2^n$ (e.g., allow $2^n \leq |H|^m < 2^{2n}$) or relax the requirement $|H| = n/\varepsilon(n)$. However, for the sake of simplicity, we ignore this issue in the presentation.

1. Encoding (or efficiency): *The mapping* Γ *is polynomial-time computable.*

   It is instructive to view Γ *as mapping* N*-bit long strings to sequences of length* $\ell(N)$ *over* $[q(N)]$, *and to view each* (codeword) $\Gamma(x) \in [q(|x|)]^{\ell(|x|)}$ *as a mapping from* $[\ell(|x|)]$ *to* $[q(|x|)]$.

2. Decoding (in implicit form): *There exists a polynomial* p *such that the following holds. For every* $w : [\ell(N)] \to [q(N)]$ *and every* $x \in \{0,1\}^N$ *such that* $\Gamma(x)$ *is* $(1 - \alpha(N))$*-close to* w, *there exists an oracle-aided*[18] *circuit* C *of size* $p((\log N)/\alpha(N))$ *such that, for every* $i \in [N]$, *it holds that* $C^w(i)$ *equals the* i^{th} *bit of* x.

The encoding condition implies that ℓ is polynomially bounded. The decoding condition refers to any Γ-codeword that agrees with the oracle $w : [\ell(N)] \to [q(N)]$ on an $\alpha(N)$ fraction of the $\ell(N)$ coordinates, *where* $\alpha(N)$ *may be very small.* We highlight the non-triviality of the decoding condition: There are N bits of information in x, while the size of the circuit C is only $p((\log N)/\alpha(N))$ and yet C should be able to recover any desired entry of x by making queries to w, which may be a highly corrupted version of $\Gamma(x)$. Needless to say, the number of queries made by C is upper-bounded by its size (i.e.,$p((\log N)/\alpha(N))$). On the other hand, the decoding condition does not refer to the complexity of obtaining the aforementioned oracle-aided circuits.

Let us relate the transformation of f_n to $\hat{f}_n$, which underlies Proposition 7.16, to Definition 7.17. We view f_n as a binary string of length $N = 2^n$ (representing the truth-table of $f_n : H^m \to \{0,1\}$) and analogously view $\hat{f}_n : F^m \to F$ as an element of $F^{|F|^m} = F^{N^3}$ (or as a mapping from $[N^3]$ to $[|F|]$).[19] Recall that the transformation of f_n to $\hat{f}_n$ is efficient. We mention that *this transformation also supports implicit decoding with parameters* q, ℓ, α *such that* $\ell(N) = N^3$, $\alpha(N) = \varepsilon(n)$, *and* $q(N) = (n/\varepsilon(n))^3$, where $N = 2^n$. The latter fact is highly non-trivial, but establishing it is beyond the scope of the current text (and the interested reader is referred to [213]).

We mention that the transformation of f_n to $\hat{f}_n$ enjoys additional features, which are not required in Definition 7.17 and will not be used in the current context. For example, there are at most $O(1/\alpha(2^n)^2)$ codewords (i.e., $\hat{f}_n$'s) that are $(1 - \alpha(2^n))$-close to any fixed $w : [\ell(2^n)] \to [q(2^n)]$, and the corresponding oracle-aided circuits can be constructed in probabilistic $p(n/\alpha(2^n))$-time.[20] These results are

[18] Oracle-aided circuits are defined analogously to oracle Turing machines. Alternatively, we may consider here oracle machines that take advice such that both the advice length and the machine's running time are upper-bounded by $p((\log N)/\alpha(N))$. The relevant oracles may be viewed either as blocks of binary strings that encode sequences over $[q(N)]$ or as sequences over $[q(N)]$. Indeed, in the latter case we consider non-binary oracles, which return elements in $[q(N)]$.

[19] Recall that $N = 2^n = |H|^m$ and $|F| = |H|^3$. Hence, $|F|^m = N^3$.

[20] The construction may yield also oracle-aided circuits that compute the decoding of codewords that are almost $(1 - \alpha(2^n))$-close to w. That is, there exists a probabilistic $p(n/\alpha(2^n))$-time algorithm that outputs a list of circuits that, with high probability, contains an oracle-aided circuit for the decoding of each codeword that is $(1 - \alpha(2^n))$-close to w. Furthermore, with high probability, the list contains only circuits that decode codewords that are $(1 - \alpha(2^n)/2)$-close to w.

termed "list decoding with implicit representations" (and we refer the interested reader again to [213]).

Our focus is on showing that efficient codes that supports implicit decoding suffice for worst-case to (strongly) average-case reductions. We state and prove a general result, noting that in the special case of Proposition 7.16 $g_n = \hat{f}_n$ (and $\ell(2^n) = 2^{3n}$).

Theorem 7.18 *Suppose that there exists a Boolean function f in $\mathcal{E}$ having circuit complexity that is almost-everywhere greater than S, and let $\varepsilon : \mathbb{N} \to [0,1]$. Consider a polynomial $\ell : \mathbb{N} \to \mathbb{N}$ such that $n \mapsto \log_2 \ell(2^n)$ is a 1-1 map of the integers, and let $m(n) = \log_2 \ell(2^n)$; e.g., if $\ell(N) = N^3$ then $m(n) = 3n$. Suppose that the mapping $\Gamma : \{0,1\}^* \to \{0,1\}^*$ is efficient and supports implicit decoding with parameters q, ℓ, α such that $\alpha(N) = \varepsilon(\lfloor \log_2 N \rfloor)$. Define $g_n : [\ell(2^n)] \to [q(2^n)]$ such that $g_n(i)$ equals the $i^{\rm th}$ element of $\Gamma(\langle f_n \rangle) \in [q(2^n)]^{\ell(2^n)}$, where $\langle f_n \rangle$ denotes the 2^n-bit long description of the truth-table of f_n. Then, the function $g : \{0,1\}^* \to \{0,1\}^*$, defined by $g(z) = g_{m^{-1}(|z|)}(z)$, is computable in exponential-time and for every family of circuit $\{C'_{n'}\}_{n' \in \mathbb{N}}$ of size $S'(n') = \mathrm{poly}(\varepsilon(m^{-1}(n'))/n') \cdot S(m^{-1}(n'))$ it holds that $\mathsf{Pr}[C'_{n'}(U_{n'}) = g(U_{n'})] < \varepsilon'(n') \stackrel{\rm def}{=} \varepsilon(m^{-1}(n'))$.*

Proof Sketch: First note that we can generate the truth-table of f_n in exponential-time, and by the encoding condition of Γ it follows that g_n can be evaluated in exponential-time. The average-case hardness of g is established via a reducibility argument as follows. We consider a circuit $C' = C'_{n'}$ of size S' such that $\mathsf{Pr}[C'_{n'}(U_{n'}) = g(U_{n'})] < \varepsilon'(n')$, let $n = m^{-1}(n')$, and recall that $\varepsilon'(n') = \varepsilon(n) = \alpha(2^n)$. Then, $C' : \{0,1\}^{n'} \to \{0,1\}$ (viewed as a function) is $(1-\alpha(2^n))$-close to the function g_n, which in turn equals $\Gamma(\langle f_n \rangle)$. The decoding condition of Γ asserts that we can recover each bit of $\langle f_n \rangle$ (i.e., evaluate f_n) by an oracle-aided circuit D of size $p(n/\alpha(2^n))$ that uses (the function) C' as an oracle. Combining (the circuit C') with the oracle-aided circuit D, we obtain a (standard) circuit of size $p(n/\alpha(2^n)) \cdot S'(n') < S(n)$ that computes f_n. The theorem follows (i.e., the violation of the conclusion regarding g implies the violation of the hypothesis regarding f). □

Advanced comment. For simplicity, we formulated Definition 7.17 in a crude manner that suffices for the proving Proposition 7.16, where $q(N) = ((\log_2 N)/\alpha(N))^3$. The issue is the existence of codes that satisfy Definition 7.17: In general, such codes may exist only when using a more careful formulation of the decoding condition that refers to codewords that are $(1 - ((1/q(N)) + \alpha(N)))$-close to the oracle $w : [\ell(N)] \to [q(N)]$ rather than being $(1-\alpha(N))$-close to it.[21] Needless to say, the difference is insignificant in the case that $\alpha(N) \gg 1/q(N)$ (as in Proposition 7.16),

[21]Note that this is the "right" formulation, because in the case that $\alpha(N) < 1/q(N)$ it seems impossible to satisfy the decoding condition (as stated in Definition 7.17). Specifically, a random $\ell(N)$-sequence over $[q(N)]$ is expected to be $(1 - (1/q(N)))$-close to any fixed codeword, and with overwhelmingly high probability it will be $(1 - ((1 - o(1))/q(N)))$-close to almost all the codewords, provided $\ell(N) \gg q(N)^2$. But in case $N > \mathrm{poly}(q(N))$, we cannot hope to recover almost all N-bit long strings based on $\mathrm{poly}(q(N) \log N)$ bits of advice (per each of them).

but it is significant in case we care about binary codes (i.e., $q(N) = 2$, or codes over other small alphabets). We mention that Theorem 7.18 can be adapted to this context (of $q(N) = 2$), and directly yields strongly inapproximable predicates. For details, see Exercise 7.18.

7.2.2 Amplification wrt exponential-size circuits

For the purpose of stronger derandomization of $\mathcal{BPP}$, we start with a stronger assumption regarding the worst-case circuit complexity of $\mathcal{E}$ and turn it to a stronger inapproximability result.

Theorem 7.19 *Suppose that there exists a decision problem $L \in \mathcal{E}$ having almost-everywhere exponential circuit complexity; that is, there exists a constant $b > 0$ such that, for all but finitely many n's, any circuit that correctly decides L on $\{0,1\}^n$ has size at least $2^{b \cdot n}$. Then, for some constant $c > 0$ and $T(n) \stackrel{\text{def}}{=} 2^{c \cdot n}$, there exists a T-inapproximable Boolean function in $\mathcal{E}$.*

Theorem 7.19 can be used for deriving a full derandomization of $\mathcal{BPP}$ (i.e., $\mathcal{BPP} = \mathcal{P}$) under the aforementioned assumption (see Part 1 of Theorem 8.19).

Theorem 7.19 follows as a special case of Proposition 7.16 (combined with Theorem 7.8; see Exercise 7.19). An alternative proof, which uses different ideas that are of independent interest, will be briefly reviewed next. The starting point of the latter proof is a mildly inapproximable predicate, as provided by Theorem 7.12. However, here we cannot afford to apply Yao's XOR Lemma (i.e., Theorem 7.13), because the latter relates the size of circuits that *strongly* fail to approximate a predicate defined over poly(n)-bit long strings to the size of circuits that fail to *mildly* approximate a predicate defined over n-bit long strings. That is, Yao's XOR Lemma asserts that if $f : \{0,1\}^n \to \{0,1\}$ is mildly inapproximable by S_f-size circuits then $F : \{0,1\}^{\text{poly}(n)} \to \{0,1\}$ is strongly inapproximable by S_F-size circuits, where $S_F(\text{poly}(n))$ is polynomially related to $S_f(n)$. In particular, $S_F(\text{poly}(n)) < S_f(n)$ seems inherent in this reasoning. For the case of polynomial lower-bounds, this is good enough (i.e., if S_f can be an arbitrarily large polynomial then so can S_F), but for $S_f(n) = \exp(\Omega(n))$ we cannot obtain $S_F(m) = \exp(\Omega(m))$ (but rather only obtain $S_F(m) = \exp(m^{\Omega(1)})$).

The source of trouble is that amplification of inapproximability was achieved by taking a polynomial number of independent instances. Indeed, we cannot hope to amplify hardness without applying f on many instances, but these instances need not be independent. Thus, the idea is to define $F(r) = \oplus_{i=1}^{\text{poly}(n)} f(x_i)$, where $x_1, ..., x_{\text{poly}(n)} \in \{0,1\}^n$ are generated from r and still $|r| = O(n)$. That is, we seek a "derandomized" version of Yao's XOR Lemma. In other words, we seek a "pseudorandom generator" of a type appropriate for expanding r to dependent x_i's such that the XOR of the $f(x_i)$'s is as inapproximable as it would have been for independent x_i's.[22]

[22]Indeed, this falls within the general paradigm discussed in Section 8.1. Furthermore, this suggestion provides another perspective on the connection between randomness and computational difficulty, which is the focus of much discussion in Chapter 8 (see, e.g., §8.2.7.2).

Teaching note: In continuation to Footnote 22, we note that there is a strong connection between the rest of this section and Chapter 8. On top of the aforementioned conceptual aspect, we will use technical tools from Chapter 8 towards establishing the derandomized version of the XOR Lemma. These tools include pairwise independence generators (see Section 8.5.1), random walks on expanders (see Section 8.5.3), and the Nisan-Wigderson Construction (Construction 8.17). Indeed, recall that Section 7.2.2 is advanced material, which is best left for independent reading.

The pivot of the proof is the notion of a hard region of a Boolean function. Loosely speaking, S is a hard region of a Boolean function f if f is *strongly inapproximable on a random input in* S; that is, for every (relatively) small circuit C_n, it holds that $\Pr[C_n(U_n) = f(U_n)|U_n \in S] \approx 1/2$. By definition, $\{0,1\}^*$ is a hard region of any *strongly* inapproximable predicate. As we shall see, any *mildly* inapproximable predicate has a hard region of density related to its inapproximability parameter. Loosely speaking, hardness amplification will proceed via methods for generating related instances that hit the hard region with sufficiently high probability. But, first let us study the notion of a hard region.

7.2.2.1 Hard regions

We actually generalize the notion of hard regions to arbitrary distributions. The important special case of uniform distributions (on n-bit long strings) is obtained from Definition 7.20 by letting X_n equal U_n (i.e., the uniform distribution over $\{0,1\}^n$). In general, we only assume that $X_n \in \{0,1\}^n$.

Definition 7.20 (hard region relative to arbitrary distribution): *Let* $f:\{0,1\}^* \to \{0,1\}$ *be a Boolean predicate,* $\{X_n\}_{n\in\mathbb{N}}$ *be a probability ensemble,* $s:\mathbb{N}\to\mathbb{N}$ *and* $\varepsilon:\mathbb{N}\to[0,1]$.

- *We say that a set* S *is a* hard region of f relative to $\{X_n\}_{n\in\mathbb{N}}$ *with respect to* $s(\cdot)$*-size circuits and advantage* $\varepsilon(\cdot)$ *if for every* n *and every circuit* C_n *of size at most* $s(n)$*, it holds that*
$$\Pr[C_n(X_n)=f(X_n)|X_n\in S] \leq \frac{1}{2} + \varepsilon(n).$$
- *We say that* f *has a* hard region of density $\rho(\cdot)$ *relative to* $\{X_n\}_{n\in\mathbb{N}}$ (with respect to $s(\cdot)$-size circuits and advantage $\varepsilon(\cdot)$) *if there exists a set* S *that is a hard region of* f *relative to* $\{X_n\}_{n\in\mathbb{N}}$ (with respect to the foregoing parameters) *such that* $\Pr[X_n \in S_n] \geq \rho(n)$.

Note that a Boolean function f is $(s, 1-2\varepsilon)$-inapproximable if and only if $\{0,1\}^*$ is a hard region of f relative to $\{U_n\}_{n\in\mathbb{N}}$ with respect to $s(\cdot)$-size circuits and advantage $\varepsilon(\cdot)$. Thus, *strongly* inapproximable predicates (e.g., S-inapproximable predicates for super-polynomial S) have a hard region of density 1 (with respect to a negligible advantage).[23] But this trivial observation does not provide hard regions

[23]Likewise, *mildly* inapproximable predicates have a hard region of density 1 with respect to an advantage that is noticeably smaller than 1/2.

(with respect to a small (i.e., close to zero) advantage) for *mildly* inapproximable predicates. Providing such hard regions is the contents of the following theorem.

Theorem 7.21 (hard regions for mildly inapproximable predicates): *Let $f:\{0,1\}^* \to \{0,1\}$ be a Boolean predicate, $\{X_n\}_{n\in\mathbb{N}}$ be a probability ensemble, $s:\mathbb{N}\to\mathbb{N}$, and $\rho:\mathbb{N}\to[0,1]$ such that $\rho(n) > 1/\mathrm{poly}(n)$. Suppose that, for every circuit C_n of size at most $s(n)$, it holds that $\Pr[C_n(X_n)=f(X_n)] \leq 1-\rho(n)$. Then, for every $\varepsilon:\mathbb{N}\to[0,1]$, the function f has a hard region of density $\rho'(\cdot)$ relative to $\{X_n\}_{n\in\mathbb{N}}$ with respect to $s'(\cdot)$-size circuits and advantage $\varepsilon(\cdot)$, where $\rho'(n) \stackrel{\mathrm{def}}{=} (1-o(1))\cdot\rho(n)$ and $s'(n) \stackrel{\mathrm{def}}{=} s(n)/\mathrm{poly}(n/\varepsilon(n))$.*

In particular, if f is $(s,2\rho)$-inapproximable then f has a hard region of density $\rho'(\cdot)\approx\rho(\cdot)$ relative to the uniform distribution (with respect to $s'(\cdot)$-size circuits and advantage $\varepsilon(\cdot)$).

Proof Sketch:[24] The proof proceeds by first establishing that $\{X_n\}$ is "related" to (or rather "dominates") an ensemble $\{Y_n\}$ such that f is strongly inapproximable on $\{Y_n\}$, and next showing that this implies the claimed hard region. Indeed, this notion of "related ensembles" plays a central role in the proof.

For $\rho:\mathbb{N}\to[0,1]$, we say that $\{X_n\}$ ρ-dominates $\{Y_n\}$ if for every x it holds that $\Pr[X_n=x] \geq \rho(n)\cdot\Pr[Y_n=x]$. In this case we also say that $\{Y_n\}$ is ρ-dominated by $\{X_n\}$. We say that $\{Y_n\}$ is critically ρ-dominated by $\{X_n\}$ if for every x either $\Pr[Y_n=x] = (1/\rho(n))\cdot\Pr[X_n=x]$ or $\Pr[Y_n=x]=0$.[25]

The notions of domination and critical domination play a central role in the proof, which consists of two parts. In the first part (Claim 7.21.1), we prove that, for $\{X_n\}$ and ρ as in the theorem's hypothesis, there exists a ensemble $\{Y_n\}$ that is ρ-dominated by $\{X_n\}$ such that f is strongly inapproximable on $\{Y_n\}$. In the second part (Claim 7.21.2), we prove that the existence of such a dominated ensemble implies the existence of an ensemble $\{Z_n\}$ that is *critically* ρ'-dominated by $\{X_n\}$ such that f is strongly inapproximable on $\{Z_n\}$. Finally, we note that such a critically dominated ensemble yields a hard region of f relative to $\{X_n\}$, and the theorem follows.

Claim 7.21.1: Under the hypothesis of the theorem it holds that there exists a probability ensemble $\{Y_n\}$ that is ρ-dominated by $\{X_n\}$ such that, for every $s'(n)$-size circuit C_n, it holds that

$$\Pr[C_n(Y_n)=f(Y_n)] \leq \frac{1}{2}+\frac{\varepsilon(n)}{2}. \tag{7.12}$$

Proof: We start by assuming, towards the contradiction, that for every distribution Y_n that is ρ-dominated by X_n there exists a $s'(n)$-size circuits C_n such that $\Pr[C_n(Y_n)=f(Y_n)] > 0.5+\varepsilon'(n)$, where $\varepsilon'(n)=\varepsilon(n)/2$. One key observation is that there is a correspondence between the set of all distributions that are

[24] See details in [99, Apdx. A].

[25] Actually, we should allow one point of exception; that is, relax the requirement by saying that for at most one string $x\in\{0,1\}^n$ it holds that $0<\Pr[Y_n=x]<\Pr[X_n=x]/\rho(n)$. This point has little effect on the proof, and is ignored in our presentation.

each ρ-dominated by X_n and the set of all the convex combinations of critically ρ-dominated (by X_n) distributions; that is, each ρ-dominated distribution is a convex combinations of critically ρ-dominated distributions and vice versa (cf., a special case in §D.4.1.1). Thus, considering an enumeration $Y_n^{(1)}, ..., Y_n^{(t)}$ of the critically ρ-dominated (by X_n) distributions, we conclude that for every distribution π on $[t]$ there exists a $s'(n)$-size circuits C_n such that

$$\sum_{i=1}^{t} \pi(i) \cdot \mathsf{Pr}[C_n(Y_n^{(i)}) = f(Y_n^{(i)})] > 0.5 + \varepsilon'(n). \tag{7.13}$$

Now, consider a finite game between two players, where the first player selects a critically ρ-dominated (by X_n) distribution, and the second player selects a $s'(n)$-size circuit and obtains a payoff as determined by the corresponding success probability; that is, if the first player selects the i^{th} critically dominated distribution and the second player selects the circuit C then the payoff equals $\mathsf{Pr}[C(Y_n^{(i)}) = f(Y_n^{(i)})]$. Eq. (7.13) may be interpreted as saying that for any randomized strategy for the first player there exists a deterministic strategy for the second player yielding average payoff greater than $0.5+\varepsilon'(n)$. The Min-Max Principle (cf. von Neumann [227]) asserts that in such a case there exists a randomized strategy for the second player that yields average payoff greater than $0.5 + \varepsilon'(n)$ no matter what strategy is employed by the first player. This means that there exists a distribution, denoted D_n, on $s'(n)$-size circuits such that for every i it holds that

$$\mathsf{Pr}[D_n(Y_n^{(i)}) = f(Y_n^{(i)})] > 0.5 + \varepsilon'(n), \tag{7.14}$$

where the probability refers both to the choice of the circuit D_n and to the random variable Y_n. Let $B_n = \{x : \mathsf{Pr}[D_n(x) = f(x)] \leq 0.5 + \varepsilon'(n)\}$. Then, $\mathsf{Pr}[X_n \in B_n] < \rho(n)$, because otherwise we reach a contradiction to Eq. (7.14) by defining Y_n such that $\mathsf{Pr}[Y_n = x] = \mathsf{Pr}[X_n = x]/\mathsf{Pr}[X_n \in B_n]$ if $x \in B_n$ and $\mathsf{Pr}[Y_n = x] = 0$ otherwise.[26] By employing standard amplification to D_n, we obtain a distribution D'_n over $\text{poly}(n/\varepsilon'(n)) \cdot s'(n)$-size circuits such that for every $x \in \{0,1\}^n \setminus B_n$ it holds that $\mathsf{Pr}[D'_n(x) = f(x)] > 1 - 2^{-n}$. It follows that there exists a $s(n)$-sized circuit C_n such that $C_n(x) = f(x)$ for every $x \in \{0,1\}^n \setminus B_n$, which implies that $\mathsf{Pr}[C_n(X_n) = f(X_n)] \geq \mathsf{Pr}[X_n \in \{0,1\}^n \setminus B_n] > 1 - \rho(n)$, in contradiction to the theorem's hypothesis. The claim follows. □

We next show that the conclusion of Claim 7.21.1 (which was stated for ensembles that are ρ-dominated by $\{X_n\}$) essentially holds also when allowing only critically ρ-dominated (by $\{X_n\}$) ensembles. The following precise statement involves some loss in the domination parameter ρ (as well as in the advantage ε).

Claim 7.21.2: If there exists a probability ensemble $\{Y_n\}$ that is ρ-dominated by $\{X_n\}$ such that for every $s'(n)$-size circuit C_n it holds that $\mathsf{Pr}[C_n(Y_n) =$

[26] Note that Y_n is ρ-dominated by X_n, whereas by the hypothesis $\mathsf{Pr}[D_n(Y_n) = f(Y_n)] \leq 0.5 + \varepsilon'(n)$. Using the fact that any ρ-dominated distribution is a convex combination of critically ρ-dominated distributions, it follows that $\mathsf{Pr}[D_n(Y_n^{(i)}) = f(Y_n^{(i)})] \leq 0.5 + \varepsilon'(n)$ holds for some critically ρ-dominated $Y_n^{(i)}$.

$f(Y_n)] \le 0.5 + (\varepsilon(n)/2)$, then there exists a probability ensemble $\{Z_n\}$ that is critically ρ'-dominated by $\{X_n\}$ such that for every $s'(n)$-size circuit C_n it holds that $\Pr[C_n(Z_n) = f(Z_n)] \le 0.5 + \varepsilon(n)$.

In other words, Claim 7.21.2 asserts that the function f has a hard region of density $\rho'(\cdot)$ relative to $\{X_n\}$ with respect to $s'(\cdot)$-size circuits and advantage $\varepsilon(\cdot)$, thus establishing the theorem. The proof of Claim 7.21.2 uses the Probabilistic Method (cf. [10]). Specifically, we select a set S_n at random by including each n-bit long string x with probability

$$p(x) \stackrel{\text{def}}{=} \frac{\rho(n) \cdot \Pr[Y_n = x]}{\Pr[X_n = x]} \le 1 \tag{7.15}$$

independently of the choice of all other strings. It can be shown that, with high probability over the choice of S_n, it holds that $\Pr[X_n \in S_n] \approx \rho(n)$ and that $\Pr[C_n(X_n) = f(X_n) | X_n \in S_n] < 0.5 + \varepsilon(n)$ for every circuit C_n of size $s'(n)$. The latter assertion is proved by a union bound on all relevant circuits, while showing that for each such circuit C_n, with probability $1 - \exp(-s'(n)^2)$ over the choice of S_n, it holds that $|\Pr[C_n(X_n) = f(X_n) | X_n \in S_n] - \Pr[C_n(Y_n) = f(Y_n)]| < \varepsilon(n)/2$. For details, see [99, Apdx. A]. (This completes the proof of the theorem.) □

7.2.2.2 Hardness amplification via hard regions

Before showing how to use the notion of a hard region in order to prove a derandomized version of Yao's XOR Lemma, we show how to use it in order to prove the original version of Yao's XOR Lemma (i.e., Theorem 7.13).

An alternative proof of Yao's XOR Lemma. Let f, p_1, and p_2 be as in Theorem 7.13. Then, by Theorem 7.21, for $\rho'(n) = 1/3p_2(n)$ and $s'(n) = p_1(n)^{\Omega(1)}/\mathrm{poly}(n)$, the function f has a hard region S of density ρ' (relative to $\{U_n\}$) with respect to $s'(\cdot)$-size circuits and advantage $1/s'(\cdot)$. Thus, for $t(n) = n \cdot p_2(n)$ and F as in Theorem 7.13, with probability at least $1 - (1 - \rho'(n))^{t(n)} = 1 - \exp(-\Omega(n))$, one of the $t(n)$ random (n-bit long) blocks of F resides in S (i.e., the hard region of f). Intuitively, this suffices for establishing the strong inapproximability of F. Indeed, suppose towards the contradiction that a small (i.e., $p'(t(n) \cdot n)$-size) circuit C_n can approximate F (over $U_{t(n)\cdot n}$) with advantage $\varepsilon(n) + \exp(-\Omega(n))$, where $\varepsilon(n) > 1/s'(n)$. Then, the $\varepsilon(n)$ term must be due to $t(n) \cdot n$-bit long inputs that contain a block in S. Using an averaging argument, we can first fix the index of this block and then the contents of the other blocks, and infer the following: for some $i \in [t(n)]$ and $x_1, ..., x_{t(n)} \in \{0,1\}^n$ it holds that

$$\Pr[C_n(x', U_n, x'') = F(x', U_n, x'') \,|\, U_n \in S] \;\ge\; \frac{1}{2} + \varepsilon(n)$$

where $x' = (x_1, ..., x_{i-1}) \in \{0,1\}^{(i-1)\cdot n}$ and $x'' = (x_{i+1}, ..., x_{t(n)}) \in \{0,1\}^{(t(n)-i)\cdot n}$. Hard-wiring $i \in [t(n)]$, $x' = (x_1, ..., x_{i-1})$ and $x'' = (x_{i+1}, ..., x_{t(n)})$ as well as $\sigma \stackrel{\text{def}}{=} \oplus_{j \neq i} f(x_j)$ in C_n, we obtain a contradiction to the (established) fact that

S is a hard region of f (by using the circuit $C'_n(z) = C_n(x', z, x'') \oplus \sigma$). Thus, Theorem 7.13 follows (for any $p'(t(n) \cdot n) \leq s'(n) - 1$).

Derandomized versions of Yao's XOR Lemma. We first show how to use the notion of a hard region in order to amplify very mild inapproximability to a constant level of inapproximability. Recall that our goal is to obtain such an amplification while applying the given function on many (related) instances, where each instance has length that is linearly related to the length of the input of the resulting function. Indeed, these related instances are produced by applying an adequate "pseudorandom generator" (see Chapter 8). The following amplification utilizes a pairwise independence generator (see Section 8.5.1), denoted G, that stretches $2n$-bit long seeds to sequences of n strings, each of length n.

Lemma 7.22 (derandomized XOR Lemma up to constant inapproximability): *Suppose that $f : \{0,1\}^* \to \{0,1\}$ is (T, ρ)-inapproximable, for $\rho(n) > 1/\mathrm{poly}(n)$, and assume for simplicity that $\rho(n) \leq 1/n$. Let b denote the inner-product mod 2 predicate, and G be the aforementioned pairwise independence generator. Then $F_1(s, r) = b(f(x_1) \cdots f(x_n), r)$, where $|r| = n = |s|/2$ and $(x_1, ..., x_n) = G(s)$, is (T', ρ')-inapproximable for $T'(n') = T(n'/3)/\mathrm{poly}(n')$ and $\rho'(n') = \Omega(n' \cdot \rho(n'/3))$.*

Needless to say, if $f \in \mathcal{E}$ then $F_1 \in \mathcal{E}$. By applying Lemma 7.22 for a constant number of times, we may transform an $(T, 1/\mathrm{poly})$-inapproximable predicate into an $(T'', \Omega(1))$-inapproximable one, where $T''(n'') = T(n''/O(1))/\mathrm{poly}(n'')$.

Proof Sketch: As in the foregoing proof (of the original version of Yao's XOR Lemma), we first apply Theorem 7.21. This time we set the parameters so to infer that, for $\alpha(n) = \rho(n)/3$ and $t'(n) = T(n)/\mathrm{poly}(n)$, the function f has a hard region S of density α (relative to $\{U_n\}$) with respect to $t'(\cdot)$-size circuits and *advantage* 0.01. Next, as in §7.2.1.2, we shall consider the corresponding (derandomized) direct product problem; that is, the function $P_1(s) = (f(x_1), ..., f(x_n))$, where $|s| = 2n$ and $(x_1, ..., x_n) = G(s)$. We will first show that P_1 is hard to compute on an $\Omega(n \cdot \alpha(n))$ fraction of the domain, and the quantified inapproximality of F_1 will follow.

One key observation is that, by Exercise 7.20, with probability at least $\beta(n) \stackrel{\text{def}}{=} n \cdot \alpha(n)/2$, at least one of the n strings output by $G(U_{2n})$ resides in S. Intuitively, we expect every $t'(n)$-sized circuit to fail in computing $P_1(U_{2n})$ with probability at least $0.49\beta(n)$, because with probability $\beta(n)$ the sequence $G(U_{2n})$ contains an element in the hard region of f (and in this case the value can be guessed correctly with probability at most 0.51). The actual proof relies on a reducibility argument, which is less straightforward than the argument used in the non-derandomized case.

For technical reasons[27], we use the condition $\alpha(n) < 1/2n$ (which is guaranteed by the hypothesis that $\rho(n) \leq 1/n$ and our setting of $\alpha(n) = \rho(n)/3$). In this case Exercise 7.20 implies that, with probability at least $\beta(n) \stackrel{\text{def}}{=} 0.75 \cdot n \cdot \alpha(n)$, at least one of the n strings output by $G(U_{2n})$ resides in S. We shall show that

[27] The following argument will rely on the fact that $\beta(n) - \gamma(n) > 0.51n \cdot \alpha(n)$, where $\gamma(n) = \Omega(\beta(n))$.

every $(t'(n) - \text{poly}(n))$-sized circuit fails in computing P_1 with probability at least $\gamma(n) = 0.3\beta(n)$. As usual, the claim is proved by a reducibility argument. Let $G(s)_i$ denote the i^{th} string in the sequence $G(s)$ (i.e., $G(s) = (G(s)_1, ..., G(s)_n)$), and note that given i and x we can efficiently sample $G_i^{-1}(x) \stackrel{\text{def}}{=} \{s \in \{0,1\}^{2n} : G(s)_i = x\}$. Given a circuit C_n that computes $P_1(U_{2n})$ correctly with probability $1 - \gamma(n)$, we consider the circuit C'_n that, on input x, uniformly selects $i \in [n]$ and $s \in G_i^{-1}(x)$, and outputs the i^{th} bit in $C_n(s)$. Then, by the construction (of C'_n) and the hypothesis regarding C_n, it holds that

$$\begin{aligned}
\mathsf{Pr}[C'_n(U_n) = f(U_n) | U_n \in S] &\geq \sum_{i=1}^{n} \frac{1}{n} \cdot \mathsf{Pr}[C_n(U_{2n}) = P_1(U_{2n}) | G(U_{2n})_i \in S] \\
&\geq \frac{\mathsf{Pr}[C_n(U_{2n}) = P_1(U_{2n}) \wedge \exists i\, G_i(U_{2n})_i \in S]}{n \cdot \max_i\{\mathsf{Pr}[G(U_{2n})_i \in S]\}} \\
&\geq \frac{(1-\gamma(n)) - (1-\beta(n))}{n \cdot \alpha(n)} \\
&= \frac{0.7\beta(n)}{n \cdot \alpha(n)} > 0.52\,.
\end{aligned}$$

This contradicts the fact that S is a hard region of f with respect to $t'(\cdot)$-size circuits and advantage 0.01. Thus, we have established that every $(t'(n) - \text{poly}(n))$-sized circuit fails in computing P_1 with probability at least $\gamma(n) = 0.3\beta(n)$.

Having established the hardness of P_1, we now infer the mild inapproximability of F_1, where $F_1(s,r) = b(P_1(s), r)$. It suffices to employ the simple (warm-up) case discussed at the beginning of the proof of Theorem 7.7 (where the predictor errs with probability less than 1/4, rather than the full-fledged result that refers to prediction error that is only smaller than 1/2). Denoting by $\eta_C(s) = \mathsf{Pr}_{r \in \{0,1\}^n}[C(s,r) \neq b(P_1(s), r)]$ the prediction error of the circuit C, we recall that if $\eta_C(s) \leq 0.24$ then C can be used to recover $P_1(s)$. Thus, for circuits C of size $T'(3n) = t'(n)/\text{poly}(n)$ it must hold that $\mathsf{Pr}_s[\eta_C(s) > 0.24] \geq \gamma(n)$. It follows that $\mathsf{E}_s[\eta_C(s)] > 0.24\gamma(n)$, which means that every $T'(3n)$-sized circuits fails to compute $(s,r) \mapsto b(P_1(s), r)$ with probability at least $\delta(|s| + |r|) \stackrel{\text{def}}{=} 0.24 \cdot \gamma(|r|)$. This means that F_1 is $(T', 2\delta)$-inapproximable, and the lemma follows (when noting that $\delta(n') = \Omega(n' \cdot \alpha(n'/3))$). $\square$

The next lemma offers an amplification of constant inapproximability to strong inapproximability. Indeed, combining Theorem 7.12 with Lemmas 7.22 and 7.23, yields Theorem 7.19 (as a special case).

Lemma 7.23 (derandomized XOR Lemma starting with constant inapproximability): *Suppose that $f : \{0,1\}^* \to \{0,1\}$ is (T, ρ)-inapproximable, for some constant ρ, and let b denote the inner-product mod 2 predicate. Then there exists an exponential-time computable function G such that $F_2(s,r) = b(f(x_1) \cdots f(x_n), r)$, where $(x_1, ..., x_n) = G(s)$ and $n = \Omega(|s|) = |r| = |x_1| = \cdots = |x_n|$, is T'-inapproximable for $T'(n') = T(n'/O(1))^{\Omega(1)}/\text{poly}(n')$.*

Again, if $f \in \mathcal{E}$ then $F_2 \in \mathcal{E}$.

Proof Outline:[28] As in the proof of Lemma 7.22, we start by establishing a hard region of density $\rho/3$ for f (this time with respect to circuits of size $T(n)^{\Omega(1)}/\text{poly}(n)$ and advantage $T(n)^{-\Omega(1)}$), and focus on the analysis of the (derandomized) direct product problem corresponding to computing the function $P_2(s) = (f(x_1), ..., f(x_n))$, where $|s| = O(n)$ and $(x_1, ..., x_n) = G(s)$. The "generator" G is defined such that $G(s's'') = G_1(s') \oplus G_2(s'')$, where $|s'| = |s''|$, $|G_1(s')| = |G_2(s'')|$, and the following conditions hold:

1. G_1 is the Expander Random Walk Generator discussed in Section 8.5.3. It can be shown that $G_1(U_{O(n)})$ outputs a sequence of n strings such that for any set S of density ρ, with probability $1 - \exp(-\Omega(\rho n))$, at least $\Omega(\rho n)$ of the strings hit S. Note that this property is inherited by G, provided $|G_1(s')| = |G_2(s'')|$ for any $|s'| = |s''|$. It follows that, with probability $1 - \exp(-\Omega(\rho n))$, a constant fraction of the x_i's in the definition of P_2 hit the hard region of f.

   It is tempting to say that small circuits cannot compute P_2 better than with probability $\exp(-\Omega(\rho n))$, but this is clear only in the case that the x_i's that hit the hard region are distributed independently (and uniformly) in it, which is hardly the case here. Indeed, G_2 is used to handle this problem.

2. G_2 is the "set projection" system underlying Construction 8.17; specifically, $G_2(s) = (s_{S_1}, ..., s_{S_n})$, where each S_i is an n-subset of $[|s|]$ and the S_i's have pairwise intersections of size at most $n/O(1)$.[29] An analysis as in the proof of Theorem 8.18 can be employed for showing that the dependency among the x_i's does not help for computing a particular $f(x_i)$ when given x_i as well as all the other $f(x_j)$'s. (Note that this property of G_2 is inherited by G.)

The actual analysis of the construction (via a guessing game presented in [125, Sec. 3]), links the success probability of computing P_2 to the advantage of guessing f on its hard region. The interested reader is referred to [125]. $\square$

Digest. Both Lemmas 7.22 and 7.23 are proved by first establishing corresponding derandomized versions of the "direct product" lemma (Theorem 7.14); in fact, the core of these proofs is proving adequate derandomized "direct product" lemmas. We call the reader's attention to the seemingly crucial role of this step (especially in the proof of Lemma 7.23): We cannot treat the values $f(x_1), ... f(x_n)$ as if they were independent (at least not for the generator G as postulated in these lemmas), and so we seek to avoid analyzing the probability of correctly computing the XOR of *all these values*. In contrast, we have established that it is very hard to correctly compute all n values, and thus *XORing a random subset of these values* yields a strongly inapproximable predicate. (Note that the argument used in Exercise 7.16

[28] For details, see [125].

[29] Recall that s_S denotes the projection of s on coordinates $S \subseteq [|s|]$; that is, for $s = \sigma_1 \cdots \sigma_k$ and $S = \{i_j : j = 1, ..., n\}$, we have $s_S = \sigma_{i_1} \cdots \sigma_{i_n}$.

fails here, because the x_i's are not independent, which is the reason that we XOR a random subset of these values rather than all of them.)

Chapter Notes

The notion of a one-way function was suggested by Diffie and Hellman [63]. The notion of weak one-way functions as well as the amplification of one-way functions (i.e., Theorem 7.5) were suggested by Yao [231]. A proof of Theorem 7.5 has first appeared in [84].

The concept of hard-core predicates was suggested by Blum and Micali [37]. They also proved that a particular predicate constitutes a hard-core for the "DLP function" (i.e., exponentiation in a finite field), provided that the latter function is one-way. The generic hard-core predicate (of Theorem 7.7) was suggested by Levin, and proven as such by Goldreich and Levin [96]. The proof presented here was suggested by Rackoff. We comment that the original proof has its own merits (cf., e.g., [102]).

The construction of canonical derandomizers (see Section 8.3) and, specifically, the Nisan-Wigderson framework (i.e., Construction 8.17) has been the driving force behind the study of inapproximable predicates in $\mathcal{E}$. Theorem 7.10 is due to [20], whereas Theorem 7.19 is due to [125]. Both results rely heavily of variants of Yao's XOR Lemma, to be reviewed next.

Like several other fundamental insights attributed to Yao's paper [231], Yao's XOR Lemma (i.e., Theorem 7.13) is not even stated in [231] but is rather due to Yao's oral presentations of his work. The first published proof of Yao's XOR Lemma was given by Levin (see [99, Sec. 3]). The proof presented in §7.2.1.2 is due to Goldreich, Nisan and Wigderson [99, Sec. 5].

The notion of a hard region and its applications to proving the original version of Yao's XOR Lemma as well as the first derandomization of it (i.e., Lemma 7.22) are due to Impagliazzo [123]. The second derandomization (i.e., Lemma 7.23) as well as Theorem 7.19 are due to Impagliazzo and Wigderson [125].

Theorem 7.12 is due to [20], and the presentation in §7.2.1.1 is based on this work. The connection between list decoding and hardness amplification (i.e., §7.2.1.3), yielding an alternative proof of Theorem 7.19, is due to Sudan, Trevisan, and Vadhan [213].

Hardness amplification for $\mathcal{NP}$ has been the subject of recent attention: An amplification of mild inapproximability to strong inapproximability is provided in [118], and an indication to the impossibility of a worst-case to average-case reductions (at least non-adaptive ones) is provided in [40].

Exercises

Exercise 7.1 Prove that if one way-functions exist then there exists one-way functions that are length preserving (i.e., $|f(x)| = |x|$ for every $x \in \{0,1\}^n$).

Guideline: Clearly, for some polynomial p, it holds that $|f(x)| < p(|x|)$ for all x. Assume, without loss of generality that $n \mapsto p(n)$ is 1-1 and increasing, and let $p^{-1}(m) = n$ if $p(n) \leq m < p(n+1)$. Define $f'(z) = f(x)01^{|z|-|f(x)|-1}$, where x is the $p^{-1}(|z|)$-bit long prefix of z.

Exercise 7.2 Prove that if a function f is hard to invert in the sense of Definition 7.3 then it is hard to invert in the sense of Definition 7.1.

Guideline: Consider a sequence of internal coin tosses that maximizes the probability in Eq. (7.1).

Exercise 7.3 Assuming the existence of one-way functions, prove that there exists a weak one-way function that is not strongly one-way.

Exercise 7.4 (a universal one-way function (by L. Levin)) Using the notion of a universal machine, present a polynomial-time computable function that is hard to invert (in the sense of Definition 7.1) if and only if there exist one-way functions.

Guideline: Consider the function F that parses its input into a pair (M, x) and emulates $|x|^3$ steps of M on input x. Note that if there exists a one-way function that can be evaluated in cubic time then F is a weak one-way function. Using padding, prove that there exists a one-way function that can be evaluated in cubic time if and only if there exist one-way functions.

Exercise 7.5 For $\ell > 1$, prove that the following $2^\ell - 1$ samples are pairwise independent and uniformly distributed in $\{0,1\}^n$. The samples are generated by uniformly and independently selecting ℓ strings in $\{0,1\}^n$. Denoting these strings by $s^1, ..., s^\ell$, we generate $2^\ell - 1$ samples corresponding to the different *non-empty* subsets of $\{1, 2, ..., \ell\}$ such that for subset J we let $r^J \stackrel{\text{def}}{=} \oplus_{j \in J} s^j$.

Guideline: For $J \neq J'$, it holds that $r^J \oplus r^{J'} = \oplus_{j \in K} s^j$, where K denotes the symmetric difference of J and J'. See related material in Section 8.5.1.

Exercise 7.6 (a variant on the proof of Theorem 7.7) Provide a detailed presentation of the alternative procedure outlined in Footnote 5. That is, prove that for every $x \in \{0,1\}^n$, given oracle access to any $B_x : \{0,1\}^n \to \{0,1\}$ that satisfies Eq. (7.6), this procedure makes $\text{poly}(n/\varepsilon)$ steps and outputs a list of strings that, with probability at least $1/2$, contains x.

Exercise 7.7 (proving Theorem 7.8) Recall that the proof of Theorem 7.7 establishes the existence of a $\text{poly}(n/\varepsilon)$-time oracle machine M such that, for every $B : \{0,1\}^n \to \{0,1\}$ and every $x \in \{0,1\}^n$ that satisfy $\mathsf{Pr}_r[B(r) = b(x,r)] \geq \frac{1}{2} + \varepsilon$, it holds that $\mathsf{Pr}[M^B(n,\varepsilon) = x] = \Omega(\varepsilon^2/n)$. Show that this implies Theorem 7.8. (Indeed, an alternative proof can be derived by adapting Exercise 7.6.)

Guideline: Apply a "coupon collector" argument.

Exercise 7.8 A polynomial-time computable predicate $b : \{0,1\}^* \to \{0,1\}$ is called a universal hard-core predicate if for every one-way function f, the predicate b is a hard-core of f. Note that the predicate presented in Theorem 7.7 is "almost universal" (i.e., for every one-way function f, that predicate is a hard-core of $f'(x,r) = (f(x),r)$, where $|x| = |r|$). Prove that there exist no universal hard-core predicate.

Guideline: Let b be a candidate universal hard-core predicate, and let f be an arbitrary one-way function. Then consider the function $f'(x) = (f(x), b(x))$.

Exercise 7.9 Prove that if $\mathcal{NP}$ is not contained in $\mathcal{P}$/poly then neither is $\mathcal{E}$. Furthermore, for every $S : \mathbb{N} \to \mathbb{N}$, if some problem in $\mathcal{NP}$ does not have circuits of size S then for some constant $\varepsilon > 0$ there exists a problem in $\mathcal{E}$ that does not have circuits of size S', where $S'(n) = S(n^\varepsilon)$. Repeat the exercise for the "almost everywhere" case.

Guideline: Although $\mathcal{NP}$ is not known to be in $\mathcal{E}$, it is the case that SAT is in $\mathcal{E}$, which implies that $\mathcal{NP}$ is reducible to a problem in $\mathcal{E}$. For the "almost everywhere" case, address the fact that the said reduction may not preserve the length of the input.

Exercise 7.10 For every function $f : \{0,1\}^n \to \{0,1\}$, present a linear-size circuit C_n such that $\mathsf{Pr}[C(U_n) = f(U_n)] \geq 0.5 + 2^{-n}$. Furthermore, for every $t \leq 2^{n-1}$, present a circuit C_n of size $O(t \cdot n)$ such that $\mathsf{Pr}[C(U_n) = f(U_n)] \geq 0.5 + t \cdot 2^{-n}$. Warning: you may not assume that $\mathsf{Pr}[f(U_n) = 1] = 0.5$.

Exercise 7.11 (self-correction of low-degree polynomials) Let d, m be integers, and F be a finite field of cardinality greater than $t \stackrel{\text{def}}{=} dm + 1$. Let $p : F^m \to F$ be a polynomial of individual degree d, and $\alpha_1, ..., \alpha_t$ be t distinct non-zero elements of F.

1. Show that, for every $x, y \in F^m$, the value of $p(x)$ can be efficiently computed from the values of $p(x + \alpha_1 y), ..., p(x + \alpha_t y)$, where x and y are viewed as m-ary vectors over F.
2. Show that, for every $x \in F^m$ and $\alpha \in F \setminus \{0\}$, if we uniformly select $r \in F^m$ then the point $x + \alpha r$ is uniformly distributed in F^m.

Conclude that $p(x)$ can be recovered based on t random points, where each point is uniformly distributed in F^m.

Exercise 7.12 (low degree extension) Prove that for any $H \subset F$ and every function $f : H^m \to F$ there exists an m-variate polynomial $\hat{f} : F^m \to F$ of individual degree $|H| - 1$ such that for every $x \in H^m$ it holds that $\hat{f}(x) = f(x)$.

Guideline: Define $\hat{f}(x) = \sum_{a \in H^m} \delta_a(x) \cdot f(a)$, where δ_a is an m-variate of individual degree $|H|-1$ such that $\delta_a(a) = 1$ whereas $\delta_a(x) = 0$ for every $x \in H^m \setminus \{a\}$. Specifically, $\delta_{a_1,...,a_m}(x_1, ..., x_m) = \prod_{i=1}^m \prod_{b \in H \setminus \{a_i\}} ((x_i - b)/(a_i - b))$.

Exercise 7.13 Suppose that $\hat{f}$ and S' are as in the conclusion of Theorem 7.12. Prove that there exists a Boolean function g in $\mathcal{E}$ that is (S'', ε)-inapproximable for $S''(n' + O(\log n')) = S'(n')/n'$ and $\varepsilon(m) = 1/m^3$.

Guideline: Consider the function g defined such that $g(x, i)$ equals the i^{th} bit of $\hat{f}(x)$.

Exercise 7.14 (a generic application of Theorem 7.8) For any $\ell : \mathbb{N}\to\mathbb{N}$, let $h : \{0,1\}^* \to \{0,1\}^*$ be a function such that $|h(x)| = \ell(|x|)$ for every $x \in \{0,1\}^*$, and $\{X_n\}_{n\in\mathbb{N}}$ be a probability ensemble. Suppose that, for some $s : \mathbb{N} \to \mathbb{N}$ and $\varepsilon : \mathbb{N} \to [0,1]$, for every family of s-size circuits $\{C_n\}_{n\in\mathbb{N}}$ and all sufficiently large n it holds that $\mathsf{Pr}[C_n(X_n) = h(X_n)] \le \varepsilon(n)$. Suppose that $s' : \mathbb{N} \to \mathbb{N}$ and $\varepsilon' : \mathbb{N} \to [0,1]$ satisfy $s'(n + \ell(n)) \le s(n)/\mathrm{poly}(n/\varepsilon'(n + \ell(n)))$ and $\varepsilon'(n + \ell(n)) \ge 2\varepsilon(n)$. Show that Theorem 7.8 implies that for every family of s'-size circuits $\{C'_{n'}\}_{n'\in\mathbb{N}}$ and all sufficiently large $n' = n + \ell(n)$ it holds that

$$\mathsf{Pr}[C'_{n+\ell(n)}(X_n, U_{\ell(n)}) = b(h(X_n), U_{\ell(n)})] \le \frac{1}{2} + \varepsilon'(n + \ell(n)).$$

Note that if X_n is uniform over $\{0,1\}^n$ then the predicate $h'(x,r) = b(h(x),r)$, where $|r| = |h(x)|$, is $(s', 1 - 2\varepsilon')$-inapproximable. Conclude that, in this case, if $\varepsilon(n) = 1/s(n)$ and $s'(n + \ell(n)) = s(n)^{\Omega(1)}/\mathrm{poly}(n)$, then h' is s'-inapproximable.

Exercise 7.15 (derandomization via averaging arguments) Let $C : \{0,1\}^n \times \{0,1\}^m \to \{0,1\}^\ell$ be a circuit, which may represent a "probabilistic circuit" that processes the first input using a sequence of choices that are given as a second input. Let X and Z be two independent random variables distributed over $\{0,1\}^n$ and $\{0,1\}^m$, respectively, and let χ be a Boolean predicate (which may represent a success event regarding the behavior of C). Prove that there exists a string $z \in \{0,1\}^m$ such that for $C_z(x) \stackrel{\text{def}}{=} C(x,z)$ it holds that $\mathsf{Pr}[\chi(X, C_z(X)) = 1] \ge \mathsf{Pr}[\chi(X, C(X,Z)) = 1]$.

Exercise 7.16 (from "selective XOR" to "standard XOR") Let f be a Boolean function, and $b(y,r)$ denote the inner-product modulo 2 of the equal-length strings y and r. Suppose that $F'(x_1, ..., x_{t(n)}, r) \stackrel{\text{def}}{=} b(f(x_1)\cdots f(x_{t(n)}), r)$, where $x_1, ..., x_{t(n)} \in \{0,1\}^n$ and $r \in \{0,1\}^{t(n)}$, is T'-inapproximable. Assuming that $n \mapsto t(n)\cdot n$ is 1-1, prove that $F(x) \stackrel{\text{def}}{=} F'(x, 1^{t'(|x|)})$, where $t'(t(n)\cdot n) = t(n)$, is T-inapproximable for $T(m) = T(m + t'(m)) - O(m)$.

Guideline: Reduce the approximation of F' to the approximation of F. An important observation is that for any $x = (x_1, ..., x_{t(n)})$, $x' = (x'_1, ..., x'_{t(n)})$, and $r = r_1 \cdots r_{t(n)}$ such that $x'_i = x_i$ if $r_i = 1$, it holds that $F'(x,r) = F(x') \oplus \oplus_{i:r_i=0} f(x'_i)$. This suggests a non-uniform reduction of F' to F, which uses "adequate" $z_1, ..., z_{t(n)} \in \{0,1\}^n$ as well as the corresponding values $f(z_i)$'s as advice. On input $x_1, ..., x_{t(n)}, r_1 \cdots r_{t(n)}$, the reduction sets $x'_i = x_i$ if $r_i = 1$ and $x'_i = z_i$ otherwise, makes the query $x' = (x'_1, ..., x'_{t(n)})$ to F, and returns $F(x') \oplus_{i:r_i=0} f(z_i)$. Analyze this reduction in the case that $z_1, ..., z_{t(n)} \in \{0,1\}^n$ are uniformly distributed, and infer that they can be set to some fixed values (see Exercise 7.15).

Exercise 7.17 (Theorem 7.14 versus Theorem 7.5) Consider a generalization of Theorem 7.14 in which f and P are functions from strings to sets of strings such that $P(x_1, ..., x_t) = f(x_1) \times \cdots \times f(x_t)$.

1. Prove that if for every family of p_1-size circuits, $\{C_n\}_{n\in\mathbb{N}}$, and all sufficiently large $n \in \mathbb{N}$, it holds that $\mathsf{Pr}[C_n(U_n) \notin f(U_n)] > 1/p_2(n)$ then for every

family of p'-size circuits, $\{C'_m\}_{m\in\mathbb{N}}$, it holds that $\Pr[C'_m(U_m) \in P(U_m)] < \varepsilon'(m)$, where ε' and p' are as in Theorem 7.14. Further generalize the claim by replacing $\{U_n\}_{n\in\mathbb{N}}$ with an arbitrary distribution ensemble $\{X_n\}_{n\in\mathbb{N}}$, and replacing U_m by a $t(n)$-fold Cartesian product of X_n (where $m = t(n) \cdot n$).

2. Show that the foregoing generalizes both Theorem 7.14 and a non-uniform complexity version of Theorem 7.5.

Exercise 7.18 (refinement of the main theme of §7.2.1.3) Consider the following modification of Definition 7.17, in which the decoding condition refers to an agreement threshold of $(1/q(N)) + \alpha(N)$ rather than to a threshold of $\alpha(N)$. The modified definition reads as follows (where p is a fixed polynomial): *For every $w:[\ell(N)]\rightarrow[q(N)]$ and $x\in\{0,1\}^N$ such that $\Gamma(x)$ is $(1-((1/q(N))+\alpha(N)))$-close to w, there exists an oracle-aided circuit C of size $p((\log N)/\alpha(N))$ such that $C^w(i)$ yields the i^{th} bit of x for every $i \in [N]$.*

1. Formulate and prove a version of Theorem 7.18 that refers to the modified definition (rather than to the original one).

   Guideline: The modified version should refer to computing $g(U_{m(n)})$ with success probability greater than $(1/q(n)) + \varepsilon(n)$ (rather than greater than $\varepsilon(n)$).

2. Prove that, when applied to binary codes (i.e., $q \equiv 2$), the version in Item 1 yields S''-inapproximable predicates, for $S''(n') = S(m^{-1}(n'))^{\Omega(1)}/\text{poly}(n')$.

3. Prove that the Hadamard Code allows implicit decoding under the modified definition (but not according to the original one).[30]

   Guideline: This is the actual contents of Theorem 7.8.

Show that if $\Gamma : \{0,1\}^N \rightarrow [q(N)]^{\ell(N)}$ is a (non-binary) code that allows implicit decoding then encoding its symbols by the Hadamard code yields a binary code ($\{0,1\}^N \rightarrow \{0,1\}^{\ell(N)\cdot 2^{\lceil \log_2 q(N)\rceil}}$) that allows implicit decoding. Note that efficient encoding is preserved only if $q(N) \leq \text{poly}(N)$.

Exercise 7.19 (using Proposition 7.16 to prove Theorem 7.19) Prove Theorem 7.19 by combining Proposition 7.16 and Theorem 7.8.

Guideline: Note that, for some $\gamma > 0$, Proposition 7.16 yields an exponential-time computable function $\hat{f}$ such that $|\hat{f}(x)| \leq |x|$ and for every family of circuit $\{C'_{n'}\}_{n'\in\mathbb{N}}$ of size $S'(n') = S(n'/3)^\gamma/\text{poly}(n')$ it holds that $\Pr[C'_{n'}(U_{n'}) = \hat{f}(U_{n'})] < 1/S'(n')$. Combining this with Theorem 7.8, infer that $P(x,r) = b(\hat{f}(x), r)$, where $|r| = |\hat{f}(x)| \leq |x|$, is S''-inapproximable for $S''(n'') = S(n''/2)^{\Omega(1)}/\text{poly}(n'')$. Note that if $S(n) = 2^{\Omega(n)}$ then $S''(n'') = 2^{\Omega(n'')}$.

Exercise 7.20 Let G be a pairwise independent generator (i.e., as in Lemma 7.22), $S \subset \{0,1\}^n$ and $\alpha \stackrel{\text{def}}{=} |S|/2^n$. Prove that, with probability at least $\min(n{\cdot}\alpha, 1)/2$, at

[30] Needless to say, the Hadamard Code is not efficient (for the trivial reason that its codewords have exponential length).

least one of the n strings output by $G(U_{2n})$ resides in S. Furthermore, if $\alpha \leq 1/2n$ then this probability is at least $0.75 \cdot n \cdot \alpha$.

Guideline: Using the pairwise independence property and employing the Inclusion-Exclusion formula, we lower-bound the aforementioned probability by $n \cdot \alpha - \binom{n}{2} \cdot \alpha^2$. If $\alpha \leq 1/n$ then the claim follows, otherwise we employ the same reasoning to the first $1/\alpha$ elements in the output of $G(U_{2n})$.

Exercise 7.21 (one-way functions versus inapproximable predicates) Prove that the existence of a non-uniformly hard one-way function (as in Definition 7.3) implies the existence of an exponential-time computable predicate that is T-inapproximable (as per Definition 7.9), for every polynomial T.

Guideline: Suppose first that the one-way function f is length-preserving and 1-1. Consider the hard-core predicate b guaranteed by Theorem 7.7 for $g(x, r) = (f(x), r)$, define the Boolean function h such that $h(z) = b(g^{-1}(z))$, and show that h is T-inapproximable for every polynomial T. For the general case a different approach seems needed. Specifically, given a (length preserving) one-way function f, consider the Boolean function h defined as $h(z, i, \sigma) = 1$ if and only if the i^{th} bit of the lexicographically first element in $f^{-1}(z) = \{x : f(x) = z\}$ equals σ. (In particular, if $f^{-1}(z) = \emptyset$ then $h(z, i, \sigma) = 0$ for every i and σ.)[31] Note that h is computable in exponential-time, but is not (worst-case) computable by polynomial-size circuits. Applying Theorem 7.10, we are done.

[31] Thus, h may be easy to computed in the average-case sense (e.g., if $f(x) = 0^{|x|} f'(x)$ for some one-way function f').