Chapter 9

Cryptography

The importance of cryptography in today's online world needs no introduction. Here we focus on the complexity issues that underlie this field.

Techniques for encrypting messages have been invented throughout history. In the post NP-completeness era, researchers tried to invent encryption techniques whose security is based upon the (presumed) intractability of NP-complete problems. This effort has not succeeded to date, seemingly because NP-completeness says something about the intractability of *worst-case* instances of the problem whereas cryptography needs problems that are intractable on *most instances*. After all, when we encrypt email, we require that decryption should be difficult for an eavesdropper for almost all messages, not just for a few messages.

A problem's average-case and worst-case complexities can differ radically. For instance, 3-COLOR is NP-complete on general graphs, but on a random graph (every potential edge is picked to be in the graph with probability 1/2) is solvable in quadratic time or less. For a detailed study of average-case complexity see Chapter 15. However, the current chapter is also self-contained.

In addition to average-case complexity, another important ingredient of cryptography is an answer to the question: "What is a random string and how can we generate one?" We explain this connection more in Section 9.2.

Caveat: All topics in this chapter fall under the heading of *private-key cryptography,* which is more interesting in complexity theory than *public-key cryptography* (e.g., using the famous RSA function described in Example 18).

9.1 Problems that are hard on average

A basic cryptographic primitive is a *one-way function*, which is a function f that is easy to compute but hard to invert on "many" outputs (when the function is not one-to-one, we define inversion to mean that that given $f(x)$ we desire any preimage, namely, an element of $f^{-1}(f(x))$). See Definition 36. Now we give a few examples and discuss the evidence that they are hard "on average inputs."

EXAMPLE 16 Consider a family of functions $\{f_n\}$ where $f_n : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^{2n}$ is defined as $f([x]_2, [y]_2) = [x \cdot y]_2$. Computing this function is easy, but inverting it is the famous FACTORING problem, for which the best current algorithm has running time

about $2^{O(\sqrt[3]{n})}$ (and even that running time relies upon unproven conjectures in number theory). In fact f seems hard to invert for many inputs, not just for some some. If x and y are primes —which by the Prime Number Theorem happens with probability $\Theta(1/n^2)$ when x, y are random n-bit integers— then f seems hard to invert.

An even harder version of the above function is obtained by using the existence of a randomized polynomial-time algorithm A (which we do not describe) that, given 1^n, generates a random n-bit primes number. The function $\tilde{f}$ maps $(\tilde{x}, \tilde{y})$ to $[x \cdot y]_2$, where $\tilde{x}, \tilde{y}$ are random strings that cause A to output primes x, y respectively. This seems hard to invert for almost all inputs.

EXAMPLE 17 Let $p_1, p_2, \ldots$ be a sequence of primes where p_i has i bits. Let g_i be the generator of the group Z_p^* the set of numbers that're nonzero mod p. Since g_i is a generator, for every $y \in 1, .., p_i - 1$, there is a unique $x \in \{1, .., p-1\}$ such that

$$g_i^x \equiv y \pmod{p_i}.$$

Then $x \to g_i^x \pmod{p_i}$ is a permutation on $1, .., p_i - 1$ and is conjectured to be one-way. The inversion problem is called the *DISCRETE LOG* problem. We show below using random self-reducibility that if it is hard on worst-case inputs, then it is hard on average.

We list some more conjectured one-way functions.

EXAMPLE 18 *RSA function.* Let $m = pq$ where p, q are large random primes and e be a random number coprime to $\phi(m) = (p-1)(q-1)$. Let Z_m^* be the set of integers in $[1, \ldots, m]$ coprime to m. Then the function is defined to be $f_{p,q}(x) = x^e \pmod{m}$. This function is used in the famous RSA public-key cryptosystem.

Rabin function. For a composite number m, define $f_m(x) = x^2 \pmod{m}$. If we can invert this function on a $1/\text{poly}(\log m)$ fraction of inputs then we can factor m in $\text{poly}(\log m)$ time (see exercises).

9.1.1 Random-self-reducibility

Roughly speaking, a problem is *random-self-reducible* if solving the problem on any input x reduces to solving the problem on a sequence of random inputs $y_1, y_2, \ldots,$ where each y_i is uniformly distributed among all inputs. To put it more intuitively, the worst-case can be reduced to the average case. Hence the problem is either easy on *all* inputs, or hard on *many* inputs. (In other words, we can exclude the possibility that problem is easy on almost all the inputs but not all.) This is best illustrated with an example.

THEOREM 35
Suppose A is an algorithm with running time $t(n)$ that, given a prime p, a generator g for $\mathbf{Z}_p^*$, and an input $g^x \pmod{p}$, manages to find x for δ fraction of $x \in \mathbf{Z}_p^*$. Then there is a randomized algorithm A' with running time $O(\frac{1}{\delta \log 1/\epsilon}(t(n) + \text{poly}(n)))$ that solves DISCRETE LOG on every input with probability at least $1 - \epsilon$.

PROOF: Suppose we are given $y = g^x \pmod{p}$ and we are trying to find x. Repeat the following trial $O(1/(\delta \log 1/\epsilon))$ times: "Randomly pick $r \in \{0, 1, \ldots, p-2\}$ and use A to

try to compute the logarithm of $y \cdot g^r (\mathrm{mod} p)$. Suppose A outputs z. Check if $g^{z-r} (\mathrm{mod} p)$ is y, and if so, output $z - r (\mathrm{mod}(p-1))$ as the answer."

The main observation is that if r is randomly chosen, then $y \cdot g^r (\mathrm{mod} p)$ is randomly distributed in $\mathbf{Z}_p^*$ and hence the hypothesis implies that A has a δ chance of finding its discrete log. After $O(1/(\delta \log 1/\epsilon)$ trials, the probability that A failed every time is at most ϵ. □

COROLLARY 36
If DISCRETE LOG mod p is hard on worst-case $x \in \mathbf{Z}_p^*$, then it is hard on almost all x.

Later as part of the proof of Theorem 39 we give another example: linear functions over $GF(2)$.

9.2 What is a random-enough string?

Cryptography often becomes much easier if we have a ready supply of random bits. Here is an example.

EXAMPLE 19 (ONE-TIME PAD) Suppose the message sender and receiver share a long string r of random bits that is not available to eavesdroppers. Then secure communication is easy. To encode message $m \in \{0,1\}^n$, take the first n bits of r, say the string s. Interpret both strings as vectors in $GF(2)^n$ and encrypt m by the vector $m+s$. The receiver decrypts this message by adding s to it (note that $s+s=0$ in $GF(2)^n$). If s is statistically random, then so is $m+s$. Hence the eavesdropper provably cannot obtain even a single bit of information about m *regardless of how much computational power he expends.*

Note that reusing s is a strict no-no (hence the name "one-time pad"). If the sender ever reuses s to encrypt another message m' then the eavesdropper can add the two vectors to obtain $(m+s)+(m'+s) = m+m'$, which is some nontrivial information about the two messages.

Of course, the one-time pad is just a modern version of the old idea of using "codebooks" with a new key prescribed for each day.

One-time pads are conceptually simple, but impractical to generate because sources of quality random bits (e.g., those based upon quantum phenomena) are often slow. Also, distributing them securely to users poses problems. Cryptography's suggested solution to such problems is to use a pseudorandom generator. This is a deterministically computable function $g : \{0,1\}^n \to \{0,1\}^{n^c}$ such that if $x \in \{0,1\}^n$ is randomly chosen, then $g(x)$ "looks" random. Thus users can stretch a shared n-bit random string to n^c "random looking" bits.

Clearly, at this point we need an answer to the question posed in the Section's title! Philosophers and statisticians have long struggled with this question.

EXAMPLE 20 What is a random-enough string? Here is Kolmogorov's definition: *A string of length n is random if no Turing machine whose description length is $< 0.99n$ (say) outputs this string when started on an empty tape.* This is not very useful because checking if a string is random is undecidable.

Statisticians have also attempted definitions which boil down to checking if the string has the "right number" of patterns that one would expect by the laws of statistics, e.g. the number of times 11100 appears as a substring. (See Knuth Volume x for a comprehensive discussion.) Such definitions are too weak in the cryptographic setting.

9.2.1 Blum-Micali and Yao definitions

Now we introduce two complexity-theoretic definitions of pseudorandomness due to Blum-Micali and Yao in the early 1980s. Let $y|_{\leq i}$ denotes the first i bits of string y.

The Blum-Micali definition is motivated by the observation that one property (in fact, the defining property) of a statistically random sequence of bits y is that given $y|_{\leq i}$, we cannot predict y_{i+1} with odds better than 50/50 *regardless of the computational power available to us*. Thus one definition of a "pseudorandom string" would be that no resource-bounded Turing machine can achieve odds much better than 50/50 in predicting y_{i+1} from $y|_{\leq i}$. Of course, this definition has the shortcoming that any single finite string would be predictable for a trivial reason: it could be hardwired into the program of the predictor Turing machine. To get around this difficulty the Blum-Micali definition (and also Yao's definition below) defines pseudorandomness for *ensembles* (or distributions) of strings rather than for individual strings.

DEFINITION 34 (BLUM-MICALI) *A function family* $\{g_n\}$ *where* $g_n : \{0,1\}^n \to \{0,1\}^m$ *is a polynomial-time computable function is* $(e(n), t(n))$-unpredictable *if for every probabilistic polynomial-time algorithm* A *that runs in time* $t(n)$ *and every large enough input size* n*,*

$$\Pr[A \text{ predicts } g(x)_{i+1} \text{ given } g(x)_{\leq i}] \leq \frac{1}{2} + \epsilon(n),$$

where the probability is over the choice of $x \in \{0,1\}^n, i \in \{1, \ldots, n\}$*, and the randomness used by* A*.*

If for every fixed k*, the family* $\{g_n\}$ *is* $(1/n^c, n^k)$*-unpredictable for every* $c > 1$*, then we say in short that it is* unpredictable by polynomial-time algorithms.

REMARK 3 Allowing the testor to be an arbitrary polynomial-time machine makes perfect sense in a cryptographic setting where we wish to assume nothing about the adversary except an upperbound on her computational power.

Yao gave an alternative definition in which testor machine is given access to the entire string at once. This definition implicitly sets up a test of randomness analogous to the more famous Turing test for intelligence. The testor machine A is given a string $y \in \{0,1\}^{n^c}$ that is produced in one of two ways: it is either drawn from the uniform distribution on $\{0,1\}^{n^c}$ or generated by taking a random string $x \in B^n$ and stretching it using a deterministic function $g : \{0,1\}^n \to \{0,1\}^{n^c}$. The testor is asked to output "1" if the string looks random to it and 0 otherwise. We say that g is a pseudorandom generator if no polynomial-time testor machine A is able determine with great certainty which of the two distributions the string came from.

DEFINITION 35 (YAO'82) *A family* $\{g_n\}$, $g_n\ \{0,1\}^n \mapsto \{0,1\}^{n^c}$, *is called an* $(\delta(n), s(n))$-pseudorandom generator *if for every probabilistic algorithm* A *running in time* $s(n)$ *and for all large enough* n

$$|\mathbf{Pr}_{y\in\{0,1\}^{n^c}}[A(y)=1] - \mathbf{Pr}_{x\in\{0,1\}^n}[A(g_n(x))=1]| \leq \delta(n), \tag{9.1}$$

where $\delta(n)$ *is the* distinguishing probability *and* $s(n)$ *is the* security parameter.

If for every k, *the family is* $(1/n^{c'}, n^k)$*-pseudorandom for every* $c' > 1$ *then we say in short that it is a* pseudorandom generator

9.2.2 Equivalence of the two definitions

Yao showed that the above two definitions are equivalent. The *hybrid argument* used in this proof has become a central idea of cryptography and complexity theory.

THEOREM 37 (YAO)
A family g_n *is pseudorandom iff it is unpredictable by polynomial-time algorithms.*

PROOF: We show that if a family is $(\epsilon(n), t(n))$-unpredictable then it is ??-pseudorandom and conversely, if it is..

Suppose g is not $(\epsilon(n), t(n))$-pseudorandom, and A is a PPT algorithm such that

$$\left| \Pr_{x\in B^n}[A(g(x))=1] - \Pr_{y\in\{0,1\}^N}[A(y)=1] \right| > \epsilon(n). \tag{9.2}$$

By considering either A or the algorithm that is A with the answer flipped, we can assume that the $|\cdot|$ can be removed and in fact

$$\Pr_{x\in B^n}[A(g(x))=1] - \Pr_{y\in\{0,1\}^N}[A(y)=1] > \epsilon(n). \tag{9.3}$$

Consider B, the following bit-prediction algorithm. Let its input be $g(x)|_{\leq i}$ where $x \in \{0,1\}^n$ and $i \in \{0,\ldots,N-1\}$ are chosen uniformly at random. B's program is: "*Pick bits* $u_{i+1}, u_{i+2}, \ldots, u_N$ *randomly and run* A *on the input* $g(x)|_{\leq i}u_{i+1}u_{i+2}\ldots u_N$. *If* A *outputs* 1, *output* u_{i+1} *else output* $\overline{u_{i+1}}$." To complete the proof we show that B predicts $g(x)_{i+1}$ correctly with probability at least $\frac{1}{2} + \frac{\epsilon(n)}{N}$.

Consider a sequence of $N+1$ distributions $\mathcal{D}_0$ through $\mathcal{D}_N$ defined as follows (in all cases, $x \in \{0,1\}^n$ and $u_1, u_2, \ldots, u_N \in \{0,1\}$ are assumed to be chosen randomly)

$$\mathcal{D}_0 = u_1u_2u_3u_4\cdots u_N \tag{9.4}$$
$$\mathcal{D}_1 = g(x)_1u_2u_3\cdots u_N \tag{9.5}$$
$$\vdots \quad \vdots \tag{9.6}$$
$$\mathcal{D}_i = g(x)_{\leq i}u_{i+1}\cdots u_N \tag{9.7}$$
$$\vdots \quad \vdots \tag{9.8}$$
$$\mathcal{D}_N = g(x)_1g(x)_2\cdots g(x)_N \tag{9.9}$$

Furthermore, we denote by $\overline{\mathcal{D}_i}$ the distribution obtained from $\mathcal{D}_i$ by flipping the ith bit (i.e., replacing $g(x)_i$ by $\overline{g(x)_i}$). If $\mathcal{D}$ is any of these $2(N+1)$ distributions then we denote $\Pr_{y\in\mathcal{D}}[A(y)=1]$ by $q(\mathcal{D})$. With this notation we rewrite (9.3) as

$$q(\mathcal{D}_N) - q(\mathcal{D}_0) > \epsilon(n). \tag{9.10}$$

Furthermore, in $\mathcal{D}_i$, the $(i+1)$th bit is equally likely to be $g(x)_{i+1}$ and $\overline{g(x)_{i+1}}$, so

$$q(\mathcal{D}_i) = \tfrac{1}{2}(q(\mathcal{D}_{i+1}) + q(\overline{\mathcal{D}_{i+1}})), \tag{9.11}$$

Now we analyze the probability that B predicts $g(x)_{i+1}$ correctly. Since i is picked randomly we have

$$\begin{aligned}
\Pr[B \text{ is correct}] = \frac{1}{N}\sum_{i=0}^{n-1}\frac{1}{2}(&\Pr_{x,\vec{u}}[B\text{'s guess for } g(x)_{i+1} \text{ is correct} \mid u_{i+1} = g(x)_{i+1}] \\
&+ \Pr_{x,\vec{u}}[B\text{'s guess for } g(x)_{i+1} \text{ is correct} \mid u_{i+1} = \overline{g(x)_{i+1}}]) \\
= \frac{1}{2N}\sum_{i=0}^{N-1}&(q(\mathcal{D}_{i+1}) + 1 - q(\overline{\mathcal{D}_{i+1}})) \\
= \frac{1}{2} + \frac{1}{2N}\sum_{i=0}^{N-1}&(q(\mathcal{D}_{i+1}) - q(\overline{\mathcal{D}_{i+1}}))
\end{aligned}$$

From (9.11), $q(\mathcal{D}_{i+1}) - q(\overline{\mathcal{D}_{i+1}}) = 2(q(\mathcal{D}_{i+1}) - q(\mathcal{D}_i))$, so this becomes

$$\begin{aligned}
&= \frac{1}{2} + \frac{1}{2N}\sum_{i=0}^{N-1} 2(q(\mathcal{D}_{i+1}) - q(\mathcal{D}_i)) \\
&= \frac{1}{2} + \frac{1}{N}(q(\mathcal{D}_N) - q(\mathcal{D}_0)) \\
&> \frac{1}{2} + \frac{\epsilon(n)}{N}.
\end{aligned}$$

The converse direction is easier since a bit prediction algorithm can in particular be used to distinguish $g(x)$ from random strings of the same length. It is left to the reader. □

9.3 One-way functions and pseudorandom number generators

Do pseudorandom generators exist? Surprisingly the answer (though we will not prove it in full generality) is that they do if and only if *one-way functions* exist. In fact most tasks in private-key cryptography hinge on the existence of one-way functions.

Earlier we introduced one-way functions informally; now we give a definition.

DEFINITION 36 (ONE-WAY FUNCTION) *A family of functions $\{f_n : \{0,1\}^n \mapsto \{0,1\}^{n^c}\}$ is* $\epsilon(n)$ one-way *with* security $s(n)$ *if there is a polynomial time Turing machine that computes them and furthermore for every algorithm A that runs in time $s(n)$,*

$$\mathbf{Pr}_{x\in\{0,1\}^n}[A \textit{ inverts } f_n(x)] \leq \epsilon(n). \tag{9.12}$$

REMARK 4 A *weak* one-way function is one that is $(1-1/n^c)$ one-way (i.e., difficult to invert on only $1/n^c$ fraction of inputs) with a security parameter $s(n)$ that is a superpolynomial function of n. Yao has proven that such a weak one-way function can be converted into a *strong* one-way function which is $1/s(n^\delta)^\gamma$-one-way with security $s(n)^\gamma$ for some constant $\delta, \gamma > 0$. Although we postpone this result to a subsequent chapter, we mention it here because Theorem 39 below uses a strong one-way permutation.

THEOREM 38 (HILL)
One-way functions exist iff pseudorandom generators do.

The "if" direction of this theorem is trivial: if g is a pseudorandom generator then it must also be a one-way function since otherwise the algorithm that inverts g would be able to distinguish its outputs from random strings. The "only if" direction is more difficult and involves using a one-way function to explicitly construct a pseudorandom generator. We will do this only for the special case of one-way functions that are *permutations*, namely, they map $\{0,1\}^n$ to $\{0,1\}^n$ in a one-to-one and onto fashion. As a first step, we describe the Goldreich-Levin theorem, which gives an easy way to produce one pseudorandom bit, and then describe how to produce n^c pseudorandom bits.

9.3.1 Goldreich-Levin hardcore bit

Let $\{f_n\}$ be a one-way permutation where $f_n : \{0,1\}^n \to \{0,1\}^n$. Clearly, the function $g : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^{2n}$ defined as $g(x,r) = (f(x), r)$ is also a one-way permutation. Since f is a permutation, $(f(x), r)$ contains all the information information-theoretically equivalent to x, r. Thus even though $f(x), r$ contains definition of one-way permutation we cannot s no computationally efficient method to obtain Nevertheless, Goldreich and Levin showed that given $(f(x), r)$, it is difficult for a polynomial-time algorithm to predict $x \odot r$, the scalar product of x and r (mod 2). This bit is called a *hardcore* bit for the permutation. Prior to the Goldreich-Levin result we knew of hardcore bits for specific (conjectured) one-way permutations, not all.

THEOREM 39 (GOLDREICH, LEVIN '86)
Suppose that f_n is a one-way permutation and has security $s(n)$. Then for all algorithms A running in time $s^{1/4}(n)$

$$\mathbf{Pr}_{x,r\in\{0,1\}^n}[A(f_n(x), r) = x \odot r] \leq \frac{1}{2} + O(\frac{1}{s(n)}). \tag{9.13}$$

PROOF: Suppose that A can predict $x \odot r$ with probability $1/2 + \delta$. We show how to invert $f_n(x)$ for $O(\delta)$ fraction of the inputs in $O(n/\delta^2)$ time, from which the theorem follows.

CLAIM 40
Suppose that

$$\mathbf{Pr}_{x,r\in\{0,1\}^n}[A(f_n(x), r) = x \odot r] \geq \frac{1}{2} + \delta. \tag{9.14}$$

Then for at least δ fraction of x's

$$\mathbf{Pr}_{r\in\{0,1\}^n}[A(f_n(x), r) = x \odot r] \geq \frac{1}{2} + \frac{\delta}{2}. \tag{9.15}$$

PROOF: We use an averaging argument. Suppose that p is the fraction of x's satisfying (9.15). We have $p \cdot 1 + (1-p)(1/2 + \delta/2) \geq 1/2 + \delta$. Solving this with respect to p, we obtain

$$p \geq \frac{\delta}{2(1/2 - \delta/2)} \geq \delta.$$

□

We construct an inversion algorithm that given $f_n(x)$, where $x \in_R \{0,1\}^n$, will try to recover x. It "succeeds" with high probability if x is such that (9.15) holds. Note that the algorithm can always check the correctness of its answer, since it has $f_n(x)$ available to it and it can apply f_n to its answer and see if $f_n(x)$ is obtained.

WARMUP: *Reconstruction when the probability in (9.15) is* $\geq 3/4 + \delta$.

Let P be any program that computes some unknown linear function over $GF(2)^n$ but errs on some inputs. Specifically, there is an unknown vector $a \in GF(2)^n$ such that

$$\Pr_x[P(x) = a \cdot x] = 3/4 + \delta. \tag{9.16}$$

Then we can use this program to compute $a \cdot x$ for every x with high probability. "On input x, repeat the following trial $O(1/\delta^2 \log 1/\epsilon)$ times. Pick y randomly from $GF(2)^n$ and compute the bit $P(x+y) + P(y)$. At the end, output the majority value."

The main observation is that when y is randomly picked from $GF(2)^n$ then $x+y$ and y are both randomly distributed in $GF(2)^n$, and hence the probability that $P(x+y) \neq a \cdot (x+y)$ or $P(y) \neq a \cdot y$ is at most $2 \cdot (1/4 - \delta) = 1/2 - 2\delta$. Thus each trial produces the correct bit with probability at least $1/2 + 2\delta$, and then Chernoff bounds imply that probability is at least $1 - \epsilon$ that the final majority is correct.

GENERAL CASE:

The idea for the general case is very similar, the only difference being that this time we want to pick $r_1, \ldots, r_m$ so that we already "know" $x \odot r_i$. The preceding statement may appear ridiculous, since knowing the inner product of x with $m \geq n$ random vectors is, with high probability, enough to reconstruct x (check this!). The catch will of course be that the r_i's will not be completely random. Instead, they will be pairwise independent. Recall the following construction of a set of pairwise independent vectors: Pick k random vectors $t_1, t_2, \ldots, t_k \in GF(2)^n$ and for each nonempty $S \subseteq \{1, \ldots, k\}$ define $Y_S = \sum_{i \in S} t_i$. This gives $2^k - 1$ vectors and for $S \neq S'$ we have $Y_S, Y_{S'}$ are independent.

Now let us describe the observation at the heart of the proof. Suppose $m = 2^k - 1$ and our random strings $r_1, \ldots, r_m$ are $\{Y_S\}$ from the previous example. Then $x \odot Y_S = x \odot (\sum_{i \in S} t_i) = \sum_{i \in S} x \odot t_i$. Now, if we know $x \odot t_i$ for $i = 1, \ldots, k$, we also know $x \odot Y_S$. Of course, we don't know $x \odot t_i$ for $i = 1, \ldots, k$, but we can just try all 2^k possibilities for this vector and run the rest of the algorithm for each of them. This multiplies the running time by a factor 2^k, which is only m. This is how we can assume that we know $x \odot Y_S$ for each subset S.

The details of the rest of the algorithm are similar to before. By Lemma 2, we know that if the theorem were not true, then for at least $\delta = \alpha/s(n)$ fraction of x's, where α is some constant, the probability over r that A gives the correct answer is at lest $1/2 + \delta/2$. We now concentrate our attention on those x's. Pick m pairwise independent vectors Y_S's

as described above, calculate $A(f_n(x), Y_S \oplus e_i) - x \odot Y_S$, and take the majority vote. The expected number of correct answers is $m(1/2+\delta/2)$, so for the majority vote to result in the incorrect answer it must be the case that the number of incorrect values deviates from its expectation by more than $m\delta/2$. Now, we can bound the variance of this random variable and apply Chebyshev's inequality.

Formally, let ξ_S denote the event that A produces the correct answer on Y_S; we have $E(\xi_S) = 1/2+\delta/2$ and $Var(\xi_S) = E(\xi_S)(1-E(\xi_S)) < 1$. Let $\xi = \sum_S \xi_S$ denote the number of correct answers on a sample of size m. By linearity of expectation, $E[\xi] = m(1/2+\delta/2)$. Furthermore, the Y_S's are pairwise independent, which implies that the same is true for the outputs ξ_S's produced by the algorithm A on them. Hence by pairwise independence $Var(\xi) < m$. Now, by Chebyshev's inequality, the probability that the majority vote is incorrect is at most $\frac{4Var(\xi)}{m^2\delta^2} \leq \frac{4}{m\delta^2}$. Recalling that $\delta = \alpha/s(n)$, we see that if we set $m = \Omega(ns^2(n))$, the probability of guessing the ith bit incorrectly is at most $1/2n$, and by the union bound, the probability of guessing the whole word incorrectly is at most $1/2$. Hence, on a "good" x, we can find the preimage of $f(x)$ with a good probability, and the number of "good" x's is non-negligible, which contradicts our assumption that f is one-way. □

9.3.2 Pseudorandom number generation

We saw that if f is a one-way permutation, then $g(x,r) = (f(x), r, x \odot r)$ is a pseudorandom generator that stretches $2n$ bits to $2n+1$ bits. Stretching even more not much more difficult, as we now show. Let $f^i(x)$ denote the i-th iterate of f on x (i.e., $f(f(f(\cdots(f(x)))))$ where f is applied i times).

THEOREM 41
If f is a one-way permutation then $g_N(x,r) = (r, x \odot r, f(x) \odot r, f^2(x) \odot r, \ldots, f^N(x) \odot r)$ is a pseudorandom generator for $N = n^c$ for any constant $c > 0$.

PROOF: Since any distinguishing machine could just reverse the string as a first step, it clearly suffices to show that the string $(r, f^N(x) \odot r, f^{N-1}(x) \odot r, \ldots, f(x) \odot r, x \odot r)$ looks pseudorandom. By Yao's theorem (Theorem 37), it suffices to show the difficulty of bit-prediction. For contradiction's sake, assume there is a PPT machine A such that when $x, r \in \{0,1\}^n$ and $i \in \{1, \ldots, N\}$ are randomly chosen,

$$\Pr[A \text{ predicts } f^i(x) \odot r \text{ given } (r, f^N(x) \odot r, f^{N-1}(x) \odot r, \ldots, f^{i+1}(x) \odot r)] \geq \frac{1}{2} + \epsilon.$$

We describe an algorithm B that given $f(z), r$ where $z, r \in \{0,1\}^n$ are randomly chosen, predicts the hardcore bit $z \odot r$ with reasonable probability, which contradicts Theorem 39.

Algorithm B picks $i \in \{1, \ldots, N\}$ randomly. Let $x \in \{0,1\}^n$ be such that $f^i(x) = z$. There is of course no efficient way for B to find x, but for any $l \geq 1$, B can efficiently compute $f^{i+l}(x) = f^{l-1}(f(z))$! So it produces the string $r, f^N(x) \odot r, f^{N-1}(x) \odot r, \ldots, f^{i+1}(x) \odot r$ and uses it as input to A. By assumption, A predicts $f^i(x) \odot r = z \odot r$ with good odds. Thus we have derived a contradiction to Theorem 39. □

9.4 Applications

Now we give some applications of the ideas introduced in the chapter.

9.4.1 Playing poker over the phone

How can two parties A and B play poker over the phone? Specifically, how do they deal the cards in a fair way? Clearly, they need a way to toss a fair coin over the phone. If only one of them actually tosses a coin, there is nothing to prevent him from lying about the result. The following fix suggests itself: both players toss a coin and they take the XOR as the shared coin. Even if B does not trust A to use a fair coin, he knows that as long as his bit is random, the XOR is also random. Unfortunately, this idea does not work because the player who reveals his bit first is at a disadvantage: the other player could just "adjust" his answer to get his desired coin toss.

This problem is addressed by the following scheme, which assumes that A and B are polynomial time turing machines that cannot invert one-way permutations. The protocol itself is called *bit commitment*. First, A chooses two strings x_A and r_A of length n and sends a message $(f_n(x_A), r_A)$, where f_n is a one-way permutation. This way, A commits the string x_A without revealing it. Now B selects a random bit b and conveys it. Then A reveals x_A and they agree to use the XOR of b and $(x_A \odot r_A)$ as their coin toss. Note that B can verify that x_A is the same as in the first message by applying f_n, therefore A cannot change her mind after learning B's bit. On the other hand, by the Goldreich–Levin theorem, B cannot predict $x_A \odot r_A$ from A's first message, so this scheme is secure.

9.4.2 Derandomization

The existence of pseudorandom generators implies subexponential deterministic algorithms for BPP: this is usually referred to as *derandomization* of BPP. (In this case, the derandomization is only partial since it results in a subexponential deterministic algorithm. Stronger complexity assumptions imply a full derandomization of BPP, as we will see later.)

THEOREM 42
If there is a pseudorandom generator that is secure against circuits of size n^c, then BPP $\subseteq$ $\cap_{\varepsilon>0}$ DTIME(2^{n^ε}).

PROOF: Let us fix an $\varepsilon > 0$ and show that BPP $\subseteq$ DTIME(2^{n^ε}).

Suppose that M is a BPP machine running in n^k time. We can build another probabilistic machine M' that takes n^ε random bits, streches them to n^k bits using the pseudorandom generator and then simulates M using this n^k bits as a random string. Obviously, M' can be simulated by going over all binary strings n^ε, running M' on each of them, and taking the majority vote.

It remains to prove that M and M' accept the same language. Suppose otherwise. Then there exists an infinite sequence of inputs $x_1, \ldots, x_n, \ldots$ on which M distinguishes a truly random string from a pseudorandom string with a high probability, because for M and M' to produce different results, the probability of acceptance should drop from 2/3 to below

1/2. Hence we can build a distinguisher similar to the one described in the previous theorem by hardwiring these inputs into a circuit family. □

The relationship between hardness and randomness is a subject of recent research.

REMARK 5 There is an interesting connection to *discrepancy theory*, a field of mathematics. Let $\mathcal{S}$ be a set of subsets of $\{0,1\}^n$. Subset $A \subset \{0,1\}^n$ has *discrepancy* ϵ *with respect to* $\mathcal{S}$ if for every $s \in \mathcal{S}$,

$$\left| \frac{|s \cap A|}{|S|} - \frac{|A|}{2^n} \right| \leq \epsilon.$$

Our earlier result that BPP $\subseteq$ P/*poly* showed the *existence* of polynomial-size sets A that have low discrepancy for all sets defined by polynomial-time Turing machines (we only described discrepancy for the universe $\{0,1\}^n$ but one can define it for all input sizes using lim sup). The goal of derandomization is to explicitly construct such sets.

9.4.3 Pseudorandom functions

Pseudorandom functions are a natural generalization of (and are easily constructed using) pseudorandom generators. This is a function $g : \{0,1\}^m \times \{0,1\}^n \to \{0,1\}^n$. For each $K \in \{0,1\}^m$ we denote by $g|_K$ the function from $\{0,1\}^n$ to $\{0,1\}^n$ defined by $g|_K(x) = g(K,x)$. Thus the family contains 2^m functions from $\{0,1\}^n$ to $\{0,1\}^n$, one for each K. Recall that the set of all functions from $\{0,1\}^n$ to $\{0,1\}^n$, denoted $\mathcal{F}_n$, has cardinality $(2^n)^{2^n}$.

We say g is a pseudorandom function generator if it passes a "Turing test" of randomness analogous to that in Yao's definition of a pseudorandom generator (Definition 35). The PPT machine is presented with an "oracle" for a function from $\{0,1\}^n$ to $\{0,1\}^n$. The function is one of two types: either a function chosen randomly from $\mathcal{F}_n$, or a function $f|_K$ where $K \in \{0,1\}^m$ is randomly chosen. The PPT machine is allowed to query the oracle in any points of its choosing. We require that the PPT has negligible probability of detecting which of the two cases holds.

Now we describe a construction of a pseudorandom function generator g from a length-doubling pseudorandom generator $f : \{0,1\}^m \to \{0,1\}^{2m}$. For any $K \in \{0,1\}^m$ let T_K be a complete binary tree of depth n whose each node is labelled with an m-bit string. The root is labelled K. If a node in the tree has label y then its left child is labelled with the first n bits of $f(y)$ and the right child is labelled with the last n bits of $f(y)$. Now we define $g(K,x)$. For any $x \in \{0,1\}^n$ interpret x as a label for a path from root to leaf in T_K in the obvious way and output the label at the leaf.

We leave it as an exercise to prove that this construction is correct.

A very interesting application of pseudorandom function generators to the P versus NP question appears in Chapter 21.

9.4.4 Lowerbounds for machine learning

In *machine learning* the goal is to learn a succinct function $f : \{0,1\}^n \to \{0,1\}$ from a sequence of type $(x_1, f(x_1)), (x_2, f(x_2)), \ldots$, where the x_i's are randomly-chosen inputs. Clearly, this is impossible in general since a random function has no succinct description. But suppose f has a succinct description, e.g. as a small circuit. Can we learn f in that case?

The existence of pseudorandom functions implies that even though a function may be polynomial-time computable, there is no way to learn it from examples in polynomial time. In fact it is possible to extend this impossibility result (though we do not attempt it) to more restricted function families such as NC^1 (see Kearns and Valiant, ...).

9.5 Recent developments

The earliest cryptosystems were designed using the SUBSET SUM problem. They were all shown to be insecure by the early 1980s. In the last few years, interest in such problems —and also the related problems of computing approximate solutions to the shortest and nearest lattice vector problems— has revived thanks to a result of Ajtai, improved since then by other researchers.

brief description to be written

Chapter notes

To be written.

Exercises

§1 Show that if P = NP then one-way functions and pseudorandom generators do not exist.

§2 (Requires just a little number theory). Prove that if some algorithm inverts the Rabin function $f_m(x) = x^2 \pmod m$ on a $1/\text{poly}(\log m)$ fraction of inputs then we can factor m in $\text{poly}(\log m)$ time. (Hint: Suppose $m = pq$ where p, q are prime numbers. Then x^2 has 4 "square roots" modulo m.)

§3 Show that if f is a one-way permutation then so is f^k (namely, $f(f(f(\cdots(f(x)))))$ where f is applied k times).

§4 Show that the above fails for one-way functions. (You may assume that one-way functions exist.)

§5 Show the correctness of the pseudorandom function generator in Section 9.4.3. (Hint: Use a hybrid argument.)