PROFESSOR: I'm going to spend a couple of minutes reviewing the major things that we talked about last time and then get into discrete source coding, which is the major topic for today. The first major thing that we talked about last time, along with all of the philosophy and all those other things, was the sense of what digital communication really is. I said that what digital communication is, is it's communication where there's a binary interface between source and destination. The source is very often analog. The most interesting sources are analog. The channel is often analog, most interesting. Channels are analog, and we'll say more about what I mean by analog later. What's important is that you have this binary interface between source and channel coding.

We said a little bit about why we wanted a binary interface, aside from the fact that it's there now, there's nothing you can do about it even if you don't like it. One reason is standardization, which means it simplifies implementation, which means you can do everything in the same way. If you have ten different kinds of channel coding and you have ten different kinds of source coding and you have a binary interface, it means you need to develop 20 different things -- ten at the source and ten at the decoder. If you don't have that standardization with a binary interface between them, you need 100 different things. You need to match every kind of source with every kind of destination. That raises the price of all chips enormously. One of the other things we said is the price of chips is very much the cost of development divided by the number of them that you stamp out. That's not quite true, but it's a good first approximation. In other words, standardization is important.

Layering. Layering is in many ways very similar to standardization because this binary interface is also a layer between source and destination. But the idea there is not that it standardizes to make things cheaper, but it simplifies the

conceptualization of what's going on. You can look at a source and only focus on one thing. How do I take that source and turn it into the smallest number of binary digits possible? We'll talk a good deal about what that means later because there's something stochastic involved in there and will take us awhile to really understand that. Finally, using a binary interface loses nothing in performance. That's what Shannon said, it's what he proved. There's some questions there when you get to networks, but the important thing is the places where you want to study non-binary interfaces, you will never get a clue of what it is that you're looking at or why if you don't first very well understand why you want a binary interface to start with. In other words, if you look at these other cases, there's exceptions to the rule, and if you don't know what the rule is, you certainly can't understand what the exceptions are.

So for today we're going to start out by studying this part of the problem in here. Namely, how do you turn a source and put a general source input into binary digits that you're going to put into the channel. How do I study this without studying that? Well, one thing is these are binary digits here. But the other thing is we're going to assume that what binary digits go in here come out here. In other words, there aren't any errors. It's an error-free system. Part of the purpose of studying channel encoding and channel decoding is to say how is it that you get that error-free performance. You can't quite get error-free performance, you get almost error-free performance, but the idea is when errors come out here, it's not this guy's fault, it's this guy's fault. Therefore, what we're going to study here is how we do our job over here. Namely, how we deal with decoding, the same string of bits that went into there and decode them coming out. So that's where we'll be for the next three weeks or so.

We talked a little bit last time about how do you layer source coding itself. I want to come back, because we were talking about so many things last time, and emphasize what this means a little bit. We're going to break source coding up into three different layers again. You start out with some kind of input wave form or image or video or whatever the heck it is. You're going to do something like sampling it or expanding it in some kind of expansion, and we'll talk a great deal

about that later. That's not an obvious thing, how to do that. When you finish doing that, you wind up with an analog sequence. In other words, you wind up with a sequence of real numbers or sequence of complex numbers. Those go into a quantizer. What the quantizer does is to turn an uncountably infinite set of things into a finite set of things. When you turn an uncountably infinite set of possibilities into a finite set of possibilities, you get distortion. There's no way you can avoid it. So that's a part of what happens there. Then at this point you have a finite alphabet of symbols. That goes into the discrete coder, goes through what we're now calling a reliable binary channel and comes out here.

What we're going to be studying for the next two weeks or so is this piece of the system right in here. Again, what we're going to be doing is assuming a reliable binary channel to the right of this, which is what we already assumed. We're going to assume that these things do whatever they have to. But this problem here, this isolated problem is important because this is dealing with the entire problem of text, and you know what text is, it's computer files, it's English language text, it's Chinese text, it's whatever kind of text. If we understand how to do that, we can then go on to talk about quantization because we'll have some idea of what we're trying to accomplish with quantization. Without that we won't know what the purpose of quantization is. Without the quantization we won't know what we're trying to accomplish over here.

There's another reason for studying this problem, which is that virtually all the ideas that come into this whole bunch of things are all tucked into this one subject in the simplest possible way. One of the nice things about information theory, which we're going to touch on I said in this course, is that one of the reasons for studying these simple things first is that information theory is really like a symphony. You see themes coming out, those themes get repeated, they get repeated again with more and more complexity each time, and when you understand the simple idea of the theme, then you understand what's going on. So, that's the other reason for dealing with that.

To summarize those things -- most of this I already said. Examples of analog

sources are voice, music, video, images. We're going to restrict this to just wave form sources, which is voice and music. In other words, an image is something where you're mapping from two dimensions this way and this way into a sequence of binary digits. So it's a mapping after you get done sampling from r square, which is this axis and this axis, into your outut. Namely, for each point in this plane, there's some real number that represents the amplitude at that point. Video is a three-dimensional to one-dimensional thing, namely, you have time. You also have this way, you have this way, so you're mapping from r cubed into r. We're not going to deal with those because really all the ideas are just contained in dealing with wave form sources. In other words, the conventional functions that you're used to seeing. Namely, things that you can draw on a piece of paper and you can understand what's going on with them.

These are usually samples or expanded into series expansions almost invariably, and we'll understand why later. That, in fact, is a major portion of the course. That's where all of the stuff from signals and systems come in. We'll have to expand that a whole lot because you didn't learn enough there. We need a lot of other things, and that's what we need to deal with wave forms. We'll take the sequence of numbers that comes out of the sampler. We're then going to quantize that sequence of numbers. That's the next thing we're going to study. Then we're going to get into analog and discrete sources, which is the topic we will study right now. So we're going to study this. After we get done this, we're going to study this also.

When we study this, we'll have what we know about this as a way of knowing how to deal with the whole problem from here out to here. Finally, we'll deal with wave forms and deal with the whole problem from here out to here. So that's our plan. In fact, this whole course is devoted to studying this problem, then this problem, then this problem -- that's the source part of the course. Then dealing with -- if I can find it again -- with the various parts of this problem. So first we study sources, then we study channels. Because of the binary interface, when we're all done with that we understand digital communication. When we get towards the end of the term we'll be looking at more sophisticated kinds of channels than we look at earlier, which are really models for wireless channels. So that's where we're going to end up.

So discrete source coding, which is what we want to deal with now. What's the objective? We're going to map a sequence of symbols into a binary sequence and we're going to do it with unique decodability. I'm not going to define unique decodability at this point. I'm going to define it a little bit later. But roughly what it means is this. We have a sequence of symbols which come into the encoding encoder. They go through this binary channel. They come out as a sequence of binary digits. Unique decodability says if this guy does his job, can this guy do his job? If this guy can always do his job when these digits are correct, then you have something called unique decodability. Namely, you can guarantee that whatever comes out here, whatever comes in here, will turn into a sequence of binary digits. That sequence of binary digits goes through here. These symbols are the same as these symbols. In other words, you are reproducing things error-free if, in fact, this reproduces things error-free. So that's our objective.

There's a very trivial approach to this, and I hope all of you will agree that this is really, in fact, trivial. You map each source symbol into an l-tuple of binary digits. If you have an alphabet of size m, how many different binary strings are there of length l? Well, there are 2 to the l of them. If l is equal to 2, you have 0, 0, 0, 1, 1, 0, and 1, 1. If l is equal to 3, you have strings of 3, which is 0, 0, 0, 0, 0, 1, 0, 1, 0, blah, blah, blah, blah, blah. What comes out to be 2 to the 3, which is equal to 8. So what we need if we're going to use this approach, which is the simplest possible approach, which is called the fixed length approach, is you need the alphabet size to be less than or equal to the number of binary digits that you use in these strings. Now, is that trivial or isn't it trivial? I hope it's trivial.

We don't want to waste bits when we're doing this, particularly, so we don't want to make l any bigger than we have to, because for every symbol that comes in, we get l symbols coming out. So we'd like to minimize l subject to this constraint that 2 to the l has to be bigger, greater than or equal to m. So, what we want to do is we want to choose l as the smallest integer which satisfies this. In other words, when you take the logarithm to the base 2 of this, you get log to the base 2 of m has to be less than or equal to l, and l is then going to be less than log to the base 2 of m plus

1. This is the constraint which says you don't make l any bigger than you have to make it. So in other words, we're going to choose l equal to the ceiling function of log to the base m. In other words, this is the integer which is greater than or equal to log to the base 2 of m.

So let me give you a couple of examples of that. Excuse me for boring you with something which really is trivial, but there's notation here you have to get used to. You get confused with this because there's the alphabet size which we call m, there's the string length which we call l, and you keep getting mixed up between these two. Everybody gets mixed up between them. I had a doctoral student the other day who got mixed up in it, and I read what she had written four times and I didn't catch it either. So this does get confusing at times.

If you have an alphabet, which is five different kinds of the letter a -- that's one reason why these source codes get messy, you have too many different kinds of each letter, which technical people who like a lot of jargon use all of them. In fact, when people start writing papers and books you find many more than five there. In terms of Latex, you get math cow, you get math gold, you get math blah, blah, blah. Everything in little and big. You get the Greek version. You get the Roman version and the Arabic version, if you're smart enough to know that language, those languages. What we mean by code is alpha gets mapped into 0, 0, 0. a gets mapped into 0, 0, 1. Capital A into this and so forth. Does it make any difference what mapping you use here? Can you find any possible reason why it wouldn't make a difference whether I map alpha into 0, 0, 0, and a into 0, 0, 1 or vice versa? I can't find any reason for that. Would it make any difference of instead of having this alphabet I had beta b, capital B, script b, and capital B with a line over it? I can't see any reason why that would make a difference either.

In other words, when we're talking about fixed length codes, there are only two things of importance. One of them is how big is the alphabet -- that's why we talk about alphabets all the time. After you know how big the alphabet is and after you know you want to do a fixed length binary encoding, then you just assign a binary string to each of these letters. In other words, there's nothing important in these

symbols. This is a very important principle of information theory. It sort of underlines the whole subject. I'm not really talking about information theory here, as I said, we're talking about communication.

But communication these days is built on these information theoretic ideas. Symbols don't have any inherent meaning. As far as communication is concerned, all you're interested in is what is the set of things -- I could call this a1, a2, a3, a4, a5, and we're going to start doing this after awhile because we will recognize that the name of the symbols don't make any difference. If you listen to a political speech if it's by a Republican there are n different things they might say, and you might as well number them a1 to a sub n. If you listen to one of the Democratic candidates there are m different things they might say. You can number them 1 to m, and you can talk to other people about it and say oh, he said a1 today, which is how do we get out of the war in Iraq. Or he said number 2 today, which is we need more taxes or less taxes and so forth. So it's not what they say as far as communication is concerned, it's just distinguishing the different possible symbols.

So, you can easily decode this -- you see three bits and you decode them. Can I? Is this right or is there something missing here? Of course, there's something missing. You need synchronization if you're going to do this. If I see a very long string of binary digits and I'm going to decode them into these letters here, I need to know where the beginning is. In other words, if it's a semi-infinite string of binary digits, I don't know how to look at it. So, inherently, we believe that somebody else gives us synchronization. This is one of these things we always assume. When you start building a system after you decide how to do this kind of coding, somebody at some point has to go through and decide where do you get the synchronization from. But you shouldn't think of the synchronization first. If I'm encoding 10 million symbols and it takes me 1,000 bits to achieve the synchronization, that 1,000 bits gets amortized over 10 million different symbols, and therefore, it doesn't make any difference, and therefore, we're going to ignore it. It's an important problem but we ignore it.

The ASCII code is a more important example in this. It was invented many, many

years ago. It was a mapping from 256 different symbols which are all the letters, all the numbers, all the things that people used on typewriters. Anybody remember what a typewriter is? Well, it's something people used to use before they had computers, and these typewriters had a lot of different keys on them and they had a lot of special things you could do with them. And somebody dreamed up 256 different things that they might want to do. Why do they use l equals 8? Nothing to do with communication or with information theory or with any of these things. It was that 8 is a nice number. It's 2 to the 3. In other words, this was a standard length of both computer words and of lots of other things. Everybody likes to deal with 8 bits, which you call a byte, rather than 7 bits which is sort of awkward or 6 bits which was an earlier standard, which would have been perfectly adequate for most things that people wanted. But no, they had to go to 8 bits because it just sounded nicer.

These codes are called fixed length codes. I'd like to say more about them but there really isn't much more to say about them. There is a more general version of them, which we'll call generalized fixed length codes. The idea there is to segment the source sequence. In other words, we're always visualizing now having a sequence of symbols which starts at time zero, runs forever. We want to segment that into blocks of length n. Namely, you pick off the first n symbols, you find the code word for those n symbols, then you find the code word for the next n symbols, then you find the code word for the next n symbols and so forth. So it's really the same problem that we looked at before. It's just that the alphabet before had the number of symbols as the alphabet size. Now, instead of having an alphabet size which is m, we're looking at blocks of m symbols and how many possible combinations are there of blocks where every symbol is one of m different things. Well, if you have two symbols, the first one can be any one of m things, the second one can be any one of m things. So there are m squared possible combinations for the first two symbols, there are m cubed possible combinations for the first three symbols and so forth. So we're going to have an alphabet on blocks of m to the n different n tuples of source letters.

Well, once you see that we're done because what we're going to do is find a binary sequence for every one of these blocks of m to the n symbols. As I said before, the

only thing important is how many are there. It doesn't matter that they're blocks or that they're stacked this way or that they're stacked around in a circle or anything else. All you're interested in is how many of them are there. So there are m to the n of them. So, what we want to do is make the binary length that we're dealing with equal to log to the base 2 of m to the n, the ceiling function of that. Which says log to the base 2 of m is less than or equal to l bar where l bar is going to be the bits per source symbol. I'm going to abbreviate that bits per source symbol. I would like to abbreviate it bps, but I and everyone else will keep thinking that bps means bits per second. We don't have to worry about seconds here, seconds had nothing to do with this problem. We're just dealing with sequences of things and we don't care how often they occur. They might just be sitting in a computer file and we're doing them offline, so seconds has nothing to do with this problem.

So, log to the base 2 of m is less than or equal to l over n, which is less than log to the base 2 of m plus 1 over n. In other words, we're just taking this dividing it by n, we're taking this dividing by n, the ceiling function is between log to the base 2 of m to the n, and log to the base 2 of m to the n plus 1. When we divide by n, that 1 becomes 1 over n. What happens when you make n large? l approaches log to the base 2 of m from above. Therefore, fixed length coding requires log to the base 2 of n bits per source symbol if, in fact, you make n large enough. In other words, for the example of five different kinds of a's, we had m equal to 5. So if you have m equal to 5, that leads to m squared equals 25, that leads to l equals -- what's the ceiling function of log of this? It's 5. l bar is equal to -- what's half of 5? 2 and 1/2, yes. As you get older you can't do arithmetic anymore.

So look what we've accomplished. We've gone from three bits per symbol down to two and and half bits per symbol, isn't that exciting? Well, you look at it and you say no, that's not very exciting. I mean yes, you can do it, but most people don't do that. So why do we bother with this? Well, it's the same reason we bother with a lot of things in this course, and the whole first two weeks of this course will be dealing with things where when you look at them and you ask is this important, you have to answer no, it's not important, it doesn't really have much to do with anything, it's a

mathematical idea. What it does have to do with is the principle involved here is important. It says that the lower limit of what you can do with fixed coding is log to the base 2 of m. You have an alphabet of size m, you can get as close to this as you want to. We will find out later that if you have equally likely symbols when we get to talking about probability, we will find out that nothing in the world can do any better than this. That's the more important thing, because what we're eventually interested in is what's the best you can do if you do things very complicated. Why do you want to know what the best is if you do something very complicated? Because if you can do that simply then you know you don't have to look any further. So that's the important thing. Namely, it lets you do something simple and know that, in fact, what you're doing makes sense.

That's why we do all of that. But then after we say well there's no place else to go on fixed length codes, we say well, let's look at variable length codes. The motivation for variable length codes is that probable symbols should probably have shorter code words than very unlikely symbols. And Morse thought of this a long, long time ago when Morse code came along. Probably other people thought of it earlier, but he actually developed the system and it worked. Everyone since then has understood that if you have a symbol that only occurs very, very, very rarely, you would like to do something, make a code word which is very long for it so it doesn't interfere with other code words. Namely, one of the things that you often do when you're developing a code is think of a whole bunch of things which are sort of exceptions. They hardly ever happen.

You use the fixed length code for all the things that happen all the time, and you make one extra code word for all these exceptions. Then you have this exception and paste it on at the end of the exception is a number which represents which exception you're looking at. Presto, you have a variable length code. Namely, you have two different possible code lengths -- one of them for all of the likely things and the indication that there is an exception, and two, all the unlikely things. There's an important feature there. You can't drop out having the code word saying this is an exception. If you just have a bunch of short code words and a bunch of long code words, then you see a short code word and you don't know -- well, if you see a long

code word starting or you have a short code word, you don't know which it is and you're stuck.

So one example of a variable length code -- we'll use some jargon here. We'll call the code a script c. We'll think of script c as a mapping which goes from the symbols onto binary strings. In other words, c of x is the code word corresponding to the symbol x. So for each x in the alphabet, capital X, and we have to think of what the capital X is. But as we say, the only thing we're really interested in is how big is this alphabet -- that's the only thing of importance. So if we have an alphabet which consists of the three letters a, b and c, we might make a code where the code word for a is equal to zero, the code word for b is equal to 1, zero, and the code word for c is equal to 1,1. Now it turns out that's a perfectly fine code and that works. Let me show you another example of a code.

Let me just show you an example of a code here so we can see that not everything works. Suppose c of a is zero, c of b is 1, and c of c is -- this is a script c, that's a little c -- is 1, zero. Does that work? Well, all of the symbols have different code words, but this is an incredibly stupid thing to do. It's an incredibly stupid thing to do because if I send a b followed by an a, what the poor decoder sees is 1 followed by zero. In other words, one of the things that I didn't tell you about is when we're using variable length codes we're just concatenating all of these code words together. We don't put any spaces between them. We don't put any commas between them. If, in fact, I put a space between them, I would really have not a binary alphabet but a ternary alphabet. I would have zeros and I would have 1's and I would have spaces, and you don't like to do that because it's much harder. When we start to study channels we'll see that ternary alphabets are much more difficult to work with than binary alphabets. So this doesn't work, this does work. Part of what we're going to be interested in is what are the conditions under why this works and why this doesn't work.

Again, when you understand this problem you will say it's very simple, and then you come back to look at it again and you'll say it's complicated and then it looks simple. It's one of these problems that looks simple when you look at it in the right way, and

it looks complicated when you get turned around and you look at it backwards. So the success of code words of a variable length code are all transmitted just as a continuing sequence of bits. You don't have any of these commas or spaces in them. If I have a sequence of symbols which come into the encoder, those get mapped into a sequence of bits, variable length sequences of bits which come out. They all get pushed together and just come out one after the other.

Buffering can be a problem here, because when you have a variable length code -- I mean look at what happens here. If I've got a very long string of a's coming in, I got a very short string of bits coming out. If I have a long string of b's and c's coming in, I have a very long string of bits coming out. Now usually the way the channels work is that you put in bits at a fixed rate in time. Usually the way that sources work is that symbols arrive at a fixed rate in time. Therefore, here, if symbols are coming in at a fixed rate in time, they're going out at a non-fixed rate in time. We have to bring them into a channel at a fixed rate in time, so we need a buffer to take care of the difference between the rate at which they come out and the rate at which they go in.

We will talk about that problem later, but for now we just say OK, we have a buffer, we'll put them all in a buffer. If the buffer ever empties out -- well, that's sort of like the problem of initial synchronization. It's something that doesn't happen very often, and we'll put some junior engineer on that because it's a hard problem, and seeing your engineers never deal with the hard problems, they always give those to the junior engineers so that they can assert their superiority over the junior engineers. It's a standard thing you find in the industry.

We also require unique decodability. Namely, the encoded bit stream has to be uniquely deparsed at the decoder. I have to have some way of taking that long string of bits and figuring out where the commas would have gone if I put commas in it. Then from that I have to decode things. In other words, it means that every symbol in the alphabet has to have a distinct code word connected with it. We have that here. We have that here. Every symbol has a distinct code word. But it has to be more than that. I'm not even going to talk about precisely what that more means

for a little bit. We also assume to make life easy for the decoder that it has initial synchronization.

There's another obvious property that we have. Namely, both the encoder and the decoder know what the code is to start with. In other words, the code is built into these devices. When you design a coder and a decoder, what you're doing is you figure out what an appropriate code should be, you give it to both the encoder and the decoder, both of them know what the code is and therefore, both of them can start decoding.

A piece of confusion. We have an alphabet here which has a list of symbols in it. So there's a symbol a1, a2, a3, up to a sub m. We're sending a sequence of symbols, and we usually call the sequence of symbols we're sending x1, x2, x3, x4, x5 and so forth. The difference is the symbols in the alphabet are all distinct, we're listing them one after the other. Usually there's a finite number of them. Incidentally, we could have a countable number of symbols. You could try to do everything we're doing here say with the integers, and there's a countable number of integers. All of this theory pretty much carries through with various little complications. We're leaving that out here because after you understand what we're doing, making it apply to integers is straightforward. Putting in the integers to start with, you'll always be fussing about various silly little special cases, and I don't know a single situation where anybody deals with a countable alphabet, except by truncating it. When you truncate an infinite alphabet you get a finite alphabet.

So, we'll assume initial synchronization, we'll also assume that there's a finite alphabet. You should always make sure that you know whether you're talking about a listing of the symbols in the alphabet or a listing of the symbols in a sequence. The symbols in a sequence can all be the same, they can all be different. They can be anything at all. The listing of symbols in the alphabet, there's just one for each symbol.

We're going to talk about a very simple case of uniquely decodable codes which are called prefix-free codes. A code is prefix-free if no code word is a prefix of any other

code word. In other words, a code word is a string of binary digits. A prefix of a string of binary digits. For example, if we have the binary string 1, 0, 1, 1, 1. What are the prefixes of that? Well, one prefix is 1, 0, 1, 1. Another one is 1, 0, 1. Another one is 1, 0. Another is 1. In other words, it's what you get by starting out at the beginning and not quite getting to the end. All of these things are called prefixes. If you want to be general you could call 1, 0, 1, 1, 1, a prefix of itself. We won't bother to do that because it just is -- that's the kind of things that mathematicians do to save a few words in the proofs that they give and we won't bother with that. We will rely a little more on common sense.

Incidentally, I prove a lot of things in these notes here. I will ask you to prove a lot of things. One of the questions that people always have is what does a proof really mean? I mean what is a proof and what isn't a proof? When you take mathematics courses you get one idea of what a proof is, which is appropriate for mathematics courses. Namely, you prove things using the correct terminology for proving them. Namely, everything that you deal with you define it ahead of time so that all of the terminology you're using all has correct definitions. Then everything should follow from those definitions and you should be able to follow a proof through without any insight at all about what is going on. You should be able to follow a mathematical proof step-by-step without knowing anything about what this is going to be used for, why anybody is interested in it or anything else, and that's an important thing to learn.

That's not what we're interested in here. What we're interested in here for a proof -- I mean yes, you know all of the things around this particular proof that we're dealing with, and what you're trying to do is to construct a proof that covers all possible cases. You're going to use insight for that, you're going to use common sense, you're going to use whatever you have to use. And eventually you start to get some sort of second sense about when you're leaving something out that really should be there. That's what we're going to be focusing on when we worry about trying to be precise here. When I start proving things about prefix codes, I think you'll see this because you will look at it and say that's not a proof, and, in fact, it really is a proof. Any good mathematician would look at it and say yes, that is a proof. Bad

mathematicians sometimes look at it and say well, it doesn't look like proof so it can't be a proof. But they are.

So here we have prefix-free codes. The definition is no code word is a prefix of any other code word. If you have a prefix-free code, you can express it in terms of a binary tree. Now a binary tree starts at a root, this is the beginning, moves off to the right -- you might have it start at the bottom and move up or whatever direction you want to go in, it doesn't make any difference. If you take the zero path you come to some leaf. If you take the one path you come to some intermediate node here. From the intermediate node, you either go up or you go down. Namely, you have a 1 or a zero. From this intermediate node you go up and you go down. In other words, a binary tree, every node in it is either an intermediate node, which means there are two branches going out from it, or it's a leaf which means there aren't any branches going out from it. You can't, in a binary tree, have just one branch coming out of a node. There are either no branches or two branches, just by definition of what we mean by a binary tree -- binary says two.

So, here this tree corresponds where we label various ones of the leafs. It corresponds to the code where a corresponds to the string zero, b corresponds to the string 1, 1, and c corresponds to the string 1, 0, 1. Now you look at this and when you look at the tree, when you look at this as a code, it's not. Obvious that it's something really stupid about it. When you look at the tree, it's pretty obvious that there's something stupid about it, because here we have this c here, which is sitting off on this leaf, and here we have this leaf here which isn't doing anything for us at all. We say gee, we could still keep this prefix condition if we moved this into here and we drop this off. So any time that there's something hanging here without corresponding to a symbol, you would really like to shorten it. When you shorten these things and you can't shorten anything else, namely, when every leaf has a symbol on it you call it a full tree. So a full tree is more than a tree, a full tree is a code tree where the leaves correspond to symbols. So a full tree has no empty leaves. Empty leaves can be shortened just like I showed you here, so we'll talk about full trees, and full trees are sort of the good trees. But prefix-free codes don't

necessarily have to worry about that.

Well, now I'm going to prove something to you, and at this point you really should object, but I don't care. We will come back and you'll get straightened out on it later. I'm going to prove that prefix-free codes are uniquely decodable, and you should cry foul because I really haven't defined what uniquely decodable means yet. You think you know what uniquely decodable means, which is good. It means physically that you can look at a string of code words and you can pick out what all of them are. We will define it later and you'll find out it's not that simple. As we move on, when we start talking about Lempel Ziv codes and things like that. You will start to really wonder what uniquely decodable means. So it's not quite as simple as it looks. But anyway, let's prove that prefix-free codes are uniquely decodable anyway, because prefix-free codes are a particularly simple example of uniquely decodable codes, and it's sort of clear that you can, in fact, decode them because of one of the properties that they have.

The way we're going to prove this is we want to look at a sequence of symbols or a string of symbols that come out of the source. As that string of symbols come out of the source, each symbol in the string gets mapped into a binary string, and then we concatenate all those binary strings together. That's a big mouthful. So let's look at this code we were just talking about where the code words are b, c and a. So if a 1 comes out of the source and then another 1, it corresponds to the first letter b. If a 1, zero comes out, it corresponds to the first letter c. If a zero comes out, that corresponds to the letter a. Well now the second symbol comes in and what happens on that second symbol is if the first symbol was an a, the second symbol could be a b or a c or an a, which gives rise to this little sub-tree here. If the first letter is a b, the second letter could be either an a, b or a c, which gives rise to this little sub-tree here. If we have a c followed by anything, that gives rise to this little sub-tree here. You can imagine growing this tree as far as you want to, although it gets hard to write down. How do you decode this? Well, as many things, you want to start at the beginning, and we know where the beginning is. That's a basic assumption on all of this source coding. So knowing where the beginning is, you sit there and you look at it, and you see a zero as the first letter as a first binary digit,

and zero says I move this way in the tree, and presto, I say gee, an a must have occurred as the first source letter.

So what do I do? I remove the a, I print out a, and then I start to look at this point. At this point I'm back where I started at, so if I can decode the first letter, I can certainly decode everything else. If the first letter is a b, what I see is a 1 followed by a 1. Namely, when I see the first binary 1 come out of the channel, I don't know what was said. I know either a b or c was sent. I have to look at the second letter, the second binary digit resolves my confusion. I know that the first source letter was in a b, if it's 1 1, or a c, if it's 1 zero. I decode that first source letter and then where am I? I'm either on this tree or on this tree, each of which goes extending off into the wild blue yonder. So this says if I know where the beginning is, I can decode the first letter. But if I can decode the first letter, I know where the beginning is for everything else. Therefore, I can decode that also. Well, aside from any small amount of confusion about what uniquely decodable means, that's a perfectly fine mathematical proof.

So, prefix-free codes are, in fact, uniquely decodable and that's nice. So then there's a question. What is the condition on the lengths of a prefix-free code which allow you to have unique decodability? The Kraft inequality is a test on whether there are prefix-free codes or there are not prefix-free codes connected with any given set a code word lengths. This is a very interesting enough inequality. This is one of the relatively few things in information theory that was not invented by Claude Shannon. You sit there and you wonder why didn't Claude Shannon realize this? Well, it's because I think he sort of realized that it was trivial. He sort of understood it and he was really eager to get on to the meat of things, which is unusual for him because he was somebody, more than anyone else I know, who really understood why you should understand the simple things before you go on to the more complicated thing. But anyway, he missed this. Bob Fano, who some of you might know, who was a professor emeritus over in LCS, was interested in information theory. Then he was teaching a graduate course back in the '50s here at MIT, and as he often did, he threw out these problems and said nobody knows

how to figure this out. How kinds of lengths can you have on prefix-free codes, and what kinds of lengths can't you have? Kraft was a graduate student at the time. The next day he came in with this beautiful, elegant proof and everybody's always known who Kraft is ever since then. Nobody's ever known what he did after that. But at least he made his mark on the world as a graduate student. So, in a sense, those were good days to be around, because all the obvious things hadn't been done yet.

But the other thing is you never know what the obvious things are until you do them. This didn't look like an obvious problem ahead of time. Don't talk about a number of other obvious things that cuts off, because somebody was looking at it in a slightly different way than other people were looking at it. You see, back then people said we want to look at these variable length codes because we want to have some capability of mapping improbable symbols into long code words and probable symbols into short code words. You'll notice that I've done something strange here. That was our motivation for looking at variable length codes, but I haven't said a thing about probability. All I'm dealing with now is the question of what is possible and what is not possible. We'll bring in probability later, but now all we're trying to figure out is what are the sets of code word lengths you can use, and what are the sets of code word lengths you can't use.

So what Kraft said is every prefix-free code for an alphabet x with code word lengths l of x for each letter in the alphabet x satisfies the sum 2 to the minus length less than or equal to 1. In other words, you take all of the code words in the alphabet, you take the length of each of those code words, you take 2 to the minus l of that length. And if this inequality is not satisfied, your code does not satisfy the prefix condition, there's no way you can create a prefix-free code which has these lengths, so you're out of luck. So you better create a new set of lengths which satisfies this inequality. There's also a simple procedure you can go through which lets you construct a code which has these lengths. So, in other words, this, in a sense, is a necessary and sufficient condition on the possibility of constructing codes with a particular set of lengths. It has nothing to do with probability. So it's, in a sense, cleaner than these other results. So, conversely, if this inequality is satisfied, you can construct a prefix-free code, and even more strangely, you can construct it

very, very easily, as we'll see. Finally, a prefix-free code is full -- you remember what a full prefix-free code is? It's a code where the tree has nothing that's unused if and only if this inequality is satisfied with a quality. So it's a neat result. It's useful in a lot of places other than source coding. If you ever get involved with designing protocols for computer networks or protocols for any kind of computer communication, you'll find that you use this all the time, because this says you can do some things, you can't do other things.

So let's see why it's true. I'll give you another funny proof that doesn't look like a proof but it really is. What I'm going to do is to associate code words with base 2 expansions. There's a little Genie that early in the morning leaves things out of these slides when I make them. It wasn't me, I put it in. So we're going to prove this by associating code words with base 2 expansions, which are like decimals, but decimals to the base 2. In other words, we're going to take a code word, y1, y2 up to y sub m where y1 is a binary digit, y2 is a binary digit. This is a string of binary digits, and we're going to represent this as a real number. The real number is the decimal, but it's not a decimal, it's a becimal, if you will, which is dot y1, y2 up to y sub m. Which means y1 over 2 plus y2 over 4 plus dot dot dot plus y sub m over 2 to the minus m. If you think of it, an ordinary becimal, y1, y2 up to y sub m, means y1 over 10 plus y2 over 100 plus y3 over 1,000 and so forth. So this is what people would have developed for decimals if, in fact, we lived in a base 2 world instead of a base 10 world. If you were born without fingers and you only had two fingers, this is the number system you would use.

When we think about decimals there's something more involved. We use decimals all the time to approximate things. Namely, if I say that a number is 0.12, I don't mean usually that it's exactly 12 one hundredths. Usually I mean it's about 12 one hundredths. The easiest way to do this is to round things down to two decimal points. In other words, when I say 0.12, what I really mean is I am talking about a real number which lies between 12 one hundredths and 13 one hundredths. It's greater than or equal to 12 one hundredths and it's less than 13 one hundredths. I'll do the same thing in base 2. As soon as I do this you'll see where the Kraft

inequality comes from.

So I'm going to have this interval here, which the interval associated with a binary expansion to m digits, there's a number associated with it which is this number here. There's also an interval associated with it, which is 2 to the minus m. So if I have a code consisting of 0, 0, 0, 1 and 1, what I'm going to do is represent zero zero as a binary expansion, so 0, 0, is a binary expansion is 0.00, which is zero. But also as an approximation it's between zero and 1/4. So I have this interval associated with 0, 0, which is the interval from zero up to 1/4. For the code word zero 1, if I'm trying to see whether that is part of a prefix code, I map it into a number, 0.01 as a binary expansion. This number corresponds to the number 1/4, and it also corresponds into sub length 2 to an interval of size 1/4. So we go from 1/4 up to 1/2. Finally, I have 1, which corresponds to the number 1/2, and since it's only one binary digit long, it corresponds to the interval 1/2 to 1. Namely, if I truncate thing to one binary digit, I'm talking about the entire interval from 1/2 to 1.

So where does the Kraft inequality come from and what does it have to do with this? Incidentally, this isn't the way that Kraft proved it. Kraft was very smart. He did this as his Master's thesis, too, I believe, and since he wanted it to be his Master's thesis he didn't want to make it look quite that trivial or Bob Fano would have said oh, you ought to do something else for a Master's thesis also. So he was cagey and made his proof look a little more complicated.

So, if a code word x is a prefix of code word y, in other words, y has some binary expansion, x has some binary expansion which is the first few letters of y. Then the number corresponding to x and the interval corresponding to x, namely, x covers that entire range of decimal expansions which start with x and goes up to something which differs from x only in that mth binary digit. In other words, let me show you what that means in terms of here. If I tried to create a code word 0, 0, 0, 1, 0, 0, 0, 1 would correspond to the number 1/16. 1/16 lies in that interval there. In other words, any time I create a code word which lies in the interval corresponding to another code word, it means that this code word has a prefix of that code word. Sure enough it does -- 0, 0, 0, 1, this has this as a prefix. In other words, there is a

perfect mapping between intervals associated with code words and prefixes of code words. So in other words, if we have a prefix-free code, the intervals for each of these code words has to be distinct.

Well, now we're in nice shape because we know what the size of each of these intervals is. The size of the interval associated with a code word of length 2 is 2 to the minus 2. To be a prefix-free code, all these intervals have to be disjoint. But everything is contained here between zero and 1, and therefore, when we add up all of these intervals we get a number which is at most 1. That's the Kraft inequality. That's all there is to it. There was one more thing in it. It's a full code if and only if the Kraft inequality is satisfied with a quality. Where was that? The code is full if and only if the expansion intervals fill up zero and 1. In other words, suppose this was 1 zero, which would lead into 0.1 with an interval 1/2 to 3/4, and this was all you had, then this interval up here would be empty, and, in fact, since this interval is empty you could shorten the code down. In other words, you'd have intervals which weren't full which means that you would have code words that could be put in there which are not there. So, that completes the proof.

So now finally, it's time to define unique decodability. The definition in the notes is a mouthful, so I broke it apart into a bunch of different pieces here. A code c for a discrete source is uniquely decodable if for each string of source letters, x1 up to x sub m, these are not distinct letters of the alphabet, these are just the things that might come out of the source. x1 could be the same as x2, it could be different from x2. If all of these letters coming out of the source, that corresponds to some concatenation of these code words, namely, c of x1, c of x2 up to c of x sub m. So I have this coming out of the source, this is a string of binary digits that come out corresponding to this, and I require that this differs from the concatenation of the code words c of x1 prime up to c of xm prime. For any other string, x1 prime x2 prime, x of m prime of source letters. Example of this, the thing that we were trying to construct before c of a equals 1c of b equals zero, c of c equals 1 zero, doesn't work because the concatenation of a and b yields 1 zero, c of x1 -- take x1 to be a, take x2 to be b. This concatenation, c of x1, c of x2 is c of a, c of b equals 1 zero. C of c equals 1 zero, and therefore, you don't have something that works. Note that n

here can be different from m here. You'll deal with that in the homework a little bit, not this week's set. But that's what unique decodability says.

Let me give you an example. Here's an example. Turns out that all uniquely decodable codes have to satisfy the Kraft inequality also. Kraft didn't prove this. In fact, it's a bit of a bear to prove it, and we'll prove it later. I suspect that about 2/3 of you will see the proof and say ugh, and 1/3 of you will say oh, this is really, really interesting. I sort of say gee, this is interesting sometimes, and more often I say ugh, why do we have to do this? But one example of a code which is uniquely decodable is first code word is 1, second code word is 1, 0, third is 1, 0, 0, and the fourth is 1, 0, 0, 0. It doesn't satisfy the Kraft inequality with the quality, it satisfies it with inequality. It is uniquely decodable. How do I know it's uniquely decodable by just looking at it? Because any time I see a 1 I know it's the beginning of a code word. So I look at some along binary string, it starts out with the 1, I just read digits till it comes to the next one, I say ah-ha, that next 1 is the first binary digit in the second code word, the third 1 that I see is the first digit in the third code word and so forth.

You might say why don't I make the 1 the end of the code word instead of the beginning of the code word and then we'll have the prefix condition again. All I can say is because I want to be perverse and I want to give you an example of something that is uniquely decodable but doesn't satisfy the Kraft inequality. So it's a question. Why don't we just stick the prefix-free codes and forget about unique decodability? You won't understand the answer to that really until we start looking at things like Lempel Ziv codes, which are, in fact, a bunch of different things all put together which are, in fact, very, very practical codes. But they're not prefix-free codes, and you'll see why they're not prefix-free codes when we study them. Then you will see why we want to have a definition of something which is more involved than that.

So don't worry about that for the time being. For the time being, the correct idea to take away from this is that why not just use prefix-free codes, and the answer is for quite a while we will. We know that anything we can do with prefix-free codes we

can also do with uniquely decodable codes, anything we can do with uniquely decodable codes, we can do with prefix-free codes. Namely, any old code that you invent has like certain set of lengths associated with the code words, and if it satisfies the Kraft inequality, you can easily develop a prefix-free code which has those lengths and you might as well do it because then it makes the coding a lot easier.

Namely, if we have a prefix-free code -- let's go back and look at that because I never mentioned it and it really is one of the important advantages of prefix-free codes. When I look at this picture and I look at the proof of how I saw that this was uniquely decodable, what we said was you start at the beginning and as soon as the decoder sees the last binary digit of a code word, the decoder can say ah-ah, it's that code word. So it's instantaneously decodable. In other words, all you need to see is the end of the code word and at that point you know it's the end. Incidentally, that makes figuring out when you have a long sequence of code words and you want to stop the whole thing, it makes things a little bit easier. This example we started out with of -- I can't find it anymore -- but the example of a uniquely decodable, but non-prefix-free code, you always had to look at the first digit of the next code word to know that the old code word was finished. So, prefix-free codes have that advantage also.

The next topic that we're going to take up is discrete memoryless sources. Namely, at this point we have gone as far as we can in studying prefix-free codes and uniquely decodable codes strictly in terms of their non-probabalistic properties. Namely, the question of what set of lengths can you use in a prefix-free code or uniquely decodable code, and what sets of lengths can't you use. So the next thing we want to do is to start looking at the probabilities of these different symbols and looking at the probabilities of the different symbols. We want to find out what sort of lengths we want to choose. There will be a simple answer to that. In fact, there'll be two ways of looking at it, one of which will lead to the idea of entropy, and the other which will lead to the idea of generating an optimal code. Both of those approaches are extremely interesting. But to do that we have to think about a very simple kind of

source. The simple kind of source is called a discrete memoryless source. We know what a discrete source is -- it's a source which spews out a sequence of symbols from this finite alphabet that we know and the decoder knows.

The next thing we have to do is to put a probability measure on the output of the source. There's a little review of probability at the end of this lecture. You should read it carefully. When you study probability, you have undoubtedly studied it like most students do, as a way of learning how to do the problems. You don't necessarily think of the generalizations of this, you don't necessarily think of why is it that when you define a probability space you start out with a sample space and you talk about elements in the sample space, what's their sample points. What do those sample points have to do with random variables and all of that stuff? That's the first thing you forget when you haven't been looking at probability for a while. Unfortunately, it's something you have to understand when we're dealing with this because we have a bunch of things which are not random variables here.

These letters here are things which we will call chance variables. A chance variable is just like a random variable but the set of possible values that it has are not necessarily numbers, they're just events, as it turns out. So the sample space is just some set of letters, as we call them, which are really events in this probability space. The probability space assigns probabilities to sequences of letters. What we're assuming here is that the sequence of letters are all statistically independent of each other.

So for example, if you go to Las Vegas and you're reporting the outcome of some gambling game and you're sending it back your home computer and your home computer is figuring out what your odds are in black jack or something, then every time the dice are rolled you get an independent -- we hope it's independent if the game is fair -- outcome of the dice. So that what we're sending then, what we're going to encode is a sequence of independent, random -- not random variables because it's not necessarily numbers that you're interested in, it's this sequence of symbols. But if we deal with the English text, for example, the idea that the letters in English text are independent of each other is absolutely ludicrous. If it's early

enough in the term that you're not overloaded already, I would suggest that those of you with a little time go back and read at least the first part of Shannon's original article about information theory where he talks about the problem of modeling English. It's a beautiful treatment, because he starts out same way we are, dealing with sources which are independent, identically distributed chance variables. Then he goes from there, as we will, to looking at Markov chains of source variables. Some of you will cringe at this because you might have seen Markov chains and forgotten about them or you might have never seen them. Don't worry about it, there's not that much that's peculiar about them. Then he goes on from there to talk about actual English language. But the point that he makes is that when you want to study something as complicated as the English language, the way that you do it is not to start out by taking a lot of statistics about English. If you want to encode English, you start out by making highly simplifying assumptions, like the assumption that we're making here that we're dealing with a discrete memoryless source.

You then learn how to encode discrete memoryless sources. You then look at blocks of letters out of these sources, and if they're not independent you look at the probabilities of these blocks. If you know how to generate an optimal code for IID letters, then all you have to do is take these blocks of length m where you'd have a probability on each possible block, and your generate a code for the block. You don't worry about the statistical relationships between different blocks. You just say well, if I make my block long enough I don't care about what happens at the edges, and I'm going to get everything of interest.

So the idea is by starting out here you have all the clues you need to start looking at the more interesting cases. As it turns out with source coding there's another advantage involved -- looking at independent letters is in some sense a worst case. When you look at this worst case, in fact, presto, you will say if the letters are statistically related, fine. I'd do even better. I could do better if I took that into account, but if I'm not taking it into account, I know exactly how well I can do.

So what's the definition of that? Source output is an unending sequence -- x1, x2, x3 -- of randomly selected letters, and these randomly selected letters are called

chance variables. Each source output is selected from the alphabet using a common probability measure. In other words, they're identically distributed. Each source output is statistically independent of the other source outputs, x1 up to x k plus 1. We will call that independent identically distributed, and we'll abbreviate it IID. It doesn't mean that the probability measure is 1 over m for each letter, it's not what we were assuming before. It means you can have an arbitrary probability assignment on the different letters, but every letter has the same probability assignment on it and they're all independent of each other.

So that's the kind of source we're going to be dealing with first. We will find out everything we want to know about how we deal with that source. You will understand that source completely and the other sources you will half understand a little later.