CS 810: Complexity Theory 2/26/2007

Lecture 17: Worst-Case to Average-Case Reductions

Instructor: Dieter van Melkebeek Scribe: Tom Watson

In the last lecture we discussed some applications of the Nisan-Wigderson pseudorandom generator, showing that if some language in $L \in \mathrm{E}$ is sufficiently average-case hard for nonuniform circuits, then BPP can be efficiently simulated deterministically, where the efficiency of the simulation depends on the hardness of L. We also introduced the idea of using error correcting codes to allow us to relax our hypothesis from the existence of an average-case hard language to the existence of a worst-case hard language, and we described a local decoding procedure for Reed-Muller codes. Today we will discuss the paradigm of list decoding and describe a local list decoding algorithm for the Hadamard code. Then we will show how local list decoding algorithms can be used to obtain very strong worst-case to average-case hardness reductions in E. Finally, we will introduce the notion of randomness extraction.

1 Worst-Case to Average-Case Reductions via Error Correcting Codes

1.1 Local List Decoding

Suppose there is a language $L \in \mathrm{E}$ such that no family of small circuits can compute L. Our goal is to show that then there exists another language $L' \in \mathrm{E}$ such that not only can no family of small circuits compute L', but no family of small circuits can even succeed in computing L' on noticeably more than half the inputs. Our strategy is to consider the characteristic sequence $\chi_{L|_m}$ at some input length m, which is a string of length 2^m, and encode it using a good binary error correcting code (ECC) to obtain a string of length $2^{m'}$ for some m' that's not too much larger than m. We will then interpret the encoding of $\chi_{L|_m}$ as the characteristic sequence $\chi_{L'|_{m'}}$ of some other language L' at input length m'.

The intuition is that if $L'|_{m'}$ can be solved on noticeably more than half the inputs by a small circuit, then the characteristic sequence of the function computed by this circuit can be viewed as a corrupted version of $\chi_{L'|_{m'}}$. Then the original information word $\chi_{L|_m}$ could be obtained by an efficient decoding procedure, allowing us to solve $L|_m$ on all inputs.

Our ECC needs to have the following properties.

(1) We want to argue that if L' is not sufficiently hard, then we can construct a small circuit computing L. This circuit will be given a string x of length m and required to compute the bit of $\chi_{L|_m}$ corresponding to that string. However, the "received word" $\chi_{L'|_{m'}}$ is exponentially longer than x. This seems to be a problem since traditional decoding algorithms need to look at the entire received word, even to compute a single bit of the information word. We get around this by using a *local decoding* algorithm, which, given the index (x, in our case) of a bit of the information word, computes that bit while only making a few randomized queries to the received word. In the last lecture we described local decoding algorithms for the Hadamard and Reed-Muller codes.

(2) Since we would like L' to be hard to solve on even a little more than half the inputs, we will need to be able to decode a fraction of errors that is as large as $1/2 - \epsilon$ for some small ϵ. This seems problematic since to be able to correct a fraction η of errors, we need the (relative) minimum distance of the ECC to be greater than 2η. This would imply that we need an ECC with minimum distance close to 1, which is a problem since, although we will not argue it, a distance of $1/2$ is the best one can hope for in the case of binary codes. We will get around this by using a *list decoding* algorithm, which, given a received word, computes all information words whose encodings are within a certain distance from it. Since we are dealing with circuits, we will be able to then nonuniformly select which information word is the correct one. Naturally, as the fraction η gets closer to half, the number of codewords within distance η grows, but we will be able to show that this number does not grow too large. We will describe a list decoding algorithm for the Hadamard code today.

(3) Finally, we will require an efficient encoding procedure for our ECC. We will need to show that if L is in E, then L' is also in E, and this will be shown by combining an exponential-time algorithm for L with an efficient encoding procedure for the ECC. In fact, we would like m' to be only a constant factor larger than m. This will allow us to show that the average-case hardness of L' is quite comparable to the worst-case hardness of L. Our ultimate goal is to show that if L has linear exponential worst-case circuit complexity, then L' has linear exponential average-case circuit complexity, since as we saw previously, the existence of such an L' allows us to obtain a quick pseudorandom generator with logarithmic seed length and conclude that BPP = P.

Properties (1) and (2) indicate that we will need an ECC with an efficient *local list decoding* algorithm. We will not give the full details of the necessary construction. All the desired properties can be realized for the concatenation of the Hadamard code with the Reed-Muller code. A local list decoder for the Reed-Muller code can be obtained using the local decoding approach discussed in the last lecture, combined with a list decoder for the Reed-Solomon code. The latter decoder involves a technical procedure for factoring bivariate polynomials. Today we describe a local list decoding procedure for the Hadamard code. The local list decoders for these two codes can easily be combined to form a local list decoder for the concatenation. In Section 1.4, we will show how this leads to the desired worst-case to average-case reduction.

1.2 Error Correcting Code Constructions

Recall that the Hadamard code is a $[2^K, K]_2$ code that encodes an information word $a \in \{0,1\}^K$ as the codeword $((a,x))_{x\in\{0,1\}^K}$. That is, it takes the inner product of a with every bit string of length K and outputs the list of all 2^K results. The minimum distance of this code is $1/2$, which in the case of binary codes is the best distance one can hope for asymptotically. Although it has a very good minimum distance, the Hadamard code has a horrible rate and is of no practical value. However, it is useful in complexity theory. This code can handle up to a $1/4$ fraction of errors if we require unique decoding. This is not good enough to achieve the strong hardness results we're after. We will show that by contenting ourselves with list decoding, we will be able to handle a fraction of errors that is almost half.

The Reed-Solomon code is another useful ECC. It has a minimum distance that is close to 1. However, it is not locally decodable — we need to query at least K positions of the received word in order to reconstruct even one position of the information word. Furthermore, the Reed-Solomon code requires that the field size be at least as large as the codeword length, which is undesirable since we are interested in codes over $GF(2)$. However, we can handle this by concatenating with a binary code such as the Hadamard code.

The Reed-Muller code is one for which we do have a good local decoding algorithm. Recall that Reed-Solomon encoding is achieved by interpreting the information word as a low-degree univariate polynomial and evaluating it at all points of the field, while Reed-Muller encoding is achieved by interpreting the information word as a low-degree polynomial in ℓ variables (say) and evaluating it at all ℓ-tuples of field elements. Intuitively, we are packing the information into an ℓ-dimensional cube. We can then locally decode by picking a random line through the point in the cube where we want to evaluate the original polynomial, querying the received word at all points along this line, and using a Reed-Solomon decoder to reconstruct a univariate polynomial corresponding to the original polynomial restricted to this line. Using a similar approach together with a list decoder for the Reed-Solomon code, a local list decoder for the Reed-Muller code can be obtained. We will not explore this result in this course.

Finally, concatenating the Reed-Muller code with the Hadamard code yields a code that satisfies our desired properties. Concatenating with the Hadamard code allows us to get a binary code, and the exponential blow-up of the Hadamard code is compensated for by the fact that the field size required by the Reed-Muller code doesn't grow too fast. The locally decodability property is the key for getting a worst-case to average-case reduction The list decodability property is the key for overcoming the upper bound of 1/2 on the distance of the code in order to acheive very strong worst-case to average-case reductions.

1.3 Local List Decoding of the Hadamard Code

We develop a local list decoding algorithm for the Hadamard code. We will not need to worry about the local decoding aspect; this will be a natural feature of our algorithm. Given a received word $r \in \{0,1\}^{2^K}$ and an error bound η, we wish to find a list of all information words whose encodings are within Hamming distance at most η from r. If $\eta < \delta/2$, where δ is the minimum distance of the code, then this list can contain at most one information word. As η gets larger, the list will also naturally get larger, but we want to show that it does not get too large. Specifically, we wish to be able to handle up to a $1/2 - \epsilon$ fraction of errors in randomized time $poly(\frac{K}{\epsilon})$. Since the received word is of length 2^K, we clearly need random access to it.

Theorem 1. *There is a randomized algorithm that, given random access to a received word $r \in \{0,1\}^{2^K}$, runs in time $poly(\frac{K}{\epsilon})$ and outputs a list of information words that with high probability contains all $a \in \{0,1\}^K$ such that*

$$Pr_{x\in\{0,1\}^K}\left[(a,x) = r(x)\right] \geq \frac{1}{2} + \epsilon.$$

That is, it outputs a list of all information words whose Hadamard encodings are at relative distance at most $1/2 - \epsilon$ from r.

Proof. We focus on obtaining one particular $a \in \{0,1\}^K$ satisfying $Pr_x[(a,x) = r(x)] \geq \frac{1}{2} + \epsilon$. By running the procedure a few more times and concatenating the lists, we can get a list that with high probability contains all information words having the desired level of agreement.

Recall that in the last lecture we described a local unique decoder for the Hadamard code. The idea was to retrieve the ith bit of a by picking a random x and querying $r(x)$ and $r(x + e_i)$ where

e_i is the string with a 1 in the ith position and 0's elsewhere. Since we were assuming that the encoding of a differed from r in at most a fraction of $1/4 - \epsilon$ positions, we were able to conclude by a union bound that with probability at least $1/2 + 2\epsilon$, both x and $x + e_i$ were positions where r was correct, in which case $r(x) + r(x + e_i) = a_i$. By picking several x's independently, and taking the majority vote of the values $r(x) + r(x + e_i)$, we were able to obtain the correct a_i with high probability. In the present settting, however, r may be wrong in a fraction of $1/2 - \epsilon$ positions, so we are only able to conclude that with probability at least 2ϵ, both x and $x + e_i$ are positions where r is correct. Thus obtaining a_i using this idea would require too many samples x. We will now describe a more elaborate approach that uses the power of list decoding to reduce the number of samples needed.

We focus on retrieving one particular component a_i of the information word a. Consider the following idea. Select $x_1, x_2, \ldots, x_t \in \{0,1\}^K$ uniformly at random (for some small t to be determined later), and obtain $2^t - 1$ strings by adding together all possible combinations of the x_j's (except the empty combination). More formally, for the $2^t - 1$ nonzero values of $c \in \{0,1\}^t$, take the string

$$y_c = \sum_{j=1}^{t} c_j x_j.$$

Note that c_j is one bit of c, whereas x_j is a string of length K. Then y_c is just the sum of some of the x_j's, namely those corresponding to the locations of the 1's in c. Now since c is nonzero, it follows that $e_i + y_c$ is uniformly distributed. The proof of the following claim follows since the event under consideration holds whenever $e_i + y_c$ is an index of a position where r agrees with the encoding of a.

Claim 1. *For all nonzero c, $Pr[(a, y_c) + r(e_i + y_c) = a_i] \geq 1/2 + \epsilon$.*

Thus *if we knew the values (a, y_c) for all c*, then we would be able to pick an arbitrary c, query $r(e_i + y_c)$, add (a, y_c) to the result, and conclude that we had a_i with confidence at least $1/2 + \epsilon$, which is much better than the 2ϵ we got in our first attempt. In other words, we are circumventing the inherent unreliability of the two-query approach by assuming we always have the "correct" answer for one of the two queries. We will show later how we can handle this hypothesis using the power of list decoding.

Another issue we need to handle is that we would naturally like to boost our confidence by making not one, but several queries to r. We can just use $e_i + y_c$ for all choices of c, and take the majority vote of the values $(a, y_c) + r(e_i + y_c)$. There are $2^t - 1$ choices for c, and one might wonder if this means we would have to make too many queries. However, it turns out that t can be chosen small enough that this isn't a problem. A more important issue to be alarmed about is the fact that these $2^t - 1$ strings are definitely not fully independent; after all, they were generated using only tK bits of randomness. However, they are *pairwise independent.* To see this, note that if $c_1 \neq c_2$ then there is an index j where they differ. Since c_1 and c_2 are nonzero we have

$$Pr[y_{c_1} = z_1] = Pr[y_{c_2} = z_2] = \frac{1}{2^K},$$

and the equality

$$Pr\big[(y_{c_1} = z_1) \wedge (y_{c_2} = z_2)\big] = \frac{1}{2^K}$$

can be seen by conditioning on the values $x_1, \ldots, x_{j-1}, x_{j+1}, \ldots, x_t$.

It turns out that pairwise independence is good enough.

Claim 2. *For all i,*

$$Pr[MAJ_i \neq a_i] \leq \frac{1}{2^t \epsilon^2}$$

where $MAJ_i = majority(b_{i,c} : c \neq 0^t)$ and $b_{i,c} = (a, y_c) + r(e_i + y_c)$.

Proof. Let $X_{i,c}$ be the indicator random variable for the event that $b_{i,c} = a_i$, i.e. our guess for a_i is correct when we use c. In order for $MAJ_i \neq a_i$ to happen, it needs to be the case that

$$\frac{1}{2^t - 1} \sum_{c \neq 0^t} X_{i,c} \leq \frac{1}{2}.$$

However, by Claim 1 we know that $E[X_{i,c}] \geq 1/2 + \epsilon$ for all $c \neq 0^t$, and thus $E[\frac{1}{2^t-1} \sum_{c \neq 0^t} X_{i,c}] \geq 1/2 + \epsilon$ by linearity of expectation. It follows that

$$Pr[MAJ_i \neq a_i] \leq Pr\left[\left|\frac{1}{2^t - 1} \sum_{c \neq 0^t} X_{i,c} - E\left[\frac{1}{2^t - 1} \sum_{c \neq 0^t} X_{i,c}\right]\right| \geq \epsilon\right].$$

Since the $X_{i,c}$'s are pairwise independent for each i, we can apply Chebyshev's inequality and the fact that every indicator random variable has variance at most 1/4 to conclude that

$$\begin{aligned} Pr[MAJ_i \neq a_i] &\leq \frac{\sigma^2\left(\frac{1}{2^t-1} \sum_{c \neq 0^t} X_{i,c}\right)}{\epsilon^2} \\ &= \frac{1}{(2^t - 1)^2} \frac{\sum_{c \neq 0^t} \sigma^2(X_{i,c})}{\epsilon^2} \\ &\leq \frac{1}{(2^t - 1)^2} \frac{2^t - 1}{4\epsilon^2} \\ &\leq \frac{1}{2^t \epsilon^2}. \end{aligned}$$

□

Now by a union bound, the probability that there exists an i such that $MAJ_i \neq a_i$ is at most $\frac{K}{2^t \epsilon^2}$. This can be made at most 1/2 by choosing

$$t = \Theta\left(\log \frac{K}{\epsilon^2}\right).$$

To summarize the algorithm up to this point, we first choose $x_1, \ldots, x_t \in \{0,1\}^K$ uniformly at random. We then recover each bit a_i separately (this will allow for local decoding) by forming the $2^t - 1$ queries $r(e_i + y_c)$ corresponding to different c's, for each query guessing that a_i equals $b_{i,c} = (a, y_c) + r(e_i + y_c)$, and taking the majority vote over all these guesses $b_{i,c}$. As argued above, we succeed in recovering the information word a correctly with probability at least 1/2 over the choice of $x_1, \ldots, x_t$. Since recovering each position a_i involves $2^t - 1 = poly(\frac{K}{\epsilon})$ queries, the entire procedure runs in time $poly(\frac{K}{\epsilon})$, as desired.

However, there is one critical issue we have yet to resolve — the entire procedure assumed we knew the values (a, y_c) for all c, which seems ridiculous since a is what we're trying to find! This is where list decoding comes in: since our algorithm needs a sequence of values $((a, y_c))_{c \neq 0^t}$, we can try all possibilities for this sequence, run our algorithm for each possibility, and output the list of all information words obtained. Then with probability at least 1/2, a appears on the list (namely, it appears when we try the correct values for the sequence $((a, y_c))_{c \neq 0^t}$).

There is a problem with this, however: there are $2^t - 1$ different c's, leading to $2^{2^t - 1}$ possibilities and making the running time exponential in $\frac{K}{\epsilon}$. This is easily remedied by recalling that the inner product is *linear*, and so

$$(a, y_c) = (a, \sum_{j=1}^{t} c_j x_j) = \sum_{j=1}^{t} c_j (a, x_j).$$

Thus the $2^t - 1$ values (a, y_c) are uniquely determined by the t values (a, x_j). It follows that we can reduce our list to size $2^t = poly(\frac{K}{\epsilon})$, since we only need to try all possible values for the sequence $((a, x_j))_j$. The overall running time remains $poly(\frac{K}{\epsilon})$.

This explains why we didn't choose $2^t - 1$ strings x independently but rather chose t strings and looked at all combinations of them: with the former approach our list would have been too long — on the order of 2^{2^t} entries — whereas with the latter approach we can get by with a list size of 2^t at the expense of having our $2^t - 1$ samples be only pairwise independent, which as we argued above, is not a big problem. $\square$

As with the local decoding algorithm discussed in the last lecture, the basic idea of the above proof is to try to recover a_i by querying a random location in the received word, querying the location whose index has the ith bit flipped, and XORing the results. In the present setting there are too many errors in the received word for this to be reliable. One key idea is that we can drastically increase the reliability *if we know that one of the two queries is uncorrupted*. Since unique decoding is not required, we can try all possibilities for the "correct" values of these queries. The other key idea is that we can keep the list size small by only choosing a small number of query locations, and deterministically generating the rest of the query locations by adding together all possible combinations. Chebyshev's inequality allows us to conclude that the reliability doesn't deteriorate too much when we do this.

As a corollary to the above result, we note that for all received words r, the number of information words a whose Hadamard encodings agree with r on at least a $1/2 + \epsilon$ fraction of positions is bounded by $poly(\frac{K}{\epsilon})$, the running time of the algorithm.

Finally, we note that the list output by our decoding algorithm may contain information words whose encodings do not have the $1/2 + \epsilon$ agreement with the received word. We can try to weed these out by randomly querying r to ensure that with high probability, r agrees with the encoding in at least a fraction $1/2 + \epsilon - \epsilon'$ locations, for some small ϵ'.

1.4 Worst-Case to Average-Case Reductions

With the Hadamard decoder described in the previous section and a local list decoder for the Reed-Muller code, one can obtain a local list decoder for the concatenation of the two codes. The precise result is stated below, without proof.

Theorem 2. *For each $\epsilon > 0$ and K there exists an error correcting code with the following properties. There is a polynomial-time encoder mapping information words of length K to codewords of length $poly(K)$. The codeword length can be assumed to be a power of 2 when K is. There is* *a randomized algorithm that runs in time $poly(\frac{K}{\epsilon})$ and outputs a list of randomized oracle Turing machines $M_1, M_2, \ldots$ that take as input a position in the information word, have oracle access to the received word, and run in time $poly(\frac{\log K}{\epsilon})$. These machines have the property that for all received words r and all information words w such that r agrees with the encoding of w in at least a fraction $1/2 + \epsilon$ positions, there exists an i such that M_i^r computes w.*

We now show how to use Theorem 2 to get worst-case to average-case reductions.

Theorem 3. *For every $L \in \mathrm{E}$ there exists a language $L' \in \mathrm{E}$ such that*

$$H_{L'}(m) \geq \frac{C_L(m)^{\Omega(1)}}{m^{O(1)}}.$$

Proof. Let L be a language in E. Applying the ECC from Theorem 2 to $\chi_{L|_m}$ yields a string of length $2^{m'}$ for some $m' = O(m)$. We define $\chi_{L'|_{m'}}$ to be this string. We can solve L' in E by taking an input of length m', computing the corresponding length $m < m'$, explicitly writing out $\chi_{L|_m}$, encoding it, and extracting the bit corresponding to our input. Writing out $\chi_{L|_m}$ takes $2^{O(m')}$ since there are 2^m positions, each of which can be computed in $2^{O(m)}$ time since $L \in \mathrm{E}$. Encoding $\chi_{L|_m}$ takes $2^{O(m')}$ time since the ECC from Theorem 2 is polynomial-time encodable. Solving L' incurs an exponential factor blowup in running time, which doesn't take us out of E, but does prevent us from using this technique to get worst-case to average-case reductions for smaller classes.

We will show that

$$H_{L'}(m') \geq \frac{C_L(m)^{\Omega(1)}}{m^{O(1)}},$$

and the theorem will follow from the fact that $m' = O(m)$ (and the fact that $C_L(m)$ is at most exponential, and so changing the input length by a constant factor only makes $C_L(m)$ change by a polynomial factor).

Set $\epsilon = 1/C_L(m)^\alpha$ for some α to be determined later. Suppose there exists a circuit of size s' that, given an input of length m', computes the corresponding bit of $\chi_{L'|_{m'}}$ for at least a $1/2 + \epsilon$ fraction of inputs. Now for some i, the machine M_i^r from Theorem 2 takes an input of length m and outputs the corresponding position of $\chi_{L|m}$ with high probability, i.e. it solves L, provided r is a string of length $2^{m'}$ that agrees with $\chi_{L'|_{m'}}$ on at least a $1/2 + \epsilon$ fraction of positions. This probability may be amplified so that there there is some choice of randomness for which M_i^r solves L on all inputs of length m. By hard-wiring this choice of randomness, we can obtain an oracle circuit of size $(\frac{m}{\epsilon})^{O(1)}$ solving L at input length m. All its oracle gates are queries to $\chi_{L'|_{m'}}$ and can thus be replaced by our hypothesized circuit of size s'. By Theorem 2, this circuit of size $(\frac{m}{\epsilon})^{O(1)} \cdot s'$ computes L exactly provided the oracle subcircuit solves L' on at least a $1/2 + \epsilon$ fraction of inputs, which it does by hypothesis. We conclude that

$$C_L(m) \leq \left(\frac{m}{\epsilon}\right)^\beta \cdot s',$$

for some constant $\beta \geq 1$ (and sufficiently large m). It follows that every circuit of size less than

$$\frac{C_L(m)^{1-\alpha\beta}}{m^\beta}$$

succeeds in computing L' at length m' for less than a

$$\frac{1}{2} + \frac{1}{C_L(m)^\alpha}$$

fraction of inputs. This implies that

$$H_{L'}(m') \geq \min\Big(C_L(m)^\alpha, \frac{C_L(m)^{1-\alpha\beta}}{m^\beta}\Big).$$

If $1 - \alpha\beta < \alpha$, i.e. $\alpha > \frac{1}{\beta+1}$, then the second term definitely dictates the minimum, so choosing $\alpha < \frac{1}{\beta}$ gives the desired result

$$H_{L'}(m') \geq \frac{C_L(m)^{\Omega(1)}}{m^{O(1)}}.$$

□

Corollary 1. *If there exists a language* $L \in \mathrm{E}$ *with* $C_L(m) \geq m^{\omega(1)}$ *then there exists a language* $L' \in \mathrm{E}$ *with* $H_{L'}(m) \geq m^{\omega(1)}$ *and thus there exists a quick PRG with subpolynomial seed length, implying that* $\mathrm{BPP} \subseteq \mathrm{SUBEXP}$.

Corollary 2. *If there exists a language* $L \in \mathrm{E}$ *with* $C_L(m) \geq 2^{m^{\Omega(1)}}$ *then there exists a language* $L' \in \mathrm{E}$ *with* $H_{L'}(m) \geq 2^{m^{\Omega(1)}}$ *and thus there exists a quick PRG with polylogarithmic seed length, implying that* $\mathrm{BPP} \subseteq \mathrm{DTIME}(n^{\log^{O(1)} n})$.

Corollary 3. *If there exists a language* $L \in \mathrm{E}$ *with* $C_L(m) \geq 2^{\Omega(m)}$ *then there exists a language* $L' \in \mathrm{E}$ *with* $H_{L'}(m) \geq 2^{\Omega(m)}$. *and thus there exists a quick PRG with logarithmic seed length, implying that* $\mathrm{BPP} = \mathrm{P}$.

2 Randomness Extraction

We have seen evidence that randomness is not very powerful in terms of reducing the complexity of solving decision problems. We have seen an unconditional pseudorandom generator that fools space-bounded computations, and a conditional pseudorandom generator that fools time-bounded computations under the hypothesis that there exists a language in E requiring linear exponential size circuits. It is conjectured that using randomness can only lead to a polynomial factor savings in time and a constant factor savings in space. This does not mean that randomness is useless in practice. On the contrary, a quadratic speedup achieved with randomness may be very attractive in practice. Additionally, many randomized algorithms are simpler and easier to implement than deterministic algorithms for the same problems.

We now turn to a different question. Most randomized algorithm assume access to a perfect source of unbiased, and more importantly uncorellated, random bits. How do we run such algorithms with access to an imperfect random source? The goal of randomness extraction is to take samples from a weak random source — one where samples may not be uniformly distributed — and generate samples that are close to being uniformly distributed. Such weak random sources will be our models for physical sources of randomness, such as keystrokes or delays over networks.

An *extractor* is an efficient procedure for taking a sample from such an imperfect source and "extracting" the randomness from it, producing an output string that is shorter but much closer to being uniformly distributed. Such a procedure can be used to run randomized algorithms with weak random sources. The fact that the output distribution of an extractor is only "close" to uniform will have only a small effect on the output distribution of the randomized algorithm.

Before we embark on the task of constructing an extractor, we need to formalize what we mean by the "amount of randomness" contained in our weak random source, and by "closeness to uniform" of the output distribution obtained by applying our extractor to our weak random source. For the latter, we will use the standard measure of statistical distance. For the former, one idea is to use the measure of entropy from physics.

Definition 1. *The* entropy *of a discrete random variable X is*

$$H(X) = E\Big[\log\frac{1}{p_i}\Big] = \sum_i p_i \log\frac{1}{p_i}$$

where the sum is over the range of X, and $p_i = Pr[X = i]$.

However, this measure of randomness does not work in our setting. Indeed, suppose that the range of X is $\{0,1\}^m$ and that for nonzero $x \in \{0,1\}^m$, $p_x = 2^{-(m+1)}$, and the rest of the probability is concentrated on 0^m. Then the entropy measure indicates that X has a fair amount of randomness, but X is useless for simulating a BPP algorithm — if 0^m is in the bad set for a particular input, then the probability of error on that input is greater than half!

Instead, we will require that for X to have a large "amount of randomness", it must be the case that no string is given too much weight. This suggests the following measure.

Definition 2. *The* min entropy *of a discrete random variable X is*

$$H_\infty(X) = \min_i \log\frac{1}{p_i}.$$

Equivalently, $H_\infty(X)$ is the largest value of k such that all outcomes have probability at most 2^{-k} under X.

We will say that a source X with $H_\infty(X) \geq k$ has at least k bits of randomness.

Our goal is to construct extractors such that given a source with min entropy at least k, the output distribution of the extractor is statistically close to the uniform distribution on strings of length as close to k as possible. A good extractor can be obtained by viewing the input sample as the characteristic string of a function and using this function in the Nisan-Wigderson pseudorandom generator construction. We will see more details about this construction in the next lecture.