7

Pseudorandom Generators

7.1 Motivation and Definition

In the previous sections, we have seen a number of interesting derandomization results:

- Derandomizing specific algorithms, such as the ones for MaxCut and Undirected S-T Connectivity;
- Giving explicit (efficient, deterministic) constructions of various pseudorandom objects, such as expanders, extractors, and list-decodable codes, as well as showing various relations between them;
- Reducing the randomness needed for certain tasks, such as sampling and amplifying the success probability of randomized algorithm; and
- Simulating **BPP** with any weak random source.

However, all of these still fall short of answering our original motivating question, of whether *every* randomized algorithm can be efficiently derandomized. That is, does $\mathbf{BPP} = \mathbf{P}$?

As we have seen, one way to resolve this question in the positive is to use the following two-step process: First show that the number of

random bits for any **BPP** algorithm can be reduced from $\text{poly}(n)$ to $O(\log n)$, and then eliminate the randomness entirely by enumeration.

Thus, we would like to have a function G that stretches a seed of $d = O(\log n)$ truly random bits into $m = \text{poly}(n)$ bits that "look random." Such a function is called a *pseudorandom generator.* The question is how we can formalize the requirement that the output should "look random" in such a way that (a) the output can be used in place of the truly random bits in *any* **BPP** algorithm, and (b) such a generator exists.

Some candidate definitions for what it means for the random variable $X = G(U_d)$ to "look random" include the following:

- *Information-theoretic or statistical measures*: For example, we might measure entropy of $G(U_d)$, its statistical difference from the uniform distribution, or require pairwise independence. All of these fail one of the two criteria. For example, it is impossible for a deterministic function to increase entropy from $O(\log n)$ to $\text{poly}(n)$. And it is easy to construct algorithms that fail when run using random bits that are only guaranteed to be pairwise independent.
- *Kolmogorov complexity*: A string x "looks random" if it is incompressible (cannot be generated by a Turing machine with a description of length less than $|x|$). An appealing aspect of this notion is that it makes sense of the randomness in a fixed string (rather than a distribution). Unfortunately, it is not suitable for our purposes. Specifically, if the function G is computable (which we certainly want) then all of its outputs have Kolmogorov complexity $d = O(\log n)$ (just hardwire the seed into the TM computing G), and hence are very compressible.
- *Computational indistinguishability*: This is the measure we will use. Intuitively, we say that a random variable X "looks random" if no *efficient* algorithm can distinguish X from a truly uniform random variable. Another perspective comes from the definition of statistical difference:

$$\Delta(X,Y) = \max_T |\Pr[X \in T] - \Pr[Y \in T]|.$$

With computational indistinguishability, we simply restrict the max to be taken only over "efficient" statistical tests T — that is, Ts for which membership can be efficiently tested.

7.1.1 Computational Indistinguishability

Definition 7.1 (computational indistinguishability). Random variables X and Y taking values in $\{0,1\}^m$ are (t,ε) *indistinguishable* if for every *nonuniform* algorithm T running in time at most t, we have

$$|\Pr[T(X)=1] - \Pr[T(Y)=1]| \leq \varepsilon.$$

The left-hand side above is called also the *advantage* of T.

Recall that a nonuniform algorithm is an algorithm that may have some nonuniform advice hardwired in. (See Definition 3.10.) If the algorithm runs in time t we require that the advice string is of length at most t. Typically, to make sense of complexity measures like running time, it is necessary to use asymptotic notions, because a Turing machine can encode a huge lookup table for inputs of any bounded size in its transition function. However, for nonuniform algorithms, we can avoid doing so by using Boolean circuits as our nonuniform model of computation. Similarly to Fact 3.11, every nonuniform Turing machine algorithm running in time $t(n)$ can be simulated by a sequence of Boolean circuit C_n of size $\tilde{O}(t(n))$ and conversely every sequence of Boolean circuits of size $s(n)$ can be simulated by a nonuniform Turing machine running in time $\tilde{O}(s(n))$. Thus, to make our notation cleaner, from now on, by "nonuniform algorithm running in time t," we mean "Boolean circuit of size t," where we measure the size by the number of AND and OR gates in the circuit. (For convenience, we don't count the inputs and negations in the circuit size.) Note also that in Definition 7.1 we have not specified whether the distinguisher is deterministic or randomized; this is because a probabilistic distinguisher achieving advantage greater than ε can be turned into a deterministic distinguisher achieving advantage greater than ε by nonuniformly fixing the randomness. (This is another example of how "nonuniformity is more powerful than randomness," like in Corollary 3.12.)

It is also of interest to study computational indistinguishability and pseudorandomness against uniform algorithms.

Definition 7.2 (uniform computational indistinguishability). Let X_m, Y_m be some sequences of random variables on $\{0,1\}^m$ (or $\{0,1\}^{\mathrm{poly}(m)}$). For functions $t : \mathbb{N} \to \mathbb{N}$ and $\varepsilon : \mathbb{N} \to [0,1]$, we say that $\{X_m\}$ and $\{Y_m\}$ are $(t(m), \varepsilon(m))$ *indistinguishable for uniform algorithms* if for all probabilistic algorithms T running in time $t(m)$, we have that

$$|\Pr[T(X_m) = 1] - \Pr[T(Y_m) = 1]| \leq \varepsilon(m)$$

for all sufficiently large m, where the probabilities are taken over X_m, Y_m and the random coin tosses of T.

We will focus on the nonuniform definition in this survey, but will mention results about the uniform definition as well.

7.1.2 Pseudorandom Generators

Definition 7.3. A deterministic function $G : \{0,1\}^d \to \{0,1\}^m$ is a (t, ε) *pseudorandom generator (PRG)* if

(1) $d < m$, and
(2) $G(U_d)$ and U_m are (t, ε) indistinguishable.

Also, note that we have formulated the definition with respect to nonuniform computational indistinguishability, but an analogous uniform definition can be given.

People attempted to construct pseudorandom generators long before this definition was formulated. Their generators were tested against a battery of statistical tests (e.g., the number of 1s and 0s are approximately the same, the longest run is of length $O(\log m)$, etc.), but these fixed set of tests provided no guarantee that the generators will perform well in an arbitrary application (e.g., in cryptography or derandomization). Indeed, most classical constructions (e.g., linear

congruential generators, as implemented in the standard C library) are known to fail in some applications.

Intuitively, the above definition guarantees that the pseudorandom bits produced by the generator are as good as truly random bits for *all* efficient purposes (where efficient means time at most t). In particular, we can use such a generator for derandomizing any algorithm of running time at most t. For the derandomization to be efficient, we will also need the generator to be efficiently computable.

Definition 7.4. We say a sequence of generators $\{G_m : \{0,1\}^{d(m)} \to \{0,1\}^m\}$ is *computable in time* $t(m)$ if there is a *uniform and deterministic* algorithm M such that for every $m \in \mathbb{N}$ and $x \in \{0,1\}^{d(m)}$, we have $M(m,x) = G_m(x)$ and $M(m,x)$ runs in time at most $t(m)$. In addition, $M(m)$ (with no second input) should output the value $d(m)$ in time at most $t(m)$.

Note that even when we define the pseudorandomness property of the generator with respect to nonuniform algorithms, the efficiency requirement refers to uniform algorithms. As usual, for readability, we will usually refer to a single generator $G : \{0,1\}^{d(m)} \to \{0,1\}^m$, with it being implicit that we are really discussing a family $\{G_m\}$.

Theorem 7.5. Suppose that for all m there exists an $(m, 1/8)$ pseudorandom generator $G : \{0,1\}^{d(m)} \to \{0,1\}^m$ computable in time $t(m)$. Then $\mathbf{BPP} \subset \bigcup_c \mathbf{DTIME}(2^{d(n^c)} \cdot (n^c + t(n^c)))$.

Proof. Let $A(x;r)$ be a **BPP** algorithm that on inputs x of length n, can be simulated by Boolean circuits of size at most most n^c, using coin tosses r. Without loss of generality, we may assume that $|r| = n^c$. (It will often be notationally convenient to assume that the number of random bits used by an algorithm equals its running time or circuit size, so as to avoid an extra parameter. However, most interesting algorithms will only actually read and compute with a smaller number of these bits, so as to leave time available for computation. Thus, one should actually think of an algorithm as only reading a prefix of its random string r.)

The idea is to replace the random bits used by A with pseudorandom bits generated by G, use the pseudorandomness property to show that the algorithm will still be correct with high probability, and finally enumerate over all possible seeds to obtain a deterministic algorithm.

Claim 7.6. For every x of length n, $A(x; G(U_{d(n^c)}))$ errs with probability smaller than $1/2$.

Proof of Claim: Suppose that there exists some x on which $A(x; G(U_{d(n^c)}))$ errs with probability at least $1/2$. Then $T(\cdot) = A(x, \cdot)$ is a Boolean circuit of size at most n^c that distinguishes $G(U_{d(n^c)})$ from U_{n^c} with advantage at least $1/2 - 1/3 > 1/8$. (Notice that we are using the input x as nonuniform advice; this is why we need the PRG to be pseudorandom against nonuniform tests.) □

Now, enumerate over all seeds of length $d(n^c)$ and take a majority vote. There are $2^{d(n^c)}$ of them, and for each we have to run both G and A. □

Notice that we can afford for the generator G to have running time $t(m) = \text{poly}(m)$ or even $t(m) = \text{poly}(m) \cdot 2^{O(d(m))}$ without affecting the time of the derandomization by than more than a polynomial amount. In particular, for this application, it is OK if the generator runs in more time than the tests it fools (which are time m in this theorem). That is, for derandomization, it suffices to have G that is mildly explicit according to the following definition:

Definition 7.7.

(1) A generator $G : \{0,1\}^{d(m)} \to \{0,1\}^m$ is *mildly explicit* if it is computable in time $\text{poly}(m, 2^{d(m)})$.
(2) A generator $G : \{0,1\}^{d(m)} \to \{0,1\}^m$ is *fully explicit* if it is computable in time $\text{poly}(m)$.

These definitions are analogous to the notions of mildly explicit and fully explicit for expander graphs in Section 4.3. The truth table

of a mildly explicit generator can be constructed in time polynomial in its size (which is of size $m \cdot 2^{d(m)}$), whereas a fully explicit generator can be evaluated in time polynomial in its input and output lengths (like the neighbor function of a fully explicit expander).

Theorem 7.5 provides a tradeoff between the seed length of the PRG and the efficiency of the derandomization. Let's look at some typical settings of parameters to see how we might simulate **BPP** in the different deterministic time classes (see Definition 3.1):

(1) Suppose that for every constant $\varepsilon > 0$, there is an $(m, 1/8)$ mildly explicit PRG with seed length $d(m) = m^{\varepsilon}$. Then $\mathbf{BPP} \subset \bigcap_{\varepsilon>0} \mathbf{DTIME}(2^{n^{\varepsilon}}) \stackrel{\text{def}}{=} \mathbf{SUBEXP}$. Since it is known that **SUBEXP** is a proper subset of **EXP**, this is already a nontrivial improvement on the current inclusion $\mathbf{BPP} \subset \mathbf{EXP}$ (Proposition 3.2).
(2) Suppose that there is an $(m, 1/8)$ mildly explicit PRG with seed length $d(m) = \text{polylog}(m)$. Then $\mathbf{BPP} \subset \bigcup_c \mathbf{DTIME}(2^{\log^c m}) \stackrel{\text{def}}{=} \tilde{\mathbf{P}}$.
(3) Suppose that there is an $(m, 1/8)$ mildly explicit PRG with seed length $d(m) = O(\log m)$. Then $\mathbf{BPP} = \mathbf{P}$.

Of course, all of these derandomizations are contingent on the question of whether PRGs exist. As usual, our first answer is yes but the proof is not very helpful — it is nonconstructive and thus does not provide for an efficiently computable PRG:

Proposition 7.8. For all $m \in \mathbb{N}$ and $\varepsilon > 0$, there exists a (nonexplicit) (m, ε) pseudorandom generator $G : \{0,1\}^d \to \{0,1\}^m$ with seed length $d = O(\log m + \log(1/\varepsilon))$.

Proof. The proof is by the probabilistic method. Choose $G : \{0,1\}^d \to \{0,1\}^m$ at random. Now, fix a time m algorithm, T. The probability (over the choice of G) that T distinguishes $G(U_d)$ from U_m with advantage ε is at most $2^{-\Omega(2^d \varepsilon^2)}$, by a Chernoff bound. There are $2^{\text{poly}(m)}$ nonuniform algorithms running in time m (i.e., circuits of size m). Thus, union-bounding over all possible T,

we get that the probability that there exists a T breaking G is at most $2^{\mathrm{poly}(m)}2^{-\Omega(2^d\varepsilon^2)}$, which is less than 1 for d being $O(\log m + \log(1/\varepsilon))$. □

Note that putting together Proposition 7.8 and Theorem 7.5 gives us another way to prove that $\mathbf{BPP} \subset \mathbf{P/poly}$ (Corollary 3.12): just let the advice string be the truth table of an $(n^c, 1/8)$ PRG (which can be described by $2^{d(n^c)} \cdot n^c = \mathrm{poly}(n)$ bits), and then use that PRG in the proof of Theorem 7.5 to derandomize the **BPP** algorithm. However, if you unfold both this proof and our previous proof (where we do error reduction and then fix the coin tosses), you will see that both proofs amount to essentially the same "construction."

7.2 Cryptographic PRGs

The theory of computational pseudorandomness discussed in this section emerged from cryptography, where researchers sought a definition that would ensure that using pseudorandom bits instead of truly random bits (e.g., when encrypting a message) would retain security against all computationally feasible attacks. In this setting, the generator G is used by the honest parties and thus should be very efficient to compute. On the other hand, the distinguisher T corresponds to an attack carried about by an adversary, and we want to protect against adversaries that may invest a lot of computational resources into trying to break the system. Thus, one is led to require that the pseudorandom generators be secure even against distinguishers with greater running time than the generator. The most common setting of parameters in the theoretical literature is that the generator should run in a fixed polynomial time, but the adversary can run in an arbitrary polynomial time.

Definition 7.9. A generator $G_m : \{0,1\}^{d(m)} \to \{0,1\}^m$ is a *cryptographic pseudorandom generator* if:

(1) G_m is fully explicit. That is, there is a constant b such that G_m is computable in time m^b.

(2) G_m is an $(m^{\omega(1)}, 1/m^{\omega(1)})$ PRG. That is, for every constant c, G_m is an $(m^c, 1/m^c)$ pseudorandom generator for all sufficiently large m.

Due to space constraints and the fact that such generators are covered in other texts (see the Chapter Notes and References), we will not do an in-depth study of cryptographic generators, but just survey what is known about them.

The first question to ask is whether such generators exist at all. It is not hard to show that cryptographic pseudorandom generators cannot exist unless $\mathbf{P} \neq \mathbf{NP}$, indeed unless $\mathbf{NP} \not\subset \mathbf{P/poly}$. (See Problem 7.3.) Thus, we do not expect to establish the existence of such generators unconditionally, and instead need to make some complexity assumption. While it would be wonderful to show that $\mathbf{NP} \not\subset \mathbf{P/poly}$ implies the existence of cryptographic pseudorandom generators, that too seems out of reach. However, we can base them on the very plausible assumption that there are functions that are easy to evaluate but hard to invert.

Definition 7.10. $f_n : \{0,1\}^n \to \{0,1\}^n$ is a *one-way function* if:

(1) There is a constant b such that f_n is computable in time n^b for sufficiently large n.
(2) For every constant c and every nonuniform algorithm A running in time n^c:

$$\Pr[A(f_n(U_n)) \in f_n^{-1}(f_n(U_n))] \leq \frac{1}{n^c}$$

for all sufficiently large n.

Assuming the existence of one-way functions seems stronger than the assumption $\mathbf{NP} \not\subset \mathbf{P/poly}$. For example, it is an *average-case* complexity assumption, as it requires that f is hard to invert when evaluated on random inputs. Nevertheless, there are a number of candidate functions believed to be one-way. The simplest is integer multiplication: $f_n(x,y) = x \cdot y$, where x and y are $n/2$-bit numbers.

Inverting this function amounts to the integer factorization problem, for which no efficient algorithm is known.

A classic and celebrated result in the foundations of cryptography is that cryptographic pseudorandom generators can be constructed from any one-way function:

Theorem 7.11. The following are equivalent:

(1) One-way functions exist.
(2) There exist cryptographic pseudorandom generators with seed length $d(m) = m - 1$.
(3) For every constant $\varepsilon > 0$, there exist cryptographic pseudorandom generators with seed length $d(m) = m^{\varepsilon}$.

Corollary 7.12. If one-way functions exist, then **BPP** $\subset$ **SUBEXP**.

What about getting a better derandomization? The proof of the above theorem is more general quantitatively. It takes any one-way function $f_\ell : \{0,1\}^\ell \to \{0,1\}^\ell$ and a parameter m, and constructs a generator $G_m : \{0,1\}^{\mathrm{poly}(\ell)} \to \{0,1\}^m$. The fact that G_m is pseudorandom is proven by a reduction as follows. Assuming for contradiction that we have an algorithm T that runs in time t and distinguishes G_m from uniform with advantage ε, we construct an algorithm T' running in time $t' = t \cdot (m/\varepsilon)^{O(1)}$ inverting f_ℓ (say with probability $1/2$). If $t' \leq \mathrm{poly}(\ell)$, then this contradicts the one-wayness of f, and hence we conclude that T cannot exist and G_m is a (t, ε) pseudorandom generator.

Quantitatively, if f_ℓ is hard to invert by algorithms running in time $s(\ell)$ and we take $m = 1/\varepsilon = s(\ell)^{o(1)}$, then we have $t' \leq s(\ell)$ for every $t = \mathrm{poly}(m)$ and sufficiently large ℓ. Thus, viewing the seed length d of G_m as a function of m, we have $d(m) = \mathrm{poly}(\ell) = \mathrm{poly}(s^{-1}(m^{\omega(1)}))$, where $m^{\omega(1)}$ denotes any superpolynomial function of m.

Thus:

- If $s(\ell) = \ell^{\omega(1)}$, we can get seed length $d(m) = m^{\varepsilon}$ for any desired constant $\varepsilon > 0$ and **BPP** $\subset$ **SUBEXP** (as discussed above).

- If $s(\ell) = 2^{\ell^{\Omega(1)}}$ (as is plausible for the factoring one-way function), then we get seed length $d(m) = \text{poly}(\log m)$ and $\mathbf{BPP} \subset \tilde{\mathbf{P}}$.

But we cannot get seed length $d(m) = O(\log m)$, as needed for concluding $\mathbf{BPP} = \mathbf{P}$, from this result. Even for the maximum possible hardness $s(\ell) = 2^{\Omega(\ell)}$, we get $d(m) = \text{poly}(\log m)$. In fact, Problem 7.3 shows that it is impossible to have a cryptographic PRG with seed length $O(\log m)$ meeting Definition 7.9, where we require that G_m be pseudorandom against all $\text{poly}(m)$-time algorithms. However, for derandomization we only need G_m to be pseudorandom against a fixed poly-time algorithm, e.g., running in time $t = m$, and we would get such generators with seed length $O(\log m)$ if the aforementioned construction could be improved to yield seed length $d = O(\ell)$ instead of $d = \text{poly}(\ell)$.

Open Problem 7.13. Given a one-way function $f : \{0,1\}^\ell \to \{0,1\}^\ell$ that is hard to invert by algorithms running in time $s = s(\ell)$ and a constant c, it is possible to construct a fully explicit (t, ε) pseudorandom generator $G : \{0,1\}^d \to \{0,1\}^m$ with seed length $d = O(\ell)$ and pseudorandomness against time $t = s \cdot (\varepsilon/m)^{O(1)}$?

The best known seed length for such a generator is $d = \tilde{O}(\ell^3 \cdot \log(m/\varepsilon)/\log^2 s)$, which is $\tilde{O}(\ell^2)$ for the case that $s = 2^{\Omega(\ell)}$ and $m = 2^{\Omega(\ell)}$ as discussed above.

The above open problem has long been solved in the positive for one-way *permutations* $f : \{0,1\}^\ell \to \{0,1\}^\ell$. In fact, the construction of pseudorandom generators from one-way permutations has a particularly simple description:

$$G_m(x,r) = (\langle x,r\rangle, \langle f(x),r\rangle, \langle f(f(x)),r\rangle, \ldots, \langle f^{(m-1)}(x),r\rangle),$$

where $|r| = |x| = \ell$ and $\langle \cdot,\cdot \rangle$ denotes inner product modulo 2. One intuition for this construction is the following. Consider the sequence $(f^{(m-1)}(U_\ell), f^{(m-2)}(U_\ell), \ldots, f(U_\ell), U_\ell)$. By the fact that f is hard to invert (but easy to evaluate) it can be argued that the $i+1$'st component of this sequence is infeasible to predict from the first i components except with negligible probability. Thus, it is a computational analogue

of a block source. The pseudorandom generator then is obtained by a computational analogue of block-source extraction, using the strong extractor $\mathrm{Ext}(x, r) = \langle x, r \rangle$. The fact that the extraction works in this computational setting, however, is much more delicate and complex to prove than in the setting of extractors, and relies on a "local list-decoding algorithm" for the corresponding code (namely the Hadamard code). See Problems 7.12 and 7.13. (We will discuss local list decoding in Section 7.6.)

Pseudorandom Functions. It turns out that a cryptographic pseudorandom generator can be used to build an even more powerful object — a family of *pseudorandom functions*. This is a family of functions $\{f_s : \{0,1\}^d \to \{0,1\}\}_{s \in \{0,1\}^d}$ such that (a) given the seed s, the function f_s can be evaluated in polynomial time, but (b) without the seed, it is infeasible to distinguish an oracle for f_s from an oracle to a truly random function. Thus in some sense, the d-bit truly random seed s is stretched to 2^d pseudorandom bits (namely the truth table of f_s)!

Pseudorandom functions have applications in several domains:

- *Cryptography*: When two parties share a seed s to a PRF, they effectively share a random function $f : \{0,1\}^d \to \{0,1\}$. (By definition, the function they share is indistinguishable from random by any poly-time third party.) Thus, in order for one party to send a message m encrypted to the other, they can simply choose a random $r \stackrel{R}{\leftarrow} \{0,1\}^d$, and send $(r, f_s(r) \oplus m)$. With knowledge of s, decryption is easy; simply calculate $f_s(r)$ and XOR it to the second part of the received message. However, the value $f_s(r) \oplus m$ would look essentially random to anyone without knowledge of s.
  This is just one example; pseudorandom functions have vast applicability in cryptography.
- *Learning Theory*: Here, PRFs are used mainly to prove negative results. The basic paradigm in computational learning theory is that we are given a list of examples of a function's behavior, $(x_1, f(x_2)), (x_2, f(x_2)), \ldots, (x_k, f(x_k)))$, where the x_is are being selected randomly from some underlying distribution, and we would like to predict what the func-

tion's value will be on a new data point x_{k+1} coming from the same distribution. Information-theoretically, correct prediction is possible after a small number of samples (with high probability), assuming that the function has a small description (e.g., is computable by a poly-sized circuit). However, it is computationally hard to predict the output of a PRF f_s on a new point x_{k+1} after seeing its value on k points (and this holds even if the algorithm gets to make "membership queries" — choose the evaluation points on its own, in addition to getting random examples from some underlying distribution). Thus, PRFs provide examples of functions that are efficiently computable yet hard to learn.

- *Hardness of Proving Circuit Lower Bounds*: One main approach to proving $\mathbf{P} \neq \mathbf{NP}$ is to show that some $f \in \mathbf{NP}$ doesn't have polynomial size circuits (equivalently, $\mathbf{NP} \not\subset \mathbf{P/poly}$). This approach has had very limited success — the only superpolynomial lower bounds that have been achieved have been using very restricted classes of circuits (monotone circuits, constant depth circuits, etc). For general circuits, the best lower bound that has been achieved for a problem in $\mathbf{NP}$ is $5n - O(n)$.
  Pseudorandom functions have been used to help explain why existing lower-bound techniques have so far not yielded superpolynomial circuit lower bounds. Specifically, it has been shown that any sufficiently "constructive" proof of superpolynomial circuit lower bounds (one that would allow us to certify that a randomly chosen function has no small circuits) could be used to distinguish a pseudorandom function from truly random in subexponential time and thus invert any one-way function in subexponential time. This is known as the "Natural Proofs" barrier to circuit lower bounds.

7.3 Hybrid Arguments

In this section, we introduce a very useful proof method for working with computational indistinguishability, known as the *hybrid argument*.

We use it to establish two important facts — that computational indistinguishability is preserved under taking multiple samples, and that pseudorandomness is equivalent to next-bit unpredictability.

7.3.1 Indistinguishability of Multiple Samples

The following proposition illustrates that computational indistinguishability behaves like statistical difference when taking many independent repetitions; the distance ε multiplies by at most the number of copies (cf. Lemma 6.3, Part 6). Proving it will introduce useful techniques for reasoning about computational indistinguishability, and will also illustrate how working with such computational notions can be more subtle than working with statistical notions.

Proposition 7.14. If random variables X and Y are (t, ε) indistinguishable, then for every $k \in \mathbb{N}$, X^k and Y^k are $(t, k\varepsilon)$ indistinguishable (where X^k represents k independent copies of X).

Note that when $t = \infty$, this follows from Lemma 6.3, Part 6; the challenge here is to show that the same holds even when we restrict to computationally bounded distinguishers.

Proof. We will prove the contrapositive: if there is an efficient algorithm T distinguishing X^k and Y^k with advantage greater than $k\varepsilon$, then there is an efficient algorithm T' distinguishing X and Y with advantage greater than ε. The difference in this proof from the corresponding result about statistical difference is that we need to preserve efficiency when going from T to T'. The algorithm T' will naturally use the algorithm T as a subroutine. Thus this is a *reduction* in the same spirit as reductions used elsewhere in complexity theory (e.g., in the theory of **NP**-completeness).

Suppose that there exists a nonuniform time t algorithm T such that

$$|\Pr[T(X^k) = 1] - \Pr[T(Y^k) = 1]| > k\varepsilon. \tag{7.1}$$

We can drop the absolute value in the above expression without loss of generality. (Otherwise we can replace T with its negation; recall that negations are free in our measure of circuit size.)

Now we will use a "hybrid argument." Consider the hybrid distributions $H_i = X^{k-i}Y^i$, for $i = 0, \ldots, k$. Note that $H_0 = X^k$ and $H_k = Y^k$. Then Inequality (7.1) is equivalent to

$$\sum_{i=1}^{k} \Pr[T(H_{i-1}) = 1] - \Pr[T(H_i) = 1] > k\varepsilon,$$

since the sum telescopes. Thus, there must exist some $i \in [k]$ such that $\Pr[T(H_{i-1}) = 1] - \Pr[T(H_i) = 1] > \varepsilon$, i.e.,

$$\Pr[T(X^{k-i}XY^{i-1}) = 1] - \Pr[T(X^{k-i}YY^{i-1}) = 1] > \varepsilon.$$

By averaging, there exists some $x_1, \ldots x_{k-i}$ and $y_{k-i+2}, \ldots y_k$ such that

$$\begin{aligned}\Pr[&T(x_1, \ldots x_{k-i}, X, y_{k-i+2}, \ldots y_k) = 1] \\ &- \Pr[T(x_1, \ldots x_{k-i}, Y, y_{k-i+2}, \ldots y_k) = 1] > \varepsilon.\end{aligned}$$

Then, define $T'(z) = T(x_1, \ldots x_{k-i}, z, y_{k-i+2}, \ldots, y_k)$. Note that T' is a nonuniform algorithm with advice i, $x_1, \ldots, x_{k-i}$, $y_{k-i+2}, \ldots y_k$ hardwired in. Hardwiring these inputs costs nothing in terms of circuit size. Thus T' is a nonuniform time t algorithm such that

$$\Pr[T'(X) = 1] - \Pr[T'(Y) = 1] > \varepsilon,$$

contradicting the indistinguishability of X and Y. □

While the parameters in the above result behave nicely, with (t, ε) going to $(t, k\varepsilon)$, there are some implicit costs. First, the amount of nonuniform advice used by T' is larger than that used by T. This is hidden by the fact that we are using the same measure t (namely circuit size) to bound both the time and the advice length. Second, the result is meaningless for large values of k (e.g., $k = t$), because a time t algorithm cannot read more than t bits of the input distributions X^k and Y^k.

We note that there is an analogue of the above result for computational indistinguishability against *uniform* algorithms (Definition 7.2), but it is more delicate, because we cannot simply hardwire i, $x_1, \ldots, x_{k-i}$, $y_{k-i+2}, \ldots, y_k$ as advice. Indeed, a direct analogue of the proposition as stated is known to be false. We need to add the

additional condition that the distributions X and Y are efficiently samplable. Then T' can choose $i \xleftarrow{R} [k]$ at random, and randomly sample $x_1, \ldots, x_{k-i} \xleftarrow{R} X$, $y_{k-i+2}, \ldots, y_k \xleftarrow{R} Y$.

7.3.2 Next-Bit Unpredictability

In analyzing the pseudorandom generators that we construct, it will be useful to work with a reformulation of the pseudorandomness property, which says that given a prefix of the output, it should be hard to predict the next bit much better than random guessing.

For notational convenience, we deviate from our usual conventions and write X_i to denote the ith bit of random variable X, rather than the ith random variable in some ensemble. We have:

Definition 7.15. Let X be a random variable distributed on $\{0,1\}^m$. For $t \in \mathbb{N}$ and $\varepsilon \in [0,1]$, we say that X is (t,ε) *next-bit unpredictable* if for every nonuniform probabilistic algorithm P running in time t and every $i \in [m]$, we have:

$$\Pr[P(X_1 X_2 \cdots X_{i-1}) = X_i] \leq \frac{1}{2} + \varepsilon,$$

where the probability is taken over X and the coin tosses of P.

Note that the uniform distribution $X \equiv U_m$ is $(t,0)$ next-bit unpredictable for every t. Intuitively, if X is pseudorandom, it must be next-bit unpredictable, as this is just one specific kind of test one can perform on X. In fact the converse also holds, and that will be the direction we use.

Proposition 7.16. Let X be a random variable taking values in $\{0,1\}^m$. If X is a (t,ε) pseudorandom, then X is $(t - O(1),\varepsilon)$ next-bit unpredictable. Conversely, if X is (t,ε) next-bit unpredictable, then it is $(t, m \cdot \varepsilon)$ pseudorandom.

Proof. Here U denotes an r.v. uniformly distributed on $\{0,1\}^m$ and U_i denotes the ith bit of U.

pseudorandom ⇒ next-bit unpredictable. The proof is by reduction. Suppose for contradiction that X is not $(t-3,\varepsilon)$ next-bit unpredictable, so we have a predictor $P:\{0,1\}^{i-1}\to\{0,1\}$ that succeeds with probability at least $1/2+\varepsilon$. We construct an algorithm $T:\{0,1\}^m\to\{0,1\}$ that distinguishes X from U_m as follows:

$$T(x_1x_2\cdots x_m)=\begin{cases}1 & \text{if } P(x_1x_2\cdots x_{i-1})=x_i\\ 0 & \text{otherwise.}\end{cases}$$

T can be implemented with the same number of $\wedge$ and $\vee$ gates as P, plus 3 for testing equality (via the formula $(x\wedge y)\vee(\neg x\wedge\neg y)$).

next-bit unpredictable ⇒ pseudorandom. Also by reduction. Suppose X is not pseudorandom, so we have a nonuniform algorithm T running in time t s.t.

$$\Pr[T(X)=1]-\Pr[T(U)=1]>\varepsilon,$$

where we have dropped the absolute values without loss of generality as in the proof of Proposition 7.14.

We now use a hybrid argument. Define $H_i=X_1X_2\cdots X_iU_{i+1}U_{i+2}\cdots U_m$. Then $H_m=X$ and $H_0=U$. We have:

$$\sum_{i=1}^{m}(\Pr[T(H_i)=1]-\Pr[T(H_{i-1})=1])>\varepsilon,$$

since the sum telescopes. Thus, there must exist an i such that

$$\Pr[T(H_i)=1]-\Pr[T(H_{i-1})=1]>\varepsilon/m.$$

This says that T is more likely to output 1 when we put X_i in the ith bit than when we put a random bit U_i. We can view U_i as being X_i with probability $1/2$ and being $\overline{X_i}$ with probability $1/2$. The only advantage T has must be coming from the latter case, because in the former case, the two distributions are identical. Formally,

$$\begin{aligned}\varepsilon/m &< \Pr[T(H_i)=1]-\Pr[T(H_{i-1})=1]\\ &= \Pr[T(X_1\cdots X_{i-1}X_iU_{i+1}\cdots U_m)=1]\end{aligned}$$

$$
\begin{aligned}
&- \Big(\frac{1}{2} \cdot \Pr[T(X_1 \cdots X_{i-1} X_i U_{i+1} \cdots U_m) = 1] \\
&+ \frac{1}{2} \cdot \Pr[T(X_1 \cdots X_{i-1} \overline{X_i} U_{i+1} \cdots U_m) = 1] \Big) \\
&= \frac{1}{2} \cdot (\Pr[T(X_1 \cdots X_{i-1} X_i U_{i+1} \cdots U_m) = 1] \\
&- \Pr[T(X_1 \cdots X_{i-1} \overline{X_i} U_{i+1} \cdots U_m) = 1]).
\end{aligned}
$$

This motivates the following next-bit predictor: $P(x_1 x_2 \cdots x_{i-1})$:

(1) Choose random bits $u_i, \ldots, u_m \xleftarrow{R} \{0,1\}$.
(2) Compute $b = T(x_1 \cdots x_{i-1} u_i \cdots u_m)$.
(3) If $b = 1$, output u_i, otherwise output $\overline{u_i}$.

The intuition is that T is more likely to output 1 when $u_i = x_i$ than when $u_i = \overline{x_i}$. Formally, we have:

$$
\begin{aligned}
&\Pr[P(X_1 \cdots X_{i-1}) = X_i] \\
&\quad = \frac{1}{2} \cdot (\Pr[T(X_1 \cdots X_{i-1} U_i U_{i+1} \cdots U_m) = 1 | U_i = X_i] \\
&\qquad + \Pr[T(X_1 \cdots X_{i-1} U_i U_{i+1} \cdots U_m) = 0 | U_i \neq X_i]) \\
&\quad = \frac{1}{2} \cdot (\Pr[T(X_1 \cdots X_{i-1} X_i U_{i+1} \cdots U_m) = 1] \\
&\qquad + 1 - \Pr[T(X_1 \cdots X_{i-1} \overline{X_i} U_{i+1} \cdots U_m) = 1]) \\
&\quad > \frac{1}{2} + \frac{\varepsilon}{m}.
\end{aligned}
$$

Note that as described P runs in time $t + O(m)$. Recalling that we are using circuit size as our measure of nonuniform time, we can reduce the running time to t as follows. First, we may nonuniformly fix the coin tosses $u_i, \ldots, u_m$ of P while preserving its advantage. Then all P does is run T on $x_1 \cdots x_{i-1}$ concatenated with some fixed bits and and either output what T does or its negation (depending on the fixed value of u_i). Fixing some input bits and negation can be done without increasing circuit size. Thus we contradict the next-bit unpredictability of X. □

We note that an analogue of this result holds for uniform distinguishers and predictors, provided that we change the definition of

next-bit predictor to involve a random choice of $i \stackrel{R}{\leftarrow} [m]$ instead of a fixed value of i, and change the time bounds in the conclusions to be $t - O(m)$ rather than $t - O(1)$ and t. (We can't do tricks like in the final paragraph of the proof.) In contrast to the multiple-sample indistinguishability result of Proposition 7.14, this result does not need X to be efficiently samplable for the uniform version.

7.4 Pseudorandom Generators from Average-Case Hardness

In Section 7.2, we surveyed cryptographic pseudorandom generators, which have numerous applications within and outside cryptography, including to derandomizing **BPP**. However, for derandomization, we can use generators with weaker properties. Specifically, Theorem 7.5 only requires $G : \{0,1\}^{d(m)} \to \{0,1\}^m$ such that:

(1) G fools (nonuniform) distinguishers running in time m (as opposed to all poly(m)-time distinguishers).
(2) G is computable in time poly$(m, 2^{d(m)})$ (i.e., G is mildly explicit). In particular, *the PRG may take more time than the distinguishers it is trying to fool.*

Such a generator implies that every **BPP** algorithm can be derandomized in time $\text{poly}(n) \cdot 2^{d(\text{poly}(n))}$.

The benefit of studying such generators is that we can hope to construct them under weaker assumptions than used for cryptographic generators. In particular, a generator with the properties above no longer seems to imply $\mathbf{P} \neq \mathbf{NP}$, much less the existence of one-way functions. (The nondeterministic distinguisher that tests whether a string is an output of the generator by guessing a seed needs to evaluate the generator, which takes more time than the distinguishers are allowed.)

However, as shown in Problem 7.1, such generators still imply nonuniform circuit lower bounds for exponential time, something that is beyond the state of the art in complexity theory.

Our goal in the rest of this section is to construct generators as above from assumptions that are as weak as possible. In this section, we will construct them from boolean functions computable in exponential time that are hard on average (for nonuniform algorithms), and in the section after we will relax this to only require worst-case hardness.

7.4.1 Average-Case Hardness

A function is *hard on average* if it is hard to compute correctly on randomly chosen inputs. Formally:

Definition 7.17. For $s \in \mathbb{N}$ and $\delta \in [0,1]$, we say that a Boolean function $f : \{0,1\}^\ell \to \{0,1\}$ is (s,δ) *average-case hard* if for all nonuniform probabilistic algorithms A running in time s,

$$\Pr[A(X) = f(X)] \leq 1 - \delta,$$

where the probability is taken over X and the coin tosses of A.

Note that saying that f is (s,δ) hard for *some* $\delta > 0$ (possibly exponentially small) amounts to saying that f is *worst-case* hard.[1] Thus, we think of average-case hardness as corresponding to values of δ that are noticeably larger than zero, e.g., $\delta = 1/s$.[1] or $\delta = 1/3$. Indeed, in this section we will take $\delta = 1/2 - \varepsilon$ for $\varepsilon = 1/s$. That is, no efficient algorithm can compute f much better than random guessing. A typical setting of parameters we use is $s = s(\ell)$ somewhere in range from $\ell^{\omega(1)}$ (slightly superpolynomial) to $s(\ell) = 2^{\alpha\ell}$ for a constant $\alpha > 0$. (Note that every function is computable by a nonuniform algorithm running in time roughly 2^ℓ, so we cannot take $s(\ell)$ to be any larger.) We will also require f to be computable in (uniform) time $2^{O(\ell)}$ so that our pseudorandom generator will be computable in time exponential in its seed length. The existence of such an average-case hard function may seem like a strong assumption, but in later sections we will see how to deduce it from a worst-case hardness assumption.

Now we show how to obtain a pseudorandom generator from average-case hardness.

Proposition 7.18. If $f : \{0,1\}^\ell \to \{0,1\}$ is $(t, 1/2 - \varepsilon)$ average-case hard, then $G(x) = x \circ f(x)$ is a (t,ε) pseudorandom generator.

---

[1] For probabilistic algorithms, the "right" definition of worst-case hardness is actually that there exists an input x for which $\Pr[A(x) = f(x)] < 2/3$, where the probability is taken over the coin tosses of A. But for nonuniform algorithms two definitions can be shown to be roughly equivalent. See Definition 7.34 and the subsequent discussion.

We omit the proof of this proposition, but it follows from Problem 7.5, Part 2 (by setting $m = 1$, $a = 0$, and $d = \ell$ in Theorem 7.24). Note that this generator includes its seed in its output. This is impossible for cryptographic pseudorandom generators, but is feasible (as shown above) when the generator can have more resources than the distinguishers it is trying to fool.

Of course, this generator is quite weak, stretching by only one bit. We would like to get many bits out. Here are two attempts:

- *Use concatenation*: Define $G(x_1 \cdots x_k) = x_1 \cdots x_k f(x_1) \cdots f(x_k)$. This is a $(t, k\varepsilon)$ pseudorandom generator because $G(U_{k\ell})$ consists of k independent samples of a pseudorandom distribution and thus computational indistinguishability is preserved by Proposition 7.14. Note that already here we are relying on *nonuniform* indistinguishability, because the distribution $(U_\ell, f(U_\ell))$ is not necessarily samplable (in time that is feasible for the distinguishers). Unfortunately, however, this construction does not improve the ratio between output length and seed length, which remains very close to 1.
- *Use composition*: For example, try to get two bits out using the same seed length by defining $G'(x) = G(G(x)_{1\cdots\ell})G(x)_{\ell+1}$, where $G(x)_{1\cdots\ell}$ denotes the first ℓ bits of $G(x)$. This works for cryptographic pseudorandom generators, but not for the generators we are considering here. Indeed, for the generator $G(x) = xf(x)$ of Proposition 7.18, we would get $G'(x) = xf(x)f(x)$, which is clearly not pseudorandom.

7.4.2 The Pseudorandom Generator

Our goal now is to show the following:

Theorem 7.19. For $s : \mathbb{N} \to \mathbb{N}$, suppose that there is a function $f \in \mathbf{E} = \mathbf{DTIME}(2^{O(\ell)})$[2] such that for every input length $\ell \in \mathbb{N}$, f

[2] $\mathbf{E}$ should be contrasted with the larger class $\mathbf{EXP} = \mathbf{DTIME}(2^{\mathrm{poly}(\ell)})$. See Problem 7.2.

is $(s(\ell), 1/2 - 1/s(\ell))$ average-case hard, where $s(\ell)$ is computable in time $2^{O(\ell)}$. Then for every $m \in \mathbb{N}$, there is a mildly explicit $(m, 1/m)$ pseudorandom generator $G : \{0,1\}^{d(m)} \to \{0,1\}^m$ with seed length $d(m) = O(s^{-1}(\text{poly}(m))^2 / \log m)$ that is computable in time $2^{O(d(m))}$.

Note that this is similar to the seed length $d(m) = \text{poly}(s^{-1}(\text{poly}(m)))$ mentioned in Section 7.2 for constructing cryptographic pseudorandom generators from one-way functions, but the average-case assumption is incomparable (and will be weakened further in the next section). In fact, it is known how to achieve a seed length $d(m) = O(s^{-1}(\text{poly}(m)))$, which matches what is known for constructing pseudorandom generators from one-way permutations as well as the converse implication of Problem 7.1. We will not cover that improvement here (see the Chapter Notes and References for pointers), but note that for the important case of hardness $s(\ell) = 2^{\Omega(\ell)}$, Theorem 7.19 achieves seed length $d(m) = O(O(\log m)^2 / \log m) = O(\log m)$ and thus $\mathbf{P} = \mathbf{BPP}$. More generally, we have:

Corollary 7.20. Suppose that $\mathbf{E}$ has a $(s(\ell), 1/2 - 1/s(\ell))$ average-case hard function $f : \{0,1\}^\ell \to \{0,1\}$.

(1) If $s(\ell) = 2^{\Omega(\ell)}$, then $\mathbf{BPP} = \mathbf{P}$.
(2) If $s(\ell) = 2^{\ell^{\Omega(1)}}$, then $\mathbf{BPP} \subset \tilde{\mathbf{P}}$.
(3) If $s(\ell) = \ell^{\omega(1)}$, then $\mathbf{BPP} \subset \mathbf{SUBEXP}$.

The idea is to apply f repeatedly, but on *slightly dependent* inputs, namely ones that share very few bits. The sets of seed bits used for each output bit will be given by a *design*:

Definition 7.21. $S_1, \ldots, S_m \subset [d]$ is an (ℓ, a)-*design* if

(1) $\forall i, |S_i| = \ell$
(2) $\forall i \neq j, |S_i \cap S_j| \leq a$

We want lots of sets having small intersections over a small universe. We will use the designs established by Problem 3.2:

Lemma 7.22. For every constant $\gamma > 0$ and every $\ell, m \in \mathbb{N}$, there exists an (ℓ, a)-design $S_1, \ldots, S_m \subset [d]$ with $d = O(\frac{\ell^2}{a})$ and $a = \gamma \cdot \log m$. Such a design can be constructed deterministically in time $\text{poly}(m, d)$.

The important points are that intersection sizes are only logarithmic in the number of sets, and the universe size d is at most quadratic in the set size ℓ (and can be linear in ℓ in case we take $m = 2^{\Omega(\ell)}$).

Construction 7.23. (Nisan–Wigderson Generator) Given a function $f : \{0,1\}^\ell \to \{0,1\}$ and an (ℓ, a)-design $S_1, \ldots, S_m \subset [d]$, define the *Nisan–Wigderson generator* $G : \{0,1\}^d \to \{0,1\}^m$ as

$$G(x) = f(x|_{S_1}) f(x|_{S_2}) \cdots f(x|_{S_m})$$

where if x is a string in $\{0,1\}^d$ and $S \subset [d]$, then $x|_S$ is the string of length $|S|$ obtained from x by selecting the bits indexed by S.

Theorem 7.24. Let $G : \{0,1\}^d \to \{0,1\}^m$ be the Nisan–Wigderson generator based on a function $f : \{0,1\}^\ell \to \{0,1\}$ and some (ℓ, a) design. If f is $(s, 1/2 - \varepsilon/m)$ average-case hard, then G is a (t, ε) pseudorandom generator, for $t = s - m \cdot a \cdot 2^a$.

Theorem 7.19 follows from Theorem 7.24 by setting $\varepsilon = 1/m$ and $a = \log m$, and observing that for $\ell = s^{-1}(m^3)$, then $t = s(\ell) - m \cdot a \cdot 2^a \geq m$, so we have an $(m, 1/m)$ pseudorandom generator. The seed length is $d = O(\ell^2 / \log m) = O(s^{-1}(\text{poly}(m))^2 / \log m)$.

Proof. Suppose G is not a (t, ε) pseudorandom generator. By Proposition 7.16, there is a nonuniform time t next-bit predictor P such that

$$\Pr[P(f(X|_{S_1}) f(X|_{S_2}) \cdots f(X|_{S_{i-1}})) = f(X|_{S_i})] > \frac{1}{2} + \frac{\varepsilon}{m}, \qquad (7.2)$$

for some $i \in [m]$. From P, we construct A that computes f with probability greater than $1/2 + \varepsilon/m$.

Let $Y = X|_{S_i}$. By averaging, we can fix all bits of $X|_{\overline{S_i}} = z$ (where $\overline{S_i}$ is the complement of S) such that the prediction probability remains greater than $1/2 + \varepsilon/m$ (over Y and the coin tosses of the predictor P). Define $f_j(y) = f(x|_{S_j})$ for $j \in \{1, \ldots, i-1\}$. (That is, $f_j(y)$ forms x by placing y in the positions in S_i and z in the others, and then applies f to $x|_{S_j}$). Then

$$\Pr_Y[P(f_1(Y) \cdots f_{i-1}(Y)) = f(Y)] > \frac{1}{2} + \frac{\varepsilon}{m}.$$

Note that $f_j(y)$ depends on only $|S_i \cap S_j| \leq a$ bits of y. Thus, we can compute each f_j with a look-up table, which we can include in the advice to our nonuniform algorithm. Indeed, every function on a bits can be computed by a boolean circuit of size at most $a \cdot 2^a$. (In fact, size at most $O(2^a/a)$ suffices.)

Then, defining $A(y) = P(f_1(y) \cdots f_{i-1}(y))$, we deduce that $A(y)$ can be computed with error probability smaller than $1/2 - \varepsilon/m$ in nonuniform time less than $t + m \cdot a \cdot 2^a = s$. This contradicts the hardness of f. Thus, we conclude G is an (m, ε) pseudorandom generator. □

Some additional remarks on this proof:

(1) This is a very general construction that works for any average-case hard function f. We only used $f \in \mathbf{E}$ to deduce G is computable in $\mathbf{E}$.
(2) The reduction works for any nonuniform class of algorithms $\mathcal{C}$ where functions of logarithmically many bits can be computed efficiently.

Indeed, in the next section we will use the same construction to obtain an *unconditional* pseudorandom generator fooling constant-depth circuits, and will later exploit the above "black-box" properties even further.

As mentioned earlier, the parameters of Theorem 7.24 have been improved in subsequent work, but the newer constructions do not have the clean structure of Nisan–Wigderson generator, where the seed of the generator is used to generate m random but correlated evaluation points, on which the average-case hard function f is evaluated. Indeed,

each output bit of the improved generators depends on the entire truth-table of the function f, translating to a construction of significantly higher computational complexity. Thus the following remains an interesting open problem (which would have significance for hardness amplification as well as constructing pseudorandom generators):

Open Problem 7.25. For every $\ell, s \in \mathbb{N}$, construct an explicit generator $H : \{0,1\}^{O(\ell)} \to (\{0,1\}^{\ell})^m$ with $m = s^{\Omega(1)}$ such that if f is $(s, 1/2 - 1/s)$ average-case hard and we define $G(x) = f(H_1(x)) f(H_2(x)) \cdots f(H_m(x))$ where $H_i(x)$, denotes the ith component of $H(x)$, then G is an $(m, 1/m)$ pseudorandom generator.

7.4.3 Derandomizing Constant-depth circuits

Definition 7.26. An *unbounded fan-in circuit* $C(x_1, \ldots, x_n)$ has input gates consisting of variables x_i, their negations $\neg x_i$, and the constants 0 and 1, as well as computation gates, which can compute the AND or OR of an unbounded number of other gates (rather than just 2, as in usual Boolean circuits).[3] The *size* of such a circuit is the number of computation gates, and the *depth* is the maximum of length of a path from an input gate to the output gate.

$\mathbf{AC^0}$ is the class of functions $f : \{0,1\}^* \to \{0,1\}$ for which there exist constants c and k and a uniformly constructible sequence of unbounded fan-in circuits $(C_n)_{n \in \mathbb{N}}$ such that for all n, C_n has size at most n^c and depth at most k, and for all $x \in \{0,1\}^n$, $C_n(x) = f(x)$. *Uniform constructibility* means that there is an efficient (e.g., polynomial-time) uniform algorithm M such that for all n, $M(1^n) = C_n$ (where 1^n denotes the number n in unary, i.e., a string of n 1s). $\mathbf{BPAC^0}$ defined analogously, except that C_n may have poly(n) extra inputs, which are interpreted as random bits, and we require $\Pr_r[C_n(x,r) = f(x)] \geq 2/3$.

---

[3] Note that it is unnecessary to allow internal NOT gates, as these can always be pushed to the inputs via DeMorgan's Laws at no increase in size or depth.

$\mathbf{AC^0}$ is one of the richest circuit classes for which we have superpolynomial lower bounds:

Theorem 7.27. For all constant $k \in \mathbb{N}$ and every $\ell \in \mathbb{N}$, the function $\textsc{Par}_\ell : \{0,1\}^\ell \to \{0,1\}$ defined by $\mathrm{PAR}_\ell(x_1, \ldots, x_\ell) = \bigoplus_{i=1}^{\ell} x_i$ is $(s_k(\ell), 1/2 - 1/s_k(\ell))$-average-case hard for nonuniform unbounded fan-in circuits of depth k and size $s_k(\ell) = 2^{\Omega(\ell^{1/k})}$.

The proof of this result is beyond the scope of this survey; see the Chapter Notes and References for pointers.

In addition to having an average-case hard function against $\mathbf{AC^0}$, we also need that $\mathbf{AC^0}$ can compute arbitrary functions on a logarithmic number of bits.

Lemma 7.28. Every function $g : \{0,1\}^a \to \{0,1\}$ can be computed by a depth 2 circuit of size 2^a.

Using these two facts with the Nisan–Wigderson pseudorandom generator construction, we obtain the following pseudorandom generator for constant-depth circuits.

Theorem 7.29. For every constant k and every m, there exists a $\mathrm{poly}(m)$-time computable $(m, 1/m)$-pseudorandom generator $G_m : \{0,1\}^{\log^{O(k)} m} \to \{0,1\}^m$ fooling unbounded fan-in circuits of depth k (and size m).

Proof. This is proven similarly to Theorems 7.19 and 7.24, except that we take $f = \mathrm{PAR}_\ell$ rather than a hard function in $\mathbf{E}$, and we observe that the reduction can be implemented in a way that increases the depth by only an additive constant. Specifically, to obtain a pseudorandom generator fooling circuits of depth k and size m, we use the hardness of PAR_ℓ against unbounded fan-in circuits of depth $k' = k + 2$ and size m^2, where $\ell = t_{k'}^{-1}(m^2) = O(\log^{k'} m)$. Then the seed length of G is $O(\ell^2/a) < O(\ell^2) = \log^{O(k)} m$.

We now follow the steps of the proof of Theorem 7.19 to go from an adversary T of depth k violating the pseudorandomness of G to a circuit A of depth k' calculating the parity function PAR_ℓ.

If T has depth k, then it can be verified that the next-bit predictor P constructed in the proof of Proposition 7.16 also has depth k. (Recall that negations and constants can be propagated to the inputs so they do not contribute to the depth.) Next, in the proof of Theorem 7.24, we obtain A from P by $A(y) = P(f_1(y)f_2(y)\cdots f_{i-1}(y))$ for some $i \in \{1,\ldots,m\}$ and where each f_i depends on at most a bits of y. Now we observe that A can be computed by a small constant-depth circuit (if P can). Specifically, applying Lemma 7.28 to each f_i, the size of A is at most $(m-1)\cdot 2^a + m = m^2$ and the depth of A is at most $k' = k+2$. This contradicts the hardness of PAR_ℓ. □

Corollary 7.30. $\mathbf{BPAC^0} \subset \tilde{\mathbf{P}}$.

With more work, this can be strengthened to actually put $\mathbf{BPAC^0}$ in $\widetilde{\mathbf{AC}}^{\mathbf{0}}$, i.e., uniform constant-depth circuits of quasipolynomial size. (The difficulty is that we use majority voting in the derandomization, but small constant-depth circuits cannot compute majority. However, they can compute an "approximate" majority, and this suffices.)

The above pseudorandom generator can also be used to give a quasipolynomial-time derandomization of the randomized algorithm we saw for approximately counting the number of satisfying assignments to a DNF formula (Theorem 2.34); see Problem 7.4.

Improving the running time of either of these derandomizations to polynomial is an intriguing open problem.

Open Problem 7.31. Show that $\mathbf{BPAC^0} = \mathbf{AC^0}$ or even $\mathbf{BPAC^0} \subset \mathbf{P}$.

Open Problem 7.32 (Open Problem 2.36, restated). Give a deterministic polynomial-time algorithm for approximately counting the number of satisfying assignments to a DNF formula.

We remark that it has recently been shown how to give an average-case $\mathbf{AC^0}$ simulation of $\mathbf{BPAC^0}$ (i.e., the derandomized algorithm is correct on *most* inputs); see Problem 7.5.

Another open problem is to construct similar, unconditional pseudorandom generators as Theorem 7.29 for circuit classes larger than $\mathbf{AC^0}$. A natural candidate is $\mathbf{AC^0}[\mathbf{2}]$, which is the same as $\mathbf{AC^0}$ but augmented with unbounded-fan-in parity gates. There are known explicit functions $f : \{0,1\}^\ell \to \{0,1\}$ (e.g., Majority) for which every $\mathbf{AC^0}[\mathbf{2}]$ circuit of depth k computing f has size at least $s_k(\ell) = 2^{\ell^{\Omega(1/k)}}$, but unfortunately the average-case hardness is much weaker than we need. These functions are only $(s_k(\ell), 1/2 - 1/O(\ell))$-average-case hard, rather than $(s_k(\ell), 1/2 - 1/s_k(\ell))$-average-case hard, so we can only obtain a small stretch using Theorem 7.24 and the following remains open.

Open Problem 7.33. For every constant k and every m, construct a (mildly) explicit $(m, 1/4)$-pseudorandom generator $G_m : \{0,1\}^{m^{o(1)}} \to \{0,1\}^m$ fooling $\mathbf{AC^0}[\mathbf{2}]$ circuits of depth k and size m.

7.5 Worst-Case/Average-Case Reductions and Locally Decodable Codes

In the previous section, we saw how to construct pseudorandom generators from boolean functions that are very hard on average, where every nonuniform algorithm running in time t must err with probability greater than $1/2 - 1/t$ on a random input. Now we want to relax the assumption to refer to worst-case hardness, as captured by the following definition.

Definition 7.34. A function $f : \{0,1\}^\ell \to \{0,1\}$ is *worst-case hard for time t* if, for all nonuniform probabilistic algorithms A running in time t, there exists $x \in \{0,1\}^\ell$ such that $\Pr[A(x) \neq f(x)] > 1/3$, where the probability is over the coin tosses of A.

Note that, for *deterministic* algorithms A, the definition simply says $\exists x\ A(x) \neq f(x)$. In the nonuniform case, restricting to deterministic

algorithms is without loss of generality because we can always derandomize the algorithm using (additional) nonuniformity. Specifically, following the proof that $\mathbf{BPP} \subset \mathbf{P/poly}$, it can be shown that if f is worst-case hard for nonuniform deterministic algorithms running in time t, then it is worst-case hard for nonuniform probabilistic algorithms running in time t' for some $t' = \Omega(t/\ell)$.

A natural goal is to be able to construct an average-case hard function from a worst-case hard function. More formally, given a function $f : \{0,1\}^\ell \to \{0,1\}$ that is worst-case hard for time $t = t(\ell)$, construct a function $\hat{f} : \{0,1\}^{O(\ell)} \to \{0,1\}$ such that $\hat{f}$ is average-case hard for time $t' = t^{\Omega(1)}$. Moreover, we would like $\hat{f}$ to be in $\mathbf{E}$ if f is in $\mathbf{E}$. (Whether we can obtain a similar result for $\mathbf{NP}$ is a major open problem, and indeed there are negative results ruling out natural approaches to doing so.)

Our approach to doing this will be via error-correcting codes. Specifically, we will show that if $\hat{f}$ is the encoding of f in an appropriate kind of error-correcting code, then worst-case hardness of f implies average-case hardness of $\hat{f}$.

Specifically, we view f as a message of length $L = 2^\ell$, and apply an error-correcting code $\mathrm{Enc} : \{0,1\}^L \to \Sigma^{\hat{L}}$ to obtain $\hat{f} = \mathrm{Enc}(f)$, which we view as a function $\hat{f} : \{0,1\}^{\hat{\ell}} \to \Sigma$, where $\hat{\ell} = \log \hat{L}$. Pictorially:

$$\boxed{\text{message } f : \{0,1\}^\ell \to \{0,1\}} \longrightarrow \boxed{\text{Enc}} \longrightarrow \boxed{\text{codeword } \hat{f} : \{0,1\}^{\hat{\ell}} \to \Sigma}.$$

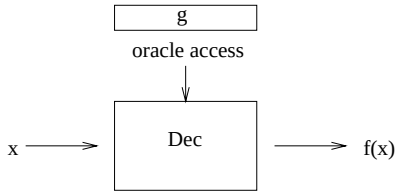
(Ultimately, we would like $\Sigma = \{0,1\}$, but along the way we will discuss larger alphabets.)

Now we argue the average-case hardness of $\hat{f}$ as follows. Suppose, for contradiction, that $\hat{f}$ is not δ average-case hard. By definition, there exists an efficient algorithm A with $\Pr[A(x) = \hat{f}(x)] > 1 - \delta$. We may assume that A is deterministic by fixing its coins. Then A may be viewed as a received word in $\Sigma^{\hat{L}}$, and our condition on A becomes $d_H(A, \hat{f}) < \delta$. So if Dec is a δ-decoding algorithm for Enc, then $\mathrm{Dec}(A) = f$. By assumption A is efficient, so if Dec is efficient, then f may be efficiently computed everywhere. This would contradict our worst-case hardness assumption, assuming that $\mathrm{Dec}(A)$ gives a time $t(\ell)$ algorithm for f. However, the standard notion of decoding requires reading all $2^{\hat{\ell}}$ values of the received word A and writing all 2^ℓ

values of the message $\mathrm{Dec}(A)$, and thus $\mathrm{Time}(\mathrm{Dec}(A)) \gg 2^\ell$. But every function on ℓ bits can be computed in nonuniform time 2^ℓ, and even in the uniform case we are mostly interested in $t(\ell) \ll 2^\ell$. To solve this problem we introduce the notion of local decoding.

Definition 7.35. A *local δ-decoding algorithm* for a code $\mathrm{Enc} : \{0,1\}^L \to \Sigma^{\hat{L}}$ is a probabilistic oracle algorithm Dec with the following property. Let $f : [L] \to \{0,1\}$ be any message with associated codeword $\hat{f} = \mathrm{Enc}(f)$, and let $g : [\hat{L}] \to \Sigma$ be such that $d_H(g, \hat{f}) < \delta$. Then for all $x \in [L]$ we have $\Pr[\mathrm{Dec}^g(x) = f(x)] \geq 2/3$, where the probability is taken over the coins flips of Dec.

In other words, given oracle access to g, we want to efficiently compute any desired bit of f with high probability. So both the input (namely g) and the output (namely f) are treated implicitly; the decoding algorithm does not need to read/write either in its entirety. Pictorially:



This makes it possible to have sublinear-time (or even polylogarithmic-time) decoding. Also, we note that the bound of 2/3 in the definition can be amplified in the usual way. Having formalized a notion of local decoding, we can now make our earlier intuition precise.

Proposition 7.36. Let Enc be an error-correcting code with local δ-decoding algorithm Dec that runs in nonuniform time at most t_{Dec} (meaning that Dec is an boolean circuit of size at most t_{Dec} equipped with oracle gates), and let f be worst-case hard for nonuniform time t. Then $\hat{f} = \mathrm{Enc}(f)$ is (t', δ) average-case hard, where $t' = t/t_{\mathrm{Dec}}$.

Proof. We do everything as explained before except with Dec^A in place of $\mathrm{Dec}(A)$, and now the running time is at most $\mathrm{Time}(\mathrm{Dec}) \cdot \mathrm{Time}(A)$. (We substitute each oracle gate in the circuit for Dec with the circuit for A.) □

We note that the reduction in this proof does not use nonuniformity in an essential way. We used nonuniformity to fix the coin tosses of A, making it deterministic. To obtain a version for hardness against uniform probabilistic algorithms, the coin tosses of A can be chosen and fixed randomly instead. With high probability, the fixed coins will not increase As error by more than a constant factor (by Markov's Inequality); we can compensate for this by replacing the (t', δ) average-case hardness in the conclusion with, say, $(t', \delta/3)$ average-case hardness.

In light of the above proposition, our task is now to find an error-correcting code $\mathrm{Enc} : \{0,1\}^{\mathbf{L}} \to \Sigma^{\hat{L}}$ with a local decoding algorithm. Specifically, we would like the following parameters.

(1) We want $\hat{\ell} = O(\ell)$, or equivalently $\hat{L} = \mathrm{poly}(L)$. This is because we measure hardness as a function of input length (which in turn translates to the relationship between output length and seed length of pseudorandom generators obtained via Theorem 7.19). In particular, when $t = 2^{\Omega(\ell)}$, we'd like to achieve $t' = 2^{\Omega(\hat{\ell})}$. Since $t' < t$ in Proposition 7.36, this is only possible if $\hat{\ell} = O(\ell)$.
(2) We would like Enc to be computable in time $2^{O(\hat{\ell})} = \mathrm{poly}(\hat{L})$, which is $\mathrm{poly}(L)$ if we satisfy the requirement $\hat{L} = \mathrm{poly}(L)$. This is because we want $f \in \mathbf{E}$ to imply $\hat{f} \in \mathbf{E}$.
(3) We would like $\Sigma = \{0,1\}$ so that $\hat{f}$ is a boolean function, and $\delta = 1/2 - \varepsilon$ so that $\hat{f}$ has sufficient average-case hardness for the pseudorandom generator construction of Theorem 7.24.
(4) Since $\hat{f}$ will be average-case hard against time $t' = t/t_{\mathrm{Dec}}$, we would want the running time of Dec to be $t_{\mathrm{Dec}} = \mathrm{poly}(\ell, 1/\varepsilon)$ so that we can take $\varepsilon = t^{\Omega(1)}$ and still have $t' = t^{\Omega(1)}/\mathrm{poly}(\ell)$.

Of course, achieving $\delta = 1/2 - \varepsilon$ is not possible with our current notion of local *unique* decoding (which is only harder than the standard notion of unique decoding), and thus in the next section

we will focus on getting δ to be just a fixed constant. In Section 7.6, we will introduce a notion of local *list* decoding, which will enable decoding from distance $\delta = 1/2 - \varepsilon$.

In our constructions, it will be more natural to focus on the task of decoding *codeword* symbols rather than message symbols. That is, we replace the message f with the codeword $\hat{f}$ in Definition 7.35 to obtain the following notion:

Definition 7.37 (Locally Correctible Codes).[4] A *local δ-correcting algorithm* for a code $\mathcal{C} \subset \Sigma^{\hat{L}}$ is a probabilistic oracle algorithm Dec with the following property. Let $\hat{f} \in \mathcal{C}$ be any codeword, and let $g : [\hat{L}] \to \Sigma$ be such that $d_H(g, \hat{f}) < \delta$. Then for all $x \in [\hat{L}]$ we have $\Pr[\mathrm{Dec}^g(x) = \hat{f}(x)] \geq 2/3$, where the probability is taken over the coin flips of Dec.

This implies the standard definition of locally decodable codes under the (mild) constraint that the message symbols are explicitly included in the codeword, as captured by the following definition (see also Problem 5.4).

Definition 7.38 (Systematic Encodings). An encoding algorithm $\mathrm{Enc} : \{0,1\}^L \to \mathcal{C}$ for a code $\mathcal{C} \subset \Sigma^{\hat{L}}$ is *systematic* if there is a polynomial-time computable function $I : [L] \to [\hat{L}]$ such that for all $f \in \{0,1\}^L$, $\hat{f} = \mathrm{Enc}(f)$, and all $x \in [L]$, we have $\hat{f}(I(x)) = f(x)$, where we interpret 0 and 1 as elements of Σ in some canonical way.

Informally, this means that the message f can be viewed as the restriction of the codeword $\hat{f}$ to the coordinates in the image of I.

Lemma 7.39. If $\mathrm{Enc} : \{0,1\}^L \to \mathcal{C}$ is systematic and $\mathcal{C}$ has a local δ-correcting algorithm running in time t, then Enc has a local δ-decoding algorithm (in the standard sense) running in time $t + \mathrm{poly}(\log L)$.

Proof. If Dec_1 is the local corrector for $\mathcal{C}$ and I the mapping in the definition of systematic encoding, then $\mathrm{Dec}_2^g(x) = \mathrm{Dec}_1^g(I(x))$ is a local decoder for Enc. □

[4] In the literature, these are often called *self-correctible codes*.

7.5.1 Local Decoding Algorithms

Hadamard Code. Recall the Hadamard code of message length m, which consists of the truth tables of all $\mathbb{Z}_2$-linear functions $c : \{0,1\}^m \to \{0,1\}$ (Construction 5.12).

Proposition 7.40. The Hadamard code $\mathcal{C} \subset \{0,1\}^{2^m}$ of message length m has a local $(1/4 - \varepsilon)$-correcting algorithm running in time $\mathrm{poly}(m, 1/\varepsilon)$.

Proof. We are given oracle access to $g : \{0,1\}^m \to \{0,1\}$ that is at distance less than $1/4 - \varepsilon$ from some (unknown) linear function c, and we want to compute $c(x)$ at an arbitrary point $x \in \{0,1\}^m$. The idea is *random self-reducibility*: we can reduce computing c at an arbitrary point to computing c at uniformly random points, where g is likely to give the correct answer. Specifically, $c(x) = c(x \oplus r) \oplus c(r)$ for every r, and both $x \oplus r$ and r are uniformly distributed if we choose $r \overset{R}{\leftarrow} \{0,1\}^m$. The probability that g differs from c at either of these points is less than $2 \cdot (1/4 - \varepsilon) = 1/2 - 2\varepsilon$. Thus $g(x \oplus r) \oplus g(r)$ gives the correct answer with probability noticeably larger than $1/2$. We can amplify this success probability by repetition. Specifically, we obtain the following local corrector:

Algorithm 7.41 (Local Corrector for Hadamard Code). Input: An oracle $g : \{0,1\}^m \to \{0,1\}$, $x \in \{0,1\}^m$, and a parameter $\varepsilon > 0$

(1) Choose $r_1, \ldots, r_t \overset{R}{\leftarrow} \{0,1\}^m$, for $t = O(1/\varepsilon^2)$.
(2) Query $g(r_i)$ and $g(r_i \oplus x)$ for each $i = 1, \ldots, t$.
(3) Output $\mathrm{maj}_{1 \le i \le t}\{g(r_i) \oplus g(r_i \oplus x)\}$.

If $d_H(g,c) < 1/4 - \varepsilon$, then this algorithm will output $c(x)$ with probability at least $2/3$. □

This local decoding algorithm is optimal in terms of its decoding distance (arbitrarily close to $1/4$) and running time (logarithmic in

the blocklength), but the problem is that the Hadamard code has exponentially small rate.

Reed–Muller Code. Recall that the q-ary Reed–Muller code of degree d and dimension m consists of all multivariate polynomials $p : \mathbb{F}_q^m \to \mathbb{F}_q$ of total degree at most d. (Construction 5.16.) This code has minimum distance $\delta = 1 - d/q$. Reed–Muller Codes are a common generalization of both Hadamard and Reed–Solomon codes, and thus we can hope that for an appropriate setting of parameters, we will be able to get the best of both kinds of codes. That is, we want to combine the efficient local decoding of the Hadamard code with the good rate of Reed–Solomon codes.

Theorem 7.42. The q-ary Reed–Muller Code of degree d and dimension m has a local $1/12$-correcting algorithm running in time $\mathrm{poly}(m, q)$ provided $d \leq q/9$ and $q \geq 36$.

Note the running time of the decoder is roughly the mth root of the block length $\hat{L} = q^m$. When $m = 1$, our decoder can query the entire string and we simply obtain a global decoding algorithm for Reed–Solomon Codes (which we already know how to achieve from Theorem 5.19). But for large enough m, the decoder can only access a small fraction of the received word. (In fact, one can improve the running time to $\mathrm{poly}(m, d, \log q)$, but the weaker result above is sufficient for our purposes.)

The key idea behind the decoder is to do restrictions to random *lines* in $\mathbb{F}^m$. The restriction of a Reed–Muller codeword to such a line is a Reed–Solomon codeword, and we can afford to run our global Reed–Solomon decoding algorithm on the line.

Formally, for $x, y \in \mathbb{F}^m$, we define the *(parameterized) line through x in direction y* as the function $\ell_{x,y} : \mathbb{F} \to \mathbb{F}^m$ given by $\ell_{x,y}(t) = x + ty$. Note that for every $a \in F, b \in \mathbb{F} \setminus \{0\}$, the line $\ell_{x+ay,by}$ has the same set of points in its image as $\ell_{x,y}$; we refer to this set of points as an *unpolymerized line*. When $y = 0$, the parameterized line contains only the single point x.

If $g : \mathbb{F}^m \to \mathbb{F}$ is any function and $\ell : \mathbb{F} \to \mathbb{F}^m$ is a line, then we use $g|_\ell$ to denote the *restriction of g to ℓ*, which is simply the composition $g \circ \ell : \mathbb{F} \to \mathbb{F}$. Note that if p is any polynomial of total degree at most d, then $p|_\ell$ is a (univariate) polynomial of degree at most d.

So we are given an oracle g of distance less than δ from some degree d polynomial $p : \mathbb{F}^m \to \mathbb{F}$, and we want to compute $p(x)$ for some $x \in \mathbb{F}^m$. We begin by choosing a random line ℓ through x. Every point of $\mathbb{F}^m \setminus \{x\}$ lies on exactly one parameterized line through x, so the points on ℓ (except x) are distributed uniformly at random over the whole domain, and thus g and p are likely to agree on these points. Thus we can hope to use the points on this line to reconstruct the value of $p(x)$. If δ is sufficiently small compared to the degree (e.g., $\delta = 1/3(d+1)$), we can simply interpolate the value of $p(x)$ from $d+1$ points on the line. This gives rise to the following algorithm.

Algorithm 7.43 (Local Corrector for Reed–Muller Code I). Input: An oracle $g\colon \mathbb{F}^m \to \mathbb{F}$, an input $x \in \mathbb{F}^m$, and a degree parameter d

(1) Choose $y \stackrel{R}{\leftarrow} \mathbb{F}^m$. Let $\ell = \ell_{x,y} : \mathbb{F} \to \mathbb{F}^m$ be the line through x in direction y.
(2) Query g to obtain $\beta_0 = g|_\ell(\alpha_0) = g(\ell(a_0)), \ldots, \beta_d = g|_\ell(\alpha_d) = g(\ell(a_d))$, where $\alpha_0, \ldots, \alpha_d \in \mathbb{F} \setminus \{0\}$ are any fixed points
(3) Interpolate to find the unique univariate polynomial q of degree at most d s.t. $\forall i, q(\alpha_i) = \beta_i$
(4) Output $q(0)$

Claim 7.44 If g has distance less than $\delta = 1/3(d+1)$ from some polynomial p of degree at most d, then Algorithm 7.43 will output $p(x)$ with probability greater than $2/3$.

Proof of Claim: Observe that for all $x \in \mathbb{F}^m$ and $\alpha_i \in \mathbb{F} \setminus \{0\}$, $\ell_{x,y}(\alpha_i)$ is uniformly random in $\mathbb{F}^m$ over the choice of $y \stackrel{R}{\leftarrow} \mathbb{F}^m$. This

implies that for each i,

$$\Pr_{\ell}[g|_{\ell}(\alpha_i) \neq p|_{\ell}(\alpha_i)] < \delta = \frac{1}{3(d+1)}.$$

By a union bound,

$$\Pr_{\ell}[\exists i, g|_{\ell}(\alpha_i) \neq p|_{\ell}(\alpha_i)] < (d+1) \cdot \delta = \frac{1}{3}.$$

Thus, with probability greater than 2/3, we have $\forall i, q(\alpha_i) = p|_{\ell}(\alpha_i)$ and hence $q(0) = p(x)$. The running time of the algorithm is $\text{poly}(m, q)$. □

We now show how to improve the decoder to handle a larger fraction of errors, up to distance $\delta = 1/12$. We alter Steps 7.43 and 7.43 in the above algorithm. In Step 7.43, instead of querying only $d+1$ points, we query over *all* points in ℓ. In Step 7.43, instead of interpolation, we use a *global* decoding algorithm for Reed–Solomon codes to decode the univariate polynomial $p|_{\ell}$. Formally, the algorithm proceeds as follows.

Algorithm 7.45 (Local Corrector for Reed–Muller Codes II).
Input: An oracle $g\colon \mathbb{F}^m \to \mathbb{F}$, an input $x \in \mathbb{F}^m$, and a degree parameter d, where $q = |\mathbb{F}| \geq 36$ and $d \leq q/9$.

(1) Choose $y \overset{R}{\leftarrow} \mathbb{F}^m$. Let $\ell = \ell_{x,y} : \mathbb{F} \to \mathbb{F}^m$ be the line through x in direction y.
(2) Query g at all points on ℓ to obtain $g|_{\ell} : \mathbb{F} \to \mathbb{F}$.
(3) Run the 1/3-decoder for the q-ary Reed–Solomon code of degree d on $g|_{\ell}$ to obtain the (unique) polynomial q at distance less than 1/3 from $g|_{\ell}$ (if one exists).[5]
(4) Output $q(0)$.

---

[5] A 1/3-decoder for Reed–Solomon codes follows from the $(1 - 2\sqrt{d/q})$ list-decoding algorithm of Theorem 5.19. Since $1/3 \leq 1 - 2\sqrt{d/q}$, the list-decoder will produce a list containing all univariate polynomials at distance less than 1/3, and since 1/3 is smaller than half the minimum distance $(1 - d/q)$, there will be only one good decoding.

Claim 7.46. If g has distance less than $\delta = 1/12$ from some polynomial p of degree at most d, and the parameters satisfy $q = |\mathbb{F}| \geq 36$, $d \leq q/9$, then Algorithm 7.45 will output $p(x)$ with probability greater than $2/3$.

Proof of Claim: The expected distance (between $g|_\ell$ and $p|_\ell$) is small:

$$\mathop{\mathrm{E}}_{\ell}[d_H(g|_\ell, p|_\ell)] < \frac{1}{q} + \delta = \frac{1}{36} + \frac{1}{12} = \frac{1}{9},$$

where the term $1/q$ is due to the fact that the point x is not random. Therefore, by Markov's Inequality,

$$\Pr[d_H(g|_\ell, p|_\ell) \geq 1/3] \leq 1/3.$$

Thus, with probability at least $2/3$, we have that $p|_\ell$ is the unique polynomial of degree at most d at distance less than $1/3$ from $g|_\ell$ and thus q must equal $p|_\ell$. □

7.5.2 Low-Degree Extensions

Recall that to obtain locally decodable codes from locally correctible codes (as constructed above), we need to exhibit systematic encoding: (Definition 7.38.) Thus, given $f : [L] \to \{0,1\}$, we want to encode it as a Reed–Muller codeword $\hat{f} : [\hat{L}] \to \Sigma$ s.t.:

- The encoding time is $2^{O(\ell)} = \mathrm{poly}(L)$.
- $\hat{\ell} = O(\ell)$, or equivalently $\hat{L} = \mathrm{poly}(L)$.
- The code is systematic in the sense of Definition 7.38.

Note that the usual encoding for Reed–Muller codes, where the message gives the coefficients of the polynomial, is not systematic. Instead the message should correspond to evaluations of the polynomial at certain points. Once we settle on the set of evaluation points, the task becomes one of interpolating the values at these points (given by the message) to a low-degree polynomial defined everywhere.

The simplest approach is to use the boolean hypercube as the set of evaluation points.

Lemma 7.47. (multilinear extension) For every $f\colon \{0,1\}^\ell \to \{0,1\}$ and every finite field $\mathbb{F}$, there exists a (unique) polynomial $\hat{f}\colon \mathbb{F}^\ell \to \mathbb{F}$ such that $\hat{f}|_{\{0,1\}^\ell} \equiv f$ and $\hat{f}$ has degree at most 1 in each variable (and hence total degree at most ℓ).

Proof. We prove the existence of the polynomial $\hat{f}$. Define

$$\hat{f}(x_1,\ldots,x_\ell) = \sum_{\alpha\in\{0,1\}^\ell} f(\alpha)\delta_\alpha(x)$$

for

$$\delta_\alpha(x) = \left(\prod_{i\colon \alpha_i=1} x_i\right)\left(\prod_{i\colon \alpha_i=0} (1-x_i)\right)$$

Note that for $x \in \{0,1\}^\ell$, $\delta_\alpha(x) = 1$ only when $\alpha = x$, therefore $\hat{f}|_{\{0,1\}^\ell} \equiv f$. We omit the proof of uniqueness. The bound on the individual degrees is by inspection. □

Thinking of $\hat{f}$ as an encoding of f, let's inspect the properties of this encoding.

- Since the total degree of the multilinear extension can be as large as ℓ, we need $q \geq 9\ell$ for the local corrector of Theorem 7.42 to apply.
- The encoding time is $2^{O(\hat{\ell})}$, as computing a single point of $\hat{f}$ requires summing over 2^ℓ elements, and we have $2^{\hat{\ell}}$ points on which to compute $\hat{f}$.
- The code is systematic, since $\hat{f}$ is an extension of f.
- However, the input length is $\hat{\ell} = \ell \log q = \Theta(\ell \log \ell)$, which is slightly larger than our target of $\hat{\ell} = O(\ell)$.

To solve the problem of the input length $\hat{\ell}$ in the multilinear encoding, we reduce the dimension of the polynomial $\hat{f}$ by changing the embedding of the domain of f: Instead of interpreting $\{0,1\}^\ell \subset \mathbb{F}^\ell$ as an embedding of the domain of f in $\mathbb{F}^\ell$, we map $\{0,1\}^\ell$ to $\mathbb{H}^m$ for some subset $\mathbb{H} \subset \mathbb{F}$, and as such embed it in $\mathbb{F}^m$.

More precisely, we fix a subset $\mathbb{H} \subset \mathbb{F}$ of size $|\mathbb{H}| = \lceil \sqrt{q} \rceil$. Choose $m = \lceil \ell / \log |\mathbb{H}| \rceil$, and fix some efficient one-to-one mapping from $\{0,1\}^\ell$ into $\mathbb{H}^m$. With this mapping, view f as a function $f \colon \mathbb{H}^m \to \mathbb{F}$.

Analogously to before, we have the following.

Lemma 7.48. (low-degree extension) For every finite field $\mathbb{F}$, $\mathbb{H} \subset \mathbb{F}$, $m \in \mathbb{N}$, and function $f : \mathbb{H}^m \to \mathbb{F}$, there exists a (unique) $\hat{f} \colon \mathbb{F}^m \to \mathbb{F}$ such that $\hat{f}|_{\mathbb{H}^m} \equiv f$ and $\hat{f}$ has degree at most $|\mathbb{H}| - 1$ in each variable (and hence has total degree at most $m \cdot (|\mathbb{H}| - 1)$).

Using $|\mathbb{H}| = \lceil \sqrt{q} \rceil$, the total degree of $\hat{f}$ is at most $d = \ell \sqrt{q}$. So we can apply the local corrector of Theorem 7.42, as long as $q \geq 81\ell^2$ (so that $d \leq q/9$). Inspecting the properties of $\hat{f}$ as an encoding of f, we have:

- The input length is $\hat{\ell} = m \cdot \log q = \lceil \ell / \log |\mathbb{H}| \rceil \cdot \log q = O(\ell)$, as desired. (We can use a field of size 2^k for $k \in \mathbb{N}$, so that $\hat{L}$ is a power of 2 and we incur no loss in encoding inputs to $\hat{f}$ as bits.)
- The code is systematic as long as our mapping from $\{0,1\}^\ell$ to $\mathbb{H}^m$ is efficient.

Note that not every polynomial of total degree at most $m \cdot (|\mathbb{H}| - 1)$ is the low-degree extension of a function $f : \mathbb{H}^m \to \mathbb{F}$, so the image of our encoding function $f \mapsto \hat{f}$ is only a *subcode* of the Reed–Muller code. This is not a problem, because any subcode of a locally correctible code is also locally correctible, and we can afford the loss in rate (all we need is $\hat{\ell} = O(\ell)$).

7.5.3 Putting It Together

Combining Theorem 7.42 with Lemmas 7.48, and 7.39, we obtain the following locally decodable code:

Proposition 7.49. For every $L \in \mathbb{N}$, there is an explicit code $\mathrm{Enc} : \{0,1\}^L \to \Sigma^{\hat{L}}$, with blocklength $\hat{L} = \mathrm{poly}(L)$ and alphabet size $|\Sigma| = \mathrm{poly}(\log L)$, that has a local $(1/12)$-decoder running in time $\mathrm{poly}(\log L)$.

Using Proposition 7.36, we obtain the following conversion from worst-case hardness to average-case hardness:

Proposition 7.50. If there exists $f\colon \{0,1\}^\ell \to \{0,1\}$ in **E** that is worst-case hard against (nonuniform) time $t(\ell)$, then there exists $\hat{f}\colon \{0,1\}^{O(\ell)} \to \{0,1\}^{O(\log \ell)}$ in **E** that is $(t'(\ell), 1/12)$ average-case hard for $t'(\ell) = t(\ell)/\mathrm{poly}(\ell)$.

This differs from our original goal in two ways: $\hat{f}$ is not Boolean, and we only get hardness $1/12$ (instead of $1/2 - \varepsilon$). The former concern can be remedied by concatenating the code of Proposition 7.49 with a Hadamard code, similarly to Problem 5.2. Note that the Hadamard code is applied on message space Σ, which is of size $\mathrm{polylog}(L)$, so it can be $1/4$-decoded by brute-force in time $\mathrm{polylog}(L)$ (which is the amount of time already taken by our decoder).[6] Using this, we obtain:

Theorem 7.51. For every $L \in \mathbb{N}$, there is an explicit code $\mathrm{Enc} : \{0,1\}^L \to \{0,1\}^{\hat{L}}$ with blocklength $\hat{L} = \mathrm{poly}(L)$ that has a local $(1/48)$-decoder running in time $\mathrm{poly}(\log L)$.

Theorem 7.52. If there exists $f\colon \{0,1\}^\ell \to \{0,1\}$ in **E** that is worst-case hard against time $t(\ell)$, then there exists $\hat{f}\colon \{0,1\}^{O(\ell)} \to \{0,1\}$ in **E** that is $(t'(\ell), 1/48)$ average-case hard, for $t'(\ell) = t(\ell)/\mathrm{poly}(\ell)$.

An improved decoding distance can be obtained using Problem 7.7.

We note that the local decoder of Theorem 7.51 not only runs in time $\mathrm{poly}(\log L)$, but also makes $\mathrm{poly}(\log L)$ queries. For some applications (such as Private Information Retrieval, see Problem 7.6), it is important to have the number q of queries be as small as possible, ideally a constant. Using Reed–Muller codes of constant degree, it is possible to obtain constant-query locally decodable codes, but the

[6] Some readers may recognize this concatenation step as the same as applying the "Goldreich–Levin hardcore predicate" to $\hat{f}$. (See Problems 7.12 and 7.13.) However, for the parameters we are using, we do not need the power of these results, and can afford to perform brute-force unique decoding instead.

blocklength will be $\hat{L} = \exp(L^{1/(q-1)})$. In a recent breakthrough, it was shown how to obtain constant-query locally decodable codes with blocklength $\hat{L} = \exp(L^{o(1)})$. Obtaining polynomial blocklength remains open.

Open Problem 7.53. Are there binary codes that are locally decodable with a constant number of queries (from constant distance $\delta > 0$) and blocklength polynomial in the message length?

7.5.4 Other Connections

As shown in Problem 7.6, locally decodable codes are closely related to protocols for *private information retrieval*. Another connection, and actually the setting in which these local decoding algorithms were first discovered, is to *program self-correctors*. Suppose we have a program for computing a function, such as the Determinant, which happens to be a codeword in a locally decodable code (e.g., the determinant is a low-degree multivariate polynomial, and hence a Reed–Muller codeword). Then, even if this program has some bugs and gives the wrong answer on some small fraction of inputs, we can use the local decoding algorithm to obtain the correct answer on *all* inputs with high probability.

7.6 Local List Decoding and PRGs from Worst-Case Hardness

7.6.1 Hardness Amplification

In the previous section, we saw how to use locally decodable codes to convert worst-case hard functions into ones with constant average-case hardness (Theorem 7.52). Now our goal is to amplify this hardness (e.g., to $1/2 - \varepsilon$).

There are some generic techniques for hardness amplification. In these methods, we evaluate the function on many independent inputs. For example, consider f' that concatenates the evaluations of $\hat{f}$ on k independent inputs:

$$f'(x_1, \ldots, x_k) = (\hat{f}(x_1), \ldots, \hat{f}(x_k)).$$

Intuitively, if $\hat{f}$ is 1/12 average-case hard, then f' should be $(1 - (11/12)^k)$-average case hard because any efficient algorithm can solve each instance correctly with probability at most 11/12. Proving this is nontrivial (because the algorithm trying to compute f' need not behave independently on the k instances), but there are Direct Product Theorems showing that the hardness does get amplified essentially as expected. Similarly, if we take the XOR on k independent inputs, the XOR Lemma says that the hardness approaches 1/2 exponentially fast.

The main disadvantage of these approaches (for our purposes) is that the input length of f' is $k\ell$ while we aim for input length of $O(\ell)$. To overcome this problem, it is possible to use derandomized products, where we evaluate $\hat{f}$ on *correlated* inputs instead of independent ones.

We will take a different approach, generalizing the notion and algorithms for locally decodable codes to locally *list*-decodable codes, and thereby directly construct $\hat{f}$ that is $(1/2 - \varepsilon)$-hard. Nevertheless, the study of hardness amplification is still of great interest, because it (or variants) can be employed in settings where doing a global encoding of the function is infeasible (e.g., for amplifying the average-case hardness of functions in complexity classes lower than **E**, such as **NP**, and for amplifying the security of cryptographic primitives). We remark that results on hardness amplification can be interpreted in a coding-theoretic language as well, as converting locally decodable codes with a small decoding distance into locally list-decodable codes with a large decoding distance. (See Section 8.2.3.)

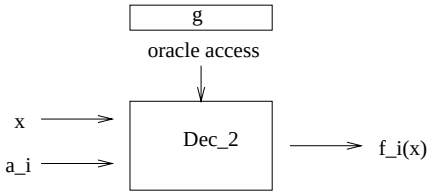
7.6.2 Definition

We would like to formulate a notion of local *list*-decoding to enable us to have binary codes that are locally decodable from distances close to 1/2. This is somewhat tricky to define — what does it mean to produce a "list" of decodings when only asked to decode a particular coordinate? Let g be our received word, and $\hat{f}_1, \hat{f}_2, \ldots, \hat{f}_s$ the codewords that are close to g. One option would be for the decoding algorithm, on input x, to output a set of values $\mathrm{Dec}^g(x) \subset \Sigma$ that is guaranteed to contain $\hat{f}_1(x), \hat{f}_2(x), \ldots \hat{f}_s(x)$ with high probability. However, this is not very useful; in the common case that $s \geq |\Sigma|$, the list could always

be $\mathrm{Dec}^g(x) = \Sigma$. Rather than outputting all of the values, we want to be able to specify to our decoder *which* $\hat{f}_i(x)$ to output. We do this with a two-phase decoding algorithm $(\mathrm{Dec}_1, \mathrm{Dec}_2)$, where both phases can be randomized.

(1) Dec_1, using g as an oracle and not given any other input other than the parameters defining the code, returns a list of advice strings $a_1, a_2, \ldots, a_s$, which can be thought of as "labels" for each of the codewords close to g.
(2) Dec_2 (again, using oracle access to g), takes input x and a_i, and outputs $\hat{f}_i(x)$.

The picture for Dec_2 is much like our old decoder, but it takes an extra input a_i corresponding to one of the outputs of Dec_1:



More formally:

Definition 7.54. A *local δ-list-decoding algorithm* for a code Enc is a pair of probabilistic oracle algorithms $(\mathrm{Dec}_1, \mathrm{Dec}_2)$ such that for all received words g and all codewords $\hat{f} = \mathrm{Enc}(f)$ with $d_H(\hat{f}, g) < \delta$, the following holds. With probability at least $2/3$ over $(a_1, \ldots, a_s) \leftarrow \mathrm{Dec}_1^g$, there exists an $i \in [s]$ such that

$$\forall x, \Pr[\mathrm{Dec}_2^g(x, a_i) = f(x)] \geq 2/3.$$

Note that we don't explicitly require a bound on the list size s (to avoid introducing another parameter), but certainly it cannot be larger than the running time of Dec_1.

As we did for locally (unique-)decodable codes, we can define a *local δ-list-correcting algorithm*, where Dec_2 should recover arbitrary symbols of the codeword $\hat{f}$ rather than the message f. In this case, we don't require that for all j, $\mathrm{Dec}_2^g(\cdot, a_j)$ is a codeword, or that it is

close to g; in other words, some of the a_js may be junk. Analogously to Lemma 7.39, a local δ-list-correcting algorithm implies local δ-list-decoding if the code is systematic.

Proposition 7.36 shows how locally decodable codes convert functions that are hard in the worst case to ones that are hard on average. The same is true for local list-decoding:

Proposition 7.55. Let Enc be an error-correcting code with local δ-list-decoding algorithm $(\mathrm{Dec}_1, \mathrm{Dec}_2)$ where Dec_2 runs in time at most t_{Dec}, and let f be worst-case hard for non-uniform time t. Then $\hat{f} = \mathrm{Enc}(f)$ is (t', δ) average-case hard, where $t' = t/t_{\mathrm{Dec}}$.

Proof. Suppose for contradiction that $\hat{f}$ is not (t', δ)-hard. Then some nonuniform algorithm A running in time t' computes $\hat{f}$ with error probability smaller than δ. But if Enc has a local δ list-decoding algorithm, then (with A playing the role of g) that means there exists a_i (one of the possible outputs of Dec_1^A), such that $\mathrm{Dec}_2^A(\cdot, a_i)$ computes $f(\cdot)$ everywhere. Hardwiring a_i as advice, $\mathrm{Dec}_2^A(\cdot, a_i)$ is a nonuniform algorithm running in time at most $\mathrm{time}(A) \cdot \mathrm{time}(\mathrm{Dec}_2) \leq t$. □

Note that, in contrast to Proposition 7.36, here we are using nonuniformity more crucially, in order to select the right function from the list of possible decodings. As we will discuss in Section 7.7.1, this use of nonuniformity is essential for "black-box" constructions, that do not exploit any structure in the hard function f or the adversary (A in the above proof). However, there are results on hardness amplification against uniform algorithms, which use structure in the hard function f (e.g., that it is complete for a complexity class like **E** or **NP**) to identify it among the list of decodings without any nonuniform advice.

7.6.3 Local List-Decoding Reed–Muller Codes

Theorem 7.56. There is a universal constant c such that the q-ary Reed–Muller code of degree d and dimension m over can be locally $(1 - \varepsilon)$-list-corrected in time $\mathrm{poly}(q, m)$ for $\varepsilon = c\sqrt{d/q}$.

Note that the distance at which list decoding can be done approaches 1 as $q/d \to \infty$. It matches the bound for list-decoding Reed–Solomon codes (Theorem 5.19) up to the constant c. Moreover, as the dimension m increases, the running time of the decoder ($\text{poly}(q, m)$) becomes much smaller than the block length ($q^m \cdot \log q$), at the price of a reduced rate ($\binom{m+d}{m}/q^m$).

Proof. Suppose we are given an oracle $g : \mathbb{F}^m \to \mathbb{F}$ that is $(1 - \varepsilon)$ close to some unknown polynomial $p : \mathbb{F}^m \to \mathbb{F}$, and that we are given an $x \in \mathbb{F}^m$. Our goal is to describe two algorithms, Dec_1 and Dec_2, where Dec_2 is able to compute $p(x)$ using a piece of Dec_1's output (i.e., advice).

The advice that we will give to Dec_2 is the value of p on a single point. Dec_1 can easily generate a (reasonably small) list that contains one such point by choosing a random $y \in \mathbb{F}^m$, and outputting all pairs (y, z), for $z \in \mathbb{F}$. More formally:

Algorithm 7.57 (Reed–Muller Local List-Decoder Dec_1).
Input: An oracle $g\colon \mathbb{F}^m \to \mathbb{F}$ and a degree parameter d

(1) Choose $y \xleftarrow{R} \mathbb{F}^m$
(2) Output $\{(y, z) : z \in \mathbb{F}\}$

This first-phase decoder is rather trivial in that it doesn't make use of the oracle access to the received word g. It is possible to improve both the running time and list size of Dec_1 by using oracle access to g, but we won't need those improvements below.

Now, the task of Dec_2 is to calculate $p(x)$, given the value of p on some point y. Dec_2 does this by looking at g restricted to the line through x and y, and using the list-decoding algorithm for Reed–Solomon Codes to find the univariate polynomials $q_1, q_2, \ldots, q_t$ that are close to g. If exactly one of these polynomials q_i agrees with p on the test point y, then we can be reasonably confident that $q_i(x) = p(x)$.

In more detail, the decoder works as follows:

Algorithm 7.58 (Reed–Muller Local List-Corrector Dec_2).
Input: An oracle $g\colon \mathbb{F}^m \to \mathbb{F}$, an input $x \in \mathbb{F}^m$, advice $(y, z) \in \mathbb{F}^m \times \mathbb{F}$,

and a degree parameter d

(1) Let $\ell = \ell_{x,y-x} : \mathbb{F} \to \mathbb{F}^m$ be the line through x and y (so that $\ell(0) = x$ and $\ell(1) = y$).
(2) Run the $(1 - \varepsilon/2)$-list-decoder for Reed–Solomon Codes (Theorem 5.19) on $g|_\ell$ to get all univariate polys $q_1, \ldots, q_t$ that agree with $g|_\ell$ in greater than an $\varepsilon/2$ fraction of points.
(3) If there exists a unique i such that $q_i(1) = z$, output $q_i(0)$. Otherwise, fail.

Now that we have fully specified the algorithms, it remains to analyze them and show that they decode p correctly. Observe that it suffices to compute p on greater than an 11/12 fraction of the points x, because then we can apply the unique local correcting algorithm of Theorem 7.42. Therefore, to finish the proof of the theorem we must prove the following.

Claim 7.59. Suppose that $g : \mathbb{F}^m \to \mathbb{F}$ has agreement greater than ε with a polynomial $p : \mathbb{F}^m \to \mathbb{F}$ of degree at most d. For at least half of the points $y \in \mathbb{F}^m$ the following holds for greater than an 11/12 fraction of lines ℓ going through y:

(1) $\mathrm{agr}(g|_\ell, p|_\ell) > \varepsilon/2$.
(2) There does not exist any univariate polynomial q of degree at most d other than $p|_\ell$ such that $\mathrm{agr}(g|_\ell, q) > \varepsilon/2$ and $q(y) = p(y)$.

Proof of Claim: It suffices to show that Items 7.59 and 7.59 hold with probability 0.99 over the choice of a random point $y \xleftarrow{R} \mathbb{F}^m$ and a random line ℓ through y; then we can apply Markov's inequality to finish the job.

Item 7.59 holds by pairwise independence. If the line ℓ is chosen randomly, then the q points on ℓ are pairwise independent samples of $\mathbb{F}^m$. The expected agreement between $g|_\ell$ and $p|_\ell$ is simply the

agreement between g and p, which is greater than ε by hypothesis. So by the Pairwise-Independent Tail Inequality (Prop. 3.28),

$$\Pr[\mathrm{agr}(g|_\ell, p|_\ell) \le \varepsilon/2] < \frac{1}{q \cdot (\varepsilon/2)^2},$$

which can be made smaller than 0.01 for a large enough choice of the constant c in $\varepsilon = c\sqrt{d/q}$.

To prove Item 7.59, we imagine first choosing the line ℓ uniformly at random from all lines in $\mathbb{F}^m$, and then choosing y uniformly at random from the points on ℓ (reparameterizing ℓ so that $\ell(1) = y$). Once we choose ℓ, we can let $q_1, \ldots, q_t$ be all polynomials of degree at most d, other than $p|_\ell$, that have agreement greater than $\varepsilon/2$ with $g|_\ell$. (Note that this list is independent of the parametrization of ℓ, i.e., if $\ell'(x) = \ell(ax + b)$ for $a \neq 0$ then $p|_{\ell'}$ and $q_i'(x) = q_i(ax + b)$ have agreement equal to $\mathrm{agr}(p|_\ell, q_i)$.) By the list-decodability of Reed–Solomon Codes (Proposition 5.15), we have $t = O(\sqrt{q/d})$.

Now, since two distinct polynomials can agree in at most d points, when we choose a random point $y \overset{R}{\leftarrow} \ell$, the probability that q_i and p agree at y is at most d/q. After reparameterization of ℓ so that $\ell(1) = y$, this gives

$$\Pr_y[\exists i \ : \ q_i(1) = p(1)] \le t \cdot \frac{d}{q} = O\left(\sqrt{\frac{d}{q}}\right).$$

This can also be made smaller than 0.01 for large enough choice of the constant c (since we may assume $q/d > c^2$, else $\varepsilon \ge 1$ and the result holds vacuously). □

7.6.4 Putting it Together

To obtain a locally list-*decodable* (rather than list-*correctible*) code, we again use the low-degree extension (Lemma 7.48) to obtain a systematic encoding. As before, to encode messages of length $\ell = \log L$, we apply Lemma 7.48 with $|\mathbb{H}| = \lceil \sqrt{q} \rceil$ and $m = \lceil \ell / \log |\mathbb{H}| \rceil$, for total degree $d \le \sqrt{q} \cdot \ell$. To decode from a $1 - \varepsilon$ fraction of errors using Theorem 7.56, we need $c\sqrt{d/q} \le \varepsilon$, which follows if $q \ge c^2\ell^2/\varepsilon^4$. This

yields the following locally list-decodable codes:

Theorem 7.60. For every $L \in \mathbb{N}$ and $\varepsilon > 0$, there is an explicit code $\mathrm{Enc} : \{0,1\}^L \to \Sigma^{\hat{L}}$, with blocklength $\hat{L} = \mathrm{poly}(L, 1/\varepsilon)$ and alphabet size $|\Sigma| = \mathrm{poly}(\log L, 1/\varepsilon)$, that has a local $(1-\varepsilon)$-list-decoder running in time $\mathrm{poly}(\log L, 1/\varepsilon)$.

Concatenating the code with a Hadamard code, similarly to Problem 5.2, we obtain:

Theorem 7.61. For every $L \in \mathbb{N}$ and $\varepsilon > 0$, there is an explicit code $\mathrm{Enc} : \{0,1\}^L \to \{0,1\}^{\hat{L}}$ with blocklength $\hat{L} = \mathrm{poly}(L, 1/\varepsilon)$ that has a local $(1/2-\varepsilon)$-list-decoder running in time $\mathrm{poly}(\log L, 1/\varepsilon)$.

Using Proposition 7.55, we get the following hardness amplification result:

Theorem 7.62. For $s : \mathbb{N} \to \mathbb{N}$, suppose that there is a function $f\colon \{0,1\}^\ell \to \{0,1\}$ in $\mathbf{E}$ that is worst-case hard against nonuniform time $s(\ell)$, where $s(\ell)$ is computable in time $2^{O(\ell)}$, then there exists $\hat{f}\colon \{0,1\}^{O(\ell)} \to \{0,1\}$ in $\mathbf{E}$ that is $(1/2 - 1/s'(\ell))$ average-case hard against (non-uniform) time $s'(\ell)$ for $s'(\ell) = t(\ell)^{\Omega(1)}/\mathrm{poly}(\ell)$.

Combining this with Theorem 7.19 and Corollary 7.20, we get:

Theorem 7.63. For $s : \mathbb{N} \to \mathbb{N}$, suppose that there is a function $f \in \mathbf{E}$ such that for every input length $\ell \in \mathbb{N}$, f is worst-case hard for nonuniform time $s(\ell)$, where $s(\ell)$ is computable in time $2^{O(\ell)}$. Then for every $m \in \mathbb{N}$, there is a mildly explicit $(m, 1/m)$ pseudorandom generator $G : \{0,1\}^{d(m)} \to \{0,1\}^m$ with seed length $d(m) = O(s^{-1}(\mathrm{poly}(m))^2/\log m)$.

Corollary 7.64. For $s : \mathbb{N} \to \mathbb{N}$, suppose that there is a function $f \in \mathbf{E} = \mathbf{DTIME}(2^{O(\ell)})$ such that for every input length $\ell \in \mathbb{N}$, f is

worst-case hard for nonuniform time $s(\ell)$. Then:

(1) If $s(\ell) = 2^{\Omega(\ell)}$, then $\mathbf{BPP} = \mathbf{P}$.
(2) If $s(\ell) = 2^{\ell^{\Omega(1)}}$, then $\mathbf{BPP} \subset \tilde{\mathbf{P}}$.
(3) If $s(\ell) = \ell^{\omega(1)}$, then $\mathbf{BPP} \subset \mathbf{SUBEXP}$.

We note that the hypotheses in these results are simply asserting that there are problems in **E** of high circuit complexity, which is quite plausible. Indeed, many common **NP**-complete problems, such as SAT, are in **E** and are commonly believed to have circuit complexity $2^{\Omega(\ell)}$ on inputs of length ℓ (though we seem very far from proving it). Thus, we have a "win–win" situation, either we can derandomize all of **BPP** *or* SAT has significantly faster (nonuniform) algorithms than currently known.

Problem 7.1 establishes a converse to Theorem 7.63, showing that pseudorandom generators imply circuit lower bounds. The equivalence is fairly tight, except for the fact that Theorem 7.63 has seed length $d(m) = O(s^{-1}(\mathrm{poly}(m))^2/\log m)$ instead of $d(m) = O(s^{-1}(\mathrm{poly}(m)))$. It is known how to close this gap via a different construction, which is more algebraic and constructs PRGs directly from worst-case hard functions (see the Chapter Notes and References); a (positive) solution to Open Problem 7.25 would give a more modular and versatile construction. For Corollary 7.64, however, there is only a partial converse known. See Section 8.2.2.

Technical Comment. Consider Item 3 of Corollary 7.64, which assumes that there is a problem in **E** of superpolynomial circuit complexity. This sounds similar to assuming that $\mathbf{E} \not\subset \mathbf{P/poly}$ (which is equivalent to $\mathbf{EXP} \not\subset \mathbf{P/poly}$, by Problem 7.2). However, the latter assumption is a bit weaker, because it only guarantees that there is a function $f \in \mathbf{E}$ and a function $s(\ell) = \ell^{\omega(1)}$ such that f has complexity at least $s(\ell)$ for *infinitely many* input lengths ℓ. Theorem 7.63 and Corollary 7.64 assume that f has complexity at least $s(\ell)$ for all ℓ; equivalently f is not in **i.o.-P/poly**, the class of functions that are computable by poly-sized circuits for infinitely many input lengths. We need the stronger assumptions because we want to build generators $G : \{0,1\}^{d(m)} \to \{0,1\}^m$ that are pseudorandom for all output lengths

m, in order to get derandomizations of **BPP** algorithms that are correct on all input lengths. However, there are alternate forms of these results, where the "infinitely often" is moved from the hypothesis to the conclusion. For example, if $\mathbf{E} \not\subset \mathbf{P/poly}$, we can conclude that $\mathbf{BPP} \subset \mathbf{i.o.\text{-}SUBEXP}$, where **i.o.-SUBEXP** denotes the class of languages having deterministic subexponential-time algorithms that are correct for infinitely many input lengths. Even though these "infinitely often" issues need to be treated with care for the sake of precision, it would be quite unexpected if the complexity of problems in **E** and **BPP** oscillated as a function of input length in such a strange way that they made a real difference.

7.7 Connections to Other Pseudorandom Objects

7.7.1 Black-Box Constructions

Similarly to our discussion after Theorem 7.19, the pseudorandom generator construction in the previous section is very general. The construction shows how to take *any* function $f : \{0,1\}^\ell \to \{0,1\}$ and use it as a subroutine (oracle) to compute a generator $G^f : \{0,1\}^d \to \{0,1\}^m$ whose pseudorandomness can be related to the hardness of f. The only place that we use the fact that $f \in \mathbf{E}$ is to deduce that G^f is computable in **E**. The reduction proving that G^f is pseudorandom is also very general. We showed how to take any T that distinguishes the output of $G^f(U_d)$ from U_m and use it as a subroutine (oracle) to build an efficient nonuniform algorithm Red such that Red^T computes f. The only place that we use the fact that T is itself an efficient nonuniform algorithm is to deduce that Red^T is an efficient nonuniform algorithm, contradicting the worst-case hardness of f.

Such constructions are called "black box," because they treat the hard function f and the distinguisher T as black boxes (i.e., oracles), without using the code of the programs that compute f and T. As we will see, black-box constructions have significant additional implications. Thus, we formalize the notion of a black-box construction as follows:

Definition 7.65. Let $G^f : [D] \to [M]$ be a deterministic algorithm that is defined for every oracle $f : [L] \to \{0,1\}$, let t, k be positive

integers such that $k \leq t$, and let $\varepsilon > 0$. We say that G is a (t, k, ε) *black-box PRG construction* if there is a randomized oracle algorithm Red, running in time t, such that for every $f : [L] \to \{0,1\}$ and $T : [M] \to \{0,1\}$ such that if

$$\Pr[T(G^f(U_{[D]})) = 1] - \Pr[T(U_{[M]}) = 1] > \varepsilon,$$

then there is an advice string $z \in [K]$ such that

$$\forall x \in [L] \quad \Pr[\mathrm{Red}^T(x, z) = f(x)] \geq 2/3,$$

where the probability is taken over the coin tosses of Red.

Note that we have separated the running time t of Red and the length k of its nonuniform advice into two separate parameters, and assume $k \leq t$ since an algorithm cannot read more than k bits in time t. When we think of Red as a nonuniform algorithm (like a boolean circuit), then we may as well think of these two parameters as being equal. (Recall that, up to polylog(s) factors, being computable by a circuit of size s is equivalent to being computable by a uniform algorithm running in time s with s bits of nonuniform advice.) However, separating the two parameters is useful in order to isolate the role of nonuniformity, and to establish connections with the other pseudorandom objects we are studying.[7]

We note that if we apply a black-box pseudorandom generator construction with a function f that is actually hard to compute, then the result is indeed a pseudorandom generator:

Proposition 7.66. If G is a (t, k, ε) black-box PRG construction and f has nonuniform worst-case hardness at least s, then G^f is an $(s/\tilde{O}(t), \varepsilon)$ pseudorandom generator.

[7] Sometimes it is useful to allow the advice string z to also depend on the coin tosses of the reduction Red. By error reduction via $r = O(\ell)$ repetitions, such a reduction can be converted into one satisfying Definition 7.65 by sampling $r = O(\ell)$ sequences of coin tosses, but this blows up the advice length by a factor of r, which may be too expensive.

Now, we can rephrase the pseudorandom generator construction of Theorem 7.63 as follows:

Theorem 7.67. For every constant $\gamma > 0$, and every $\ell, m \in \mathbb{N}$, and every $\varepsilon > 0$, there is a (t, k, ε) black-box PRG construction $G^f : \{0,1\}^d \to \{0,1\}^m$ that is defined for every oracle $f : \{0,1\}^\ell \to \{0,1\}$, with the following properties:

(1) (Mild) explicitness: G^f is computable in uniform time $\mathrm{poly}(m, 2^\ell)$ given an oracle for f.
(2) Seed length: $d = O((\ell + \log(1/\varepsilon))^2 / \log m)$.
(3) Reduction running time: $t = \mathrm{poly}(m, 1/\varepsilon)$.
(4) Reduction advice length: $k = m^{1+\gamma} + O(\ell + \log(m/\varepsilon))$.

In addition to asserting the black-box nature of Theorem 7.63, the above is more general in that it allows ε to vary independently of m (rather than setting $\varepsilon = 1/m$), and gives a tighter bound on the length of the nonuniform advice than just $t = \mathrm{poly}(m, 1/\varepsilon)$.

Proof Sketch: Given a function f, G^f encodes f in the locally list-decodable code of Theorem 7.61 (with decoding distance $1/2 - \varepsilon'$ for $\varepsilon' = \varepsilon/m$) to obtain $\hat{f} : \{0,1\}^{\hat{\ell}} \to \{0,1\}$ with $\hat{\ell} = O(\ell + \log(1/\varepsilon))$, and then computes the Nisan–Wigderson generator based on $\hat{f}$ (Construction 7.23) and a $(\hat{\ell}, \gamma \log m)$ design. The seed length and mild explicitness follow from the explicitness and parameters of the design and code (Lemma 7.22 and Theorem 7.61). The running time and advice length of the reduction follow from inspecting the proofs of Theorems 7.61 and 7.19. Specifically, the running time of of the Nisan–Wigderson reduction in the proof of Theorem 7.19 is $\mathrm{poly}(m)$ (given the nonuniform advice) by inspection, and the running time of the local list-decoding algorithm is $\mathrm{poly}(\ell, 1/\varepsilon') \leq \mathrm{poly}(m, 1/\varepsilon)$. (We may assume that $m > \ell$, otherwise G^f need not have any stretch, and the conclusion is trivial.) The length of the advice from the locally list-decodable code consists of a pair $(y, z) \in \mathbb{F}^v \times \mathbb{F}$, where $\mathbb{F}$ is a field of size $q = \mathrm{poly}(\ell, 1/\varepsilon)$ and $v \log |\mathbb{F}| = \hat{\ell} = O(\ell + \log(1/\varepsilon))$. The

Nisan–Wigderson reduction begins with the distinguisher-to-predictor reduction of Proposition 7.16, which uses $\log m$ bits of advice to specify the index i at which the predictor works and $m - i - 1$ bits for hardwiring the bits fed to the distinguisher in positions $i, \ldots, m$. In addition, for $j = 1, \ldots, i-1$, the Nisan–Wigderson reduction nonuniformly hardwires a truth-table for the function $f_j(y)$ which depends on the at most $\gamma \cdot \log m$ bits of y selected by the intersection of the ith and jth sets in the design. These truth tables require at most $(i-1) \cdot m^\gamma$ bits of advice. In total, the amount of advice used is at most

$$\begin{aligned} &O(\ell + \log(1/\varepsilon)) + m - i - 1 + (i-1) \cdot m^\gamma \\ &\quad = m^{1+\gamma} + O(\ell + \log(1/\varepsilon)). \end{aligned}$$

□

One advantage of a black-box construction is that it allows us to automatically "scale up" the pseudorandom generator construction. If we apply the construction to a function f that is not necessarily computable in **E**, but in some higher complexity class, we get a pseudorandom generator G^f computable in an analogously higher complexity class. Similarly, if we want our pseudorandom generator to fool tests T computable by nonuniform algorithms in some higher complexity class, it suffices to use a function f that is hard against an analogously higher class.

For example, we get the following "nondeterministic" analogue of Theorem 7.63:

Theorem 7.68. For $s : \mathbb{N} \to \mathbb{N}$, suppose that there is a function $f \in \mathbf{NE} \cap \mathbf{co\text{-}NE}$ such that for every input length $\ell \in \mathbb{N}$, f is worst-case hard for nonuniform algorithms running in time $s(\ell)$ with an **NP** oracle (equivalently, boolean circuits with SAT gates), where $s(\ell)$ is computable in time $2^{O(\ell)}$. Then for every $m \in \mathbb{N}$, there is a pseudorandom generator $G : \{0,1\}^{d(m)} \to \{0,1\}^m$ with seed length $d(m) = O(s^{-1}(\mathrm{poly}(m))^2 / \log m)$ such that G is $(m, 1/m)$-pseudorandom against nonuniform algorithms with an **NP** oracle, and G is computable in nondeterministic time $2^{O(d(m))}$ (meaning that there is a nondeterministic algorithm that on input x, outputs $G(x)$ on at least one computation path and outputs either $G(x)$ or "fail" on all computation paths).

The significance of such generators is that they can be used for derandomizing **AM**, which is a randomized analogue of **NP**, defined as follows:

Definition 7.69. A language L is in **AM** iff there is a probabilistic polynomial-time verifier V and polynomials $m(n), p(n)$ such that for all inputs x of length n,

$$x \in L \Rightarrow \Pr_{r \stackrel{R}{\leftarrow} \{0,1\}^{m(n)}} [\exists y \in \{0,1\}^{p(n)} \; V(x,r,y) = 1] \geq 2/3,$$

$$x \notin L \Rightarrow \Pr_{r \stackrel{R}{\leftarrow} \{0,1\}^{m(n)}} [\exists y \in \{0,1\}^{p(n)} \; V(x,r,y) = 1] \leq 1/3.$$

Another (non-obviously!) equivalent definition of **AM** is the class of languages having constant-round interactive proof systems, where a computationally unbounded prover ("Merlin") can convince a probabilistic polynomial-time verifier ("Arthur") that an input x is in L through an interactive protocol with of $O(1)$ rounds of polynomial-length communication.

Graph Nonisomorphism is the most famous example of a language that is in **AM** but is not known to be in **NP**. Nevertheless, using Theorem 7.68 we can give evidence that Graph Nonisomorphism is in **NP**.

Corollary 7.70. If there is a function $f \in \mathbf{NE} \cap \mathbf{co\text{-}NE}$ that, on inputs of length ℓ, is worst-case hard for nonuniform algorithms running in time $2^{\Omega(\ell)}$ with an **NP** oracle, then $\mathbf{AM} = \mathbf{NP}$.

While the above complexity assumption may seem very strong, it is actually known to be weaker than the very natural assumption that exponential time $\mathbf{E} = \mathbf{DTIME}(2^{O(\ell)})$ is not contained in subexponential space $\cap_{\varepsilon>0}\mathbf{DSPACE}(2^{\varepsilon n})$.

As we saw in Section 7.4.3 on derandomizing constant-depth circuits, black-box constructions can also be "scaled down" to apply to lower complexity classes, provided that the construction G and/or reduction Red can be shown to be computable in a lower class (e.g., $\mathbf{AC^0}$).

7.7.2 Connections to Other Pseudorandom Objects

At first, it may seem that pseudorandom generators are of a different character than the other pseudorandom objects we have been studying. We require complexity assumptions to construct pseudorandom generators, and reason about them using the language of computational complexity (referring to efficient algorithms, reductions, etc.). The other objects we have been studying are all information-theoretic in nature, and our constructions of them have been unconditional.

The notion of black-box constructions will enable us to bridge this gap. Note that Theorem 7.67 is unconditional, and we will see that it, like all black-box constructions, has an information-theoretic interpretation. Indeed, we can fit black-box pseudorandom generator constructions into the list-decoding framework of Section 5.3 as follows:

Construction 7.71. Let $G^f : [D] \to [M]$ be an algorithm that is defined for every oracle $f : [n] \to \{0,1\}$. Then, setting $N = 2^n$, define $\Gamma : [N] \times [D] \to [M]$, by

$$\Gamma(f, y) = G^f(y),$$

where we view the truth table of f as an element of $[N] \equiv \{0,1\}^n$.

It turns out that if we allow the reduction unbounded running time (but still bound the advice length), then pseudorandom generator constructions have an exact characterization in our framework:

Proposition 7.72. Let G^f and Γ be as in Construction 7.71. Then G^f is an (∞, k, ε) black-box PRG construction iff for every $T \subset [M]$, we have

$$|\mathrm{LIST}_\Gamma(T, \mu(T) + \varepsilon)| \leq K,$$

where $K = 2^k$.

Proof.

$\Rightarrow$. Suppose G^f is an (∞, k, ε) black-box PRG construction. Then f is in $\mathrm{LIST}_\Gamma(T, \mu(T) + \varepsilon)$ iff T distinguishes $G^f(U_{[D]})$ from $U_{[M]}$ with

advantage greater than ε. This implies that there exists a $z \in [K]$ such that $\mathrm{Red}^T(\cdot, z)$ computes f everywhere. Thus, the number of functions f in $\mathrm{LIST}_\Gamma(T, \mu(T) + \varepsilon)$ is bounded by the number of advice strings z, which is at most K.

$\Leftarrow$. Suppose that for every $T \subset [M]$, we have $L = |\mathrm{LIST}_\Gamma(T, \mu(T) + \varepsilon)| \leq K$. Then we can define $\mathrm{Red}^T(x, z) = f_z(x)$, where $f_1, \ldots, f_L$ are any fixed enumeration of the elements of $\mathrm{LIST}_\Gamma(T, \mu(T) + \varepsilon)$. □

Notice that this characterization of black-box PRG constructions (with reductions of unbounded running time) is the same as the one for averaging samplers (Proposition 5.30) and randomness extractors (Proposition 6.23). In particular, the black-box PRG construction of Theorem 7.67 is already a sampler and extractor of very good parameters:

Theorem 7.73. For every constant $\gamma > 0$, every $n \in \mathbb{N}$, $k \in [0, n]$, and every $\varepsilon > 0$, there is an explicit (k, ε) extractor $\mathrm{Ext} : \{0,1\}^n \times \{0,1\}^d \to \{0,1\}^m$ with seed length $d = O(\log^2(n/\varepsilon)/\log k)$ and output length $m \geq k^{1-\gamma}$.

Proof Sketch: Without loss of generality, assume that n is a power of 2, namely $n = 2^\ell = L$. Let $G^f(y) : \{0,1\}^d \to \{0,1\}^m$ be the (t, k_0, ε_0) black-box PRG construction of Theorem 7.67 which takes a function $f : \{0,1\}^\ell \to \{0,1\}$ and has $k_0 = m^{1+\gamma} + O(\ell + \log(m/\varepsilon_0))$, and let $\mathrm{Ext}(f, y) = \Gamma(f, y) = G^f(y)$. By Propositions 7.72 and 6.23, Ext is a $(k_0 + \log(1/\varepsilon_0), 2\varepsilon_0)$ extractor. Setting $\varepsilon = 2\varepsilon_0$ and $k = k_0 + \log(1/\varepsilon_0)$, we have a (k, ε) extractor with output length

$$m = (k - O(\ell + \log(m/\varepsilon)))^{1-\gamma} \geq k^{1-\gamma} - O(\ell + \log(m/\varepsilon)).$$

We can increase the output length to $k^{1-\gamma}$ by increasing the seed length by $O(\ell + \log(m/\varepsilon))$. The total seed length then is

$$d = O\left(\frac{(\ell + \log(1/\varepsilon))^2}{\log m} + \ell + \log(m/\varepsilon)\right) = O\left(\frac{\log^2(n/\varepsilon)}{\log k}\right). \qquad \square$$

The parameters of Theorem 7.73 are not quite as good as those of Theorem 6.36 and Corollary 6.39, as the output length is $k^{1-\gamma}$ rather

than $(1-\gamma)k$, and the seed length is only $O(\log n)$ when $k = n^{\Omega(1)}$. However, these settings of parameters are already sufficient for many purposes, such as the simulation of **BPP** with weak random sources. Moreover, the extractor construction is much more direct than that of Theorem 6.36. Specifically, it is

$$\mathrm{Ext}(f,y) = (\hat{f}(y|_{S_1}), \ldots, \hat{f}(y|_{S_m})),$$

where $\hat{f}$ is an encoding of f in a locally list-decodable code and $S_1, \ldots, S_m$ are a design. In fact, since Proposition 7.72 does not depend on the running time of the list-decoding algorithm, but only the amount of nonuniformity, we can use any $(1/2 - \varepsilon/2m, \mathrm{poly}(m/\varepsilon))$ list-decodable code, which will only require an advice of length $O(\log(m/\varepsilon))$ to index into the list of decodings. In particular, we can use a Reed–Solomon code concatenated with a Hadamard code, as in Problem 5.2.

We now provide some additional intuition for why black-box pseudorandom generator constructions are also extractors. A black-box PRG construction G^f is designed to use a *computationally hard* function f (plus a random seed) to produce an output that is *computationally indistinguishable* from uniform. When we view it as an extractor $\mathrm{Ext}(f,y) = G^f(y)$, we instead are feeding it a function f that is chosen randomly from a high min-entropy distribution (plus a random seed). This can be viewed as saying that f is *information-theoretically hard*, and from this stronger hypothesis, we are able to obtain the stronger conclusion that the output is *statistically indistinguishable* from uniform. The information-theoretic hardness of f can be formalized as follows: if f is sampled from a source F of min-entropy at least $k + \log(1/\varepsilon)$, then for every fixed function A (such as $A = \mathrm{Red}^T$), the probability (over $f \leftarrow F$) that there exists a string z of length k such that $A(\cdot, z)$ computes f everywhere is at most ε. That is, a function generated with min-entropy larger than k is unlikely to have a description of length k (relative to any fixed "interpreter" A).

Similarly to black-box PRG constructions, we can also discuss converting worst-case hard functions to average-case hard functions in

a black-box manner:

Definition 7.74. Let $\mathrm{Amp}^f : [D] \to [q]$ be a deterministic algorithm that is defined for every oracle $f : [n] \to \{0,1\}$. We say that Amp is a (t, k, ε) *black-box worst-case-to-average-case hardness amplifier* if there is a probabilistic oracle algorithm Red, called the *reduction*, running in time t such that for every function $g : [D] \to [q]$ such that

$$\Pr[g(U_{[D]}) = \mathrm{Amp}^f(U_{[D]})] > 1/q + \varepsilon,$$

there is an advice string $z \in [K]$, where $K = 2^k$, such that

$$\forall x \in [n] \qquad \Pr[\mathrm{Red}^g(x, z) = f(x)] \geq 2/3,$$

where the probability is taken over the coin tosses of Red.

Note that this definition is almost identical to that of a locally $(1 - 1/q - \varepsilon)$-list-decodable code (Definition 7.54), viewing Amp^f as $\mathrm{Enc}(f)$, and Red^g as Dec_2^g. The only difference is that in the definition of locally list-decodable code, we require that there is a first-phase decoder Dec_1^g that efficiently produces a list of candidate advice strings (a property that is natural from a coding perspective, but is not needed when amplifying hardness against nonuniform algorithms). If we remove the constraint on the running time of Red, we simply obtain the notion of a $(1 - 1/q - \varepsilon, K)$ list-decodable code. By analogy, we can view black-box PRG constructions (with reductions of bounded running time) as simply being extractors (or averaging samplers) with a kind of efficient local list-decoding algorithm (given by Red, again with an advice string that need not be easy to generate).

In addition to their positive uses illustrated above, black-box reductions and their information-theoretic interpretations are also useful for understanding the limitations of certain proof techniques. For example, we see that a black-box PRG construction $G^f : \{0,1\}^d \to \{0,1\}^m$ must have a reduction that uses $k \geq m - d - \log(1/\varepsilon) - 1$ bits of advice. Otherwise, by Propositions 7.72 and 6.23, we would obtain a $(k, 2\varepsilon)$ extractor that outputs m almost-uniform bits when given a source of min-entropy less than $k - d - 1$, which is impossible if $\varepsilon < 1/4$.

Indeed, notions of black-box reduction have been used in other settings as well, most notably to produce a very fine understanding of the relations between different cryptographic primitives, meaning which ones can be constructed from each other via black-box constructions.

7.8 Exercises

Problem 7.1 (PRGs imply hard functions). Suppose that for every m, there exists a mildly explicit $(m, 1/m)$ pseudorandom generator $G_m : \{0,1\}^{d(m)} \to \{0,1\}^m$. Show that **E** has a function $f : \{0,1\}^\ell \to \{0,1\}$ with nonuniform worst-case hardness $t(\ell) = \Omega(d^{-1}(\ell - 1))$. In particular, if $d(m) = O(\log m)$, then $t(\ell) = 2^{\Omega(\ell)}$ (Hint: look at a prefix of Gs output.)

Problem 7.2 (Equivalence of lower bounds for EXP and E). Show that **E** contains a function $f : \{0,1\}^\ell \to \{0,1\}$ of circuit complexity $\ell^{\omega(1)}$ if and only if **EXP** does. (Hint: consider $f'(x_1 \cdots x_{\ell^\epsilon}) = f(x_1 \cdots x_{\ell^\epsilon})$.)
Does the same argument work if we replace $\ell^{\omega(1)}$ with $2^{\ell^{\Omega(1)}}$? How about $2^{\Omega(\ell)}$?

Problem 7.3 (Limitations of Cryptographic Generators).

(1) Prove that a cryptographic pseudorandom generator cannot have seed length $d(m) = O(\log m)$.
(2) Prove that cryptographic pseudorandom generators (even with seed length $d(m) = m - 1$) imply $\mathbf{NP} \not\subseteq \mathbf{P/poly}$.
(3) Note where your proofs fail if we only require that G is an $(m^c, 1/m^c)$ pseudorandom generator for a fixed constant c.

Problem 7.4 (Deterministic Approximate Counting). Using the PRG for constant-depth circuits of Theorem 7.29, give deterministic quasipolynomial-time algorithms for the problems below.

(The running time of your algorithms should be $2^{\mathrm{poly}(\log n, \log(1/\varepsilon))}$, where n is the size of the circuit/formula given and ε is the accuracy parameter mentioned.)

(1) Given a constant-depth circuit C and $\varepsilon > 0$, approximate the fraction of inputs x such that $C(x) = 1$ to within an additive error of ε.
(2) Given a DNF formula φ and $\varepsilon > 0$, approximate the number of assignments x such that $\varphi(x) = 1$ to within a *multiplicative* fraction of $(1 + \varepsilon)$. You may restrict your attention to φ in which all clauses contain the same number of literals. (Hint: Study the randomized DNF counting algorithm of Theorem 2.34.)

Note that these are *not* decision problems, whereas classes such as **BPP** and $\mathbf{BPAC_0}$ are classes of decision problems. One of the points of this problem is to show how derandomization can be used for other types of problems.

Problem 7.5 (Strong Pseudorandom Generators). By analogy with strong extractors, call a function $G : \{0,1\}^d \to \{0,1\}^m$ a (t, ε) *strong pseudorandom generator* iff the function $G'(x) = (x, G(x))$ is a (t, ε) pseudorandom generator.

(1) Show that there do not exist strong cryptographic pseudorandom generators.
(2) Show that the Nisan–Wigderson generator (Theorem 7.24) is a strong pseudorandom generator.
(3) Suppose that for all constants $\alpha > 0$, there is a strong and fully explicit $(m, \varepsilon(m))$ pseudorandom generator $G : \{0,1\}^{m^\alpha} \to \{0,1\}^m$. Show that for every language $L \in \mathbf{BPP}$, there is a deterministic polynomial-time algorithm A such that for all n, $\Pr_{x \stackrel{R}{\leftarrow} \{0,1\}^n}[A(x) \neq \chi_L(x)] \leq 1/2^n + \varepsilon(\mathrm{poly}(n))$. That is, we get a *polynomial-time* average-case derandomization even though the seed length of G is $d(m) = m^\alpha$.

(4) Show that for every language $L \in \mathbf{BPAC^0}$, there is an $\mathbf{AC^0}$ algorithm A such that $\Pr_{x \xleftarrow{R} \{0,1\}^n}[A(x) \neq \chi_L(x)] \leq 1/n$. (Warning: be careful about error reduction.)

Problem 7.6 (Private Information Retrieval). The goal of *private information retrieval* is for a user to be able to retrieve an entry of a remote database in such a way that the server holding the database *learns nothing* about which database entry was requested. A trivial solution is for the server to send the user the entire database, in which case the user does not need to reveal anything about the entry desired. We are interested in solutions that involve much less communication. One way to achieve this is through replication.[8] Formally, in a *q-server private information-retrieval (PIR) scheme*, an arbitrary database $D \in \{0,1\}^n$ is duplicated at q noncommunicating servers. On input an index $i \in [n]$, the *user algorithm* U tosses some coins r and outputs queries $(x_1, \ldots, x_q) = U(i, r)$, and sends x_j to the jth server. The jth server algorithm S_j returns an answer $y_j = S_j(x_j, D)$. The user then computes its output $U(i, r, x_1, \ldots, x_q)$, which should equal D_i, the ith bit of the database. For privacy, we require that the distribution of each query x_j (over the choice of the random coin tosses r) is the same regardless of the index i being queried.

It turns out that q-query locally decodable codes and q-server PIR are essentially equivalent. This equivalence is proven using the notion of *smooth codes*. A code $\mathrm{Enc} : \{0,1\}^n \to \Sigma^{\hat{n}}$ is a *q-query smooth code* if there is a probabilistic oracle algorithm Dec such that for every message x and every $i \in [n]$, we have $\Pr[\mathrm{Dec}^{\mathrm{Enc}(x)}(i) = x_i] = 1$ and Dec makes q nonadaptive queries to its oracle, each of which is uniformly distributed in $[\hat{n}]$. Note that the oracle in this definition is a valid codeword, with no corruptions. Below you will show that smooth codes imply locally decodable codes and PIR schemes; converses are also known (after making some slight relaxations to the definitions).

[8] Another way is through computational security, where we only require that it be *computationally infeasible* for the database to learn something about the entry requested.

(1) Show that the decoder for a q-query smooth code is also a local $(1/3q)$-decoder for Enc.
(2) Show that every q-query smooth code $\mathrm{Enc} : \{0,1\}^n \to \Sigma^{\hat{n}}$ gives rise to a q-server PIR scheme in which the user and servers communicate at most $q \cdot (\log \hat{n} + \log |\Sigma|)$ bits for each database entry requested.
(3) Using the Reed–Muller code, show that there is a polylog(n)-server PIR scheme with communication complexity polylog(n) for n-bit databases. That is, the user and servers communicate at most polylog(n) bits for each database entry requested. (For constant q, the Reed–Muller code with an optimal systematic encoding as in Problem 5.4 yields a q-server PIR with communication complexity $O(n^{1/(q-1)})$.)

Problem 7.7 (Better Local Decoding of Reed–Muller Codes). Show that for every constant $\varepsilon > 0$, there is a constant $\gamma > 0$ such that there is a local $(1/2 - \varepsilon)$-decoding algorithm for the q-ary Reed–Muller code of degree d and dimension m, provided that $d \leq \gamma q$. (Here we are referring to unique decoding, not list decoding.) The running time of the decoder should be poly(m, q).

Problem 7.8 (Hitting-Set Generators). A set $H_m \subset \{0,1\}^m$ is a (t, ε) *hitting set* if for every nonuniform algorithm T running in time t that accepts greater than an ε fraction of m-bit strings, T accepts at least one element of H_m.

(1) Show that if, for every m, we can construct an $(m, 1/2)$ hitting set H_m in time $s(m) \geq m$, then $\mathbf{RP} \subset \bigcup_c \mathbf{DTIME}(s(n^c))$. In particular, if $s(m) = \mathrm{poly}(m)$, then $\mathbf{RP} = \mathbf{P}$.
(2) Show that if there is a (t, ε) pseudorandom generator $G_m : \{0,1\}^d \to \{0,1\}^m$ computable in time s, then there is a (t, ε) hitting set H_m constructible in time $2^d \cdot s$.

(3) Show that if, for every m, we can construct an $(m, 1/2)$ hitting set H_m in time $s(m) = \text{poly}(m)$, then $\mathbf{BPP} = \mathbf{P}$. (Hint: this can be proven in two ways. One uses Problem 3.1 and the other uses a variant of Problem 7.1 together with Corollary 7.64. How do the parameters for general $s(m)$ compare?)
(4) Define the notion of a (t, k, ε) black-box construction of hitting set-generators, and show that, when $t = \infty$, such constructions are equivalent to constructions of *dispersers* (Definition 6.19).

Problem 7.9 (PRGs versus Uniform Algorithms $\Rightarrow$ Average–Case Derandomization). For functions $t : \mathbb{N} \to \mathbb{N}$ and $\varepsilon : \mathbb{N} \to [0,1]$, we say that a sequence $\{G_m : \{0,1\}^{d(m)} \to \{0,1\}^m\}$ of is a $(t(m), \varepsilon(m))$ pseudorandom generator against uniform algorithms iff the ensembles $\{G(U_{d(m)})\}_{m\in\mathbb{N}}$ and $\{U_m\}_{m\in\mathbb{N}}$ are uniformly computationally indistinguishable (Definition 7.2).

Suppose that we have a mildly explicit $(m, 1/m)$ pseudorandom generator against uniform algorithms that has seed length $d(m)$. Show that for every language L in **BPP**, there exists a deterministic algorithm A running in time $2^{d(\text{poly}(n))} \cdot \text{poly}(n)$ on inputs of length n such that:

(1) $\Pr[A(X_n) = L(X_n)] \geq 1 - 1/n^2$, where $X_n \stackrel{R}{\leftarrow} \{0,1\}^n$ and $L(\cdot)$ is the characteristic function of L. (The exponent of 2 in n^2 is arbitrary, and can be replaced by any constant.) Hint: coming up with the algorithm A is the "easy" part; proving that it works well is a bit trickier.
(2) $\Pr[A(X_n) = L(X_n)] \geq 1 - 1/n^2$, for any random variable X_n distributed on $\{0,1\}^n$ that is samplable in time n^2.

Problem 7.10 (PRGs are Necessary for Derandomization).

(1) Call a function $G : \{0,1\}^d \to \{0,1\}^m$ a (t, ℓ, ε) *pseudorandom generator against bounded-nonuniformity algorithms* iff for

every probabilistic algorithm T that has a program of length at most ℓ and that runs in time at most t on inputs of length n, we have

$$|\Pr[T(G(U_d)) = 1] - \Pr[T(U_m) = 1]| \leq \varepsilon.$$

Consider the promise problem Π whose YES instances are truth tables of functions $G : \{0,1\}^d \to \{0,1\}^m$ that are $(m, \log m, 1/m)$ pseudorandom generators against bounded-nonuniformity algorithms, and whose NO instances are truth tables of functions that are not $(m, \log m, 2/m)$ pseudorandom generators against bounded-nonuniformity algorithms. (Here m and d are parameters determined by the input instance G.) Show that Π is in **prBPP**.

(2) Using Problem 2.11, show that if $\mathbf{prBPP} = \mathbf{prP}$, then there is a mildly explicit $(m, 1/m)$ pseudorandom generator against uniform algorithms with seed length $O(\log m)$. (See Problem 7.9 for the definition. It turns out that the hypothesis $\mathbf{prBPP} = \mathbf{prP}$ here can be weakened to obtain an equivalence between PRGs vs. uniform algorithms and average-case derandomization of **BPP**.)

Problem 7.11 (Composition). For simplicity in this problem, only consider constant t in this problem (although the results do have generalizations to growing $t = t(\ell)$).

(1) Show that if $f : \{0,1\}^\ell \to \{0,1\}^\ell$ is a one-way permutation, then for any constant t, $f^{(t)}$ is a one-way permutation, where $f^{(t)}(x) \stackrel{\text{def}}{=} \underbrace{f(f(\cdots f}_{t}(x)))$.

(2) Show that the above fails for one-way functions. That is, assuming that there exists a one-way function g, construct a one-way function f which doesn't remain one way under composition. (Hint: for $|x| = |y| = \ell/2$, set $f(x,y) = 1^{|g(y)|}g(y)$ unless $x \in \{0^\ell, 1^\ell\}$.)

(3) Show that if G is a cryptographic pseudorandom generator with seed length $d(m) = m^{\Omega(1)}$, then for any constant t, $G^{(t)}$ is a cryptographic pseudorandom generator. Note where your proof fails for fully explicit pseudorandom generators against time m^c for a fixed constant c.

Problem 7.12 (Local List Decoding the Hadamard Code). For a function $f : \mathbb{Z}_2^m \to \mathbb{Z}_2$, A *parameterized subspace* $x + V$ of $\mathbb{Z}_2^m$ of dimension d is given by a linear map $V : \mathbb{Z}_2^d \to \mathbb{Z}_2^m$ and a shift $x \in \mathbb{Z}_2^m$. (We do not require that the map V be full rank.) We write V for $0 + V$. For a function $f : \mathbb{Z}_2^m \to \mathbb{Z}_2$, we define $f|_{x+V} : \mathbb{Z}_2^d \to \mathbb{Z}_2^m$ by $f|_{x+V}(y) = f(x + V(y))$.

(1) Let $c : \mathbb{Z}_2^m \to \mathbb{Z}_2$ be a codeword in the Hadamard code (i.e., a linear function), $r : \mathbb{Z}_2^m \to \mathbb{Z}_2$ a received word, V a parameterized subspace of $\mathbb{Z}_2^m$ of dimension d, and $x \in \mathbb{Z}_2^m$. Show that if $d_H(r|_{x+V}, c|_{x+V}) < 1/2$, then $c(x)$ can be computed from x, V, $c|_V$, and oracle access to r in time $\mathrm{poly}(m, 2^d)$ with $2^d - 1$ queries to r.
(2) Show that for every $m \in \mathbb{N}$ and $\varepsilon > 0$, the Hadamard code of dimension m has a $(1/2 - \varepsilon)$ local list-decoding algorithm $(\mathrm{Dec}_1, \mathrm{Dec}_2)$ in which both Dec_1 and Dec_2 run in time $\mathrm{poly}(m, 1/\varepsilon)$, and the list output by Dec_1 has size $O(1/\varepsilon^2)$. (Hint: consider a random parameterized subspace V of dimension $2\log(1/\varepsilon) + O(1)$, and how many choices there are for $c|_V$.)
(3) Show that Dec_2 can be made to be deterministic and run in time $O(m)$.

Problem 7.13 (Hardcore Predicates). A *hardcore predicate* for a one-way function $f : \{0,1\}^\ell \to \{0,1\}^\ell$ is a $\mathrm{poly}(\ell)$-time computable function $b : \{0,1\}^\ell \to \{0,1\}$ such that for every constant c, every

nonuniform algorithm A running in time ℓ^c, we have:

$$\Pr[A(f(U_\ell)) = b(U_\ell)] \leq \frac{1}{2} + \frac{1}{\ell^c},$$

for all sufficiently large ℓ. Thus, while the one-wayness of f only guarantees that it is hard to compute *all* the bits of fs input from its output, b specifies a particular bit of information about the input that is very hard to compute (one can't do noticeably better than random guessing).

(1) Let $\mathrm{Enc} : \{0,1\}^\ell \to \{0,1\}^{\hat{L}}$ be a code such that given $x \in \{0,1\}^\ell$ and $y \in [\hat{L}]$, $\mathrm{Enc}(x)_y$ can be computed in time $\mathrm{poly}(\ell)$. Suppose that for every constant c and all sufficiently large ℓ, Enc has a $(1/2 - 1/\ell^c)$ local list-decoding algorithm $(\mathrm{Dec}_1, \mathrm{Dec}_2)$ in which both Dec_1 and Dec_2 run in time $\mathrm{poly}(\ell)$. Prove that if $f : \{0,1\}^\ell \to \{0,1\}^\ell$ is a one-way function, then $b(x,y) = \mathrm{Enc}(x)_y$ is a hardcore predicate for the one-way function $f'(x,y) = (f(x), y)$.

(2) Show that if $b : \{0,1\}^\ell \to \{0,1\}^\ell$ is a hardcore predicate for a one-way *permutation* $f : \{0,1\}^\ell \to \{0,1\}^\ell$, then for every $m = \mathrm{poly}(\ell)$, the following function $G : \{0,1\}^\ell \to \{0,1\}^m$ is a cryptographic pseudorandom generator:

$$G(x) = (b(x), b(f(x)), b(f(f(x))), \ldots, b(f^{(m-1)}(x))).$$

(Hint: show that G is "previous-bit unpredictable.")

(3) Using Problem 7.12, deduce that if $f : \{0,1\}^\ell \to \{0,1\}^\ell$ is a one-way permutation, then for every $m = \mathrm{poly}(\ell)$, the following is a cryptographic pseudorandom generator:

$$G_m(x,r) = (\langle x,r\rangle, \langle f(x),r\rangle, \langle f(f(x)),r\rangle, \ldots, \langle f^{(m-1)}(x),r\rangle).$$

Problem 7.14 (PRGs from 1–1 One-Way Functions). A random variable X has (t,ε) *pseudoentropy* at least k if it is (t,ε) indistinguishable from some random variable of min-entropy at least k.

(1) Suppose that X has (t,ε) pseudoentropy at least k and that $\mathrm{Ext}:\{0,1\}^n \times \{0,1\}^d \to \{0,1\}^m$ is a (k,ε')-extractor computable in time t'. Show that $\mathrm{Ext}(X,U_d)$ is an $(t-t',\varepsilon+\varepsilon')$ indistinguishable from U_m.

(2) Let $f:\{0,1\}^\ell \to \{0,1\}^{\ell'}$ be a one-to-one one-way function (not necessarily length-preserving) and $b:\{0,1\}^\ell \to \{0,1\}$ a hardcore predicate for f (see Problem 7.13). Show that for every constant c and all sufficiently large ℓ, the random variable $f(U_\ell)b(U_\ell)$ has $(\ell^c, 1/\ell^c)$ pseudoentropy at least $\ell+1$.

(3) (*) Show how to construct a cryptographic pseudorandom generator from any one-to-one one-way function. (Any seed length $\ell(m) < m$ is fine.)

7.9 Chapter Notes and References

Other surveys on pseudorandom generators and derandomization include [162, 209, 226, 288].

Descriptions of classical constructions of pseudorandom generators (e.g., linear congruential generators) and the batteries of statistical tests that are used to evaluate them can be found in [245, 341]. Linear congruential generators and variants were shown to be cryptographically insecure (e.g., not satisfy Definition 7.9) in [56, 80, 144, 252, 374]. Current standards for pseudorandom generation in practice can be found in [51].

The modern approach to pseudorandomness described in this section grew out of the complexity-theoretic approach to cryptography initiated by Diffie and Hellman [117] (who introduced the concept of one-way functions, among other things). Shamir [360] constructed a generator achieving a weak form of unpredictability based on the conjectured one-wayness of the RSA function [336]. (Shamir's generator outputs a sequence of long strings, such that none of the string can be predicted from the others, except with negligible probability, but individual bits may be easily predictable.) Blum and Micali [72] proposed the criterion of next-bit unpredictability (Definition 7.15) and constructed a generator satisfying it based on the conjectured

hardness of the Discrete Logarithm Problem. Yao [421] gave the now-standard definition of pseudorandomness (Definition 7.3) based on the notion of computational indistinguishability introduced in the earlier work of Goldwasser and Micali [176] (which also introduced hybrid arguments). Yao also proved the equivalence of pseudorandomness and next-bit unpredictability (Proposition 7.16), and showed how to construct a cryptographic pseudorandom generator from any one-way permutation. The construction described in Section 7.2 and Problems 7.12 and 7.13 uses the hardcore predicate from the later work of Goldreich and Levin [168]. The construction of a pseudorandom generator from an arbitrary one-way function (Theorem 7.11) is due to Håstad, Impagliazzo, Levin, and Luby [197]. The most efficient (and simplest) construction of pseudorandom generators from general one-way functions to date is in [198, 401]. Goldreich, Goldwasser, and Micali [164] defined and constructed pseudorandom functions, and illustrated their applicability in cryptography. The application of pseudorandom functions to learning theory is from [405], and their application to circuit lower bounds is from [323]. For more about cryptographic pseudorandom generators, pseudorandom functions, and their applications in cryptography, see the text by Goldreich [157].

Yao [421] demonstrated the applicability of pseudorandom generators to derandomization, noting in particular that cryptographic pseudorandom generators imply that $\mathbf{BPP} \subset \mathbf{SUBEXP}$, and that one can obtain even $\mathbf{BPP} \subset \tilde{\mathbf{P}}$ under stronger intractability assumptions. Nisan and Wigderson [302] observed that derandomization only requires a mildly explicit pseudorandom generator, and showed how to construct such generators based on the average-case hardness of $\mathbf{E}$ (Theorem 7.24). A variant of Open Problem 7.25 was posed in [202], who showed that it also would imply stronger results on hardness amplification; some partial negative results can be found in [214, 320].

The instantiation of the Nisan–Wigderson pseudorandom generator that uses the parity function to fool constant-depth circuits (Theorem 7.29) is from the earlier work of Nisan [298]. (The average-case hardness of parity against constant-depth circuits stated in Theorem 7.27 is due Boppana and Håstad [196].) The first unconditional

pseudorandom generator against constant-depth circuits was due to Ajtai and Wigderson [12] and had seed length $\ell(m) = m^{\varepsilon}$ (compared to polylog(m) in Nisan's generator). Recently, Braverman [81] proved that any polylog(m)-wise independent distribution fools $\mathbf{AC^0}$, providing a different way to obtain polylogarithmic seed length and resolving a conjecture of Linial and Nisan [265]. The notion of strong pseudorandom generators (a.k.a. seed-extending pseudorandom generators) and the average-case derandomization of $\mathbf{AC^0}$ (Problem 7.5) are from [242, 355]. Superpolynomial circuit lower bounds for $\mathbf{AC^0}[\mathbf{2}]$ were given by [322, 368]. Viola [412] constructed pseudorandom generators with superpolynomial stretch for $\mathbf{AC^0}[\mathbf{2}]$ circuits that are restricted to have a logarithmic number of parity gates.

Detailed surveys on locally decodable codes and their applications in theoretical computer science are given by Trevisan [391] and Yekhanin [424]. The notion grew out of several different lines of work, and it took a couple of years before a precise definition of locally decodable codes was formulated. The work of Goldreich and Levin [168] on hardcore predicates of one-way permutations implicitly provided a local list-decoding algorithm for the Hadamard code. (See Problems 7.12 and 7.13.) Working on the problem of "instance hiding" introduced in [2], Beaver and Feigenbaum [54] constructed a protocol based on Shamir's "secret sharing" [359] that effectively amounts to using the local decoding algorithm for the Reed–Muller code (Algorithm 7.43) with the multilinear extension (Lemma 7.47). Blum, Luby, and Rubinfeld [71] and Lipton [266] introduced the concept of self-correctors for functions, which allow a one to convert a program that correctly computes a function on most inputs to one that correctly computes the function on all inputs.[9] Both papers gave self-correctors for group homomorphisms, which, when applied to homomorphisms from $\mathbb{Z}_2^n$ to $\mathbb{Z}_2$, can be interpreted as a local corrector for the Hadamard code

[9] Blum, Luby, and Rubinfeld [71] also defined and constructed self-testers for functions, which allow one to efficiently determine whether a program does indeed compute a function correctly on most inputs before attempting to use self-correction. Together a self-tester and self-corrector yield a "program checker" in the sense of [70]. The study of self-testers gave rise to the notion of *locally testable codes*, which are intimately related to probabilistically checkable proofs [41, 42], and to the notion of *property testing* [165, 337, 340], which is an area within sublinear-time algorithms.)

(Proposition 7.40). Lipton [266] observed that the techniques of Beaver and Feigenbaum [54] yield a self-corrector for multivariate polynomials, which, as mentioned above, can be interpreted as a local corrector for the Reed–Muller code. Lipton pointed out that it is interesting to apply these self-correctors to presumably intractable functions, such as the Permanent (known to be #**P**-complete [404]), and soon it was realized that they could also be applied to complete problems for other classes by taking the multilinear extension [42]. Babai, Fortnow, Nisan, and Wigderson [43] used these results to construct pseudorandom generators from the worst-case hardness of **EXP** (or **E**, due to Problem 7.2), and thereby obtain subexponential-time or quasipolynomial-time simulations of **BPP** under appropriate worst-case assumptions (Corollary 7.64, Parts 2 and 3). All of these works also used the terminology of random self-reducibility, which had been present in the cryptography literature for a while [29], and was known to imply worst-case/average-case connections. Understanding the relationship between the worst-case and average-case complexity of **NP** (rather than "high" classes like **EXP**) is an important area of research; see the survey [74].

Self-correctors for multivariate polynomials that can handle a constant fraction of errors (as in Theorem 7.42) and fraction of errors approaching 1/2 (as in Problem 7.7) were given by Gemmell et al. [149] and Gemmell and Sudan [150], respectively. Babai, Fortnow, Levin, and Szegedy [41] reformulated these results as providing error-correcting codes with efficient local decoding (and "local testing") algorithms. Katz and Trevisan [239] focused attention on the exact query complexity of locally decodable codes (separately from computation time), and proved that locally decodable codes cannot simultaneously have the rate, distance, and query complexity all be constants independent of the message length. Constructions of 3-query locally decodable codes with subexponential blocklength were recently given by Yekhanin [423] and Efremenko [128]. Private Information Retrieval (Problem 7.6) was introduced by Chor, Goldreich, Kushilevitz, and Sudan [99]. Katz and Trevisan [239] introduced the notion of smooth codes and showed their close relation to both private information retrieval and locally decodable codes (Problem 7.6). Recently, Saraf, Kopparty,

and Yekhanin [249] constructed the first locally decodable codes with sublinear-time decoding and rate larger 1/2.

Techniques for Hardness Amplification (namely, the Direct Product Theorem and XOR Lemma) were first described in oral presentations of Yao's paper [421]. Since then, these results have been strengthened and generalized in a number of ways. See the survey [171] and Section 8.2.3. The first local list-decoder for Reed–Muller codes was given by Arora and Sudan [35] (stated in the language of program self-correctors). The one in Theorem 7.56 is due to Sudan, Trevisan, and Vadhan [381], who also gave a general definition of locally list-decodable codes (inspired by a list-decoding analogue of program self-correctors defined by Ar et al. [30]) and explicitly proved Theorems 7.60, 7.61, and 7.62.

The result that $\mathbf{BPP} = \mathbf{P}$ if $\mathbf{E}$ has a function of nonuniform worst-case hardness $s(\ell) = 2^{\Omega(\ell)}$ (Corollary 7.64, Part 1) is from the earlier work of Impagliazzo and Wigderson [215], who used derandomized versions of the XOR Lemma to obtain sufficient average-case hardness for use in the Nisan–Wigderson pseudorandom generator. An optimal construction of pseudorandom generators from worst-case hard functions, with seed length $d(m) = O(s^{-1}(\mathrm{poly}(m)))$ (cf., Theorem 7.63), was given by Shaltiel and Umans [356, 399].

For more background on $\mathbf{AM}$, see the Notes and References of Section 2. The first evidence that $\mathbf{AM} = \mathbf{NP}$ was given by Arvind and Köbler [37], who showed that one can use the Nisan–Wigderson generator with a function that is $(2^{\Omega(\ell)}, 1/2 - 1/2^{\Omega(\ell)})$-hard for nondeterministic circuits. Klivans and van Melkebeek [244] observed that the Impagliazzo–Wigderson pseudorandom generator construction is "black box" and used this to show that $\mathbf{AM}$ can be derandomized using functions that are worst-case hard for circuits with an $\mathbf{NP}$ oracle (Theorem 7.68). Subsequent work showed that one only needs worst-case hardness against a nonuniform analogue of $\mathbf{NP} \cap \mathbf{co\text{-}NP}$ [289, 356, 357].

Trevisan [389] showed that black-box pseudorandom generator constructions yield randomness extractors, and thereby obtained the extractor construction of Theorem 7.73. This surprising connection between complexity-theoretic pseudorandomness and information-theoretic pseudorandomness sparked much subsequent work, from

which the unified theory presented in this survey emerged. The fact that black-box hardness amplifiers are a form of locally list-decodable codes was explicitly stated (and used to deduce lower bounds on advice length) in [397]. The use of black-box constructions to classify and separate cryptographic primitives was pioneered by Impagliazzo and Rudich [213]; see also [326, 330].

Problem 7.1 (PRGs imply hard functions) is from [302]. Problem 7.2 is a special case of the technique called "translation" or "padding" in complexity theory. Problem 7.4 (Deterministic Approximate Counting) is from [302]. The fastest known deterministic algorithms for approximately counting the number of satisfying assignments to a DNF formula are from [280] and [178] (depending on whether the approximation is relative or additive, and the magnitude of the error). The fact that hitting set generators imply $\mathbf{BPP} = \mathbf{P}$ (Problem 7.8) was first proven by Andreev, Clementi, and Rolim [27]; for a more direct proof, see [173]. Problem 7.9 (that PRGs vs. uniform algorithms imply average-case derandomization) is from [216]. Goldreich [163] showed that PRGs are necessary for derandomization (Problem 7.10). The result that one-to-one one-way functions imply pseudorandom generators is due to Goldreich, Krawczyk, and Luby [167]; the proof in Problem 7.14 is from [197].

For more on Kolmogorov complexity, see [261]. In recent years, connections have been found between Kolmogorov complexity and derandomization; see [14]. The tighter equivalence between circuit size and nonuniform computation time mentioned after Definition 7.1 is due to Pippenger and Fischer [311]. The $5n - O(n)$ lower bound on circuit size is due to Iwama, Lachish, Morizumi, and Raz [218, 254]. The fact that single-sample indistinguishability against uniform algorithms does not imply multiple-sample indistinguishability unless we make additional assumptions such as efficient samplability (in contrast to Proposition 7.14), is due to Goldreich and Meyer [169]. (See also [172].)