CS225: Pseudorandomness	Prof. Salil Vadhan

Lecture 17: Pseudorandom Generators

April 10, 2007

Based on scribe notes by Alexandr Andoni, Saurabh Sanghvi, David Woodruff, Yan-Cheng Chang, and Kevin Matulef. Lecture given by Dan Gutfreund..

1 Motivation

Our accomplishments in derandomization in the class so far include the following:

- Derandomizing specific algorithms, such as the ones for Max Cut 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 error reduction of randomized algorithms and sampling; 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 **BPP** = **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 n^c to $O(\log n)$, and then eliminate the randomness entirely by enumeration.

Thus, we would like to have a function $G : \{0,1\}^{O(\log n)} \to \{0,1\}^{n^c}$ that stretches a seed of $O(\log n)$ truly random bits into n^c 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 "looks random" include the following:

- Information-theoretic or statistical measures: e.g., entropy, statistical difference from uniform distribution, 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 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, which is defined as follows: a string "looks random" if it is incompressible (cannot be generated by a Turing machine with a representation of length less than

n). However, if the function G is computable (which we certainly want!) then all of its outputs have Kolmogorov complexity $O(\log n)$ (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 way to look at it is as follows. Recall 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 (T's for which membership can be efficiently tested).

2 Computational Indistinguishability

Definition 1 (computational indistinguishability) *Random variables X and Y taking values in $\{0,1\}^n$ are (t, ε)* indistinguishable *if for every* nonuniform *algorithm running in time 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. 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 (e.g. because a Turing machine can encode a huge lookup table in its transition function). However, for nonuniform algorithms, we can avoid doing so by using Boolean circuits as our nonuniform model of computation. As mentioned earlier in the course, 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, by 'nonuniform algorithm running in time t', we mean 'Boolean circuit of size t' (where the size is measured by the number of AND and OR gates in the circuit).

That said, it is of interest to study computational indistinguishability and pseudorandomness against uniform algorithms.

Definition 2 (uniform computational indistinguishability) *Let X_n, Y_n be some sequences of random variables on $\{0,1\}^n$ (or $\{0,1\}^{\mathrm{poly}(n)}$). For functions $t : \mathbb{N} \to \mathbb{N}$ and $\varepsilon : \mathbb{N} \to [0,1]$, we say that $\{X_n\}$ and $\{Y_n\}$ are $(t(n), \varepsilon(n))$* indistinguishable for uniform algorithms *if for all probabilistic algorithms T running in time $t(n)$, we have that*
$$|\Pr[T(X_n) = 1] - \Pr[T(Y_n) = 1]| \leq \varepsilon(n)$$
for all sufficiently large n, where the probabilities are taken over X_n, Y_n and the random coin tosses of T.

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

3 Pseudorandom Generators

Definition 3 *A deterministic function* $G : \{0,1\}^\ell \to \{0,1\}^n$ *is a* (t, ε) pseudorandom generator (PRG) *if*

1. $\ell < n$, *and*
2. $G(U_\ell)$ *and* U_n *are* (t, ε) *indistinguishable.*

Also, note that we have formulated the definition with respect to nonuniform computational indistinguishability, but there is a natural uniform analogue of this definition.

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 1's and 0's are approximately the same, the longest run is of length $O(\log n)$, 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 less than t. For the derandomization to be efficient, we will also need the generator to be efficiently computable.

Definition 4 *We say a sequence of generators* $\{G_n : \{0,1\}^{\ell(n)} \to \{0,1\}^n\}$ *is* computable in time $t(n)$ *if there is a* uniform and deterministic *algorithm* M *such that for every* $n \in \mathbb{N}$ *and* $y \in \{0,1\}^{\ell(n)}$, *we have* $M(1^n, x) = G_n(x)$ *and* $M(1^n, x)$ *runs in time at most* $t(n)$.

Note that even when though 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 $G_n : \{0,1\}^{\ell(n)} \to \{0,1\}^n$, with it being implicit that we are discussing a family $\{G_n\}$.

Theorem 5 *Suppose that for all* n *there exists an* $(n, 1/8)$ *pseudorandom generator* $G_n : \{0,1\}^{\ell(n)} \to \{0,1\}^n$ *computable in time* $t(n)$. *Then* $\mathbf{BPP} \subseteq \bigcup_c \mathbf{DTIME}(2^{\ell(n^c)} \cdot (n^c + t(n^c)))$.

By nonuniform tests, we of course mean that we will allow the program T that is trying to distinguish the PRGs output to be non-uniform (in the sense that the program can differ arbitrarily for each n). This assumption of course makes the theorem weaker.

Proof: A **BPP** algorithm starts with some algorithm A taking an input x of length n and a sequence of random bits r, and then accepting or rejecting after at most n^c steps for some c. We can of course assume that the algorithm uses at most n^c random bits.

The idea will be to plug in the pseudorandom generator G_{n^c} to produce this sequence of random bits, then to use the pseudorandomness assumption to show that the algorithm will do just as well with that sequence, and then to enumerate over all possible seeds to produce a derandomization.

Claim 6 *For every x of length n, $A(x; G_{n^c}(U_{\ell(n^c)}))$ errs with probability smaller than $1/2$.*

> **Proof of claim:** Suppose not. Then $T(\cdot) = A(x, \cdot)$ is a nonuniform algorithm running in time n^c that distinguishes $G_{n^c}(U_{\ell(n^c)})$ from U_{n^c} with advantage at least $1/2 - 1/3 > 1/8$. Notice that we are using x here as nonuniform advice; this is why we need the PRG to be robust against nonuniform tests. □

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

Notice that we can afford for the generator G_n have running time $t(n) = \text{poly}(n)$ or even $t(n) = 2^{O(\ell(n))}$ without affecting the time of the derandomization polynomially. In particular, for this application, it is OK if the generator runs in more time than the tests it fools (which are time n in this theorem).

The theorem provides a mechanism to produce various different theorems, relating the existence of PRGs for certain seed lengths with the ability to derandomize. Let's look at some typical settings of parameters to see what we might imagine proving with this theorem. Assuming throughout that $t(n) = \text{poly}(n)$ or $t(n) = 2^{O(\ell(n))}$, we have:

1. Suppose that for every ϵ you can create a PRG with $\ell(n) = n^\epsilon$. Then $\mathbf{BPP} \subseteq \bigcap_{\epsilon>0} \mathbf{DTIME}(2^{n^\epsilon}) \stackrel{\text{def}}{=} \mathbf{SUBEXP}$.

   Since we know that **SUBEXP** is a proper subset of **EXP**, this would be a nontrivial improvement on the current inclusion $\mathbf{BPP} \subseteq \mathbf{EXP}$.

2. Suppose we had a PRG with $\ell(n) = \text{polylog}(n)$. Then $\mathbf{BPP} \subseteq \bigcup_c \mathbf{DTIME}(2^{\log^c n}) \stackrel{\text{def}}{=} \tilde{\mathbf{P}}$. ($\tilde{\mathbf{P}}$ is known as "quasi-polynomial" time).

3. Suppose we had a PRG with $\ell(n) = O(\log n)$. Then $\mathbf{BPP} = \mathbf{P}$.

4 Existence of PRGs

Of course, the previous section makes it incumbent on us to determine if PRGs exist. As usual in this course, the answer is yes but the proof is not very helpful—it is nonconstructive and thus does not provide for an efficient PRG.

Proposition 7 *For all $t \in \mathbb{N}$ and $\varepsilon > 0$, there exists a (t, ε) PRG $G : \{0,1\}^\ell \to \{0,1\}^t$ with seed length $O(\log t + \log(1/\varepsilon))$.*

Proof: The proof is by the probabilistic method. Choose $G : \{0,1\}^\ell \to \{0,1\}^n$ at random. Now, fix a time t algorithm, T.

The probability (over choice of G) that T distinguishes $G(U_\ell)$ from U_n with advantage ϵ is at most $2^{-\Omega(2^\ell \epsilon^2)}$, by a Chernoff bound argument.

There are $2^{\text{poly}(t)}$ nonuniform algorithms running in time t (i.e. circuits of size t). Thus, union-bounding over all possible T, and setting $\epsilon = 1/t$, we get that the probability that there exists a T breaking G is at most $2^{\text{poly}(t)} 2^{-\Omega(2^\ell \varepsilon^2)}$, which is less than 1 for ℓ being $O(\log t + \log(1/\varepsilon))$. ■

Note that putting together Proposition 7 and Theorem 5 gives us another way to prove that $\mathbf{BPP} \subseteq \mathbf{P/poly}$. Just let the advice string be the truth table of the PRG for the proper length, and then one can use that PRG and the proof of Theorem 5 to derandomize **BPP**. 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 exactly the same 'construction'.

5 Cryptographic PRGs

The theory of computational pseudorandomness developed in this lecture 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 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 adversaries with greater running time. 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 8 *A generator* $G_n : \{0,1\}^{\ell(n)} \to \{0,1\}^n$ *is a* cryptographic pseudorandom generator *if*

- *There is a constant c such that G_n is computable in time n^c.*
- *For every constant d, G_n is an $(n^d, 1/n^d)$ pseudorandom generator for all sufficiently large n.*

Due to time constraints and the fact that such generators are covered in cryptography courses here and at MIT, 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 $\mathbf{NP} \not\subseteq \mathbf{BPP}$. 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 $\mathbf{NP} \not\subseteq \mathbf{BPP}$ implies that 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 9 $f_n : \{0,1\}^n \to \{0,1\}^n$ *is a* one-way function *if:*

1. *There is a constant c such that f is computable in time n^c.*
2. *For every constant d and every nonuniform algorithm A running in time n^d:*
$$\Pr[A(f(U_n)) \in f^{-1}(f(U_n))] \leq \frac{1}{n^d}$$
for all sufficiently large n.

Assuming the existence of one-way functions seems stronger than the assumption $\mathbf{NP} \not\subseteq \mathbf{BPP}$. 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 multipication: $f_n(x, y) = x \cdot y$, where x and y are $n/2$-bit numbers. Inverting this function is the integer factorization problem, for which no efficient algorithm is known.

A classic and celebrated result in the foundations of cryptography, which we unfortunately do not have time to cover is that cryptographic pseudorandom generators can be constructed from any one-way function:

Theorem 10 *The following are equivalent:*

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

Corollary 11 *If one-way functions exist, then* $\mathbf{BPP} \subseteq \mathbf{SUBEXP}$.

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 proof that G_m is pseudorandom is proven by a reduction as follows. Given any 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).

Thus if f_ℓ is hard to invert by algorithms running in time $s(\ell)$, we can set $t = m = 1/\varepsilon = s(\ell)^{1/c}$ for a constant c. That is, viewing the seed length ℓ' of G_m as a function of m, we have $\ell'(m) = \mathrm{poly}(s^{-1}(m^c))$.

Thus:

- If $s(\ell)$ can be taken to be an arbitrarily large polynomial (as the definition of one-way function above), we get seed length $\ell'(m) = m^\varepsilon$ and $\mathbf{BPP} \subseteq \mathbf{SUBEXP}$ (as discussed above).
- If $s(\ell) = 2^{\ell^{\Omega(1)}}$ (as is plausible for the factoring one-way function), then we get seed length $\ell'(m) = \mathrm{poly}(\log m)$ and $\mathbf{BPP} \subseteq \tilde{\mathbf{P}}$.

But we cannot get seed length $\ell'(m) = O(\log m)$, as needed for concluding $\mathbf{BPP} = \mathbf{P}$, from this result. Even for the maximum possible hardness $s(\ell) = 2^\ell$, we get $\ell'(m) = \mathrm{poly}(\log m)$.

Open Problem 12 *Construct pseudorandom generators from arbitrary one-way functions with seed length $\ell'(m) = O(s^{-1}(m^c))$.*

It is known how to do this from any one-way *permutation* $f : \{0,1\}^\ell \to \{0,1\}^\ell$. In fact, the construction of pseudorandom generators from one-way permutations has a particularly simple description:

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

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 $(f^{(m-1)}(U_n), f^{(m-2)}(U_n), \ldots, f(U_n), U_n)$. 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 the 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 the computational setting, however, is much more delicate and complex to prove than in the setting of extractors.

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 $\{f_s : \{0,1\}^\ell \to \{0,1\}\}_{s\in\{0,1\}^\ell}$ 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 ℓ-bit truly random seed s is stretched to 2^ℓ 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 $f : \{0,1\}^\ell \to \{0,1\}$ (by definition, the function they share is indistinguishable from random by any poly-time 3rd party). Thus, in order for one party to send an encrypted message m to the other, they could simply choose a random $r \in \{0,1\}^\ell$, and send $(r, f_s(r) \oplus m)$. It is easy to see how someone with knowledge of s could decrypt- simply calculate $f_s(r)$ and add 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, $(x_1, f(x_2)), (x_2, f(x_2)), \ldots, (x_k, f(x_k)))$, and we would like to predict what the function's value will be on a new data point x_{k+1} coming from the same distribution. Information-theoretically, it should be possible to predict after a small number of samples assuming that the function has a small description (e.g. is computable by a poly-sized circuit). However, essentially by definition, it should be computationally hard to predict the output of PRFs. Thus, PRFs (if they exist) provide examples of functions that are efficiently computable yet hard to learn (even with membership queries).

- 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, $\mathbf{NP} \not\subseteq \mathbf{P/poly}$). 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 roughly $4.5n$.

  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.

6 Working with computational indistinguishability

In this section we will prove a useful property of computational indistinguishability, and along the way start getting used to working with this notion. The following lemma illustrates that computational indistinguishability behaves like statistical difference when taking many independent repetitions; the distance ε multiplies by the number of copies. 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 13 *If X and Y are (t, ε) indistinguishable, then for every k, X^k and Y^k are $(t, k\varepsilon)$ indistinguishable (where X^k represents k independent copies of X).*

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 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 (**NP**-completeness). The difference in this proof from the corresponding result about statistical difference is that we need be sure to preserve efficiency when going from T' to T.

Suppose that there exists a nonuniform algorithm T such that

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

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 1 is equivalent to

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

meaning that there exists some i such that $\Pr[T(H_{i-1}) = 1] - \Pr[T(H_i) = 1] > \varepsilon$. The latter simply says that

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

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

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

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 things actually costs nothing in terms of circuit size (because constant inputs can be propagated through the circuit, only eliminating gates). Thus T' is a 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 seem to behave nicely, with (t, ε) going to $(t, k\varepsilon)$, it is actually more costly than the corresponding result for statistical difference. 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 distribution X^k and Y^k.

For computational indistinguishability against *uniform* algorithms, we cannot use the same proof, because we cannot hardwire in the samples x_i and y_i. Indeed, the propositoin as stated is known to be false. We need to add the additional condition that the distributions X and Y are efficiently samplable.

Definition 14 *A sequence X_n of random variables is samplable in time $s(n)$ if there is a probabilistic algorithm S running in time $s(n)$ such that for every n, the output of $S(1^n)$ is distributed according to X_n.*

Proposition 15 *Suppose X_n and Y_n are $(t(n), \varepsilon(n))$ indistinguishable for uniform algorithms and that they can be sampled in time $s(n)$. Then for every $k(n)$, $X_n^{k(n)}$ and $Y_n^{k(n)}$ are $(t(n) - O(k(n)s(n)), k(n)\varepsilon(n))$ indistinguishable for uniform algorithms.*

Proof Sketch: The proof is similar to the one before. Consider that there exists T that $k\varepsilon$ distinguishes X_n^k and Y_n^k. Then we will construct T' that distinguishes X_n and Y_n.

On input z, T' chooses $i \stackrel{\text{R}}{\leftarrow} \{1, \ldots, k\}$ and outputs $T(X_n^{k-i} z Y_n^{i-1})$ (where it uses the sampling algorithms for X_n and Y_n to generate the needed samples).

We have that

$$\begin{aligned} \Pr[T'(X_n) = 1] &= \frac{1}{k} \sum_{i=1}^{k} \Pr[T(H_{i-1}) = 1] \\ \Pr[T'(Y_n) = 1] &= \frac{1}{k} \sum_{i=1}^{k} \Pr[T(H_i) = 1]. \end{aligned}$$

Therefore,

$$\Pr[T'(X_n) = 1] - \Pr[T'(Y_n) = 1] = \frac{1}{k} \left(\Pr[T(H_0) = 1] - \Pr[T(H_k) = 1]\right) > \varepsilon.$$

Note that $\mathrm{time}(T') = \mathrm{time}(T) + O(k(n) \cdot s(n))$. $\square$

This illustrates that computational indistinguishability is more subtle than statistical difference, and obtaining reductions for uniform algorithms is more difficult than obtaining reductions for nonuniform algorithms.