Today, we explore the following.

- pseudorandom function collections and applications
- pseudorandom permutations

Last time, we constructed the definition of a pseudorandom function \( f \), which is chosen from the function space \( H_n = \{ f : \{0,1\}^n \rightarrow \{0,1\}^n \} \), \(|H_n| = (2^n)^{2^n}\). Recall that a statistical test for functions \( T^f \) is a probabilistic polynomial time algorithm with black-box access to \( f \), or rather, that can request values \( f(x) \) for any \( x \) it desires.

**Definition 1 (Passing a Statistical Test for Functions)** Then, a collection of pseudorandom functions \( F_n \subseteq H_n, |F_n| = 2^n \) passes a statistical test \( T^f \) if

\[
\forall \text{ polynomials } Q(n), \forall n \text{ sufficiently large, } \left| \Pr_{f \in F_n} [T^f(1^n) = 1] - \Pr_{f \in H_n} [T^f(1^n) = 1] \right| < \frac{1}{Q(n)}.
\]

**Definition 2 (Collection of Pseudorandom Functions)** A collection of functions \( F_n \) is a pseudorandom function collection if it has the following three properties.

- **Indexing** \( \forall n, \exists \text{ unique } i \in \{0,1\}^n \text{ associated with each } f \in F_n. \) (Or rather, there is an \( n \)-bit description for each function in the collection.)
- **Efficiency** \( \exists \text{ probabilistic polynomial time algorithm } A \text{ such that } A(i,x) = f_i(x). \) (Or rather, we can evaluate all the functions in polynomial time.)
- **Pseudorandomness** \( F_n \) passes all statistical tests for functions.

Now, we will attempt to build pseudorandom function collections from one-way functions. One possible attempt would be to use a collection of trapdoor functions; however, while it is indexable by trapdoor and efficiently computable, it is not pseudorandom. Another attempt would be to let \( f_i(x) = x \)th output block of \( G(i) \), where \( G \) is a pseudorandom number generator. However, while we this is indexable by the seed \( i \), we aren’t guaranteed pseudorandomness, let alone efficient evaluation, as \( i \) nears \( 2^n \). However, it is possible to build pseudorandom function collections from one-way functions.

**Theorem 3** If one-way functions exist, then pseudorandom function collections exist.

**Proof** Let \( G(x) \) be a pseudorandom number generator that stretches from \( n \) bits to \( 2n \) bits. Call the output of \( G(x) = G_0(x) \circ G_1(x) \), where \( G_0(x) \) is the first \( n \) bits of the output, and \( G_1(x) \) is the last \( n \) bits of the output (\( \circ \) represents concatenation). We can call \( G \) recursively on both \( G_0(x) \) and \( G_1(x) \), giving us \( G_{00}(x), G_{01}(x), G_{10}(x), \) and \( G_{11}(x) \). Then, we can construct a tree of height \( n + 1 \) and with root \( x \), as shown in Figure 1 below.

This tree has \( 2^n \) leaves \( G_i(x) \) where \( i \in \{0,1\}^n \). Define a function \( f_x(z) = G_x(z) \), where each bit of \( z \) tells us whether to go left (\( z_k = 0 \)) or right (\( z_k = 1 \)) down the tree. Thus, \( f \) is a candidate for a pseudorandom function collection. Let’s check the three properties.

- **Indexing** We can index each instance of the function \( f \) by \( x \), which is the original input to \( G \).
- **Efficiency** Yes, it’s polynomial time, as we have to evaluate \( G \) \( n \) times for each answer.
- **Pseudorandomness** We’ll have to prove this.

**Lemma 4** This candidate passes all statistical tests for functions.
Proof Assume not; that is, there exists a statistical test $T^f$ such that

$$\exists \text{ polynomials } Q(n), \forall n \text{ sufficiently large}, \left| \Pr \left[ T^f \in F_n(1^n) = 1 \right] - \Pr \left[ T^f \in H_n(1^n) = 1 \right] \right| > \frac{1}{Q(n)}.$$

Define the $j$th hybrid in the following manner: $z = z_1 \circ z_2 \circ \cdots \circ z_{j-1} \circ z_j \circ z_{j+1} \circ \cdots \circ z_{n-1} \circ z_n$, where each $z_k$ is a bit, so we could follow $z_1, z_2, \ldots, z_{j-1}, z_j$ through an empty tree, choose a random $r$, fill in the node with $r$, and for $z_{j+1}, z_{j+2}, \ldots, z_{n-1}, z_n$, compute the rest of the tree using $r$, so that $f(z) = G_{z_{j+1}z_{j+2} \cdots z_{n-1}z_n}(r)$. (We can store $r$ so if a test asks the same $z$ over again, we can give the same result.)

Note that running $T^f$ on the 0th hybrid represents running $T^f$ when $f \in F_n$, whereas running $T^f$ on the $n$th hybrid is the case where $f \in H_n$. If $T^f$ can tell apart the 0th and $n$th hybrids, there must exist another algorithm that can tell apart the $j$th and $j+1$st hybrids. Also note, though, that the $j$th hybrid represents placing the output of a pseudorandom number generator in the $j+1$st level, whereas the $j+1$st hybrid represents placing strings chosen completely at random. However, we know that it’s impossible to tell apart the output from a pseudorandom number generator from something completely random (we could prove this as well using a hybrid argument), so an algorithm that could distinguish the $j$th hybrid from the $j+1$st one must not exist, so $T^f$ must also not exist. Thus, our candidate passes all statistical tests for functions.

Since our candidate satisfies the pseudorandomness property, we can say that it is a pseudorandom function collection.

There are many useful applications for collections of pseudorandom functions. For example:

- We can create hash functions to fool an adversary who is trying to create collisions. If we use $F_n = \{ f : \{0,1\}^x \rightarrow \{0,1\}^y \}$, where $x$ is the height of the tree we create, and $y$ is the length of the seed at the root of the tree. (These two parameters don’t necessarily have to be the same.)
- We can create ”identify friend vs. foe” systems. For example, credit card numbers are often based upon personal data like addresses, birthdays, or phone numbers. Credit card customers who know the credit card numbers of some of their friends or neighbors wouldn’t be able to guess more credit card numbers of others just by their personal data, because they don’t know the seed to the pseudorandom function that generates the credit card numbers. In addition, there would be no table-lookup required for the credit card company, because each credit card number is calculable based on information about the credit card holder.
• We can run symmetric encryption. Recall from last time that we did stateless symmetric encryption using a pseudorandom function, where Alice and Bob agreed upon a pseudorandom function \( f \), and Alice sent a ciphertext \( c = (r, f(r) \oplus m) \). This scheme was vulnerable to the chosen ciphertext attack; however, if we use a collection of pseudorandom functions, our scheme is secure under the chosen ciphertext attack.

• We have constructed a simple law with a short description (the index), which is not predictable. Thus, this has many learning theory applications, such as to the block cipher.

Now we’d like to extend this model to give us pseudorandom permutation collections. To do this, we can use the Feistel permutation.

**Definition 5 (Feistel Permutation)** Break up a \( 2n \)-bit string into two parts; the first \( n \) bits we’ll call \( L \) and the last \( n \) bits we’ll call \( R \). Then, the Feistel permutation \( D_f(L, R) = (R, L \oplus f(R)) \), where \( f \) is a member of a collection of pseudorandom functions.

Then, to construct the pseudorandom permutation collection, we can take three Feistel permutations on different pseudorandom functions chosen from a collection: \( D_{f_3}(D_{f_2}(D_{f_1}(L, R))) \).

\[
D_{f_3}(D_{f_2}(D_{f_1}(L, R))) = D_{f_3}(D_{f_2}(R, L \oplus f_1(R)))
\]
\[
= D_{f_3}(L \oplus f_1(R), R \oplus f_2(L \oplus f_1(R)))
\]
\[
= (R \oplus f_2(L \oplus f_1(R)), L \oplus f_1(R) \oplus f_3(R \oplus f_2(L \oplus f_1(R))))
\]

We use the permutation property of the Feistel permutation to ensure that this is a permutation. It fulfills all three properties for pseudorandom function collections as well.

**Indexing** We can index each instance of the function \( f \) by the indexes of \( f_1, f_2, \) and \( f_3 \).

**Efficiency** It’s polynomial time to compute this, considering that each of the \( f \)'s is polynomial time as well. Also note that given the \( f \)'s, our permutation is also invertible.

**Pseudorandomness** Trivial.

Next time, we’ll start off the topic of public key encryption.