Heads & Tails, summer 2009 [1]

Bonn-Aachen International Center
for Information Technology

Version: July 29, 2009

[1] This text is part of a larger set of lecture notes: cross-references to other sections are replaced by **??**.
This text is not for distribution

Contents

Chapter 1

Pseudorandom generators

Random numbers or random bit strings are essential in many areas of computer science, from sorting, routing in networks, and computer algebra to cryptography. Most computer systems provide a function like RAND that delivers elements which look "random" in some sense. However, there is no practical inexpensive way known to generate truly random numbers. One can think of measuring radioactive activity, current machine clock time or disk usage, user input like keystroke timing or mouse movement, but these are either expensive or not very random. What else can you think of?

The most popular type of random generators, based on linear congruential generation, is successfully used in many applications. But it is not good enough for cryptography. So cryptographers had to invent their own notion, called (computational) pseudorandom generators, which are the topic of this chapter.

Such a generator takes a small amount of true randomness as input and produces a large amount of pseudorandomness. The defining property is that these pseudorandom elements cannot be told apart from truly random ones by any efficient algorithm.

In this chapter, we first define and illustrate this notion of "distinguishing" between pseudorandom and truly random elements, then see that it is essentially equivalent to "predicting the next element", and finally discuss two specific generators, by Nisan & Wigderson (1994) and by Blum *et al.* (1986).

1.1. True random generators

Randomness is a vital ingredient for cryptography, from the generation of random keys to the challenges in identification schemes. There are two types of method for generating the required randomness. Both are inconvenient, expensive, and potentially insecure.

A **software-based generator** measures some process such as

- the system clock,

- key stroke or mouse movements,
- system or network parameters,
- the contents of certain registers,
- user input.

All of these have their problems. A 1 GHz machine running uninterrupted for a whole year (good luck!) goes through $365 \cdot 24 \cdot 60 \cdot 60 \cdot 10^9$ or about $2^{54.8}$ cycles. So even if we took that as random, we would only get about 54 bits. In a more realistic situation, say a smartcard engaged in an identification protocol, we can at best expect a few usable bits, certainly not enough for any reasonable protocol. Key strokes and mouse movements can possibly be observed. Some versions of PGP require a new user to execute about 15 seconds of energetic mouse pushing. That's ok, but you would not be prepared to do this every time you withdraw money from an ATM. System parameters and register contents might be predicted or simulated. The most common method are user-generated passwords. With appropriate cautions, this is quite reasonable, but again one can expect only a few "random" bits.

The second type of method are **hardware-based generators** which measure some physical process, such as

- radioactive decay,
- semi conductor thermal noise,
- capacitor charge,
- sector access times in a sealed hard disk.

All of these are expensive and face potential observation or manipulation by an adversary.

A random sequence is **correlated** if the probability that a bit is 1 depends on the previous bits. There are methods to remove correlation, but we do not go into this.

Such a sequence is **biased** if each bit equals 1 with some probability p, with $0 < p < 1$, and hence equals 0 with probability $1-p$. Von von Neumann suggested how to remove such a bias: we group the sequence into consecutive pairs, and take 10 to mean 1, 01 to mean 0, and discard 00 and 11.

1.2. Pseudorandom generators

A pseudorandom generator will be a deterministic algorithm $\mathcal{A}$ with (random) inputs from a small set X and outputs in a large set Y which are "indistinguishable" from random elements of Y. This notion is defined in the next section.

Often we will have $X = \{0,1\}^k$ and $Y = \{0,1\}^n$ for some $k < n$, and then $\mathcal{A}$ is a pseudorandom bit generator.

Thus the idea of pseudorandom generators is to take a very small amount of randomness (a random element of X) to produce a large amount of pseudorandomness (an element of Y). These new elements are not truly random, but should behave as if they were for the intended application.

A detailed discussion of random generation is in Knuth (1998). Knuth presents a large array of **statistical tests** for pseudorandomness. It seems hard to describe a general strategy for employing these tests; one has to decide each time anew which tests are appropriate for the purpose at hand.

In contrast to the underlying notion of **statistical pseudorandomness**, we will develop a theory of **computational pseudorandomness**. This is the right approach for cryptographical applications. We will see a "universal test", namely predicting the next pseudorandom element, and establish a strong connection with computational complexity, the theory that asks how "hard" it is to solve a given problem.

What is a random element, say a random bit? Is 0 a random bit? Is 1? These nonsensical questions indicate that there is no reasonable way to talk about the randomness of an individual bit, or any finite bit string. One can define randomness for infinite strings. For our purposes, it is more useful to talk about "potentially infinite strings", namely machines that produce individual bits. Then one can have such a machine produce arbitrarily long strings of "random" elements. When X is a finite set, a (uniform) truly random generator for X would produce (without any input) a uniformly random element of X, so that each element of X has the same probability $1/\#X$ of occurring. Nobody knows how to build such a generator (which is efficient).

The most popular pseudorandom generators are the **linear congruential pseudorandom generators**. We have a modulus $m \in \mathbb{N}$, two integers a, b, a **seed** $x_0 \in \mathbb{N}$, and define

$$x_i = ax_{i-1} + b \bmod m \tag{1.1}$$

for $i \geq 1$. These are good enough for many purposes, e.g. in computer algebra, but not for cryptography. Suppose that Alice and Bob are part of a cryptographic network that uses Schnorr's identification scheme; see **??** for details. Each time Alice identifies herself to Bob, he sends her a random number r as part of the protocol. Now, suppose that Bob makes the mistake of taking the r's provided by his machine's `rand` command in `C`, which is based on a linear congruential generator. If Eve listens in to the traffic and observes several consecutive values of r, she can predict future values of r, as described below. Then the identification scheme is completely broken. The same would happen if a bank computer used such a generator to produce individual transaction numbers. After observing a few of them, an adversary would be able to determine the next ones.

In the generator (1.1), we have

$$\begin{aligned} x_i &\equiv ax_{i-1} + b \bmod m, \\ x_{i+1} &\equiv ax_i + b \bmod m. \end{aligned}$$

In order to eliminate a and b, we subtract and find

$$x_i - x_{i+1} \equiv a(x_{i-1} - x_i) \bmod m.$$

Similarly we get

$$x_{i+1} - x_{i+2} \equiv a(x_i - x_{i+1}) \bmod m.$$

Multiplying by appropriate quantities, we obtain

$$\begin{aligned} (x_i - x_{i+1})^2 &\equiv a(x_i - x_{i+1})(x_{i-1} - x_i) \\ &\equiv (x_{i+1} - x_{i+2})(x_{i-1} - x_i) \bmod m. \end{aligned}$$

Thus from 4 consecutive values x_{i-1}, x_i, x_{i+1}, x_{i+2} we get a multiple

$$m' = (x_i - x_{i+1})^2 - (x_{i+1} - x_{i+2})(x_{i-1}x_i)$$

of m. If the required gcds are 1, then we can also compute guesses a' and b' for a and b, respectively. We can then compute the next values $x_{i+3}, x_{i+4}, \ldots$ with these guesses and also observe the generator. Whenever a discrepancy occurs, we refine our guesses. One can show that after a polynomial number of steps one arrives at guesses which produce the same sequence as the original generator (although the actual values of a, b, and m may be different from the guessed ones). See Boyar (1989).

Such a generator is useless for cryptographic purposes, since we can **predict** the next value after having seen enough previous ones.

There are variations of these generators that compute internally $x_0, x_1, \ldots \operatorname{rem} m$ as before, but publish only the middle half (or the top half) of the bits of x_i. These generators are also insecure; they fall prey to a short vector attack.

One may also take just one bit, say $x_i \operatorname{rem} 2$. It is not known whether this yields pseudorandom bits.

The following **RSA generator** is supposed to be secure. We have $N = pq$ and e with $\gcd(e, \phi(N)) = 1$ as in the RSA system, and a random seed $x_0 \in \mathbb{Z}_N^\times$. We define $x_1, x_2, \ldots \in \mathbb{Z}_N^\times$ by $x_{i+1} = x_i^e$.

Nothing is known about how "random" this sequence is, nor whether there is a way of predicting x_i from previous values, nor whether such a prediction algorithm would also break the RSA system.

For the **Littlewood pseudorandom number generator**, we pick (small) integers $n < d$, which are publicly known, and an n-bit string x as (truly random) seed. We can also consider x as an integer in binary, and $2^{-n}x$ is the rational number with binary representation $0.x$.

Output is the sequence of the dth bits of the binary representation of $\log_2((x+i)2^{-n}) = \log(x+i) - n$ for $i = 0, 1, \ldots$. Thus with $n = 10$, $d = 14$ and key $x = 0110100111$, the first five pseudorandom bits are 11001, produced according to the entries (all in binary) of the following table.

i	$(x+i)2^{-n}$	$\log_2(x+i) - n$
0	0.0110100111	-1.010001101000011
1	0.0110101000	-1.010001011010011
2	0.0110101001	-1.010001001100100
3	0.0110101010	-1.010000111110101
4	0.0110101011	-1.010000110000110

Littlewood (1953), page 23, proposed this number generator, actually with $n = 5$ and $d = 7$ in its decimal version and for use in a key-addition encryption scheme. He says that "it is sufficiently obvious that a *single* message cannot be unscrambled".

This looks quite attractive, but is flawed. Wilson (1979) showed a first attack, and Stehlé (2004) gives an attack on the original system and even apparently stronger variants. His approach relies on modern cryptanalytic techniques including lattice basis reduction and Coppersmith's root finding method.

1.3. Distinguishers

We now want to formalize the notion that the elements generated by a pseudorandom generator should look "random". The idea is that no efficient algorithm should be able to distinguish between these elements and truly random ones.

Recall that a **probability distribution** on a finite set A is a function $p\colon A \longrightarrow \mathbb{R}_{\geq 0}$ with $\sum_{a\in A} p(a) = 1$. The **uniform probability distribution** u has $u(a) = 1/\#A$ for all $a \in A$. Together with p, a further function $f\colon A \longrightarrow B$ gives a **random variable** X on B (that is, with values in B), which assumes the value $b \in B$ with probability $\sum_{\substack{a\in A \\ f(a)=b}} p(a)$ which we abbreviate as

$$\mathrm{prob}(b \xleftarrow{\text{🐞}} X).$$

We then also have a probability distribution q on B, with $q(b) = \sum_{f(a)=b} p(a)$. If $B \subseteq \mathbb{R}$, then the **expected value** (or average, or mean) of X is

$$E(X) = \sum_{a\in A} p(a)X(a) = \sum_{b\in B} b \cdot \mathrm{prob}(b \xleftarrow{\text{🐞}} X).$$

Example 1.2. Rolling a fair die corresponds to the uniform distribution on $A = \{1,2,3,4,5,6\}$. If $X(a) = a^2$ for $a \in A$, then

$$\text{prob}(4 \xleftarrow{\text{🎲}} X) = \frac{1}{6},$$
$$E(X) = \frac{1}{6}(1+4+9+16+25+36) = \frac{91}{6}. \qquad \Diamond$$

We denote by $\mathbb{B}^n = \{0,1\}^n$ the Boolean n-cube. The uniform probability distribution u_n on $\mathbb{B}^n$ gives every string $x \in \mathbb{B}^n$ the same probability 2^{-n}, and the uniform random variable U_n takes on every value $x \in \mathbb{B}^n$ with probability 2^{-n}. From random variables X_1 on A_1, X_2 on A_2, ..., X_k on A_k we get the product variable $X = X_1 \times \cdots \times X_k$ on $A = A_1 \times \cdots \times A_k$, which by definition takes on a value $(a_1, \ldots, a_k) \in A$ with probability $\text{prob}(a_1 \xleftarrow{\text{🎲}} X_1) \cdots \text{prob}(a_k \xleftarrow{\text{🎲}} X_k)$. As an example, we have $U_n = U_1 \times \cdots \times U_1 = U_1^n$ on $\mathbb{B}^n = \mathbb{B}^1 \times \cdots \times \mathbb{B}^1$. If we have a random variable X on A and a mapping $f\colon A \longrightarrow B$, we get a random variable $f(X)$ on B which takes a value $b \in B$ with probability $\text{prob}(b \xleftarrow{\text{🎲}} f(X)) = \sum_{\substack{a \in \mathcal{A} \\ f(a)=b}} \text{prob}(a \xleftarrow{\text{🎲}} X)$. We will use this in the scenario where $\pi\colon C \times D \longrightarrow D$ is the projection and X a random variable on $C \times D$. Then $\pi(X)$ is called the marginal value of X on B.

Now suppose that we have a random variable X on $\mathbb{B}^n$, and a probabilistic algorithm $\mathcal{A}$ with n-bit inputs $x \in \mathbb{B}^n$ and one bit of output. This gives a random variable $\mathcal{A}(X)$ on $\mathbb{B} = \{0,1\}$ whose underlying distribution consists of X and the internal randomization in $\mathcal{A}$. For a bit $b \in \mathbb{B}$, we have

$$\text{prob}(b \xleftarrow{\text{🎲}} \mathcal{A}(X)) = \sum_{x \in \mathbb{B}^n} \text{prob}(x \xleftarrow{\text{🎲}} X) \cdot \text{prob}(b \xleftarrow{\text{🎲}} \mathcal{A}(x)).$$

The *expected value* of $\mathcal{A}$ on X is

$$E(\mathcal{A}(X)) = \sum_{b \in \mathbb{B}} b \cdot \text{prob}(b \xleftarrow{\text{🎲}} \mathcal{A}(X)) = \text{prob}(1 \xleftarrow{\text{🎲}} \mathcal{A}(X)).$$

For a deterministic algorithm, $\text{prob}(1 \longleftarrow \mathcal{A}(x)) = \mathcal{A}(x)$ is either 0 or 1.

Example 1.3. Let $\mathcal{A}$ be the deterministic algorithm which outputs $\mathcal{A}((x_1, \ldots, x_6)) = x_3$ on any input $(x_1, \ldots, x_6) \in \mathbb{B}^6$. Then for the uniform random variable U_6 on $\mathbb{B}^6$ we have

$$E(\mathcal{A}(U_6)) = \text{prob}(1 \xleftarrow{\text{🎲}} \mathcal{A}(U_6)) = \text{prob}(1 \xleftarrow{\text{🎲}} U_1) = \frac{1}{2}.$$

The U_1 here is the third component of $U_6 = U_1 \times U_1 \times U_1 \times U_1 \times U_1 \times U_1 = U_1^6$. $\Diamond$

DEFINITION 1.4. *If we have two random variables X and Y on $\mathbb{B}^n$, and an algorithm $\mathcal{A}$ as above, then*

$$\Delta_{\mathcal{A}}(X,Y) = |E(\mathcal{A}(X)) - E(\mathcal{A}(Y))|$$

is the ***distinguishing power*** *of $\mathcal{A}$ (between X and Y). If $\Delta_{\mathcal{A}}(X,Y) \geq \epsilon > 0$, then we say that $\mathcal{A}$ is an* ***ϵ-distinguisher*** *between X and Y. If such an $\mathcal{A}$ exists, we say that X and Y are* ***ϵ-distinguishable****.*

The pseudorandom generators that we define below cannot produce truly random values. But we want their values to be practically indistinguishable from random ones, namely ϵ-distinguishable with tiny ϵ (for any efficient $\mathcal{A}$).

EXAMPLE 1.5. Suppose that n is even and X takes only values with exactly $n/2$ ones: if $x \in \mathbb{B}^n$ and $\text{prob}(x \xleftarrow{\text{🎲}} X) > 0$, then $w(x) = \frac{n}{2}$. Here $w(x)$ is the Hamming weight of x, that is, the number of ones in x. Then the following deterministic algorithm $\mathcal{A}$ distinguishes between X and the uniform variable U_n on $\mathbb{B}^n$: $\mathcal{A}(x) = 1 \iff w(x) = \frac{n}{2}$. We have

$$E(\mathcal{A}(X)) = \text{prob}(1 \xleftarrow{\text{🎲}} \mathcal{A}(X)) = \text{prob}(\frac{n}{2} \xleftarrow{\text{🎲}} w(X)) = 1,$$

$$\begin{aligned} E_{\mathcal{A}}(U_n) &= \text{prob}(1 \xleftarrow{\text{🎲}} \mathcal{A}(U_n)) = \text{prob}\left(\frac{n}{2} \xleftarrow{\text{🎲}} w(U_n)\right) \\ &= 2^{-n} \cdot \#\{x \in \mathbb{B}^n \colon w(x) = \frac{n}{2}\} = 2^{-n}\binom{n}{n/2}. \end{aligned}$$

Stirling's formula (see Knuth 1973, 1.2.11.2) says that

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n (1 + \frac{1}{12n} + \frac{1}{288n^2} + \ldots).$$

Substituting this into the binomial coefficient and ignoring all minor terms, we find

$$E(\mathcal{A}(U_n)) \approx 2^{-n}\frac{\sqrt{2\pi n}(\frac{n}{e})^n}{\pi n(\frac{n}{2e})^n} = 2^{-n}\frac{2^n}{\sqrt{\pi n/2}} = \frac{1}{\sqrt{\pi n/2}}.$$

Thus

$$|E(\mathcal{A}(X)) - E(\mathcal{A}(U_n))| \approx 1 - \frac{1}{\sqrt{\pi n/2}} \geq \epsilon$$

for any ϵ with, say $\frac{\sqrt{\pi}-1}{\sqrt{\pi}} > 0.43 \geq \epsilon > 0$, as soon as $n \geq 2$. For $n = 100$, X and U_n are 0.9-distinguishable. ◊

DEFINITION 1.6. *A* **bit generator** *(or* **generator** *for short) is a function $f \colon \mathbb{B}^k \longrightarrow \mathbb{B}^n$ for some $k < n$. The corresponding random variable on $\mathbb{B}^n$ is $f(U_k)$.*

Example 1.7. We consider the generator

$$f\colon \mathbb{B}^3 \longrightarrow \mathbb{B}^6$$

given by the following table

x	$f(x)$
000	001101
001	001011
010	011010
011	010110
100	101100
101	100101
110	110100
111	110010

Each image word in $f(\mathbb{B}^3)$ has Hamming weight 3. We can easily distinguish the random variable $X = f(U_3)$ from U_6 by the distinguisher $\mathcal{A}$ from the previous example. Namely, on input $y \in \mathbb{B}^6$, $\mathcal{A}$ outputs 1 if $w(y) = 3$ and 0 otherwise. Then

$$E(\mathcal{A}(U_6)) = 2^{-6} \cdot \binom{6}{3} = \frac{5}{16}, \quad E(\mathcal{A}(X)) = 1, \quad \Delta_{\mathcal{A}}(X, U_6) = 1 - \frac{5}{16} = \frac{11}{16}.$$

Thus $\mathcal{A}$ is a $\frac{11}{16}$-distinguisher. In such a small example, one can find other distinguishing properties. The following illustrates a general construction that we will see a little later.

We can use the fourth bit of y to distinguish U_6 from $f(U_3)$, by comparing it to the value of 0 or 1 which occurs less often in the first three positions, called the minority. Thus for $y \in \mathbb{B}^6$

$$\mathcal{B}(y) = \begin{cases} 1 & \text{if } y_4 = \text{minority}(y_1, y_2, y_3), \\ 0 & \text{otherwise.} \end{cases}$$

Since both values for y_4 are equally likely in U_6 (and independent of y_1, y_2, y_3), we have $E(\mathcal{B}(U_6)) = 1/2$.

We now calculate $E(\mathcal{B}(X)) = \text{prob}(1 \xleftarrow{\text{R}} \mathcal{B}(X))$. There are eight values of y which occur as values of X, each with probability 1/8.

y	$\text{prob}(1 \xleftarrow{\text{R}} \mathcal{B}(y))$
001101	1
001011	0
011010	1
010110	1
101100	0
100101	1
110100	0
110010	1

Therefore $E(\mathcal{B}(X)) = 5/8$, $|E(\mathcal{B}(X)) - E(\mathcal{B}(U_6))| = \frac{5}{8} - \frac{1}{2} = \frac{1}{8}$, and $\mathcal{B}$ is an $\frac{1}{8}$-distinguisher between U_6 and X. This is quite ok, but not as good as the distinguisher $\mathcal{A}$ from above. ◊

We now want to define pseudorandom generators. To this end, we consider a family $g = (g_k)_{k\in\mathbb{N}}$ of Boolean functions g_k with

$$g_k \colon \mathbb{B}^k \longrightarrow \mathbb{B}^{n(k)},$$

where $n(k) > k$ for all $k \in \mathbb{N}$. Thus each family member g_k is a generator from $\mathbb{B}^k$ to $\mathbb{B}^{n(k)}$. On input a uniformly random $x \in \mathbb{B}^k$, it produces a (much) longer output $y = g_k(x) \in \mathbb{B}^{n(k)}$ which should look "random". For any $k \in \mathbb{N}$, the random variable $X = g_k(U_k)$ assumes the value $y \in \mathbb{B}^{n(k)}$ with probability

$$\mathrm{prob}(y \xleftarrow{\text{🐞}} X) = 2^{-k} \cdot \#\{x \in \mathbb{B}^k \colon g_k(x) = y\}.$$

At most 2^k many y's have positive probability. Since $k < n(k)$, only "very few" values y actually occur, and X is "very far" from the uniform random variable. But still it might be quite difficult to detect this difference. However, it is always possible to detect some difference. For example, we may choose some $y_0 \in g_k(\mathbb{B}^k)$, so that $\mathrm{prob}(y \xleftarrow{\text{🐞}} X) \geq 2^{-k}$, and take an algorithm which computes the function $\mathcal{A}(y) = (y = y_0) \in \mathbb{B}$. Then

$$E(\mathcal{A}(X)) \geq 2^{-k} \gg 2^{-n(k)} = E(\mathcal{A}(U_{n(k)})).$$

Thus $\mathcal{A}$ distinguishes somewhat between the two distributions, but its distinguishing power $2^{-k} - 2^{-n(k)} \approx 2^{-k}$ is exponentially small in k. We can't be bothered with such tiny (and unavoidable) differences, and call them "negligible". We are even a bit more generous and call any function negligible if it is smaller than any inverse polynomial.

DEFINITION 1.8. *A function $t \colon \mathbb{N} \longrightarrow \mathbb{R}$ is negligible if for all $e \geq 1$ there exists k_0 such that for all $k \geq k_0$ we have*

$$|t(k)| \leq k^{-e}.$$ □

For example, t with $t(k) = k^{-\log k}$ is negligible, but not exponentially small like 2^{-k}.

Now the generators we consider have to be efficient, but there must not exist efficient distinguishers. This gives the following notion.

DEFINITION 1.9. *A family $g = (g_k)_{k\in\mathbb{N}}$ as above is a **pseudorandom generator** if*

- *it can be implemented in polynomial time $k^{O(1)}$,*

- *for all probabilistic polynomial-time algorithms $\mathcal{A}$, the distinguishing power $\Delta_{\mathcal{A}}(g_k(U_k), U_{n(k)})$ is a negligible function of k.*

Such a generator can be used in any efficient (polynomial time) algorithm that requires truly random bits. Namely, if it was ever observed that the algorithm did not perform as predicted for truly random inputs, then the algorithm would distinguish between U_n and the pseudorandom generator; but this is not possible.

This is, quite appropriately, an "asymptotic" notion. It does not depend on the first hundred (or hundred million) g_k's, only on their eventual behavior. We have seen many cryptosystems, such as RSA, which can be implemented for arbitrary key lengths. However, there are also cryptosystems like Rijndael which have fixed input lengths and are not part of an infinite family.

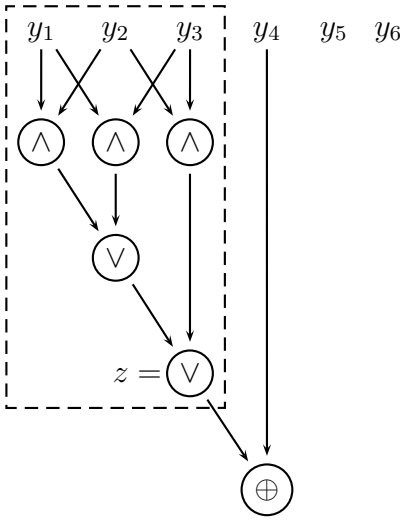
We now want to define a "finite" version of this notion. It should be applicable to individual Boolean functions such as $g\colon \mathbb{B}^3 \longrightarrow \mathbb{B}^6$ from Example 1.7. In Definition 1.9 we did not specify the notion of "algorithm". The reader should think, as usual, of Turing machines or appropriate random access machines. For our finite version, Boolean circuits are appropriate. They have (one-bit) input gates, and NOT, AND, OR, and XOR gates. The time that such a circuit takes is the number of gates in it (except for input gates). It is usually called the size of the circuit. Then "algorithm" may also be taken to mean "family of Boolean circuits". There is a technical problem with "uniformity" here; see the Notes.

Definition 1.10. *Let $k < n$ and s be integers, $\epsilon \geq 0$ real, and $f\colon \mathbb{B}^k \longrightarrow \mathbb{B}^n$ a generator. A probabilistic Boolean circuit $\mathcal{C}$ of size s and with distinguishing power $\Delta_{\mathcal{C}}(f(U_k), U_n) \geq \epsilon$ is called an (ϵ, s)-**distinguisher** between $f(U_k)$ and U_n. The function f is called an (ϵ, s)-**resilient pseudorandom generator** if no such $\mathcal{C}$ exists.*

Example 1.7 continued. We take $f\colon \mathbb{B}^3 \longrightarrow \mathbb{B}^6$ as above, and implement the two distinguishers as Boolean circuits.

We start with the second one, and first compute $z = (w(y_1, y_2, y_3) \geq 2)$ in the

circuit within dashed lines, and then output $z \oplus y_4$:

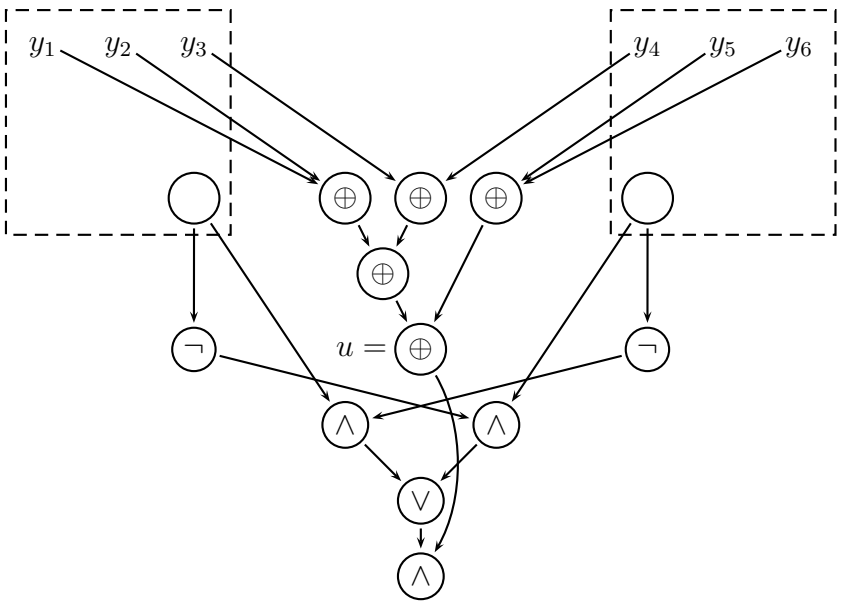


This circuit has 6 gates and is therefore a $(\frac{1}{8}, 6)$-distinguisher between $f(U_3)$ and U_6. Thus f is not a $(\frac{1}{8}, 6)$-resilient pseudorandom generator.

To implement the first distinguisher "$w(y) = 3$" as a circuit, we first compute $u = \bigoplus_{1 \le i \le 6} y_i$. Thus $u = 1$ if and only if $w(y)$ is 1, 3, or 5. If we add the condition that

$$(w(h_1) \ge 2 \text{ and } w(h_2) \le 1) \text{ or } (w(h_1) \le 1 \text{ and } w(h_2) \ge 2),$$

where $h_1 = (y_1, y_2, y_3)$ and $h_2 = (y_4, y_5, y_6)$ are the two halves of y, then we compute precisely the Boolean function "$w(y) = 3$". We re-use the 5-gate circuit from above twice in dashed lines and get the following circuit.



This circuit has $2 \cdot 5 + 11 = 21$ gates, so that f is also not $\left(\frac{11}{16}, 21\right)$-resilient. $\Diamond$

1.4. Predictors

We consider probabilistic algorithms that try to predict the next value x_i of a sequence from the previous bits $x_1, \ldots, x_{i-1}$. A good predictor can also be used as a distinguisher. The main result of this section is the converse: from any good distinguisher one can build a reasonably good predictor. The proof introduces an important tool: "hybrid" distributions which "interpolate" between two given distributions.

For two random variables X and Y on the same set B, we write

$$\mathrm{prob}(Y \xleftarrow{\text{🐞}} X) = \sum_{x \in B} \mathrm{prob}(b \xleftarrow{\text{🐞}} Y) \cdot \mathrm{prob}(b \xleftarrow{\text{🐞}} X)$$

for the probability that both produce the same value. This generalizes the notion $x \longleftarrow X$ in a natural way.

When X is a random variable on $\mathbb{B}^n$ and $i \leq n$, we want to consider the *ith successor bit* under X, namely the following one-bit random variable $X_i(y)$, for any $y \in \mathbb{B}^{i-1}$. Its value is 0 with the same probability as the one with which strings $(y, 0, z)$ occur under X, for any $z \in \mathbb{B}^{n-i}$, and 1 with the probability of $(y, 1, z)$ occurring under X. More precisely, for any $j \leq n$ and $w \in \mathbb{B}^j$, we let

(1.11)

$$\begin{aligned} p(w, *) &= \mathrm{prob}(w \longleftarrow (X_1, \ldots, X_j)) \\ &= \mathrm{prob}(\{w\} \times U_{n-j} \xleftarrow{\text{🐞}} X) = 2^{-n+j} \cdot \sum_{z \in \mathbb{B}^{n-j}} \mathrm{prob}((w, z) \xleftarrow{\text{🐞}} X) \end{aligned}$$

be the probability of w as an initial segment under X. Then for $b \in \{0, 1\}$, we set $\mathrm{prob}(b \xleftarrow{\text{🐞}} X_i(y)) = 1/2$ if $p(y, *) = 0$, and otherwise

$$\mathrm{prob}(b \xleftarrow{\text{🐞}} X_i(y)) = p((y, b), *)/p(y, *).$$

Definition 1.12. *Let $1 \leq i \leq n$ be integers.*

(i) *A* **predictor** *for the ith bit is a probabilistic algorithm with inputs from $\mathbb{B}^{i-1}$ and output in $\mathbb{B}$.*

(ii) *Let X be a random variable on $\mathbb{B}^n$, $(X_1, \ldots, X_{i-1})$ the corresponding variable on $\mathbb{B}^{i-1}$, and $\mathcal{P}$ a predictor for the ith bit. Then the* **success rate** *$\sigma_{\mathcal{P}}(X)$ of $\mathcal{P}$ on X is*

$$\sigma_{\mathcal{P}}(X) = \sum_{y \in \mathbb{B}^{i-1}} p(y, *) \cdot \mathrm{prob}(\mathcal{P}(y) \xleftarrow{\text{🐞}} X_i(y)).$$

Its **prediction power** *is* $\sigma_{\mathcal{P}}(X) - 1/2$. *If* $\sigma_{\mathcal{P}}(X) \geq \epsilon + 1/2$, *then* $\mathcal{P}$ *is an* ϵ-**predictor** *for* X.

(iii) *A family* $(X_k)_{k\in\mathbb{N}}$ *of random variables* X_k *on* $\mathbb{B}^{n(k)}$ *is* ***computationally unpredictable*** *if for any function* i_k *with* $i_k \leq n(k)$, *any probabilistic polynomial-time predictor for the* i_k*th bit of* X_k *has negligible prediction power. Here, the predictor is an algorithm which takes as input* k *(encoded in unary) and* $y \in \mathbb{B}^{i_k-1}$.

Thus $0 \leq \sigma_{\mathcal{P}} \leq 1$. A very simple (and rather useless) predictor is to output a uniformly random bit, independent of the input. It has success rate $1/2$ for any X.

If $\sigma_{\mathcal{P}}(X) \leq 1/2$, then flipping the output bit of $\mathcal{P}$ produces a predictor $\mathcal{P}'$ with $\sigma_{\mathcal{P}'}(X) = 1 - \sigma_{\mathcal{P}}(X) \geq 1/2$. For a "good" predictor $\mathcal{P}$, the goal is to make its prediction power $\sigma_{\mathcal{P}}(X) - 1/2$ as large as possible.

As in the previous section, we also have a finite version of this asymptotic notion. Now X is a random variable on $\mathbb{B}^n$, $1 \leq i \leq n$, and $\mathcal{P}$ is a probabilistic circuit of size s with $i-1$ inputs and one output, and is called an (ϵ, s)-predictor if $\sigma_{\mathcal{P}}(X) \geq \epsilon + 1/2$. We say that X is (ϵ, s)-**unpredictable** if no such i and $\mathcal{P}$ exist.

EXAMPLE 1.7 CONTINUED. We take $X = f(U_3)$ on $\mathbb{B}^6$. Since 0 and 1 occur equally often in each $f(x)$, we consider the "minority bit predictor" $\mathcal{M}_i$ for the ith bit. It predicts the bit that occurs less frequently in the history; if both occur equally often, it predicts 0 or 1, each with probability $1/2$.

Clearly this algorithm predicts the sixth bit always correctly: $\sigma_{\mathcal{M}_6}(X) = 1$, and $\mathcal{M}_6$ is a $\frac{1}{2}$-predictor. We now compute its quality as a predictor for the fourth bit:

$$\sigma_{\mathcal{M}_4}(X) = \sum_{y\in\mathbb{B}^3} p(y,*) \cdot \mathrm{prob}(X_4(y) \stackrel{\text{🎲}}{\longleftarrow} \mathcal{M}_4(y)).$$

We only have six $y \in \mathbb{B}^3$ with $p(y,*) > 0$.

y	$p(y,*)$	$X_4(y)$	$\mathcal{M}_4(y)$	$\mathrm{prob}(X_4(y) \stackrel{\text{🎲}}{\longleftarrow} \mathcal{M}_4(y))$
001	1/4	0, 1	1	1/2
011	1/8	0	0	1
010	1/8	1	1	1
101	1/8	1	0	0
100	1/8	1	1	1
110	1/4	0, 1	0	1/2

Therefore the success rate is

$$\frac{1}{4}\cdot\frac{1}{2} + \frac{1}{8}\cdot 1 + \frac{1}{8}\cdot 1 + \frac{1}{8}\cdot 0 + \frac{1}{8}\cdot 1 + \frac{1}{4}\cdot\frac{1}{2} = \frac{5}{8} > \frac{1}{2},$$

and $\mathcal{M}_4$ is a $\frac{1}{8}$-predictor. ◊

It is clear that a predictor can also serve as a distinguisher. Suppose that X is a random variable on $\mathbb{B}^n, 1 \leq i \leq \ell$, and $\mathcal{P}$ is an ϵ-predictor for the ith bit under X. Then we consider the following method for obtaining an algorithm $\mathcal{A}$.

ALGORITHM 1.13. Distinguisher $\mathcal{A}$ from predictor.

Input: $y \in \mathbb{B}^n$, and i and $\mathcal{P}$ as above.
Output: 0 or 1.

1. Compute $z = \mathcal{P}(y_1, \ldots, y_{i-1})$.
2. $\mathcal{A}$ outputs 1 if $y_i = z$ and 0 otherwise.

THEOREM 1.14. *If $\mathcal{P}$ is an (ϵ, s)-predictor for the ith bit under X, then $\mathcal{A}$ is an $(\epsilon, s+5)$-distinguisher between X and U_n.*

PROOF. The output of $\mathcal{A}$ equals $(y_i \wedge z) \vee (\neg y_i \wedge \neg z)$, which is independent of the values of $y_{i+1}, \ldots, y_n$, and $\mathcal{A}$ has size $s+5$. We have

$$\begin{aligned} E(\mathcal{A}(X)) &= \text{prob}(1 \xleftarrow{\bullet\bullet} \mathcal{A}(X)) \\ &= \text{prob}(X_i \xleftarrow{\bullet\bullet} \mathcal{P}(X_1, \ldots, X_{i-1})) \\ &= \sigma_{\mathcal{P}}(X) \geq \frac{1}{2} + \epsilon. \end{aligned}$$

On the other hand, whatever $\mathcal{P}$ computes, the probability that its output $\mathcal{P}(U_{i-1})$ equals a uniform random bit from U_1 is $1/2$. Thus the distinguishing power of $\mathcal{A}$ between X and U_n is

$$|E(\mathcal{A}(X)) - E(\mathcal{A}(U_n))| \geq \frac{1}{2} + \epsilon - \frac{1}{2} = \epsilon. \qquad \square$$

It is quite surprising that also from any good distinguisher one can obtain a reasonably good predictor. This strong result is due to Yao (1982). Thus distinguishers and predictors are essentially equivalent. In other words, predicting the next bit is a "universal test" for pseudorandomness.

THEOREM 1.15. *Let X be a random variable on $\mathbb{B}^n$, and $\mathcal{A}$ an (ϵ, s)-distinguisher between X and U_n. Then there exists an i with $1 \leq i \leq n$ and an $(\frac{\epsilon}{n}, s+1)$-predictor for the ith bit under X.*

PROOF. For $0 \leq i \leq n$, we let $\pi_i \colon \mathbb{B}^n \longrightarrow \mathbb{B}^i$ be the projection onto the first i coordinates, and

$$Y_i = \pi_i(X) \times U_{n-i}.$$

Thus Y_i is the random variable on $\mathbb{B}^n$ where the first i bits are generated according to X, and the other $n-i$ according to the uniform distribution. These

Y_i are "hybrid" variables, partly made up from X and partly from the uniform distribution, and they "interpolate" between the extremes $Y_n = X$ and $Y_0 = U_n$. For $1 \leq i \leq n$, let $e_i = E(\mathcal{A}(Y_i))$. The distinguishing power of $\mathcal{A}$ is at least ϵ, so that $|e_n - e_0| \geq \epsilon$. By flipping the output bit of $\mathcal{A}$ if necessary, we may assume that $e_0 - e_n \geq \epsilon$. Intuitively, this means that an output 1 of $\mathcal{A}$ indicates that the input is likely to come from U_n, and an output 0 that it comes from Y. Then we have

$$\epsilon \leq e_0 - e_n = \sum_{1 \leq i \leq n} (e_i - 1 - e_i) \leq n \cdot \max_{1 \leq i \leq n} e_{i-1} - e_i.$$

Hence the maximum is at least ϵ/n, and there exists some $i \leq n$ with $e_i - e_{i-1} \geq \epsilon/n$. We now choose such an i.

We now construct a predictor $\mathcal{P}$ for the ith bit under X.

ALGORITHM 1.16. Predictor $\mathcal{P}$.

Input: $y \in \mathbb{B}^{i-1}$.
Output: 0 or 1.

1. Choose $y_i, \ldots, y_n \in \mathbb{B}$ uniformly at random.
2. $y^* \longleftarrow (y, y_i, \ldots, y_n)$. [Thus $y^* \in \mathbb{B}^n$.]
3. $z \longleftarrow \mathcal{A}(y^*)$.
4. Output $y_i \oplus z$.

The intuition why this should work is as follows. If $\mathcal{A}$ outputs $z = 1$, then probably (y, y_i) comes from Y_i, since $e_i > e_{i-1}$, and if $z = 0$, then (y, y_i) is more likely to come from Y_{i-1}. Now Y_{i-1} and Y_i differ only in the ith place, where Y_i is derived from X while Y_{i-1} has a uniformly random bit. Thus we take $z = 1$ as an indication that y_i comes from X, and indeed output $y_i = y_i \oplus 1 \oplus 1 = y_i \oplus z$ as the prediction. But $z = 0$ indicates that y_i is presumably from U_1, and that the opposite bit $y_i \oplus 1 \oplus 1 = y_i \oplus z$ is a better prediction for the ith bit under X than y_i itself is.

The success rate of $\mathcal{P}$ on X is

$$\begin{aligned}
\sigma_{\mathcal{P}}(X) &= \sum_{y\in\mathbb{B}^{i-1}} p(y,*)\cdot \operatorname{prob}(\mathcal{P}(y) \xleftarrow{\text{🐞}} X_i(y)) \\
&= \sum_{\substack{y\in\mathbb{B}^{i-1}\\ y_i\in\mathbb{B}}} p(y,*)\cdot \operatorname{prob}(y_i \xleftarrow{\text{🐞}} U_1 \text{ and } y_i\oplus \mathcal{A}((y,y_i)\times U_{n-i})\oplus 1 \xleftarrow{\text{🐞}} X_i(y)) \\
&= \sum_{\substack{y\in\mathbb{B}^{i-1}\\ y_i\in\mathbb{B}}} p(y,*)\cdot \big[\operatorname{prob}(0 \xleftarrow{\text{🐞}} \mathcal{A}((y,y_i)\times U_{n-i}), y_i\oplus 1 \xleftarrow{\text{🐞}} X_i(y), \text{and } y_i \xleftarrow{\text{🐞}} U_1) \\
&\qquad\qquad + \operatorname{prob}(1 \xleftarrow{\text{🐞}} \mathcal{A}((y,y_i)\times U_{n-i}), y_i \xleftarrow{\text{🐞}} X_i(y), \text{and } y_i \xleftarrow{\text{🐞}} U_1)\big] \\
&= \sum_{\substack{y\in\mathbb{B}^{i-1}\\ y_i\in\mathbb{B}}} p(y,*)\cdot \big[\operatorname{prob}(0 \xleftarrow{\text{🐞}} \mathcal{A}((y,y_i)\times U_{n-i})) \\
&\qquad\qquad - \operatorname{prob}(0 \xleftarrow{\text{🐞}} \mathcal{A}((y,y_i)\times U_{n-i}), y_i \xleftarrow{\text{🐞}} U_1, \text{and } y_i \xleftarrow{\text{🐞}} X_i(y)) \\
&\qquad\qquad + \operatorname{prob}(1 \xleftarrow{\text{🐞}} \mathcal{A}((y,y_i)\times U_{n-i}), y_i \xleftarrow{\text{🐞}} X_i(y), \text{and } y_i \xleftarrow{\text{🐞}} U_1)\big] \\
&= \sum_{\substack{y\in\mathbb{B}^{i-1}\\ y_i\in\mathbb{B}}} p(y,*)\cdot \big[\operatorname{prob}(0 \xleftarrow{\text{🐞}} \mathcal{A}((y,y_i)\times U_{n-i})) \\
&\qquad\qquad - \operatorname{prob}(0 \xleftarrow{\text{🐞}} \mathcal{A}((y,X_i(y)))\times U_{n-i})\cdot\frac{1}{2} \\
&\qquad\qquad + \operatorname{prob}(1 \xleftarrow{\text{🐞}} \mathcal{A}((y,X_i(y))\times U_{n-i}))\cdot\frac{1}{2}\big] \\
&= \operatorname{prob}(0 \xleftarrow{\text{🐞}} \mathcal{A}(Y_{i-1})) - \frac{1}{2}\operatorname{prob}(0 \xleftarrow{\text{🐞}} \mathcal{A}(Y_i)) + \frac{1}{2}\operatorname{prob}(1 \xleftarrow{\text{🐞}} \mathcal{A}(Y_i)) \\
&= 1 - e_{i-1} - \frac{1-e_i}{2} + \frac{e_i}{2} = \frac{1}{2} + e_i - e_{i-1} \geq \frac{1}{2} + \frac{\epsilon}{n}
\end{aligned}$$

Some explanations may be useful. In the second equation, we sum over the two possible values for y_i chosen in step 1 of $\mathcal{P}$. Now $\mathcal{P}(y) = y_i \oplus z$ and $X_i(y)$ take the same value in two cases:

$$\begin{array}{lll} z=0 & \text{and} & y_i\oplus 1 = X_i(y), \text{or} \\ z=1 & \text{and} & y_i = X_i(y). \end{array}$$

These two cases lead to the third equation. In the fourth equation, the first summand of the previous expression is split into the probability that 0 occurs as value of $\mathcal{A}((y,y_i)\times U_{n-1})$, without regard to $X_i(y)$, minus the probability that y_i occurs as value of $X_i(y)$—this is the complement to the condition $y_i\oplus 1 \longleftarrow X_i(y)$. For the fifth equation, we use the fact that the event $y_i \longleftarrow U_1$ is independent of the other events, for both possible choices of y_i, and occurs with probability $1/2$. □

Corollary 1.17. *(i) Suppose that each bit of the generator $f\colon \mathbb{B}^k \longrightarrow \mathbb{B}^n$ is (ϵ, s)-unpredictable. Then $f(U_k)$ is $(\epsilon, s+1)$-indistinguishable from U_n.*

(ii) Suppose that the generator $g = (g_k)_{k\in\mathbb{N}}$ is such that each bit (that is, for each sequence $(i_k)_{k\in\mathbb{N}}$ the i_kth bit of g_k) is computationally unpredictable. Then g is a pseudorandom generator.

In other words, bit prediction is a universal test for pseudorandomness.

EXAMPLE 1.7 CONTINUED. We apply Yao's construction to $X = f(U_3)$ on $\mathbb{B}^6$ and $\mathcal{A}$ as above, with $\mathcal{A}(y) = 1$ if and only if $w(y) = 3$. We have seen above that $E_{\mathcal{A}}(X) = 1$ and $E_{\mathcal{A}}(U_6) = \frac{5}{16}$, and now have to calculate the expected value of $\mathcal{A}$ on the hybrid distributions $Y_i = \pi_i(X) \times U_{6-i}$. These distributions are depicted in Figure 1.1. At the back, we have $f(U_3) = Y_6$, a rugged landscape with eight peaks and valleys at zero level. The montains get eroded as we move forward, to Y_5, Y_4, and Y_3, until we arrive at $Y_2 = Y_1 = Y_0$, a uniformly flat seascape.

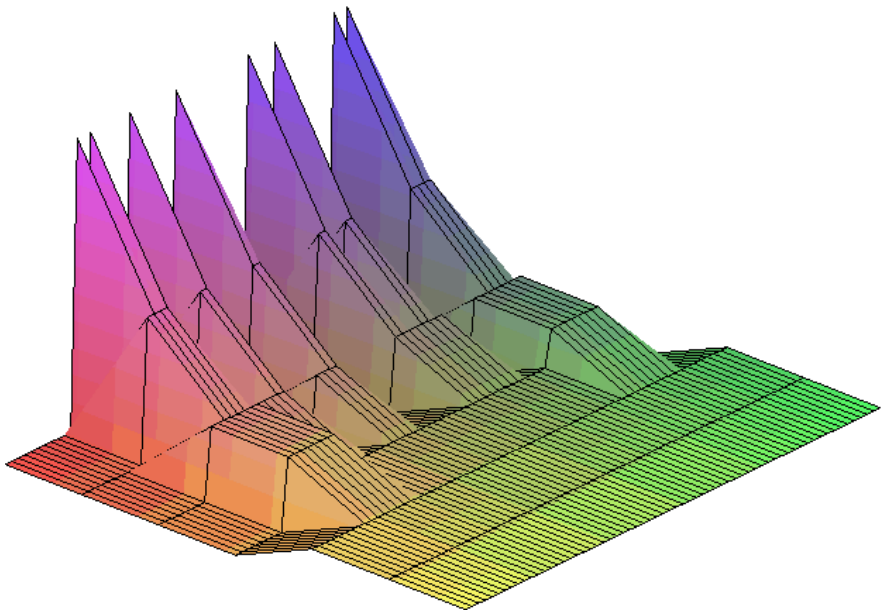


Figure 1.1: The hybrid distributions from Yao's construction.

For $0 \le i \le 6$ and any $y \in f(\mathbb{B}^3)$, we denote by

$$c_i(y) = \binom{6-i}{3-w(y_1,\ldots,y_i)}$$

the number of extensions $(z_{i+1},\ldots,z_6)$ of $(y_1,\ldots,y_i)$ that lead to total Hamming weight 3, that is, with $w(y_1,\ldots,y_i,z_{i+1},\ldots,z_6) = 3$. Then

$$\begin{aligned}\big((\pi_i(x) \times U_{6-i})(y)\big) &= \operatorname{prob}\pi_i(0)(y_1,\ldots,y_i)\cdot 2^{i-6}\\ &= 2^{i-6}\cdot 2^{-3}\#\{x\in\mathbb{B}^3\colon (f(x)_1,\ldots,f(x)_i) = (y_1,\ldots,y_i)\},\end{aligned}$$

$$\begin{aligned} e_i = E(\mathcal{A}(Y_i)) &= \text{prob}(1 \xleftarrow{\text{🐞}} \mathcal{A}(y_i)) = \text{prob}(3 \xleftarrow{\text{🐞}} w(Y_i)) \\ &= \text{prob}(3 \xleftarrow{\text{🐞}} w((X_1, \ldots, X_i) \times U_{6-i})) \\ &= 2^{-3} \cdot 2^{-(6-i)} \cdot \#\{(x,y) \in \mathbb{B}^3 \times \mathbb{B}^{6-i} : \\ &\quad w(f(x)_1, \ldots, f(x)_i, y_{i+1}, \ldots, y_6) = 3\} \\ &= 2^{i-9} \sum_{x \in \mathbb{B}^3} c_i(f(x)). \end{aligned}$$

The following two tables give the values of the $c_i(f(x))$ and e_i.

x	$f(x)$	c_0	c_1	c_2	c_3	c_4	c_5	c_6
000	001101	20	10	4	3	2	1	1
001	001011	20	10	4	3	1	1	1
010	011010	20	10	6	3	2	1	1
011	010110	20	10	6	3	2	1	1
100	101100	20	10	6	3	1	1	1
101	100101	20	10	6	3	2	1	1
110	110100	20	10	4	3	1	1	1
111	110010	20	10	4	3	2	1	1

i	0	1	2	3	4	5	6
e_i	$\frac{5}{16}$	$\frac{5}{16}$	$\frac{5}{16}$	$\frac{3}{8}$	$\frac{13}{32}$	$\frac{1}{2}$	1
$e_i - e_{i-1}$		0	0	$\frac{2}{32}$	$\frac{1}{32}$	$\frac{3}{32}$	$\frac{16}{32}$

We check that the sum of these differences equals $e_6 - e_0 = 11/16$. The largest of the differences is $e_6 - e_5 = 1/2$. Intuitively, it is clear that this points to the minority bit predictor $\mathcal{M}_6$ for the last bit, from page 13, with success probability 1 and 1/2 on $f(U_3)$ and U_6, respectively. But now we want to trace the general construction. It yields the following predictor $\mathcal{P}$ for the sixth bit under $X = f(U_3)$. We first change $\mathcal{A}$ to $\mathcal{A}'$ by flipping its output bit, so that now $e_5 - e_6 = 1/2 > 0$. On input $y \in \mathbb{B}^5$, $\mathcal{P}$ chooses $y_6 \in \mathbb{B}$ uniformly at random, calculates

$$z = \begin{cases} 0 & \text{if } w(y, y_6) = 3, \\ 1 & \text{otherwise,} \end{cases}$$

and outputs $y_6 \oplus z$. We claim that $\mathcal{P}(y) = \mathcal{M}_6(y)$ for any $y \in \pi_5(X)$. This follows from the following table of the values $(z, \mathcal{P}(y))$, where the second entry indeed always equals $\mathcal{M}_6(y)$:

		$w(y)$	
		2	3
y_6	0	(1, 1)	(0, 0)
	1	(0, 1)	(1, 0)

In particular, we have $\sigma_{\mathcal{P}}(X) = \sigma_{\mathcal{M}_6}(X) = 1$, and $\mathcal{P}$ is also an 1/2-predictor for X. However, $\mathcal{P}$ is not equal to $\mathcal{M}_6$, since on input $y = (0,0,0,0,0)$, say, we have $\mathrm{prob}(1 \xleftarrow{\$} \mathcal{P}(y)) = 1/2$ and $\mathrm{prob}(1 \xleftarrow{\$} \mathcal{M}_6(y)) = 1$. ◊

S

1.5. From short to long generators

If we have an (ϵ, s)-generator $f\colon \mathbb{B}^k \to \mathbb{B}^n$ and $k < \ell \le n$, then by composing with the projection $\pi\colon \mathbb{B}^n \to \mathbb{B}^\ell$ to the first ℓ bits we get a function $g = \pi \circ f\colon \mathbb{B}^k \longrightarrow \mathbb{B}^\ell$. It is also an (ϵ, s)-generator, since any algorithm that distinguishes $g(U_k)$ from U_ℓ can also distinguish $f(U_k)$ from U_n, with the same size and quality.

Thus it is easy as pie to shorten generators. Can we also make them longer? This is less obvious, but this section is devoted to showing that this can indeed be achieved.

We take as our starting point a generator that is as short as possible, namely $f\colon \mathbb{B}^k \to \mathbb{B}^{k+1}$, and construct from it a generator $g\colon \mathbb{B}^k \to \mathbb{B}^n$ for any $n > k$. To do this, we apply f iteratively to k-bit strings, save the first bit, and apply f again to the remaining k bits.

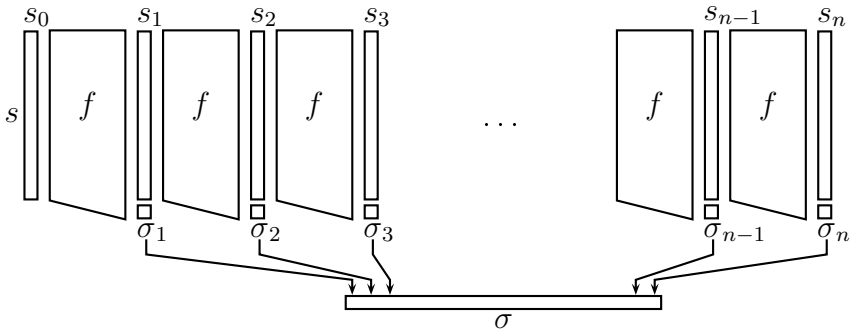


Figure 1.2: Long generator g

We define functions f_i that leave the first $i-1$ bits unchanged, for $i \ge 1$, and apply f to the last k bits:

$$f_i = \mathrm{id}_{\mathbb{B}_{i-1}} \times f\colon \begin{array}{rcl} \mathbb{B}^{k+i-1} & \longrightarrow & \mathbb{B}^{k+i}, , \\ (x_1, \ldots, x_{k+i-1}) & \longmapsto & (x_1, \ldots, x_{i-1}, f(x_i, \ldots, x_{k+i-1})). \end{array}$$

We let $g_i = f_i \circ \cdots \circ f_2 \circ f_1\colon \mathbb{B}^k \to \mathbb{B}^{k+i}$ be the composition of i of these maps. Thus $g_1 = f_1 = f$. We also set $g_0 = \mathrm{id}_{\mathbb{B}^k}$.

THEOREM 1.18. *Let $f\colon \mathbb{B}^k \to \mathbb{B}^{k+1}$ be an (ϵ, s)-resilient generator, that can be computed by a circuit of size t, let $\ell \ge 1$, and $g = g_\ell\colon \mathbb{B}^k \to \mathbb{B}^{k+\ell}$ as above. Then g is an $(\ell\epsilon, s - \ell t)$-resilient generator, and can be computed with ℓ applications of f.*

The idea of the proof is to turn a distinguisher $\mathcal{A}$ between $g(U_k)$ and $U_{k+\ell}$ into a distinguisher $\mathcal{B}$ between $f(U_k)$ and U_{k+1}. We consider hybrid random variables $Y_0, Y_1 \ldots, Y_\ell$ which interpolate between $Y_\ell = g(U_k)$ and $Y_0 = U_{k+\ell}$. If $\mathcal{A}$ distinguishes well between Y_0 and Y_ℓ, then it also distinguishes well between Y_i and Y_{i+1} for some i. But these adjacent distributions Y_i and Y_{i+1} are essentially like $f(U_k)$ and U_{k+1}, so that we can also distinguish between these two. By assumption, this can only be done with bad quality, so that also the quality of the initial $\mathcal{A}$ is bad.

Proof. For $0 \le i < \ell$, we first define an auxiliary function $h_i = \pi_1 \times g_{i-1} \colon \mathbb{B}^{k+1} \longrightarrow \mathbb{B}^{k+i}$, so that

$$h_i(x_1, x_2, \ldots, x_{k+1}) = (x_1, g_{i-1}(x_2, \ldots, x_{k+1}))$$

for all $(x_1, \ldots, x_{k+1}) \in \mathbb{B}^{k+1}$. The important property connecting f, the g's, and the h's is that for $i \ge 1$ we have $h_i \circ f = g_i$, and hence

$$h_i(f(U_k)) = g_i(U_k), \quad h_i(U_{k+1}) = (U_1, g_{i-1}(U_k)). \tag{1.19}$$

Here, and in similar situations later, the uniform distributions like U_1 and U_k are taken independently.

Now we let $\mathcal{A}$ be an (δ, s)-distinguisher between $g(U_k)$ and $U_{k+\ell}$, that is, an algorithm using time s and so that

$$E(\mathcal{A}(g(U_k))) - E(\mathcal{A}(U_{k+\ell})) \ge \delta.$$

(If the left hand quantity is at most $-\delta$, then we flip the output bit of $\mathcal{A}$ to obtain the above inequality.) We will show that the following Algorithm 1.20 distinguishes between $f(U_k)$ and U_{k+1}.

Algorithm 1.20. From long to short distinguishers.
Input: $x \in \mathbb{B}^{k+1}$.
Output: 1 or 0.

1. Choose $i \in_R \{1, \ldots, \ell\}$ uniformly at random.
2. Choose $y \xleftarrow{\text{🎲}} U_{\ell-i}$.
3. Execute $\mathcal{A}$ on input $(y, h_i(x)) \in \mathbb{B}^{\ell+k}$ and return its output.

For any input $x \in \mathbb{B}^{k+1}$ to $\mathcal{B}$, we have

$$\text{prob}(1 \xleftarrow{\text{🎲}} \mathcal{B}(x)) = \frac{1}{\ell} \sum_{1 \le i \le \ell} \text{prob}(1 \xleftarrow{\text{🎲}} \mathcal{A}(U_{\ell-i}, h_i(x))). \tag{1.21}$$

We consider for $0 \le i \le \ell$ the hybrid random variable

$$Y_i = U_{\ell-i} \times g_i(U_k)$$

with values in $\mathbb{B}^{k+\ell}$. Thus $Y_\ell = g_\ell(U_k)$ and $Y_0 = U_{k+\ell}$ are the two random variables between which $\mathcal{A}$ distinguishes. For any $i \le \ell$ we have

$$\begin{aligned} Y_i &= U_{\ell-i} \times g_i(U_k) = U_{\ell-i} \times h_i(f(U_k)) \text{ if } i \ge 0, \\ Y_{i-1} &= U_{\ell-i+1} \times g_{i-1}(U_k) = U_{\ell-i} \times U_1 \times g_{i-1}(U_k) = U_{\ell-i} \times h_i(U_{k+1}) \text{ if } i \ge 1. \end{aligned}$$

Now let $\alpha_i = \text{prob}(1 \xleftarrow{\text{🎲}} \mathcal{A}(Y_i))$ for $0 \le i \le \ell$. The assumption about $\mathcal{A}$'s distinguishing power says that $\alpha_\ell - \alpha_0 \ge \delta$. Then using (1.21) we have

$$\begin{aligned} \text{prob}(1 \xleftarrow{\text{🎲}} \mathcal{B}(f(U_k))) &= \frac{1}{\ell} \sum_{1 \le i \le \ell} \text{prob}(1 \xleftarrow{\text{🎲}} \mathcal{A}(U_{\ell-i} \times h_i(f(U_k)))) \\ &= \frac{1}{\ell} \sum_{1 \le i \le \ell} \text{prob}(1 \xleftarrow{\text{🎲}} \mathcal{A}(Y_i)) = \frac{1}{\ell} \sum_{1 \le i \le \ell} \alpha_i, \\ \text{prob}(1 \xleftarrow{\text{🎲}} \mathcal{B}(U_{k+1})) &= \frac{1}{\ell} \sum_{1 \le i \le \ell} \text{prob}(1 \xleftarrow{\text{🎲}} \mathcal{A}(U_{\ell-i} \times h_i(U_{k+1}))) \\ &= \frac{1}{\ell} \sum_{1 \le i \le \ell} \text{prob}(1 \xleftarrow{\text{🎲}} \mathcal{A}(Y_{i-1})) = \frac{1}{\ell} \sum_{1 \le i \le \ell} \alpha_{i-1}, \\ E(\mathcal{B}(f(U_k))) - E(\mathcal{B}(U_{k+1})) &= \text{prob}(1 \xleftarrow{\text{🎲}} \mathcal{B}(f(U_k))) - \text{prob}(1 \xleftarrow{\text{🎲}} \mathcal{B}(U_{k+1})) \\ &= \frac{1}{\ell} \left(\sum_{1 \le i \le \ell} \alpha_i - \sum_{1 \le i \le \ell} \alpha_{i-1} \right) = \frac{1}{\ell}(\alpha_\ell - \alpha_0) \ge \frac{\delta}{\ell}. \end{aligned}$$

Thus algorithm $\mathcal{B}$ has distinguishing power at least δ/ℓ between $f(U_k)$ and U_{k+1}. We have to determine the size of $\mathcal{B}$. The random choices in steps 1 and 2 just correspond to some further random input gates, and do not contribute to the size. For $h_i(x)$, we have to apply f exactly $i-1 \le \ell-1$ times, using size at most ℓt. The execution of $\mathcal{A}$ takes another s' gates. The total comes to $s' + \ell t$.

Since f is (ϵ, s)-resilient, we have either $\delta/\ell \le \epsilon$ or $s' + \ell t \ge s$, which is the claim. □

It is straightforward to apply this construction to the asymptotic notion of pseudorandom generator, whose output cannot be distinguished by polynomial size circuit families from the uniform distribution.

COROLLARY 1.22. *Let $f = (f_k)_{k \in \mathbb{N}}$ be a pseudorandom generator with $f_k \colon \mathbb{B}^k \longrightarrow \mathbb{B}^{k+1}$, and $p \in \mathbb{Z}[t]$ a positive polynomial. Then the above construction yields a pseudorandom generator $g = (g_k)_{k \in \mathbb{N}}$ with $g_k \colon \mathbb{B}^k \longrightarrow \mathbb{B}^{k+p(k)}$.*

Thus we have the nice result that from the smallest possible pseudorandom generators, which add only one pseudorandom bit, we can obtain pseudorandom generators with arbitrary polynomial expansion rate.

1.6. The Nisan–Wigderson generator

All known pseudorandom generators assume that some function is hard to compute, and then extend few random bits to many bits that look random to all efficient algorithms. The Nisan–Wigderson generator that we describe now starts from a fairly general assumption of this type, and produces a pseudorandom generator.

We now quantify when a function f is hard to approximate. Namely, a probabilistic Boolean circuit $\mathcal{A}$ can produce a random bit, which then will equal the value of f with probability $1/2$. Now f is difficult if nothing essentially better is possible, with small circuits. More precisely, let $f\colon \mathbb{B}^n \longrightarrow \mathbb{B}$ be a Boolean function, $\epsilon > 0$, and $s \in \mathbb{N}$. We say that f is (ϵ, s)**-hard** if for all algorithms (= Boolean circuits) $\mathcal{A}$ with n inputs and time s, we have

$$|\operatorname{prob}(f(U_n) \stackrel{\text{🐞}}{\longleftarrow} \mathcal{A}(U_n)) - \frac{1}{2}| \leq \frac{\epsilon}{2}.$$

The **hardness** H_f of f is the maximal integer $H_f = h$ such that f is (h^{-1}, h)-hard.

One can amplify the hardness of a function by XORing several copies. This is Yao's (1982) famous XOR lemma, which we state without proof and will not use later.

Theorem 1.23 (Yao's XOR Lemma). *Let $f_1, \ldots, f_k\colon \mathbb{B}^n \longrightarrow \mathbb{B}$ all be (ϵ, s)-hard, $\delta > 0$, and $f\colon \mathbb{B}^{kn} \longrightarrow \mathbb{B}$ with*

$$f(x_1, \ldots, x_k) = \bigoplus_{1 \leq i \leq k} f_i(x_i).$$

Then f is $(\epsilon^k + \delta, \delta^2(1-\epsilon)^2 s)$-hard.

If we have a hard function f, then the single bit $f(x)$, for random $x \in \mathbb{B}^n$, looks random to any efficient algorithm. We now show how to get many bits that look random by evaluating f at many different, nearly disjoint, subsets of bits of a larger input. The tool for achieving this comes from **design theory**, an area of combinatorics, and the theory of finite fields. A thorough survey of this subject is in Beth *et al.* (1993).

Let k, n, s, and t be integers. A (k, n, s, t)**-design** D is a sequence $D = (S_1, \ldots, S_n)$ of subsets of $\{1, \ldots, k\}$ such that for all $i, j \leq n$ we have

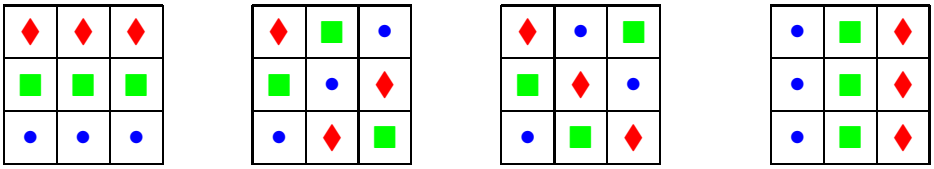
1. $\#S_i = s$,
2. $\#(S_i \cap S_j) \leq t$ if $i \neq j$.

EXAMPLE 1.24. We take $k = 9$, $n = 12$, $s = 3$, and $t = 1$, and arrange the nine elements of $\{1, \ldots, 9\}$ in a 3×3 square like this:

7	8	9
4	5	6
1	2	3

.

The reason for doing this will be explained after Theorem 1.27. In each of the four copies of the square, we have marked three subsets S_i: one with ●, one with ■, and the third one with ♦.



Thus $S_1 = \{1, 2, 3\}$, $S_2 = \{4, 5, 6\}$, $S_3 = \{7, 8, 9\}$, $S_4 = \{1, 5, 9\}$, $S_5 = \{3, 4, 8\}$, $S_6 = \{2, 6, 7\}$, $S_7 = \{1, 6, 8\}$, $S_8 = \{2, 4, 9\}$, $S_9 = \{3, 5, 7\}$, $S_{10} = \{1, 4, 7\}$, $S_{11} = \{2, 5, 8\}$, and $S_{12} = \{3, 6, 9\}$.

Now $D = \{S_1, \ldots, S_{12}\}$ is an $(9, 12, 3, 1)$-design as one easily verifies. As an example, $S_1 \cap S_5 = \{3\}$ has only one element. ◊

In design theory, one does not usually order the S_1, ..., S_n, but the above is more appropriate for our purposes. The general goal in design theory is to fix some of the four parameters and optimize the others, making n and s as large and k and t as small as possible.

If D is a (k, n, s, t)-design as above and $f\colon \mathbb{B}^s \longrightarrow \mathbb{B}$ a Boolean function, we obtain a Boolean function $f_D\colon \mathbb{B}^k \longrightarrow \mathbb{B}^n$ by evaluating f at the subsets of the bits of x given by S_1, ..., S_n. More specifically, if $x \in \mathbb{B}^k$ and $S_i = \{v_1, \ldots, v_s\}$, with $1 \le v_1 < v_2 < \cdots < v_s \le k$, then the ith bit of $f_D(x)$ is $f(x_{v_1}, \ldots, x_{v_s})$.

EXAMPLE 1.24 CONTINUED. Say we consider the parity function $f\colon \mathbb{B}^3 \longrightarrow \mathbb{B}$, so that $f(x_1, x_2, x_3) = (x_1 + x_2 + x_3) \operatorname{rem} 2$. With the design from above, the value of $f_D\colon \mathbb{B}^9 \longrightarrow \mathbb{B}^{12}$ at $x = (0, 1, 1, 1, 1, 0, 0, 0, 1) \in \mathbb{B}^9$ is

$$f_D\left(\begin{array}{|c|c|c|}\hline 0&0&1\\\hline 1&1&0\\\hline 0&1&1\\\hline\end{array}\right) = 001001010100.$$

For example, the second of the twelve values is computed as $f_D(x)_2 = f(x_4, x_5, x_6) = f(110) = 1 + 1 + 0 \operatorname{rem} 2 = 0$. ◊

We want to get rid of the two parameters ϵ and s in our notion of (ϵ, s)-resilient pseudorandom generators. To this end, we—somewhat artificially—set $\epsilon = n^{-1}$ and $s = n$. Thus we now consider pseudorandom generators $f\colon \mathbb{B}^k \longrightarrow \mathbb{B}^n$ for which there is no algorithm using time at most n and with

$$|E(\mathcal{A}(f(U_k))) - E(\mathcal{A}(U_n))| \geq n^{-1}.$$

This is seemingly more generous than the previous definition. One has to show that from a pseudorandom generator in the new sense one can construct one in the previous sense (with different values of k and n).

Theorem 1.25. *Let k, n, s be positive integers, $s \geq 2, t = \lfloor \log_s n \rfloor - 1, f\colon \mathbb{B}^s \longrightarrow \mathbb{B}$ with hardness $H_f > 2n^2$, and D an (k, n, s, t)-design. Then $f_D\colon \mathbb{B}^k \longrightarrow \mathbb{B}^n$ is an (n^{-1}, n)-resilient pseudorandom generator.*

Proof. By Theorem 1.15, any ϵ-distinguisher between $X = f_D(U_k)$ and U_n can be transformed into a $\frac{\epsilon}{n}$-predictor for some bit under X. So we now assume that we have a predictor $\mathcal{P}$ for the ith bit under X, for some $i \leq n$, with $\sigma_{\mathcal{P}}(X) \geq 1/2 + \epsilon$ and $\epsilon \geq n^{-2}$, and derive a contradiction to our hardness assumption.

By reordering the elements of $\{1, \ldots, k\}$, we may assume that $S_i = \{1, \ldots, s\}$, so that the ith bit depends only on the first s components of the values of U_k. In order to separate out the dependence on the first s and the last $k - s$ bits; we write $U_k = U_s \times U_{k-s}$. As in (1.11), we let $p(y, *) = \mathrm{prob}(y \leftarrow (X_1, \ldots, X_{i-1}))$ be the probability that y occurs as an initial segment under X, for $y \in \mathbb{B}^{i-1}$. Then

$$\begin{aligned}
1/2 + \epsilon &\leq \sigma_{\mathcal{P}}(X) \\
&= \sum_{y \in \mathbb{B}^{i-1}} \mathrm{prob}(y \xleftarrow{\text{🐞}} (X_1, \ldots, X_{i-1})) \cdot \mathrm{prob}(\mathcal{P}(y) \xleftarrow{\text{🐞}} X_i(y)) \\
&= \sum_{\substack{x' \in \mathbb{B}^s, x'' \in \mathbb{B}^{k-s} \\ y = f_D(x', x'')_{1 \ldots i-1} \in \mathbb{B}^{i-1}}} \mathrm{prob}(x' \xleftarrow{\text{🐞}} U_s) \cdot \mathrm{prob}(x'' \xleftarrow{\text{🐞}} U_{k-s}) \cdot \mathrm{prob}(f(x') \xleftarrow{\text{🐞}} \mathcal{P}(y)) \\
&= 2^{-(k-s)} \sum_{x'' \in \mathbb{B}^{k-s}} r(x''),
\end{aligned}$$

where $f_D(x', x'')_{1 \ldots i-1}$ stands for $(f_D(x', x'')_1, \ldots, f_D(x', x'')_{i-1}) \in \mathbb{B}^{i-1}$, and

$$r(x'') = 2^{-s} \sum_{\substack{x' \in \mathbb{B}^s \\ y = f_D(x', x'')_{1 \ldots i-1}}} \mathrm{prob}(f(x') \xleftarrow{\text{🐞}} \mathcal{P}(y)).$$

Thus the average of r over $\mathbb{B}^{k-s}$ is at least $1/2 + \epsilon$. Then there exists some value $z \in \mathbb{B}^{k-s}$ of x'' so that $r(z) \geq 1/2 + \epsilon$; otherwise we would have

$$2^{k-s}(1/2 + \epsilon) > 2^{k-s} \max_{x'' \in \mathbb{B}^{k-s}} r(x'') \geq \sum_{x'' \in \mathbb{B}^{k-s}} r(x'') \geq 2^{k-s}(1/2 + \epsilon).$$

This is just an instance of the general fact that some value is at least as large as the average value: "not everybody can be below average".

Now we fix such a z. Thus

$$r(z) = 2^{-s} \sum_{\substack{x' \in \mathbb{B}^s \\ y = f_D(x',z)_{1\ldots i-1}}} \operatorname{prob}(f(x') \xleftarrow{\text{🐞}} \mathcal{P}(y)) \geq 1/2 + \epsilon.$$

We now have an algorithm for approximating f: compute y as above, plug it into $\mathcal{P}$, and use $\mathcal{P}(y)$ as an approximation for $f(x')$.

ALGORITHM 1.26. Circuit $\mathcal{A}$ that approximates f.
Input: $x' = (x_1, \ldots, x_s) \in \mathbb{B}^s$.
Output: 0 or 1.

1. For $j = 1, \ldots, i-1$ do
2. $\quad y_j \longleftarrow f_D(x', z)_j$, with z as above.
3. Output $\mathcal{P}(y_1 \ldots, y_{i-1})$.

We have to show that $\mathcal{A}$ approximates f well, and that it can be built with few gates. The latter seems implausible at first, since in step 2 we have to evaluate f at some point $w_i \in \mathbb{B}^s$, given by the bits of (x', z) in the positions contained in S_i. But isn't that hard? Yes, computing f at an arbitrary input is hard, but the whole setup is designed so that these special evaluation problems become easy.

Let $1 \leq j < i$. Since $\#(S_i \cap S_j) \leq t = \lfloor \log_s n \rfloor - 1 \leq \lfloor \log_2 n \rfloor - 1$, and z is fixed, y_j depends on at most t bits. It is a general fact that any Boolean function on t bits (with one output) can be computed in time 2^{t+1}, say by writing it in disjunctive (or conjunctive) normal form. Thus y_j can be computed from x' in time $2^{t+1} \leq n$, and all of $y_1, \ldots, y_{i-1}$ can be computed with at most $n(i-1) \leq n^2$ operations.

What is the probability that $\mathcal{A}(x') = f(x')$, for $x' \longleftarrow U_s$? We are given our fixed z, and compute $y_1, \ldots, y_{i-1}$ correctly from x'. Thus $\mathcal{A}(x') = \mathcal{P}(y_1, \ldots, y_{i-1})$, and

$$\begin{aligned} 2^{-s} \sum_{x' \in \mathbb{B}^s} \operatorname{prob}(f(x') \xleftarrow{\text{🐞}} \mathcal{A}(x')) &= 2^{-s} \sum_{\substack{x' \in \mathbb{B}^s \\ y = f_D(x',z)_{1\ldots i-1}}} \operatorname{prob}(f(x') \xleftarrow{\text{🐞}} \mathcal{P}(y)) \\ &= r(z) \geq 1/2 + \epsilon \geq 1/2 + n^{-2}. \end{aligned}$$

This contradicts the assumption that $H_f \geq 2n^2$, and proves the claim. □

1.7. Construction of good designs

As in many other fields of combinatorics, finite fields are the basis for an attractive solution. Let $\mathbb{F}_q$ be a finite field with q elements, so that q is a prime power, $t < q$

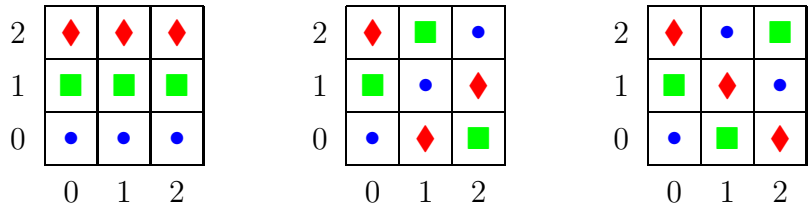
an integer,

$$\begin{aligned} P &= \{f \in \mathbb{F}_q[x]\colon \deg f \le t\}, \\ S_f &= \{(u, f(u))\colon u \in \mathbb{F}_q\} \subseteq L = \mathbb{F}_q^2 \text{ for } f \in P, \\ k &= \#L = q^2, \quad n = q^{t+1}. \end{aligned}$$

Theorem 1.27. *The collection of all these graphs S_f of $f \in P$ is a (k, n, q, t)-design.*

Proof. The only claim to verify is that $\#(S_f \cap S_g) \le t$ for distinct f and $g \in P$. But $\#(S_f \cap S_g) \ge t+1$ means that the two polynomials f and g of degree at most t have $t+1$ values in common. Then $f - g$ is a polynomial of degree at most t with at least $t+1$ roots, hence the zero polynomial, and we have $f = g$. □

Example 1.24 continued. We take $q = 3$ and $t = 1$, so that $k = 9 = q^2$, $n = 9 = q^{1+1}$, and $s = 3 = q$. The following picture shows this design.



$\bullet$: $f = 0$, $\blacksquare$: $f = 1$, $\blacklozenge$: $f = 2$,
$\bullet$: $f = x$, $\blacksquare$: $f = x + 1$, $\blacklozenge$: $f = x + 2$,
$\bullet$: $f = 2x$, $\blacksquare$: $f = 2x + 1$, $\blacklozenge$: $f = 2x + 2$.

Thus we find the first nine pieces of the design from Example 1.24. ◊

In general, this construction does not provide the best possible design, but it is very simple and sufficient for our purposes.

$s = q$	t	$k = s^2$	$n = s^{t+1}$	$\binom{k}{s}$
3	1	9	9	28
3	2	9	27	84
4	2	16	64	1820

Figure 1.3: Some design parameters. Compare n to the number $\binom{k}{s}$ of all subsets of size s.

COROLLARY 1.28.

(i) For any positive integers $s > t$, where s is a prime power, there exists an (s^2, s^{t+1}, s, t)-design.

(ii) For any positive integers k, n, t, and a prime power s with $k \geq s^2$ and $t \geq \log_s n - 1$ there exists a (k, n, s, t)-design.

PROOF. In (i) we have recorded the above construction. For (ii), we use (i) and note that $n = s^{\log_s n} \leq s^{t+1}$. □

COROLLARY 1.29. *Let n and s be positive integers, with s a prime power, and $f\colon \mathbb{B}^s \longrightarrow \mathbb{B}$ with hardness $H_f \geq 2n^2$. Then the Nisan–Wigderson generator is a pseudorandom generator from $\mathbb{B}^{s^2}$ to $\mathbb{B}^n$.*

In particular, if n is exponential in s, say $n = 2^{s/4}$, then we have a pseudorandom generator that turns short random strings into exponentially long pseudorandom ones.

The corollary has the form:

(1.30) If there is a hard problem, then a pseudorandom generator exists.

Most statements about the existence of pseudorandom generators have this form. We have a substantial collection of problems that we think are hard, but unfortunately it is even harder to **prove** this. In fact, very few such results are known; we will mention one below. On the other hand, almost all Boolean functions on s inputs require time at least $2^s/s$ to compute them exactly. This is easily proved by a counting argument; see Muller (1956) and Boppana & Sipser (1990), Theorem 2.4. Thus hard functions do exist; an unresolved difficulty is to find nice and natural such functions. But for our application we would have to solve a yet more difficult problem: to show that some functions are even hard to approximate.

One of the interesting consequences of Nisan and Wigderson's work is that this lamentable situation of relying on the hardness of functions is unavoidable: the converse of (1.30) also holds! If we can prove that something is a pseudorandom generator, then we have automatically proved some problem to be hard!

Recall the complexity classes

$$\mathcal{P} \subseteq \mathcal{ZPP} \subseteq \mathcal{RP} \subseteq \mathcal{BPP} \subseteq \mathcal{NP} \subseteq \mathcal{EXPTIME}.$$

We say that a generator $g = (g_k)_{k\in\mathbb{N}}$, with $g_k\colon \mathbb{B}^k \longrightarrow \mathbb{B}^{n(k)}$ for all k, is *quick* if it can be implemented in time exponential in k. A function $r\colon \mathbb{N} \longrightarrow \mathbb{N}$ is called *reasonable* if for all $k, k' \in \mathbb{N}$ we have

$$k \leq r(k) \leq 2^k,$$

$$r(k) \leq r(k') \text{ if } k \leq k',$$
$$(r(k))^2 \leq r(k^2).$$

Theorem 1.31. *Let $r\colon \mathbb{N} \longrightarrow \mathbb{N}$ be reasonable. Then the following statements are equivalent:*

(i) For some $c > 0$ there exists a function in $\mathcal{EXPTIME}$ with hardness $r(k^c)$.

(ii) For some $c > 0$ there exists a quick pseudorandom generator g with $g_k\colon \mathbb{B}_k \longrightarrow \mathbb{B}^{r(k^c)}$.

Proof. We only prove (i) $\Longrightarrow$ (ii). Let $f = (f_s)_{s\in\mathbb{N}}$ be a function in $\mathcal{EXPTIME}$ with $f_s\colon \mathbb{B}^s \longrightarrow \mathbb{B}$ for all s and hardness $H_{f_s} \geq r(s^c)$. We build a pseudorandom generator $g = (g_k)$ with $g_k\colon \mathbb{B}^k \longrightarrow \mathbb{B}^n$, with $n = r(k^{c/4} - 1)/2$. Let $k \in \mathbb{N}$ and $s = \lfloor k^{1/2} \rfloor$. Then $H_{f_s} \geq r(s^c) \geq r(k^{c/2}-1) \geq r((k^{c/4}-1)^2) \geq (r(k^{c/4}-1))^2 = 2n^2$. Now the corollary says that we indeed have a pseudorandom generator from $\mathbb{B}^{s^2} \longrightarrow \mathbb{B}^n$. Since $s^2 \leq k$, this gives a pseudorandom generator $\mathbb{B}^k \longrightarrow \mathbb{B}^n$. □

1.8. Deterministic simulation of probabilistic computation

A fundamental question about probabilistic computations is whether randomness really helps, or whether it can in effect be eliminated without too much cost.

Suppose that $\mathcal{A}$ is a probabilistic algorithm computing some function f in time t. In particular, it uses at most $t(n)$ random bits on inputs of size n. We can simulate $\mathcal{A}$ by deterministic algorithm $\mathcal{B}$ which makes all $2^{t(n)}$ choices of these bits one after the other, simulates $\mathcal{A}$ on each of them, and then counts the outcomes and takes the majority opinion as output. Of course, this is an exponential increase in cost, from $t(n)$ to $2^{t(n)}$.

Now if $t(n)$ is polynomial in n and we have a good pseudorandom generator, we may take its output instead of the random bits required in $\mathcal{A}$. Then we only have to try out all possible choices for the seeds to the pseudorandom generator. This may be exponentially less than $2^{t(n)}$.

Nisan and Wigderson assume that there exists a function in $\mathcal{DTIME}$ $(2^{O(n)})$ with the properties at left, and conclude the inclusions of complexity classes at right.

Not approximable by polynomial-sized circuits	$\mathcal{BPP} \subseteq \bigcap_{\epsilon>0} \mathcal{DTIME}(2^{n^\epsilon})$
For some $\epsilon > 0$, not approximable by circuits of size 2^{n^ϵ}	$\mathcal{BPP} \subseteq \mathcal{DTIME}(2^{(\log n)^c})$ for some $c > 0$
Hardness $\geq 2^{\epsilon n}$ for some $\epsilon > 0$	$\mathcal{BPP} = \mathcal{P}$

Generally speaking, (too) few hardness results have been proven. One of the few successes deals with circuits of constant depth d. Håstad proved a lower bound on approximability (of the parity function). Nisan and Wigderson use this to obtain a pseudorandom generator $\mathbb{B}^k \longrightarrow \mathbb{B}^n$ with exponential expansion $n^{k^{1/(2d+6)}}$ which no circuit of polynomial size and with depth at most d can distinguish from the uniform distribution more than negligible advantage.

1.9. The Blum–Blum–Shub generator

This generator takes $N = p \cdot q$ with distinct odd primes p and q as in RSA, as seed a random square $x_0 \in \mathbb{Z}_N^\times$, then computes $x_i \equiv x_{i-1}^2 \operatorname{rem} N$, and returns the low order bit

$$x_0 \operatorname{rem} 2, x_1 \operatorname{rem} 2, \ldots .$$

Why the hell should this be secure?

For $a \in \mathbb{Z}$ and a prime p, the **Legendre symbol** is

$$\left(\frac{a}{p}\right) = \begin{cases} 1 & \text{if } a \bmod p \in \mathbb{Z}_p^\times \text{ is a square,} \\ -1 & \text{if } a \bmod p \in \mathbb{Z}_p^\times \text{ is a nonsquare,} \\ 0 & \text{if } p|a. \end{cases}$$

Fermat's Little Theorem says that $a^{p-1} = 1$ in $\mathbb{Z}_p$ for all $a \neq 0$. If $a = b^2$, then $a^{(p-1)/2} = (b^2)^{(p-1)/2} = b^{p-1} = 1$. Thus $x^{(p-1)/2} - 1$ has the $(p-1)/2$ squares as its roots, and since its degree is $(p-1)/2$, there are no others. It follows that $\left(\frac{a}{p}\right) \equiv a^{(p-1)/2} \pmod p$.

Now $\frac{p-1}{2}$ elements of $\mathbb{Z}_p^\times$ are squares, and $\frac{p-1}{2}$ are nonsquares, so that half of the elements of $\mathbb{Z}_p^\times$ have Legendre symbol 1, and half have -1. The **Jacobi symbol** is defined in our situation as

$$\left(\frac{a}{pq}\right) = \left(\frac{a}{p}\right) \cdot \left(\frac{a}{q}\right).$$

By the Chinese Remainder Theorem, an element $a \in \mathbb{Z}_N^\times$ is a square modulo N if and only if it is a square modulo p and modulo q. We have

$$\left(\frac{-1}{p}\right) = 1 \iff p \equiv 1 \bmod 4.$$

Let $\square = \square_N = \{a \in \mathbb{Z}_N^\times \colon \exists b \in \mathbb{Z}_N^\times \; a = b^2\}$ be the set of squares modulo N, and $\boxtimes = \boxtimes_N = \{a \in \mathbb{Z}_N^\times \colon \left(\frac{a}{N}\right) = 1 \text{ and } a \notin \square\}$ be the set of nonsquares modulo N with Jacobi symbol 1. (They are also called *pseudosquares* in the literature, but this is a very different use of "pseudo" from pseudoprimes—which are usually primes—and "pseudorandom" elements—which behave like random elements; the elements of $\boxtimes$ are never squares.)

	$\left(\frac{a}{q}\right) = 1$	$\left(\frac{a}{q}\right) = -1$
$\left(\frac{a}{p}\right) = 1$	$\left(\frac{a}{N}\right) = 1$ $\square_N$ $\square_p \times \square_q$ $(u, -v)$	$\left(\frac{a}{N}\right) = -1$ $\square_p \times \boxtimes_q$ (u, v)
$\left(\frac{a}{p}\right) = -1$	$\left(\frac{a}{N}\right) = -1$ $\boxtimes_p \times \square_q$ $(-u, -v)$	$\left(\frac{a}{N}\right) = 1$ $\boxtimes_N$ $\boxtimes_p \times \boxtimes_q$ $(-u, v)$

Figure 1.4: The values u and v are explained in the text.

It is easy to compute $\left(\frac{a}{N}\right)$ by a method similar to the Euclidean algorithm. This takes $O(k^2)$ bit operations if a and N are k-bit numbers (and presumably $O(M(k)\log k)$ with fast arithmetic). Thus we can quickly tell whether $a \in \square \cup \boxtimes$. The **quadratic residuosity problem** modulo N is to decide on input $a \in \square \cup \boxtimes$ whether $a \in \square$. Of course, given the factors p and q, this becomes easy since we can compute $\left(\frac{a}{p}\right)$ and $\left(\frac{a}{q}\right)$. But no polynomial-time algorithm is known if these factors are not provided, and we will assume that in fact this is a hard problem.

Let $\square_p, \boxtimes_p \subset \mathbb{Z}_p^\times$ be the sets of squares and nonsquares, respectively, and similarly for q. Under the Chinese remainder isomorphism

$$\chi\colon \mathbb{Z}_N^\times \longrightarrow \mathbb{Z}_p^\times \times \mathbb{Z}_q^\times$$

we have

$$\begin{aligned} \chi(\square_N) &= \square_p \times \square_q, \\ \chi(\boxtimes_N) &= \boxtimes_p \times \boxtimes_q\,. \end{aligned}$$

We consider the squaring map $\sigma_p\colon \mathbb{Z}_p^\times \longrightarrow \square_p \subseteq \mathbb{Z}_p^\times$ with $\sigma_p(a) = a^2$. If $p \equiv 3 \bmod 4$, then -1 is not a square modulo p, and exactly one of the two square roots a and $-a$ of a^2 is a square.

We now assume that $p \equiv q \equiv 3 \bmod 4$. Then $N = pq$ is called a **Blum integer**, after Manuel Blum. If $\chi(a) = (u, v)$, then $\chi(a^2)$ has the four square roots

$$(u, v), (-u, v), (u, -v), (-u, -v).$$

Exactly one of them is a square, and χ^{-1} of this square is called the **principal (square) root** of a^2. If, say, u is a square modulo p and v a nonsquare modulo q, then $(u, -v)$ is the square among the four. This situation is illustrated in Figure 1.4.

EXAMPLE 1.32. We let $p = 3$ and $q = 7$, so that $N = 21$. Then $\square_3 = \{1\}$, $\boxtimes_3 = \{2\}$, $\square_7 = \{1, 2, 4\}$, and $\boxtimes_7 = \{3, 5, 6\}$. Figure 1.4 now looks as follows: Not surprisingly, 1 is the principal root of 1, and 4 that of 16. But also 16 is

	$1, 2, 4$	$3, 5, 6$
1	$1 \leftrightarrow (1,1)$	$10 \leftrightarrow (1,3)$
	$16 \leftrightarrow (1,2)$	$19 \leftrightarrow (1,5)$
	$4 \leftrightarrow (1,4)$	$13 \leftrightarrow (1,6)$
2	$8 \leftrightarrow (2,1)$	$17 \leftrightarrow (2,3)$
	$2 \leftrightarrow (2,2)$	$5 \leftrightarrow (2,5)$
	$11 \leftrightarrow (2,4)$	$20 \leftrightarrow (2,6)$

Figure 1.5:

the principal root of 4. In other words, $-5 \equiv 16 \bmod 21$ is the principal root of $25 \equiv 4 \bmod 21$. ◊

From an algebraic point of view, $\square$ is a subgroup of $\mathbb{Z}_N^\times$ with $\phi(N)/4$ elements. The squaring map $\sigma\colon \mathbb{Z}_N^\times \longrightarrow \square$ is a homomorphism which always maps four elements $(\pm u, \pm v)$ to one, namely to (u^2, v^2). $\square$ has four cosets $\square$, $\boxtimes$, and, say, C_0 and C_1, and in each coset lies exactly one of these four square roots. In particular, σ induces a bijection on $\square$. Multiplication by -1 gives a bijection between $\square$ and $\boxtimes$ (and between C_0 and C_1). The residue class group $\mathbb{Z}_N^\times/\square$ is isomorphic to $\{\pm 1\} \times \{\pm 1\} \cong \mathbb{Z}_2 \times \mathbb{Z}_2$, with $\square \leftrightarrow (1,1)$ and $\boxtimes \leftrightarrow (-1,-1)$. Here $\{\pm 1\}$ is the "multiplicative version" of $\mathbb{Z}_2$. The corresponding mapping $\mathbb{Z}_N^\times \longrightarrow \{\pm 1\} \times \{\pm 1\}$ is given by $a \longmapsto \left(\left(\frac{a}{p}\right), \left(\frac{a}{q}\right)\right)$.

Our ultimate goal is to prove the following result.

THEOREM 1.33. *Let N be a k-bit Blum integer, consider the Blum–Blum–Shub generator $g\colon \mathbb{B}^k \longrightarrow \mathbb{B}^n$ for some $n > k$, and suppose that $\mathcal{A}$ is an ϵ-distinguisher between $g(u_k)$ and u_n, for $\epsilon = n^{-e}$ for some $e > 0$. Then for any $\delta > 0$ one can test quadratic residuosity by a probabilistic algorithm $\mathcal{T}$ with error probability at most δ. If $\mathcal{A}$ uses time polynomial in n, then $\mathcal{B}$ uses time polynomial in n, ϵ^{-1} and $\log \delta^{-1}$.*

We note that $\mathcal{A}$ only has to work well on most inputs, while on any single input x to $\mathcal{B}$, the error probability is at most δ.

The proof proceeds in four steps:

distinguisher $\stackrel{\text{step 1}}{\longrightarrow}$ postdictor $\stackrel{\text{step 2}}{\longrightarrow}$ squareness distinguisher $\stackrel{\text{step 3}}{\longrightarrow}$ weak squareness test $\stackrel{\text{step 4}}{\longrightarrow}$ strong squareness test.

A **postdictor** (or **previous bit predictor**) works like our old friends the predictors, only it predicts the previous bit x_0 from $x_1, \ldots, x_n$. Yao's method yields the first step.

Step 1: From $\mathcal{A}$ we obtain an $\frac{\epsilon}{n}$-postdictor.

In **Step 2**, we build a squareness distinguisher $\mathcal{B}$ from a postdictor $\mathcal{P}$.

Algorithm 1.34. Squareness distinguisher $\mathcal{B}$.
Input: A Blum integer N and $a \in \square \cup \boxtimes \subseteq \{0, \ldots, N-1\}$.
Output: "$a \in \square$" or "$a \in \boxtimes$".

1. Compute $x_1 = a^2 \operatorname{rem} N$.
2. Compute the output $y_1 = x_1 \operatorname{rem} 2, \ldots, y_n = x_n \operatorname{rem} 2$ of the Blum–Blum–Shub generator.
3. Compute $z = \mathcal{P}(y_1, \ldots, y_n)$.
4. If $a \equiv z \bmod 2$ then output "$a \in \square$" else output "$a \in \boxtimes$".

The idea is that $\mathcal{P}$ always postdicts elements from a long sequence of repeated squares, so that the postdicted z is likely to be the low order bit of a square. The two square roots modulo N in $\square \cup \boxtimes$ of $x_1 \equiv a^2 \bmod N$ are a and $-a \operatorname{rem} N = N - a$, and $a \not\equiv N - a \bmod 2$ since N is odd.

Lemma 1.35. *Suppose that $\mathcal{P}$ is an ϵ-postdictor. Then for $a \in \square \cup \boxtimes$ chosen uniformly at random, the output of the squareness distinguisher is correct with probability at least $1/2 + \epsilon$.*

Proof. By assumption, we have

$$\begin{aligned}
\frac{1}{2} + \epsilon \leq \sigma_{\mathcal{P}}(p) &= \sum_{a \in \square \cup \boxtimes} p(a) \operatorname{prob}(a_0 = \mathcal{P}(y_1, \ldots, y_n)) \\
&= p_0 \cdot \sum_{a \in \square \cup \boxtimes} \operatorname{prob}(a_0 = \mathcal{P}(y_1, \ldots, y_n)) \\
&= p_0 \cdot \left[\sum_{a \in \square} \operatorname{prob}(\mathcal{B}(a) = \text{"}a \in \square\text{"}) + \sum_{a \in \boxtimes} \operatorname{prob}(\mathcal{B}(a) = \text{"}a \in \boxtimes\text{"}) \right] \\
&= p_0 \cdot \sum_{a \in \square \cup \boxtimes} \operatorname{prob}(\mathcal{B}(a) \text{ is correct}),
\end{aligned}$$

where a_0 is the low order bit of a_1, p is the uniform distribution on $\square \cup \boxtimes$, so that $p(a) = p_0 = (\#(\square \cup \boxtimes))^{-1} = 2/(p-1)(q-1)$ for all $a \in \square \cup \boxtimes$, $y_1, \ldots, y_n$ are

computed as in step 2 of $\mathcal{B}$, and "prob" refers to the internal distribution of the probabilistic algorithms $\mathcal{P}$ and $\mathcal{B}$. We note that $\#\mathbb{Z}_N^\times = (p-1)(q-1) = \varphi(N)$. $\square$

Step 3. Our algorithm $\mathcal{B}$ distinguishes (slightly) the squares in $\square$ from the nonsquares in $\boxtimes$. If $\mathcal{B}$ were deterministic, then for slightly more than half the inputs from $\square \cup \boxtimes$ its answer would be correct. (In general, the correctness probabilities sum to just over 1/2.) We now build a much stronger result from this: a probabilistic algorithm $\mathcal{C}$ whose success probability on **any** input is slightly more than 1/2. For any problem, any algorithm with $\mathcal{C}$'s properties also has $\mathcal{B}$'s properties, but in general one cannot go the other way around. Here we succeed in this by "smearing" the (non)squareness of a single input x uniformly across the whole of $\square \cup \boxtimes$.

ALGORITHM 1.36. Weak squareness test $\mathcal{C}$.

Input: $x \in \square \cup \boxtimes$.
Output: "$x \in \square$" or "$x \in \boxtimes$".

1. Choose $r \in \mathbb{Z}_N^\times$ and $b \in \{0,1\}$ uniformly at random.
2. Compute $z = (-1)^b r^2 x \operatorname{rem} N$.
3. Call $\mathcal{B}$ with input z, and let $c \in \{0,1\}$ be the output bit $c = (\mathcal{B}(z) = "z \in \square")$. [Thus c is 1 if and only if $\mathcal{B}$ answers "$z \in \square$".]
4. Output "$x \in \square$" if $b \oplus c = 1$ and "$x \in \boxtimes$" otherwise.

THEOREM 1.37. *For any input $x \in \square \cup \boxtimes$, this test $\mathcal{C}$ answers correctly with probability at least $1/2 + \epsilon$.*

PROOF. We first claim that if $\mathcal{B}$ answers correctly, then so does $\mathcal{C}$. Let $x \in \square$. Then

$$b = 0 \Longleftrightarrow z \in \square \Longleftrightarrow (\mathcal{B}(z) = "z \in \square") \Longleftrightarrow c = 1.$$

Thus for both possible values of b, we have $b \oplus c = 1$, and $\mathcal{C}(x)$ is correct. Similarly, for $x \in \boxtimes$ we find

$$b = 0 \Longleftrightarrow z \in \boxtimes \Longleftrightarrow (\mathcal{B}(z) = "z \in \boxtimes") \Longleftrightarrow c = 0,$$

so that $b \oplus c = 0$. This proves the claim.

Now let $x \in \square \cup \boxtimes$ be an input, and $y \in \square \cup \boxtimes$ arbitrary. We claim that there exists exactly four choices for (b, r) so that $y = (-1)^b r^2 x \operatorname{rem} N$. First suppose that $x \in \square$. The elements $r^2 \operatorname{rem} N$ form precisely the set $\square$ of squares, and each element of $\square$ comes from four values of r. Since $\square$ is a group, the elements $r^2 x \operatorname{rem} N$ also make up $\square$, each element occurring four times. Now multiplication by $1 = (-1)^0$ does not change anything, while multiplication by $-1 = (-1)^1$ maps $\square$ bijectively to $\boxtimes$.

Similarly, if $x \in \boxtimes$, then $r^2 x \operatorname{rem} N$ forms $\boxtimes$, four times, and $(-1)^b r^2 x \operatorname{rem} N$ gives $\square \cup \boxtimes$.

In particular, for any input $x \in \square \cup \boxtimes$, the element z computed in the algorithm is a uniform random element of $\square \cup \boxtimes$. The success probability of $\mathcal{B}$ on such inputs z is at least $1/2 + \epsilon$, so that $\mathcal{C}$ also has at least this success probability. □

Now comes the final **Step 4**. We have a Monte Carlo test $\mathcal{C}$ for squareness with success probability at least $1/2 + \epsilon$. We now improve this to $1 - \delta$ for any $\delta > 0$.

This method for bumping up success probabilities works for any Monte Carlo algorithm. So we have a set $\square \subseteq \mathbb{B}^n$ and a probabilistic algorithm $\mathcal{C}$ which answers "$x \in \square$" or "$x \notin \square$" on input x, and the output is correct with probability exactly $1/2 + \epsilon$ for every $x \in \mathbb{B}^n$.

We let $k = 2m + 1$ for $m \in \mathbb{N}$, and consider the test $\mathcal{T}$ which, on input x, runs $\mathcal{C}$ exactly k times and outputs the majority answer.

Theorem 1.38. *The test $\mathcal{T}$ answers correctly with probability at least $1 - (1 - 4\epsilon^2)^m/2$.*

Proof. Let x be an input. We assume that the correctness probability of $\mathcal{C}$ on input x is exactly $1/2 + \epsilon$. The probability of obtaining exactly i correct answers in k trials is

$$\binom{k}{i} \left(\frac{1}{2} + \epsilon\right)^i \left(\frac{1}{2} - \epsilon\right)^{k-i}.$$

$\mathcal{T}$ answers incorrectly if at most m correct answers were given by $\mathcal{C}$. We set $s = \frac{1}{2} + \epsilon$ and $t = \frac{1}{2} - \epsilon = 1 - s$. Thus $s/t \leq 1$, $k - m = m + 1$ and $\sum_{0 \leq i \leq n} \binom{k}{i} = 2^{2m+1}$. The probability that $\mathcal{T}$ answers incorrectly is at most

$$\begin{aligned}
\sum_{0 \leq i \leq m} \binom{k}{i} s^i t^{k-i} &= s^m t^{k-m} \sum_{0 \leq i \leq m} \binom{k}{i} (s/t)^{m-i} \\
&\leq (st)^m t \sum_{0 \leq i \leq m} \binom{k}{i} \\
&= \left(\frac{1}{4} - \epsilon^2\right)^m t \cdot 2^{2m} \\
&= (1 - 4\epsilon^2)^m t \leq (1 - 4\epsilon^2)^m/2.
\end{aligned}$$

□

Corollary 1.39. *In order to improve the correctness probability from $1/2 + \epsilon$ to $1 - \delta$, as above, it is sufficient to take $k = \lceil \epsilon^{-2} \ln((2\delta)^{-1}) \rceil + 2$.*

PROOF. It is sufficient to choose m so that

$$(1-4\epsilon^2)^m/2 \le \delta.$$

Thus

$$m \ge \left\lceil \frac{\ln 2\delta}{\ln(1-4\epsilon^2)} \right\rceil$$

is good enough. The Brook Taylor expansion of the natural logarithm gives $\ln(1-x) = -x + \frac{x^2}{2} - \frac{x^3}{3} + - \ldots \le -x/2$ for $0 \le x < 1$. (Note that the two logarithms have negative values.) Hence

$$\frac{\ln 2\delta}{\ln(1-4\epsilon^2)} \le \frac{2}{4\epsilon^2} \ln((2\delta)^{-1}).$$

Thus

$$k = 2m+1 = 2\left\lceil \frac{\ln((2\delta)^{-1})}{2\epsilon^2} \right\rceil + 1 \le \left\lceil \frac{\ln((2\delta)^{-1})}{\epsilon^2} \right\rceil + 2$$

is sufficient. □

1.10. Randomness extraction

In this section we will explore the following problem: Assume you are given a source which generates n bits of "bad" randomness and the goal is to extract m bits of "good" randomness.

The oldest approach was given by von Neumann (1951). He solved the following question: Given a coin B whose probability of giving `Heads` is p, construct out of this coin a coin C for which the probability of giving `Heads` is $1/2$. The solution he gave was to throw the coin B twice. If in the two experiments the results are different (e.g. first `Heads` then `Tails`), the value of the new coin is defined to be the value of the first throw (in our example `Heads`). Otherwise the result is discarded and the coin B is again thrown twice. The probability for the coin C giving `Heads` is now equal to the probability of C giving `Tails`. However we will have to throw the coin an expected number of $1/(2p(1-p))$ times twice in order to extract one fair coin toss. Thus the above procedure extracted in a suitable sense the randomness hidden in B.

Identify in this section constantly a random variable with its distribution and make the distinction only if necessary. We are all the time discussing probability distributions, their distance and the amount of randomness they contain. For the latter Shannon's entropy pops into ones mind, but it turns out that it is often hard to estimate the entropy in a distribution. A more convenient measure is the so called min-entropy. For a random variable X over (in our case always finite) set A, the min-entropy of X is defined by

$$H_\infty(X) := \min_{a \in A} - \log_2 \operatorname{prob}(a \xleftarrow{\bullet\bullet} X).$$

We observe that if $H_\infty(X)$ then for all $a \in A$ we have $\text{prob}(a \xleftarrow{\text{🐞}} X) \leq 2^{-k}$.

It is illustrative to compare this value to Shannon's entropy

$$H(X) = -\sum_{a \in A} \text{prob}(a \xleftarrow{\text{🐞}} X) \log_2 \text{prob}(a \xleftarrow{\text{🐞}} X).$$

While Shannon measures the "amount of randomness" on average, the min-entropy captures the "worst-case". Intuitively the min-entropy measures the number of random bits that are at least contained in X. More precisely we can think of X containing at least k random bits if and only if $H_\infty(X) \geq k$.

Now let X, Y be two distributions over the (finite) set A. We define their *statistical distance* by

$$\Delta(X, Y) := \frac{1}{2} \sum_{a \in A} |\text{prob}(a \xleftarrow{\text{🐞}} X) - \text{prob}(a \xleftarrow{\text{🐞}} Y)|.$$

We say that X is ε-close to Y if $\Delta(X, Y) \leq \varepsilon$ and we call X to be ε-quasi-random if X is ε-close to uniform.

Now what is an extractor? Ideally we would like at least to construct a function E on n-bit strings that extracts a *single bit* of randomness, i.e. a function $E : \mathbb{B}^n \to B$ such that for all distributions X on $\mathbb{B}^n$ with $H_\infty(X) \geq n - 1$, the value $E(X)$ is close to a uniform random bit. However such a function E does not exist: Every function E has a bit $b \in \mathbb{B}$ such that the set $S := \{x \in \mathbb{B}^n \,|\, E(x) = b\}$ has not less than 2^{n-1} elements. The distribution U_S that assigns to each element of S the same probability and to all elements not in S probability 0 certainly has $H_\infty(U_S) \geq n - 1$ but the output $E(X)$ is constantly b and thus not close to uniform. This means that we need to be more generous for the definition of an extractor.

In the following we will always use the following notation: n denotes the length of the input x following distribution X, m the length of the extracted element, k denotes the min-entropy of X and d the length of some additional uniform random input which will allow us to actually construct our extractors.

Definition 1.40 (Extractor). *A function $E : \mathbb{B}^n \times \mathbb{B}^d \to \mathbb{B}^m$ is called a (k, ε)-extractor if for all distributions X on $\mathbb{B}^n$ with $H_\infty(X) \geq k$ the output $E(X, U_d)$ is ε-quasi-random. Here U_d denotes the uniform distribution on d-bit strings.*

We call E *efficient* if it is computable in polynomial time (in the size of the input). This of course makes only sense if we consider E to be a whole family of functions and d and m to be suitable functions in n. The extractor is called *nontrivial* if $m > d$. This is because we simply could output the first m bits of the additional truely random input if $d \geq m$.

How do we now construct such an extractor? The idea is to hash in a suitable sense elements from $\mathbb{B}^n$ to $\mathbb{B}^m$. To do so we need certain families of hash-functions that behave (in a statistical sense) very well:

DEFINITION 1.41 (Universal family of hash functions). *Assume you have a family of functions* $H = \{h_s : \mathbb{B}^n \to B^\ell \mid s \in \mathbb{B}^d\}$. *Then* H *is called* universal *if for all* $x_1, x_2 \in \mathbb{B}^n$ *and all* $y_1, y_2 \in B^\ell$ *we have*

$$\text{prob}(y_1 \xleftarrow{\text{🐞}} H(x_1) \text{ and } y_2 \xleftarrow{\text{🐞}} H(x_1)) = 2^{-2\ell}.$$

We are now going to construct an extractor out of any universal family of hash functions:

DEFINITION 1.42. *Let* H *be a universal family of hash functions of size* 2^d. *The extractor defined by* H *is given by*

$$E(x, s) = h_s(x) \circ s,$$

the notation denotes concatenation of strings.

In abuse of notations we will write $E(x, h) = h(x) \circ h$, identifying the function h with its index. Note that the number m of produced bits is equal to $\ell + d$.

We will now show that this construction is indeed good:

THEOREM 1.43. *If* H *is a universal family of hash functions and* $\ell \in k - 2\log(1/\varepsilon) - \mathcal{O}(1)$, *then the extractor* $E(x, h)$ *defined by* H *is a* $(k, \varepsilon/2)$*-extractor.*

PROOF. For the proof we will need a certain tool, the so called *collision probability*. For a distribution X over $\mathbb{B}^n$ we define its collision probability as

$$\begin{aligned}\text{Col}(X) &= \text{prob}(x = y \text{ and } x \xleftarrow{\text{🐞}} X \text{ and } y \xleftarrow{\text{🐞}} X)\\ &= \sum_{x,y \in \mathbb{B}^n} \text{prob}(x \xleftarrow{\text{🐞}} X)\,\text{prob}(y \xleftarrow{\text{🐞}} X)\\ &= \sum_{x \in \mathbb{B}^n} \text{prob}(x \xleftarrow{\text{🐞}} X)^2\end{aligned}$$

The proof has three steps:

1. We show that if $H_\infty(X) \geq k$ then $\text{Col}(X) \leq 1/k$.

2. If $\text{Col}(X)$ is small, so is the collision probability of the output distribution of the extractor defined by H.

3. We finish by observing that if the collision probability of a distribution Y is close to the collision probability of the uniform distribution, then Y is close to uniform.

Ad 1) We have that

$$\mathrm{Col}(X) = \sum_{x\in\mathbb{B}^n} \mathrm{prob}(x \xleftarrow{\text{🐞}} X)^2 \le p_{\max} \sum_{x\in\mathbb{B}^n} \mathrm{prob}(x \xleftarrow{\text{🐞}} X) = p_{\max} \le 2^{-k}$$

Here $p_{\max}$ denotes the maximum probability for which X assumes a certain value.

Ad 2) We need to analyze $\mathrm{Col}(E(X,H)) = \mathrm{Col}(H(X)\circ H)$. We have

$$\begin{aligned}
&\mathrm{Col}(H(X)\circ H)\\
&\quad = \mathrm{prob}(H(X)\circ H = H'(X')\circ H')\\
&\quad = \mathrm{prob}(H=H')\,\mathrm{prob}(H(X)=H'(X') \mid H=H')\\
&\quad = \mathrm{prob}(H=H')\,\mathrm{prob}(H(X)=H(X'))\\
&\quad = \mathrm{prob}(H=H')\,(\mathrm{prob}(X=X') + \mathrm{prob}(X\neq X')\,\mathrm{prob}(H(X)=H(X') \mid X\neq X')))\\
&\quad \le 2^{-d}\left(2^{-k} + \mathrm{prob}(H(X)=H(X') \mid X\neq X')\right)\\
&\quad = 2^{-d}\left(2^{-k} + 2^{-\ell}\right)\\
&\quad = 2^{-(\ell+d)}\left(2^{2\log\varepsilon - \mathcal{O}(1)} + 1\right)\\
&\quad \le 2^{-(\ell+d)}\left(\varepsilon^2 + 1\right)
\end{aligned}$$

Ad 3) Let Y_1, Y_2 be any two probability distributions over $\mathbb{B}^m$. Then we have the following fact:

$$\Delta(Y_1,Y_2) \le \frac{1}{2}2^{m/2}\sqrt{\sum_{y\in\mathbb{B}^m}(\mathrm{prob}(y \xleftarrow{\text{🐞}} Y_1) - \mathrm{prob}(y \xleftarrow{\text{🐞}} Y_2))^2} \tag{1.44}$$

We are now going to estimate the last sum. We have:

$$\sum_{y\in\mathbb{B}^m}(\mathrm{prob}(y \xleftarrow{\text{🐞}} Y_1) - \mathrm{prob}(y \xleftarrow{\text{🐞}} Y_2))^2 =$$

$$\sum_{y\in\mathbb{B}^m}\mathrm{prob}(y \xleftarrow{\text{🐞}} Y_1)^2 + \sum_{y\in\mathbb{B}^m}\mathrm{prob}(y \xleftarrow{\text{🐞}} Y_2)^2 - 2\sum_{y\in\mathbb{B}^m}\mathrm{prob}(y \xleftarrow{\text{🐞}} Y_1)\,\mathrm{prob}(y \xleftarrow{\text{🐞}} Y_2))$$

If $Y_2 = U_m$ is uniform we obtain:

$$\sum_{y\in\mathbb{B}^m}\mathrm{prob}(y \xleftarrow{\text{🐞}} Y_1)^2 + 2^{-(d+\ell)} - 2\cdot 2^{-(d+\ell)}\sum_{y\in\mathbb{B}^m}\mathrm{prob}(y \xleftarrow{\text{🐞}} Y_1) = \mathrm{Col}(Y_1) - 2^{-(d+\ell)}$$

If we set now $Y_1 = H(X)\circ H$, we obtain

$$\sum_{y\in\mathbb{B}^m}(\mathrm{prob}(y \xleftarrow{\text{🐞}} H(X)\circ H) - \mathrm{prob}(y \xleftarrow{\text{🐞}} U_m))^2 = \mathrm{Col}(H(X)\circ H) - 2^{-(d+\ell)}$$

$$\le 2^{-(d+\ell)}(\varepsilon^2+1) - 2^{-(d+\ell)} = \frac{\varepsilon^2}{2^{d+\ell}}$$

From the inequality (1.44) we obtain

$$\Delta(H(X) \circ H, U_m) \leq \frac{1}{2} 2^{(d+\ell)/2} \sqrt{\frac{\varepsilon^2}{2^{d+\ell}}} = \varepsilon/2$$

This finishes the proof. □

Notes 1.2. The attack on linear congruential generators is due to Reeds (1977) and Boyar ??.

1.3. The two notions of probability distribution p on a set A and random variable X on A are equivalent in the following sense. From p we get the random variable $X = \mathrm{id}$, as described in the text, and from some X, we get p with $p(a) = \mathrm{prob}(a \xleftarrow{\text{🎲}} X)$. These associations are inverse to each other, that is, starting from some p and taking $X = \mathrm{id}$, we get the distribution p back in the way described. Similarly, for any X, the random variable corresponding to the p which corresponds to X equals X.

A fundamental notion in complexity theory is the complexity class $\mathcal{P}$ of all Boolean predicates (= one-output Boolean functions = languages) which can be computed by a (deterministic) Turing machine in polynomial time. We can also consider the class $\mathcal{P}_{\mathrm{circ}}$ of all such predicates which can be computed by a family $(\mathcal{C}_n)_{n\in\mathbb{N}}$ of Boolean circuits $\mathcal{C}_n$, where $\mathcal{C}_n$ has n inputs and its size is polynomial in n. Then $\mathcal{P} \subset \mathcal{P}_{\mathrm{circ}}$, but the two classes are not identical, because the circuit for n inputs may be constructed in a manner totally different from that for $n-1$ inputs, while a Turing machine has only "one" behavior for all input sizes. This can be mended by stipulating that the circuits $\mathcal{C}_n$ have to be "uniformly constructed" in dependence on n. With the appropriate technical definitions, we have $\mathcal{P}_{\mathrm{circ}}(\mathrm{uniform}) = \mathcal{P}$. Alternatively, we can allow Turing machines a special "advice tape"; this gives the complexity class $\mathcal{P}/\ \mathrm{poly}$, which equals $\mathcal{P}_{\mathrm{circ}}$. For the rather technical details, we refer to ???. If we think of $\mathcal{C}$ as representing an electrical circuit, then the time that a signal takes corresponds to the length of a longest path from inputs to outputs; this is called the depth of the circuit.

1.4. In Definition 1.12 (iii), the predictor actually also has to compute i_k from k, so that i_k depends "uniformly" on k (see Notes 1.3).

Acronyms

AES Advanced Encryption Standard

ATM Automatic Teller Machine

CBC Cipher Block Chaining

CESG Communications-Electronics Security Group

CFB Cipher Feedback

DEC Digital Equipment Corporation

DES Data Encryption Standard

DSA Digital Signature Algorithm

DSS Digital Signature Standard

ECB Electronic Codebook

EFF Electronic Frontiers Foundation

FIPS Federal Information Processing Standard

IBM International Business Machines

IDEA International Data Encryption Algorithm

MARS A candidate cipher for AES. missing long name

MD4 Message Digest 4

MD5 Message Digest 5

NBS National Bureau of Standards

NIST National Institute of Standards and Technology

NSA National Security Agency

OFB Output Feedback

PIN Personal Identification Number

PKCS Public Key Cryptography Standard RSA Inc. issued some of these.

PRG Pseudo Random number Generator

RSA Rivest, Shamir and Adleman Cryptosystem

RC6

SHA Secure Hash Algorithm

SHS Secure Hash Standard

TDEA Triple Data Encryption Algorithm

Bibliography

The numbers in brackets at the end of a reference are the pages on which it is cited. Names of authors and titles are usually given in the same form as on the article or book.

Thomas Beth, Dieter Jungnickel & Hanfried Lenz (1993). *Design Theory*. Cambridge University Press. First edition 1985. [22]

L. Blum, M. Blum & M. Shub (1986). A simple unpredictable pseudo-random number generator. *SIAM Journal on Computing* **15**(2), 364–383. [1]

Ravi B. Boppana & Michael Sipser (1990). The Complexity of Finite Functions. In *Handbook of Theoretical Computer Science*, J. van Leeuwen, editor, volume A, 757–804. North-Holland. [27]

Joan Boyar (1989). Inferring Sequences Produced by Pseudo-Random Number Generators. *Journal of the ACM* **36**(1), 129–141. [4]

Donald E. Knuth (1973). *The Art of Computer Programming, vol.1: Fundamental Algorithms*. Addison-Wesley, Reading MA, 2nd edition. [7]

Donald E. Knuth (1998). *The Art of Computer Programming, vol. 2, Seminumerical Algorithms*. Addison-Wesley, Reading MA, 3rd edition. ISBN 0-201-89684-2. First edition 1969. [3]

J. E. Littlewood (1953). *A Mathematician's Miscellany*. Methuen & Co. Ltd., London, 136. [5]

D. E. Muller (1956). Complexity in Electronic Switching Circuits. *IRE Transactions on Electronic Computers* **5**, 15–19. [27]

John von Neumann (1951). Various techniques used in connection with random digits. Monte Carlo methods. *National Bureau of Standards, Applied Mathematics Series* **12**, 36–38. [35]

Noam Nisan & Avi Wigderson (1994). Hardness vs Randomness. *Journal of Computer and System Sciences* **49**, 149–167. [1]

James Reeds (1977). "Cracking" a random number generator. *Cryptologia* **1**(1), 20–26. [39]

Damien Stehlé (2004). Breaking Littlewood's Cipher. *Cryptologia* **XXVIII**(4), 341–357. [5]

D. Wilson (1979). Littlewood's cipher. *Cryptologia* **3**, 120–121 and 172–176. [5]

Andrew C. Yao (1982). Theory and Applications of Trapdoor Functions. In *Proceedings of the 23rd Annual IEEE Symposium on Foundations of Computer Science,* Chicago IL, 80–91. IEEE Computer Society Press. [14]

Players

The numbers in brackets at the end of a reference are the pages on which it is cited. Names of authors and titles are usually given in the same form as on the article or book.

Lenore Blum (1943-). *?//?, New York, USA. †?//?. URL http://tulsagrad.ou.edu/statistics/biographies/LenoreBlume.htm. [iii, 29, 30, 31, 32, 34]

Manuel Blum (1938-). *26 April 1938, Caracas, Venezuela. †?//?. URL http://en.wikipedia.org/wiki/Manuel_Blum. [iii, 29, 30, 31, 32, 34]

Johan Håstad (1960-). *?//?. †?//?. URL http://www.nada.kth.se/~johanh/cv.pdf. [29]

Richard Wesley Hamming (1915-1998). *11 February 1915, Chicago, Illinois. †7 January 1998, Monterey, California. URL http://en.wikipedia.org/wiki/Richard_Hamming. [7, 8, 17]

Carl Gustav Jacob Jacobi (1804-1851). *10 December 1804, Potsdam, Kingdom of Prussia. †18 February 1851, Berlin, Kingdom of Prussia. URL http://en.wikipedia.org/wiki/Carl_Gustav_Jakob_Jacobi. [29]

Donald Ervin Knuth (1938-). *10 January 1938, Milwaukee, Wisconsin, USA. †?//?. URL http://en.wikipedia.org/wiki/Donald_Ervin_Knuth. [3]

Adrien-Marie Legendre (1752-1833). *18 September 1752, Paris, France. †10 January 1833, Paris, France. URL http://en.wikipedia.org/wiki/Legendre. [29]

John von Neumann (1903-1957). *28 December 1903, Budapest, Austria-Hungary. †8 February 1957, Washington, D.C., USA. URL http://en.wikipedia.org/wiki/Von_Neumann. [2]

Noam Nisan (????). *?//?. †?//?. URL http://www.cs.huji.ac.il/~noam/. [iii, 22, 24, 27, 28, 29]

Claus-Peter Schnorr (1943-). *4 August 1943. †?//?. URL http://de.wikipedia.org/wiki/Claus-Peter_Schnorr. [3]

Michael Shub (1942???-). *?//?. †?//?. URL http://www.math.toronto.edu/shub/. [iii, 29, 30, 31, 32, 34]

Brook Taylor (1685-1731). *18 August 1685, Edmonton, Middlesex, England. †30 November 1731, London, England. URL http://en.wikipedia.org/wiki/Brook_Taylor.

Avi Wigderson (1956-). *9 September 1956. †?//?. URL http://math.ias.edu/~avi/avicv1.pdf. [iii, 22, 24, 27, 28, 29]