[SQUEAKING]

[RUSTLING]

[CLICKING]

ANKUR MOITRA: These are fairly challenging topics because we're really just going to be proving one theorem today. But this theorem, which is called Shannon's noiseless coding theorem, will involve a lot of the probabilistic tools that we've developed earlier in the class. So I'll give you an introduction to what data compression is and really how Shannon's work put this on a rigorous mathematical foundation.

One of the cool historical facts about what I'm going to be teaching you about today is that it literally happened in this building pre the renovation. But the theorems I'm going to tell you were all laid out right here, not down the hall in a different building, literally right here in Building 2. So this will be cool.

So let me start off by giving you some historical context for the lead-in for this. So how many people here use text messages? OK, should be everyone. You're all paying attention. That's good. Of course, when we're texting each other, we use short forms, like BRB. In fact, every once in a while, I'll see a crazy acronym that I don't, but maybe you guys know.

And so you can think that the acronyms, especially as you get older and you stop being as plugged into it, at some point, you have no idea what people are talking about anymore. But this isn't even a new phenomenon. So let me give you some historical context.

Back in the day, a long, long time ago, the way that people used to send messages was through telegrams. And the problem with telegrams was that they cost a lot of money. So we're talking about the cost $1 per word. So people wanted to send each other telegrams. But the same way that when we have text messages, we have common phrases and strings that we say over and over again, so we figure out the short form, well, here, they would create some of code book to compress communication.

And some of these things were pretty obscure things. You can go back through and look at the telegram books from way back in the day. And you can find lots of gems in these code books, like if you send someone a telegram that had this word, which I can't pronounce, what it meant was it translated to the phrase has not been reinsured. That's a pretty technical concept.

They also had another one, A-Z-K-H-E. This word, if you sent it through a telegram back in the day, this would get translated to clean bill of health. So all of these phrases are saving you money because, instead of sending this many words in this phrase, I'm getting away with sending just a single word for a single dollar.

Some of these really go off the rails. So this is my favorite one when I look through these telegram books-- NBET. So what did this mean? This meant captain is insane. Hopefully, that one doesn't come up that often. But that really was one of the phrases in these code books from way back when.

So what we're going to talk about today is really the mathematical foundation of what it means to compress data. And even better, we're going to talk about what it means to find the best compression for data. So I'm going to give you some examples. I'll give you some definitions. And we'll think intuitively about what a good compression ought to look like. And we're going to be working up to this beautiful theorem, which is called Shannon's noiseless coding theorem, that's going to tell us the best possible way to do data compression.

So let's start off with the key definition at least that kicks it off. And as I mentioned, this happened all in this building. So in 1948, in this building, Claude Shannon founded the field of information theory. And this field, you can take whole graduate seminars on it. It's a beautiful, rich, thriving field. He's really one of the mathematical heroes of this entire area of modern research.

And we're going to tell you about his first beautiful theorem. In the lecture on Tuesday, I'm going to tell you about an amazing result that was proved by a graduate student here on an exam. So you guys all took an exam on Thursday. And it turns out that the theorem I'm going to tell you about on Tuesday was proved by a graduate student who was given a question for his project. And he invented something that we still teach to this day. It's amazing and beautiful. And then in the third lecture, we're going to talk about Shannon's second main theorem.

So first, let's define what we mean by our data source. This is the source of what we want to compress. And you can consider much more complicated sources. But let's keep it simple for today. So a first order source is described by the following things. So first of all, there's an alphabet, which is just the set of all possible symbols. And we'll denote that by capital A. And the individual symbols are a1, a2, all the way up to ak.

And then the other component we have is that we have a distribution, which is going to be a distribution p on the alphabet. So it'll tell you the probability of the first symbol. It'll tell you the probability of the second symbol, all the way up to the k-th symbol. So all this first order source is it's a k-sided unfair die.

You roll the die. And whatever symbol from your alphabet comes up on top, that's your next symbol in your message. So what happens is that the first order source is going to define a long string that we just take by taking IID samples. And so the way that this works is it's going to generate length n strings where each symbol is IID from this associated distribution.

So the way that I'm constructing a long phrase is I take my k-sided die that's unfair. I roll it n times. And then I just concatenate all of those letters in my alphabet together, keeping track the order in which I rolled them. Now, you can think about the alphabet as being composed of letters. But really, you could think about it as being composed of words, too.

Maybe the way that I create a sentence is by rolling my die and getting my first word, rolling my die, and getting my second word. This would be a terrible model for English language. And there are a lot richer models. You can do things like Markov chains, where a lot of this theory is going to carry over. But we're going to keep it simple and just talk about this as our data source.

So I'm trying to send a message through a telegram to someone else. And the way I decide on what message I want to send is I use my first order source to generate it. And then I care about how short I can make my communication be to transmit that message from my source. So are there any questions about the setup? Do people understand what a first order source is and what the communication problem is?

OK, so as I mentioned, there are more complex sources, like Markov chains. But now, let me define precisely what I mean by a codebook and a coding scheme. So what is a strategy for compressing communication? This will be our second key definition.

So we'll say that a coding function, phi, it's a function that maps symbols of length n from my alphabet, which are just things generated from my first order source, to 0, 1 strings. Now, if you haven't seen this notation before, this notation star here means that I can take any length I want.

So I'm mapping anything that my first order source generates to a string of a variable length. But it's composed of 0's and 1's. So maybe one of my messages gets mapped to the 00 string. Maybe one of my messages gets mapped to the 111 string. So in fact, the length of the output doesn't need to be fixed across the different messages. And that's exactly where we're going to win because somehow the intuition is that more frequent messages should be mapped to shorter strings.

Now, first of all, I need not just that this coding function maps every message to an output string because that wouldn't help me. What if I mapped everything to the string 0 or everything to the string the empty set? That wouldn't help me decode and figure out what the original message is.

So we need the property that we can successfully decode. In particular, what we want is that for any pair of different messages that are generated by our first order source, we want that their image under this mapping is not the same. So that's the key property because what this really means is just that we can decode.

So this part is a little bit subtle. Later on, when we talk about Huffman codes on Tuesday next week, we're going to have some constraints which are called prefix-free constraints that we're really going to talk about a coding function that maps symbol to symbol-- symbol to sequence.

But here, the way we're thinking about it is n is gigantic and it's fixed. And then what happens is I decide on my coding function phi. And I'm just going to one-shot transmit some variable length message. And at the end of the message, we say END, termination over. And then the key point is just that things which are different get mapped to different outputs. And we see where the actual thing ends. And then we want that we can recover x and y from this. So are there any questions about what a first order source is or what a coding function is? Does this make sense? Yeah? OK, all right.

So let's do an example. And really, our main question is going to be this question we're going to address at the end with Shannon's theorem. So what is the best coding function? And I even have to define what I mean by best here. But let's do an example to make all of this explicit.

Let's say that our first order source has two letters. So our alphabet is of size 2. And these two letters are a1 and a2. And we're going to have p1 is equal to 7/8 and p2 is equal to 1/8. So already here, there's a huge asymmetry because the first letter in my alphabet is way more likely than my second letter. And I'm going to set n equals 2. So I'm looking at length 2 messages from my first order source.

And what are all of the messages that I could want to transmit to someone? I could want to transmit a1 a1, a1 a2, a2 a1, and a2 a2. Each of these messages has a different probability, like a1 a1 is much more likely than the others. It has probability 49 over 64. This is 7 over 64, 7 over 64, and finally 1 over 64. So the chance that I get a2 a2 is very unlikely.

And now, what we can do is we can think about candidate codebooks and coding functions. So let's do the naive thing first and see how good the naive thing is. So the naive thing is just to map the symbol a1 to 0 and a2 to 1. So in this case, when I get my code, it's just going to output 00 because I literally do it just symbol by symbol.

a1 gets replaced with 0. a2 gets replaced with 1. My code would be 01 here, 10 here, and 11. And now, I should define what I mean by the best coding scheme. What we care about is how much money I pay on average with this coding scheme. And the cost of my transmission, the same way it is for things like telegrams, it will be some amount per symbol in this case, not per word.

So we can compute the expected length of this particular encoding. Notice that the expected length is a property of the first order source and the choice of the coding function. So what is the expected length? Well, we can do this. This is a very easy computation. What's the probability I output the first symbol, and then I output a length 2 message?

What's the probability I have the second message? And again, I output a length 2 message. What's the probability I output the third message from my first order source and so on. So this is a very easy computation to do because it's just 2. No matter what my first order source outputs, I always pay two symbols. So I claim this is not optimal. So can we do better?

Anyone have any intuition for how I could design a better codebook for this particular sequence? This one's not too bad because there's really one dominant event, which is the event that my first order source outputs a1 a1. So originally, when that happened, I outputted a length 2 code. I want to do better. So any ideas? Yeah?

AUDIENCE: [INAUDIBLE]

ANKUR MOITRA: Yeah, like 0, for example. So in this case, for example, my new codebook-- well, I'll have my message a1 a1, a1 a2, a2 a1, a2 a2. In this case, I could output 0. But now, what I need is I need to-- I could do-- let's just do something else for this. I could do 10, 110, and then 111. I'm not trying to get optimality yet. I'm just trying to get an improvement. And we can calculate the new expected length.

And now, with this very large probability that I output the first message, well, I'm only paying 1 with this. I can add up 7 over 64. I'm still paying 2. And then on these other messages, I'm losing because now this third message, a2a1, instead of its length being 2, its length is 3. And on this last message, I'm losing again because it used to be 2 and now it's 3.

But if you work it out, what this works out to be is 87 over 64, which indeed is less than 2. So this was an example where the naive encoding we could improve upon it. And we showed that we could do better. But really, the question now is, what's the best that we can do?

So it turns out that there's a very clean and elegant answer to what's the best we can do. And it's related to really the first key notion from information theory, which is called the entropy. So let me just define what this is. And then we'll talk about Shannon's theorem that we're going to prove.

So the binary entropy of a distribution-- in our case p, which is given by probabilities p1, p2, up to pk-- well, we write it as h of p1 up to pk. And it's the following expression. It's the negative sum from i equals 1 to k of pi log pi.

So in particular, you have to be careful with the negative signs here. So this log is base 2. This pi is less than 1. So this term inside here is negative. But we're fixing it with the negative outside. So the entropy is something that's non-negative. And what I claim is that this simple definition right here is the key to understanding what is possible in terms of data compression and what the optimal scheme is. And we're going to unravel that as we prove Shannon's theorem.

So let me now tell you Shannon's theorem. It comes in two parts. So we fix a first order source exactly as we've been doing. And let's say that this first order source has entropy H. We just plug in whatever our probabilities are that define the first order source into this binary entropy function that I defined right here. And let's say that the result we get is this numerical value H that's some constant.

And what I claim is that for any coding function, no matter how clever you try to be, any valid coding function phi, we have the property that the expected length-- and what I mean by this is the length of phi of x1 up to xn, where this x1 up to xn comes from our first order source. I claim that this is at least the entropy H times n minus little o of n.

So just to make sure-- you guys may not have all seen asymptotic notation before. What this expression means right here, little o of n just means that it's a function that grows smaller than n. So you should think about n over log n or root n. So the main point is that as n becomes large, this is the term that dominates. And H is the answer to what the per symbol length is that we pay in our encoding function.

So this bound only kicks in when n is large, because when n is small, this term might actually be more significant than this right here. But what we're saying is that, at least in the limit, we can understand what is the best possible coding scheme. So this is one part of Shannon's noiseless coding theorem. Let me state the other part, which is just the natural converse, that you can actually achieve it.

So moreover, this really is the answer. So there is a coding function phi with the property that its expected length l is defined the same way as at most the entropy times n plus little o of n. So up to this plus minus little o of n, we've resolved what the right answer is for the best possible coding scheme.

And amazingly, this kind of mysterious function called the entropy pops out. And that's really the thing that governs what the best you can do is. So are there any questions about the definition so far? We've talked about coding functions. We've talked about the binary entropy function. And now, we have this key theorem that we're going to prove today. Does this make sense? Yeah, OK, good.

All right. And just to be clear, I said this in words. But what Shannon's theorem is telling us is that H is the best per symbol from our alphabet length that you can do for a coding scheme. So before we prove this theorem, let's work up to it with some intuition first.

So let me ask you to test your intuition now that we know what target we're shooting for. So what distribution p on k symbols-- we're still fixing our alphabet size-- requires the longest encoding? This is just for intuition's sake because we already know what the answer is for the best possible per symbol length. We know it's the entropy function.

So you can think about this question mathematically. If I have any distribution p on k things, what distribution on k things is going to maximize that binary entropy function? That's one way to think about mathematically what I'm asking here. But maybe the better way to think about it is to go back to your intuition.

So if I just started off with this, and I told you that there were first order sources, and that we were looking at coding functions, intuitively, what kind of first order source would be the hardest to compress? If you think back to this example we had right here, how were we able to win over the naive coding scheme was we exploited the fact that the probabilities were very different because there was this message a1a1 which was so much more likely.

So we could encode that very frequent outcome with a very short string. But maybe that's not always possible. So you can think about this thing mathematically, in terms of the binary entropy function. Or you can think about it from first principles, just from your guess instead of doing the calculus. Yeah?

AUDIENCE: [INAUDIBLE]

ANKUR MOITRA: That's exactly right. That is completely correct. So in fact, the answer to this question is what you'd expect, what you could have guessed even before class started, the uniform distribution. So the uniform distribution, p1 equals p2 all the way up to pk is all equal to 1 over k.

And in this case, what does the entropy look like? Well, it's minus the sum of i equals 1 to k of pi log pi, which is the same thing as log k. And what this corresponds to is this corresponds to the naive encoding, at least if k is a power of 2, because one of the things I can do is I can just take my first symbol. Let's say it's ai. And I can encode this using log k bits because you can tell me which of the k symbols it is.

And I can take the second symbol and encode it using log k bits. So in fact, this is an example, the uniform distribution, where the naive encoding cannot be beaten. One of the ways you can prove that this really is the answer, at least once we have Shannon's theorem, is you can solve the calculus problem-- which pi, which is a distribution and has non-negative entries that sum to 1, has the property that it maximizes the entropy function? You'll find out that this is the answer. But it really corresponds to this intuitive fact that, when there's no discrepancy in the probabilities, you can't have any win in terms of making more common things shorter code words.

OK, good. So we have our intuition. Let's work towards proving the first part of Shannon's theorem. So let me state one of the key lemmas, which really corresponds to this thing that you have intuition for now. So what I claim is that any coding function for the uniform distribution on k symbols, it has the expected length being at least log k minus some constant. So o of 1 is some constant.

You'll see in the proof why this constant shows up. It really comes from the fact that, for my coding function, I still can encode some of my symbols using very short strings. And then I can fill out things with longer strings, too. So there is some slight improvement you can get. But for reasons that we'll get into, that'll come out in the wash.

So the sketch for this is the following. All of these symbols are interchangeable. We can think of phi as mapping symbols, each of these k symbols, to nodes in a binary tree. So in particular, I can start with this binary tree, where the root is the empty set, because I output nothing. There are no 0's and 1's. My message just says END. And then maybe down here, I output the symbol 0. Down here, I output the symbol 1. Down here, I output the code 00, 01, and so on.

So my constraint that phi is a code, is a valid code, is just the constraint that I map each of my symbols to distinct nodes in this binary tree. I can't have any collisions. As long as there are no collisions, then I meet the definition of being a valid coding function. And now, the intuition is you want to minimize the expected length for what the depth is of the symbol you're searching for wherever you map it in this tree.

So what should you do for your first symbol? Maybe you would map it to the empty set because that's pretty shallow. Maybe your next symbol you would map somewhere here, and then you'd map here, and you'd map here. And basically, you'd fill out the levels of this tree in order. You'd fill out all previous things because all these symbols are all equivalent. And this will minimize the expected length.

And you can check that most of the symbols are going to get mapped down here, where their length of the message will be log k. And maybe you're off by some constant factor that depends on the fact that there are symbols before it. But if you work out the details, this is what you can show explicitly. It's just this lemma that, for the uniform distribution for our coding function, you can't do better than log k minus some constant.

But now, we're going to use this simple lemma. And we're going to appeal to it in powerful ways. So let's do a thought experiment before we get to the first part of Shannon's theorem. And let me ask a simpler sounding question, which is going to be the heart of the lower bound.

So what if I made your life easier? You're trying to encode a general message from this first order source. But what if I made your life easier by promising you something structurally about the message you're encoding that gave you more information? So what if you know there are n1 occurrences of the symbol a1, n2 occurrences of a2, and so on?

So I'm making your life easier because instead of having to worry about a general message from your first order source, what if I told you something about that message and promised you I'm not going to tell you where these symbols of a1 appear, but I'm telling you, there are exactly n1 of them. I'm not going to tell you where these symbols a2 occur, but there are exactly n2 of them. What if I told you this for each different symbol type, I told you exactly what their count was?

So now, what we can do is we can ask, first of all, how many such sequences are there. If I promise you that this condition holds, then I can ask a counting question, which is how many possible messages could there be under this promise that I've given you? You can see now this will connect back to a lot of the topics we did earlier in the class.

So the key is really the multinomial. If we had only two symbols and I told you how many 1's there are-- I told you there were k 1's out of n things-- the answer for how many strings there would be n choose k. But in general, if I have k larger than 2, when I tell you all of those types, what you're going to get is you're going to get this multinomial, which is called n choose n1 n2 all the way up to nk.

And it's defined-- we've seen it before. It's just n factorial over n1 factorial n2 factorial all the way up to nk factorial. That just counts the number of ways of coloring a set of integers from 1 to n with k different colors once I promise you that there are n1 reds, n2 blues, and so on. So let's just denote this quantity as m because this will be important for us.

And now, we can get to a key observation. So it sounds like, so far in proving Shannon's theorem, we've only thought about special cases. We've talked about the case where we had the uniform distribution on k symbols. But what I claim here is that now we're back to the case of the uniform distribution. So we know how many sequences there are. And moreover, each of these sequences has the same probability.

So the probability that they have is p1 to the n1 p2 to the n2 times pk to the nk. So there are a ton of possible sequences. But each particular sequence you give me, well, every time I see a symbol of a1, the probability that happened was p1. So I multiply p1. Every time I see a symbol of type a2, I multiply by p2.

So what this means is we're back in the uniform distribution setting because when I made you this promise about the number of counts of each type of symbol, well, I know how many messages there are. And I know that it's the uniform distribution on those possible messages.

So the intuition behind the first part of Shannon's theorem really comes from two parts. We're going to expand what's happening with this expression. We're going to use Stirling's approximation. And the binary entropy function will pop out. But then it'll turn out that we can remove this assumption because when we think back to tail bounds, if I had a first order source where the probability was p1 for the first symbol and p2 for the second symbol-- and really, that's a coin that has bias p1-- and I flip it n times, what we know is that the empirical number of times we see heads is going to converge very quickly to p1 as a fraction.

So in fact, this assumption that you know what n1, n2, and all the way up to nk are is not really such a big assumption. Just because of large deviation principles, I should approximately know the number of symbols of each type anyways. And that's the way that we're going to piece this all together. So that's the preview of the argument. But let's do it.

So now, what we can do is, from lemma 1, we know that the expected length of my encoding, even if I condition on telling you the information that there are n1 occurrences of a1, n2 occurrences of a2 and so on, we know, by the first lemma, that the expected length must be log m minus maybe some constant.

So here, this constant isn't so bad because log m is a really gigantic thing. You should think of it as being like exponential in n And this other o of 1 is just a constant. So now, the entire name of the game is just to figure out what log m is. And this is where the binary entropy function is going to pop out.

So let's do that. So we can just plug in for our expression for m this multinomial expression we wrote down. And then we're going to use Stirling's formula to group together terms and simplify. So plugging in the multinomial, we get that log m is equal to log of n factorial minus the sum from i equals 1 to k of log ni factorial. And then I have my constant sitting right here, which is the slack in this expression. So this is just using the multinomial expression and using properties of the log.

And now, what I can do is I can use Stirling. So Stirling's formula gives us an expression that n factorial behaves like square root of 2 pi n times n over e to the n. And really, this part isn't going to matter because I'm taking the log of all of these expressions. That's just going to get thrown into the constant. The important thing is this n over e to the n.

So when we plug that in, we're going to get a very nice expression. So let's use Stirling. Well, we'll get that the expected length is at least n log n minus the sum i equals 1 to k ni log ni. And I'll have some more expressions right here. All I'm doing is I'm pulling out this n to the n term.

So when I have n factorial behaves like n to the n, I get n log n. This behaves like ni to the ni. So I get ni log ni. But now, I have to deal with all of these e to the n correction terms. So I'm going to have a minus n log of e, and then a plus sum from i equals 1 to k of ni log e. And then I'll have my o of 1 sitting in here.

So this is my expression. But now, life is good because I claim that this expression just goes away. So this is n log e. And this is the sum over i from 1 to k of ni log e. So what is the sum of all of the ni's? n Why?

AUDIENCE: Because like, that's the tall number of each, I guess--

ANKUR MOITRA: Perfect. So the important point is just to not forget what the ni's are. They count the symbols of type i. And the total number of symbols of type i when we sum over i is just the total number of symbols, which is n. So this part completely goes away. And now, we're in good shape because we can start to see where the entropy function comes in.

So how can we think about this in terms of the entropy function? Well, we can rewrite this expression right here as the sum from i equals 1 to k of ni log n because then it's just the knee that's varying. I'm just splitting n up into the sum of the ni's. And then when I arrange these expressions, what I'm going to get-- so this part right here is the sum from i equals 1 to k of ni. And then I have log n over ni.

And when I put a minus in front of this, I can switch these two. And then I'll have my minus o of 1 sitting here. So in fact, now, what I can do is I can divide through by n. And what I'll get is ni over n that looks like a probability. So it'll be equal, this expression right here, to the binary entropy function of n1 over n n2 over n all the way up to nk over n. And still now I have my little o of actually n term here. So I'm in good shape.

So you can see that when I expect each of these ni's-- the ni over n-- to behave like pi, then I'm pulling out this binary entropy function. And this is the first part of Shannon's theorem. This gives us our valid lower bound. It's actually almost there. Let me just clean up the proof.

So what this tells us really is that, even if you were given the promise that there are ni symbols of type i, then you can't do better than this expression. But in principle, the thing you might be worried about is maybe every configuration of the ni symbols is very unlikely. So now, let's turn this lower bound into an actual lower bound for Shannon's theorem. Let's see. I can do it over here.

So let me just reiterate why I'm not quite done yet. So I claim that what we've done so far doesn't quite prove what we're after because the thing we could be worried about is that there might not be any choice of ni's that's very common. So let's pop up a level.

So what I'm saying here in this proof-- the way the proof is structured is I'm saying, aha, even if I told you that ni is the number of occurrences of symbol type i, well, in that case, you can't beat the binary entropy function. The trouble is that it might be very unlikely that configuration of ni's is really what happens.

So you might tell me, all right, when that happens, I'm dead. You're right. I have to pay the binary entropy function. But maybe I just never have to-- that situation never arises. So that's the last part of the proof that we have to worry about, is we have to argue why these ni's, at least some suitable choice of them, is actually quite likely.

And so the key for this is the following definition. And then we'll connect it to our earlier discussions of tail bounds. So we'll say that a string is epsilon typical if we have the following property-- for all i, for all of the symbols in our alphabet-- well, we take that string. And we look at what its ni is. So what is its count of symbol type i? We divide it by n. And I want this to be very close to pi. In fact, I want it to be epsilon close.

So this is what I was alluding to before, was that this is the type of thing we proved for tail bounds. When k equals 2, when we just have heads or tails, we want the empirical number of heads in our string to be very close to the true bias of my coin. I want it to be close up to an additive epsilon.

And this is the same thing. It's just saying that the empirical average of the number of times we actually roll a particular side for our die is epsilon close to the true probability. So we'll say that a string from our first order source is epsilon typical if this condition holds for all i. And now, the main part is really that we can come back to Chebyshev.

Remember that we used tools like Chebyshev, Chernoff inequality, and so on to prove that the empirical number of heads converges very quickly to the expected number of heads. And this works even when we have a k-sided die instead. So what I claim is really the consequence of what we proved by Chebyshev's inequality quite a few lectures ago at this point, is that the probability that a random string from our first order source is epsilon typical is actually quite large. It's like 1 minus maybe k over epsilon squared n.

So the point was that when we had Chebyshev, we could argue that, as n was going to infinity, because the variances behaved nicely for the sum of independent random variables, that the probability of being far away from our expectation was going down rapidly with n, at a 1 over n kind of rate. And then maybe I have to do some kind of union bound over each of these different k sides for the die. Any one of those goes wrong, I fail typicality.

But at the end of the day, the point is, when you look at this expression, k and epsilon are fixed. So k is fixed by our first order source. And you should think of epsilon as being your target accuracy in Shannon's theorem. Maybe I want to get up to optimality up to a 0.01 or a 0.001. The point is just that if you take n large enough, then this term is still going to 0. So you're almost always typical.

So one of the hard parts about the results we're doing in coding theory is keeping all the parameters straight in your head, in terms of what's the order in which things go to infinity. So k is some fixed thing. Epsilon is merely tiny. And then n can be gigantic in relation to things like 1 over epsilon. Any questions about this statement that I'm asserting right here? This makes sense and it seems to jive with the things we covered for tail bounds? Yeah? OK.

So now, let's put this all together. So putting this all together, let's see what we have. So you give me some coding function. And I want to argue that that coding function cannot do super duper well. I care about the expected length of your coding function. And we can break this up into two parts using the law of total expectation.

We can look at the expected length given that the string is epsilon typical times the probability that it's epsilon typical. And we can add in the expected length given that it's not epsilon typical times the probability that it's not epsilon typical. So all I'm doing is I'm using the law of total expectation to break up my computation because what we know how to reason about is really, this part right here. When we have a typical sequence, we can reason through this log of the multinomial what the expected length must be.

So what I can do is I can just-- I'm interested in a lower bound. So I can just drop this term. So this entire thing is at least 1 minus k over epsilon squared n because that's my probability of being typical. And then I'm going to use this fact that I had about how it relates to the binary entropy function. So I'll put in minus n sum from I equals 1 to k of pi plus or minus epsilon log of pi plus or minus epsilon. And then the entire thing will be minus some little o of n.

So here, I'm being slightly sloppy because the way that I reasoned about here was I gave you what n1, n2, all the way up to nk were. And then I talked about log of capital M being the lower bound. So here, really, epsilon typicality is not any one configuration of the ni's. It's a range of configurations for those ni's. But I know that each of those ni's are equal to ni over n is equal to pi plus or minus epsilon because that's the thing we plug into the binary entropy function.

But the point is that this expression right here, it doesn't actually vary that much as you range epsilon over some tiny-- some constant. So in particular, if I take this expression and I look at epsilon, which is little o of 1-- so it's something that's actually going to 0 as the length of my code increases, the length of my message-- then what I'm going to get here is that this behaves basically like the entropy function at pi, up to some small slack.

So what this will give me is what I was after, that this is equal to n times entropy minus little o of n. So really, this proof, putting it all together, it mimics what I did in the special case where I promised you what the ni's are. But it just uses the fact that maybe I don't want to tell you the ni's. I just want to tell you it's epsilon typical. And then I want to use the fact that the binary entropy function doesn't vary much, as long as the pi's are what they're supposed to be plus or minus some little o of 1.

So these are all computations you can do. But this is just the intuition for where it is. And this gives us the first part of Shannon's theorem. We have our bona fide lower bound on the best possible coding function. So are there any questions about the proof of the first part of Shannon's theorem? This was a lot of ingredients. Yeah?

AUDIENCE: Can you explain again why we don't need to consider the square root of n? The square root of pi n?

ANKUR MOITRA: The square root 2 pi n? Yeah. Well, OK, so here, what's going to happen is that all of these things-- I'm really plugging Stirling in into this expression. So when I take log of this expression, I'm going to have these giant terms in here. Log of n to the n looks like n log n.

I'm going to add in some other terms. You're right. I could worry about what happens here. And that would be a log n term. But the point is that, at the end of the day, the thing that I want is this little o of n. So all of those log n's are just tiny by comparison. So even though, by this vantage point, that looks like a serious term-- it's a root n sitting in there-- But after I take the logs, really, the action is all happening in this part right here. But good question.

So that's what makes some of these coding theorems a bit tricky, is that I promised you, when we did things like probability and counting, that we would see ways that these tools would come up in very complicated things. This uses a lot of material from earlier in the class. If you just think about what all has happened, we used tail bounds to argue about typicality. We used combinatorics to understand what exactly this multinomial works out to be. And then we use Stirling's formula in order to massage it into something which has the binary entropy function.

And all of these things required a lot of estimates about what the error terms are and how they add up. But the intuition, at its core, really comes down to this idea that you're encoding. If it works, I mean, typicality is a common thing that happens. And in that case, the probabilities of the different outcomes are very close to each other. So then you can't really beat log of the number of strings.

So at the core, this lower bound in first part of Shannon's theorem is exactly this kind of statement that we all had intuition for, that you can't beat the naive encoding for the uniform distribution. And we're just reasoning about why this holds even when it's really close to the uniform distribution.

All right. So this was a challenging proof. Tuesday's lecture will be an easier proof. Thursday's lecture will make this look like a cakewalk. I'm sorry, but it's true. So let me pause. Any other questions about this? Does this make sense? You guys all comfortable with it? Yeah? All right.

So the good news is that now we're in a position where we can prove the other side of Shannon's theorem very easily. So the second part actually follows from roughly the same kind of argument. In fact, the way that I'm going to write it is really as a block diagram for what this encoding function looks like. So let's prove the second part of Shannon's theorem.

So let me ask a question first, which will get us started. How many epsilon typical sequences are there? Well, let's just work it out. Let's do the combinatorics. So what I claim is that the answer is a sum over signatures-- all of these n1 n2 up to nk that are epsilon typical.

And we have this multinomial here, n choose n1 n2 all the way up to nk. So this is just, by definition, the answer. I sum over all of the signatures that meet my definition of epsilon typicality. And I count the number of things that meet that signature. But let's get an upper bound on what this is.

So first of all, I can think about this expression. I'll still be a bit sloppy here, the same way I was last time, where I'm going to think about this multinomial as really being p1 plus or minus epsilon times n p2 plus or minus epsilon times n and so on. So I don't know what these n1 n2's are. But certainly, I could choose whatever choices that are close to pi that maximize this expression. And I could use it as an upper bound for every term that shows up in the sum.

So the same way I used this notation earlier just to think about all possible values that meet this condition of being close to the pi's, I'm doing the same thing here. But now, I have to worry about how many terms there are in the sum. And this is where we get a huge win. How many times does the sum execute? How many things are we summing over for this multinomial? It's actually not exponential in n. That's the key point.

So in fact, how many different signatures meet the condition of being epsilon typical? Well, I could just ignore the constraint that it's epsilon typical and just look at all of the possible signatures. How many choices are there for n1? There's n possibilities. How many choices are there for n2? There's n possibilities and so on. So I get n to the k.

So this term right here, remember that we're going to be taking the log of all of these expressions. So this term right here has the same property that it's not exponentially large in n. And that's a huge win for reasons we'll see. So I'm going to call this term right here m prime, even though it's really a function of whatever I jitter these pi's to be.

But now, I promised you that, really, our proof is, in essence, going to be a block diagram. So let's prove it. Let's explicitly come up with a good coding scheme for this first order source.

So we start off with our message. And then in my block diagram, I'm first going to ask, is the message epsilon typical? You give me the message. I can certainly compute what all the ni's are. And I can plug it into the expression and decide whether it meets my definition for some value of epsilon.

Now, in the case where the answer is yes, how am I going to encode this message? The first thing that I'm going to do is I'm going to ask you, what is n1 up to nk? And then I can ask you, what string is it? Let's worry about the no branch later. But let's compute the number of bits that this coding scheme uses.

So how long is my transmission? Well, I have to tell you whether the answer is yes or no. That's 1 bit. I tell you yes or no. What about telling you what the signature n1 up to nk is? How many bits is that at most? Yeah?

AUDIENCE: k log n.

ANKUR MOITRA: Perfect, k log n. Because I only pay the log of the number of possibilities, I just have to write out a binary string that tells me which of the possibilities it is. So this is a huge win because all of these questions, the answer to them is really hidden in this little o of n term because it's not linear in n.

The only place where I'm going to pay linear in n is this last part right here, which is what is the string? This right here will get me log of M prime. And what we just argued was that log of M prime, even when I maximize over what p prime i to put in here that's epsilon close to pi, I'm going to get something-- the log of Mi is going to behave like the binary entropy function times n up to some lower order terms.

So now, down this entire branch, I'm only paying exactly what Shannon wants me to pay. I'm paying up to leading order the binary entropy times n. Now, you guys can help me out with what happens in the rest of the way. So here, I could pay plus 1 bit because I have to tell you, no, it's not epsilon typical. And here, what I can do, as it turns out-- any guesses what I can do down this no branch? Anyone have any intuition?

So let me ask a leading question. See, Shannon's theorem, I have this slack built in, little o of n. What's the probability that I go down this branch, at least if I set parameters correctly? What do I want it to be? Do I want it to be a half probability, something closer to 1, something tiny? Maybe a little o of 1?

So if I set these parameters the right way, the probability that I fail and the thing is not epsilon typical is little o of 1 because, remember, it was this k over epsilon squared n for my Chebyshev bound. So if I set the parameters right, that's tiny. And if this is little o of 1, the probability I go down this branch, I can actually afford to do something really dumb here.

So what could I do is I could just output the trivial encoding. So how much does this trivial encoding pay? It pays a constant times n, maybe 100 times n. But so what? It only happens with probability 1 over log n that we go down this branch. So it doesn't actually prevent me from proving Shannon's second theorem. So this is very unlikely. So I can just give up whenever that happens and do the naive thing.

So let's analyze this and just write down what the expression is, even though this block diagram really does the trick. So let's compute the expected length. So the expected length-- I always have to tell you the answer to the question that's yes or no is it epsilon typical. If it's epsilon typical, which happens with 1 minus little o of 1 probability, because this n is gigantic compared to k and 1 over epsilon squared, in that case, I pay this k log n plus this log M prime.

And otherwise, I pay this k over epsilon squared n all times maybe log k times n because this is my trivial encoding, is that for each symbol I just tell you which particular symbol it is. But the point is that, once I tell you what k is-- maybe it's 10. Once I fix my target accuracy-- maybe it's 0.00001-- n can be large enough so that this term becomes arbitrarily small. And then it's multiplied by something linear in n. But then it becomes much, much smaller. Any constant times n, I can make it as small as I want. So this is the proof of Shannon's second main theorem. Any questions?

You guys should have some questions. So just to summarize, really, the way that all of this fit together was a bit magical, but it's really based on this block diagram, is that even though a priori we were worried about how do I encode an arbitrary thing that could come out of my first order source, the point is that there's this event that happens that makes it a lot easier to reason about.

This event almost always happens. And once it's epsilon typical, then the problem behaves like the uniform distribution on some universe. And we just care about how big is that universe, M prime. That's a multinomial. And when we use Stirling's approximation, the binary entropy pops out. But this block diagram is really how the proof goes, both for the upper and the lower bounds.

So we're probably going to finish a couple minutes early. But let me tell you where we're headed for this. So this theorem looks amazing. We started off with this question about what is the best encoding. We saw that there were naive encodings that we could improve upon. And then we figured out what is the natural limit, in terms of what's the best that we could ever hope for, at least in the asymptotic regime where n is going to infinity.

But this coding scheme would actually be pretty terrible from a practical standpoint, if this were all there would be. So if you think about it, when I defined what a coding function was, a coding function is just a mapping from a possible message to what I'm going to encode it as, the codebook.

But in principle, as n becomes gigantic, how large is that codebook? It's becoming exponentially large in M because there are all kinds of messages. So this would be terrible if I had to do this because I would have to pre-guess whatever message anyone would ever want to send and fix what it is I'm going to transmit along the channel.

That would be a really bad way to do things, because it's not an efficient way to do the encoding, let alone the decoding. If I actually receive some message, then how in the world am I going to figure out what was the original thing you started from other than looking at everything and figuring out what its image was?

So what we're going to cover on Tuesday is a really amazing scheme, which is going to give us a different way to think about compression. And it's a scheme that has a lot of beautiful combinatorics. It's called the Huffman code. And what will be great about the Huffman code is that it'll naturally give us efficient encodings and decodings. So that'll be the Tuesday lecture.

And then on Thursday, what we're going to talk about is the other big result of Shannon was it's not just about data compression. But when you're transmitting messages at long distances, inevitably there are errors in your transmission. So what happens when some of the symbols that you're sending along the telephone wire get corrupted? Is there a way to recover and decode what was the original thing you'd meant to send?

It'll turn out that you can do amazing things. These are really the building blocks of the starting point of what's called coding theory that has a lot of applications and other areas of computer science. And in fact, one of the biggest challenges for things like quantum computation is really how do we do these schemes in quantum computers, too?

So we'll stop early. And as I told you in the beginning, come to office hours, ask questions. We're here. Don't take your foot off the gas pedal quite yet because there's still a lot of coursework to go. But otherwise, I'll see you next week, where we'll continue our coding theory in information theory units. So see you then.