PROFESSOR: The AEP is probably one of the most difficult concepts we talk about in this course. It seems simple to start with. As I said before, it's one of those things where you think you understand it, and then you think you don't understand it.

When Shannon first came out with this theory, there were a lot of very, very good professional mathematicians who spent a long time trying to understand this. And who, in fact, blew it. Because, in fact, they were trying to look at it strictly in terms of mathematics. They were looking for strict mathematical theorems, they weren't looking to try to get the insight from it. Because of that they couldn't absorb it.

There were a lot of engineers who looked at it, and couldn't absorb it because they couldn't match it with any of the mathematics, and therefore they started thinking there was more there then there really was so, so this is, in fact, tricky.

What we're looking at is a sequence of chance variables coming from a discrete memoryless source. In other words, a discrete memoryless source is something that spits out symbols and each symbol is independent of each other symbol, each symbol has the same probability mass function as every other symbol.

We said it's a neat thing to look at this log pmf random variable, and a log pmf random variable is minus log of the probability of the particular symbol. So we have a sequence now of random variables.

And the expected value of that random variable, for each of the random variables, is the expected value of the log pmf which is, as we said before, is this thing we've called the entropy, which we're trying to get some insight into. So we have this log pmf random variable, we have the entropy, which is the expected value of it.

We then talked about the sequence of these random variables. We talked about the sample average of them, and the whole reason you want to look at this random variable is the sample average of a lot of these pmf's, since logs add when the thing you're taking the log of multiplies. What you wind up with is the sum of these log pmf's is, in fact, equal to minus the lof of the probability of the whole sequence.

In other words, you look at this whole sequence as a big, giant chance variable, which has capital X to the n different possible values. Namely, every possible sequence of length n. When you're talking about source coding, you have to find the code word for each of those m to the n different sequences.

So what you're doing here is trying to look at the probability of each of those m to the n sequences. We can then use Huffman coding, or whatever we choose to use, to try to encode those things.

OK, the weak law of large numbers applies here, and says that the probability that this sample average of the log pmf is close to the expected value of the log pmf. The probability that that's greater than or equal to epsilon is less than or equal to the variance of the log pmf random variable, divided by n times epsilon squared.

Now, we are going to take the viewpoint that n epsilon squared, is very small. Excuse me, it's very large. Epsilon we're thinking as a small number, and we're thinking of as a large number, but the game we always play here is to first pick some epsilong, as small as you want to make it, and then you make n larger and larger. And as n gets bigger and bigger, eventually this number gets very small. So that's the game that we're playing. It's y n times epsilon squared, we're thinking of it is a large number.

So what this says is the probability that this log pmf of the entire sequence, well the sample value of it, namely the log pmf divided by n, us close to H of X. The probability of that is less than or equal to this. We then define the typical set, T sub epsilon of n. This is the set of all typical sequences out of the source. So we're defining typical sequences in this way. It's a set of all sequences which are what we put into there. Well, it's what we put into there, but we're looking at the compliment of this set.

These are the exceptional things. These are the typical things. We're saying that the exceptional things here have a small probability when you make n big enough. This is saying that the exceptional things don't amount to anything, and the typical things fill up the entire probability space. So what were saying is the probability that this sequence is actually typical is greater than or equal to 1 minus something small. It says when n gets big enough the probability that you get a typical sequence out of the source is going to wan.

OK. We drew this in terms of a probability distribution, which I hope makes things look a little more straightforward. This is the distribution function of the sample average of this log pmf random variable. And what we're saying is that as n gets larger and larger, the thing that's going to happen because of the fact that the variance of the sample average -- the sample average is always a random variable. The average is not a random variable, it's a fixed number.

And what this is saying is that as n gets larger and larger, this distribution function here gets closer and closer to a stack. In other words, the thing that's happening is that as n gets big, you go along here, nothing happens suddenly you move up here and suddenly you move across there. That's what happens in the limit, mainly the sample average is, at that point, always equal to the average.

So as n goes to infinity, a typical set approaches probability 1. We express that in terms of the Chebyshev inequality in this way, but the picture says you can interpret it in any one of a hundred different ways. Because the real essence of this is not the Chebyshev inequality, the real essence of it is saying that as n gets bigger and bigger, this distribution function becomes a stack. So that's a nice way of thinking about it.

Let's summarized what we did with that. All of the major results about typical sets. The first of them, is this one. It's a bound on the number of elements in a typical set, I'm not going to rederive that again, but it comes from looking at the fact that all of the typical elements have roughly the same probability. All of them collectively fill up the whole space, and therefore you can find the probability of each of them by taking one and dividing by the probability of each of them. When you do that, you get two bounds. One of them says that this magnitude is less than 2 to the n times H of X plus epsilon. The other one says it's greater than 2 to the n H of X minus epsilon. And you have to throw in this little fudge factor here because of the fact that the typical set doesn't quite fill up the whole space.

What that all says is when n gets large the number of typical elements is approximately equal to 2 to the n times H of X. Again, think hard about what this means. This number here really isn't all like 2 to the n H of X. This n times epsilon, is going to be a big number. As n gets bigger, it gets bigger and bigger. But, at the same time, when you look at 2 to the n times H of X plus epsilon, and you know that H of X is something substantial and you know that epsilon is very small, in some sense it still says this.

And in source coding terms, it very much says this. Because in source coding terms, what you're always looking at is these exponents. Because if you have n different things you're trying to encode, it takes you log of n bits to encode them. So when you take the log of this, you see that the number of extra bits you needs to encode these sequences is on the order of n times epsilon. Which is some number of bits, but the real number bits you're looking at is n times H of X. So that's the major term, and this is just fiddly.

The next thing we found is that the probability of an element in a typical set is between 2 the minus n times H of X plus epsilon and 2 to the minus n times H of X minus epsilon. Which is saying almost the same thing as this is saying.

And again, there's this approximation which says this is about equal to 2 to the minus n of H of X. And this is an approximation in exactly the same sense that that's an approximation. Finally, the last statement is that the probability that this typical -- the probability that you get a typical sequence is greater than or equal to 1 minus this variance divided by n times epsilon squared, with the same kind of approximation.

The probability that you get the typical sequence is about 1. So what this is saying is there are hardly any exceptions in terms of probability. There are a huge number of exceptions. But they're all extraordinarily small in probability. So most of the space is all tucked into this typical set. Most of these, all of these typical sequences, have about the same probability. And the number of typical sequences is about 2 to the n times H of X.

Does that say that the entropy has any significance? It sure does. Because the entropy is really what's determining everything about this distribution, as far as probabilities are concerned. There's nothing left beyond that. If you're going to look at very long sequences, and in source coding we want to look at long sequences, that's the story. So the entropy tells a story. Despite what Huffman said.

Huffman totally ignored the idea of entropy and because of that he came up with the optimal algorithm, but his optimal algorithm didn't tell him anything about what was going on. This is not the [UNINTELLIGIBLE] Huffman, I think his algorithm is one of the neatest things I've ever seen, because he came up with it out of nowhere. Just pure thought, which is nice.

We then start to talk about fixed lenght to fized length source coding. This is not a practical thing. This is not something I would advise trying to do. It's something which is conceptually useful, because it's talks about anything you can do eventually, when you look at an almost infinite sequence of symbols from the source and you look at how many bits it takes you to represent them. Ultimately you need to turn that encoder on, at some point in pre-history, and pre-history in our present day is about one year ago. And you have to turn it off sometime in the distant future, which is maybe six months from now.

During that time you have to encode all these bits. So, in fact, when you get it all done and look at the overall picture, it's fixed length to fixed length. And all of these algorithms are just ways of doing the fixed length to fixed length without too much delay involved in them.

I didn't say everything there. What this typical set picture gives us there, is you can achieve an expected number of bits per source symbol, which is about n times H of X, with very rare failures. And if you try to achieve H of X minus epsilon bits per symbol, the interesting thing we found last time, was that the fraction of sequences you can encode was zilch. In other words, there is a very rapid transition here.

If you try to get by with too few bits per symbol, you die very, very quickly. It's not that your error probability is large, your error probability is asymptotically equal to 1. You always screw up. So that's the picture.

We want to go onto Markow sources. Let me explain why I want to do this first. When we're talking about the discrete memoryless sources, it should be obvious that that's totally a toy problem. There aren't any sources I can imagine where you would want to encode their output where you can reasonably conclude that they were discrete and memoryless. Namely, that each symbol was independent of each other symbol.

The only possibility I can think of is where you're trying to report the results of gambling or something. And gambling is so dishonest that they probably aren't independent anyway. You could use this is a way of showing they aren't independent, but it's not a very useful thing. So somehow we want to be able to talk about how do you encode sources with memory.

Well Markow sources are the easiest kind of sources with memory. And they have the nice property that you can include as much statistics in them as you want to. You can make them include as much of the structure you can find as anything else will do. So that people talk about much more general classes of sources, but these are really sufficient. These are sufficient to talk about everything useful. Not necessarily the nicest way to think about useful things but it's sufficient.

So a finite state in Markov chain, I assume you're somewhat familiar with Markov chaings from taking a probability courses. If not, you should probably review it there, because the notes go through it pretty quickly. There's nothing terribly complicated there, but a finite state in a Markov chain is a sequence of discrete chance variables, in that sense it's exactly like the discrete memoryless sources we were looking at.

The letters come from some finite alphabet, so in that sense it's like the discrete memoryless sources. But here the difference is that each letter depends on the letter before. Namely, before the Markov chain changes from one state to another state in one of these steps, it looks at the state it's in and decides where it's going to go next. So we have a transition probability matrix, you can think of this as a matrix, it's something which has values for every S in the state space, and for everys S prime in the state space. Which represents the probability that the state at time n is equal the this state S, and the state of time n minus 1 is equal to the states S prime.

So this tells you what's the probabilities are of going from one state to another. The important thing here is that this single step transition incorporates all of the statistical knowledge. In other words, this is also equal to the probability that S n is equal to this state S, given that S n minus 1 is equal to the state S prime, and also that S n minus 2 is equal to any given state of time n minus 2, and all the way back to S minus zero being any old state at time S zero.

So it says that this source loses memory, except for the first thing back. I like to think of this as a blind frog who has very good sensory perception, jumping around on lily pads. In other words, he can't see the lily pad, he can only sense the nearby lily pads. So he jumps from one lily pad to the next and when he gets to the next lily pad, he then decides which lily pad he's going go to go to next. That'll wake you up, anyway.

And we also want to define some initial pmf on the initial state. So you like to think of Markov chains as starting at some time and then proceeding forever into the future. That seems to indicate that all we've done is to replace one trivial problem with another trivial problem. In other words, on step memory is not enough to deal with things like English text and things like that. Or any other language that you like.

So the idea of a Markov source is that you create whatever set of states that you want, you can have a very large state space, but you associate the output of the source not with the states, but with the transitions from one state to another. So this gives an example of it, it's easier to explain the idea by simply looking at an example. In this particular example, which I think is the same as the one in the notes, what you're looking at is a memory of the two previous states.

So this is a binary source, it produces binary digits, just either zero or 1. If the previous two digits were zero zero, it says that the next digit is going to be a 1 with probability 0.1 and the next is going to be a zero with probability 0.9.

If you're in states 0,0 and the next thing that comes out is a zero, then at that point, namely at that point one advanced into the future, you have a zero as your last digit, a zero as a previous digit, and a zero as the digit before that. So your state has move from xn minus 2 and xn minus 1, to xn minus 1 and xn.

In other words, any time you make a transition, this digit here, which is the previous digit, has to always become that digit. You'll notice the same thing in all of these. When you go from here to there, this last digit becomes the first digit. When you go from here to here, the last digit because the first digit, and so forth. And that's a characteristic of this particular structure of having the memory represented by the last two digits.

So the kind of output you can get from this source -- I mean you see that it doesn't do anything very interesting. When you have two zeroes in the past, it tends to produce a lot more zeroes, it tends to get stuck with lots of zeroes. If you got a single 1 that goes over into this state, and from there it can either go there or there. Once it gets here, it tends to produce a large number of 1's.

So what's happening here is you have a Markov chain which goes from long sequences of zeroes, to transition regions where there are a bunch of zeros and 1's, and finally gets trapped into either the all zero state again, or the all 1 state again. And moves on from there.

So the letter is what's in this case the source output. If you know the old state, the source output specifies in this state. If you know the old state and the new state, that specifies the letter. In other words, one of the curious things about this chain is

I've arranged it so that, since we have a binary output, they're only two possible transitions from each state. Which says that the state plus the sequence of source outputs specifies the state at every point in time, the stayed at every point in time specifies the source sequence. In other words, the two are isomorphic to each other. One specifies the other.

Since one specifies the other, you can pretty much forget about the sequence and look at the state chain, if you want to. And therefore everything you know about Markov chains is useful here. Or if you like to think about the real source, you can think about the real source as producing these letters. So either way is fine because either one specifies the other.

When you don't have that property, and you look at a sequence of letters from this source, these are called partially specified Markov chains. And they're awful things to deal with. You can write theses about, but you don't get any insight about them. There's hardly anything that you would like to be true which is true. And these are just awful things to look at. So we won't look them.

One of the nice things about engineering is you can create your own models. Mathematicians have to look at the crazy models that engineers suggest to them. They have no choice. That's their job. But as an engineer, we can only look at the nice models. We can play and the mathematicians have to work. So it's nicer to be an engineer, I think. Although famous mathematicians only look at the engineering problems that appeal to them, so in fact, when they become famous the two groups come back together again. Because the good engineers are also good mathematician, so they become sort of the same group.

These transitions, mainly the transition lines that we draw on a graph like this, always indicate positive probability. In other words, that there's zero probability from going to here to here. You don't clutter up the diagram by putting a line in, which allows you to just look at these transitions to figure out what's going on.

One of the things that you learned when you study finite state Markov chains, is that a state s is accessible from some other state s prime, if the graph has some path from s prime to s. In other words, it's not saying you can go there in one step. It's saying that there's some way you can get there if you go long enough. Which means there's some probability of getting there. And the fact that there's a probability of getting there pretty much means that you're going to get there eventually.

That's not an obvious statement but let's see what that means here. This state is accessible from this state, in the sense that you can get from here to there by going over here and then going to here. If we look at the states which are accessible from each other, you get some set of states, and if you're in that set of states, you can never get out of it. Therefore, every one of those states remains with positive probability and you keep rotating back and forth between them in some way, but you never get out of them.

In other words, a Markov chain which doesn't have this property would be the following Markov chain. That's the simplest one I can think of. If you start out in this state you stay there. If you start out in this state, you stay there, That's not a very nice chain.

Is this a decent model for an engineering study? No. Because when you're looking at engineering, the thing that you're interested in, is you're looking at something that happens over a long period of time. Back at time infinity you can decide whether you're here, or whether you're here, and you might as well not worry a whole lot about what happened back at time minus infinity, as far as building your model. You may as well just build a model for this, or build a model for that.

There's another thing here, which is periodicity. In some chains, you can go from this state to this state in one step, well one, two steps. Or you can go there in one, two, three, four steps. Or you can go there in one, two, three steps and so forth. Which says, in terms of m the period of s is the greatest common denominator of path lengths from s back to s again. If that period is equal to 1, mainly if there's not some periodic structure which says the only way you can get back to a state is by coming back every two steps or every three steps or something, then again it's not a very nice Markov chain.

Because if you're modeling it, you might as well just model things over two states instead of over single states. So the upshot of that is you define these nice Markov chains, which are aperiodic, which don't have any of this periodic structure, and every state is accessible from every other state. And you call them ergodic Markov chains. And what ergodic means, sort of and as a more general principle, is that the probabilities of things are equal to the relative frequency of things. Namely, if you look at a very long sequence of things out of a Markov chain, what you see in that very long sequence should be representative of the probabilities of that very long sequence. The probabilities of transitions at various times should be the same from one time to another.

And that's whate ergodicity means. It means that the thing is stationery. You look at it at one time, it behaves the same way as at another time. It doesn't have any periodic structure, which means if you look at it at even times, it behaves differently from looking at it at odd times. That's the kind of Markov chain you would think you would have, unless you look at these odd ball examples of other things.

Everything we do is going to be based on the idea of ergodic Markov chains, because they're the nicest models to use. A Markov source then is a sequence of labeled transitions on an and ergodic Markov chain. Those are the only things we want to look at. But that's general enough to do most of the things we want to do.

And once you have ergodic Markov chains, there are a lot of nice things that happen. Mainly, if you try to solve this set of equations, namely if you want to say, well suppose there is some pmf function which gives me the relative frequency of a given state. Namely, if I look at an enormous number of states, I would like the state little s to come up with some relative frequency. All the time that I do it.

Namely, I wouldn't like to have one sample path which comes up with a relative frequency 1/2, and another sample path of almost infinite length which comes up with a different relative frequency. Because it would mean that different sequences of states are not typical of the Markov chain. That's another way of looking at what ergodicity means. It means that infinite length sequences are not typical anymore. It depends on when they start, when they stop. It depends on whether you start at an even time or an odd time.

Depends on all of these things, that -- all of these things that real sources shouldn't depend on. So, if you have relative frequencies then you should have those relative frequencies at time n and at time n minus 1. And probability of a particular symbol s, if the probabilities of the previous symbol s prime were the same values, q of s prime. We know the transition probabilities, that's q of s given s prime. The sum over s prime, of q of s prime, given capital Q of s given s prime, what is that?

That's the probability of x. In other words, if you start out at time n minus 1 with something pmf function on the states at time n minus 1, this is the formula you would use to calculate the probability the pmf function for states at time s.

This is the probability mass function for the states at the next unit of time, time n. If this probability distribution is the same as this probability distribution, then you say that you're in steady state, because you do this again, you plug this into here, it's the same thing. You get the same answer at time n plus 1. You plug it in again you got the same answer at time n plus 2, and so on forever. So you stay in steady state.

The question is, if you have a matrix here, can you solve this equation? Is it easy to solve? And what's the solution? There's a nice theorem that says, if the chain is ergodic, namely if it has these nice properties of transition probabilities, that corresponds to a particular kind of matrix here. If you have that kind of matrix, this is just a vector matrix equation. That vector matrix equation has a unique solution then for this probability little q, in terms of this transition probability.

It also has the nice property that if you start out with any old distribution, and you grind this thing away for a number of times, this q of x is going to approach the steady state solution. Which means if you start a Markov chain out in some known state, after awhile the probability that you're in state s is going to become this steady state probability. It gets closer and closer to it exponentially as time goes on.

So that's just arithmetic. These steady state probabilities are approached asymptotically from any starting state, i.e. for all s and s prime and s. The limit of the probability that s sub n, the state at time n, is equal to a given state s, given that the state at time zero was equal to s prime. This probability is equal to q of s and the limit as n goes to infinity. All of you know those things from studying Markov chains. I hope, because those are the main facts about Markov chains.

Incidentally I'm not interested in Markov chains here. We're not going to do anything with them, the only thing I want to do with them is to show you that there are ways of modeling real sources, and coming as close to good models for real sources as you want to come. That's the whole approach here.

How do you do coding for Markov sources? The simplest approach, which doesn't work very well, is to use a separate prefix-free code for each prior state. Namely, if I look at this Markov chain that I had, it says that when I'm in the state I want to somehow encode the next state that I go to, or the next letter that comes out of the Markov source. The things that can come out of the Markov source are either a 1 or a zero.

Now you see the whole problem with this approach. As soon as you look at an example, it sort of blows the cover on this. What's the best prefix-free code to encode a 1 and a zero? Where one appears with probability 0.9 and the other one appears with probability 0.1. What's the Huffman encoder do? It assigns one of those symols to 1 and one of them to zero. You might as well encode 1 to this one, and zero to that one. Which means that all of the theory, all it does is generate the same symbols that you had before.

You're not doing any compression at all. It's a nice thing to think about. In other words, by thinking about these things you then see the solution. And our solution before to these problems was that if you don't get anywhere by using a prefix-free code on a single digit, take a block of n digits and use a prefix-free code there. So that's the approach we will take here.

The general idea for single letters is this prefix-free code we're going to generate satisfies a Kraft inequality, you can use the Huffman algorithm, you get this property here. And this entropy, which is now a function of the particular state that we were in to start with, is just this entropy here. So this is a conditional entropy, which we get a different conditional entropy for each possible previous state. And that conditional entropy for each possible previous state, is what tells us exactly what we can do as far as generating a Huffman code for that next state.

This would work fine if you had a symbol alphabet of size 10,000 or something. It just doesn't work well when your symbol alphabet is binary.

If we start out in a steady state, then all of these probability stay in steady state. When we look at the number of binary digits per source symbol and we average them over all of the initial states, the initial states occur with these probabilities q of s. We have these best Huffman code. So the number of binary digits we're using per source symbol is really this average here. Because this is averaging over all the states you're going to go into. And the entropy of the source output, conditional on the chance variable s, is now, in fact, defined just as that average.

So they encoder transmits s zero, followed by the code word for s 1 using s zero. That specifies s 1, and then you encode x 2, using s 1 and so forth. And the decoder is sitting there and the decoder does exactly the same thing. Namely, the decoder first sees what s zero is, then it uses the code for s zero to decide what x 1 was, then it uses the code for s 1, which is determined by s 1, s one is determined by x 1, and it goes on and on like that.

Let me review a little bit about conditional entropy. I'm going pretty fast here and I'm not deriving these things because they're in the notes. And it's almost as if I want you to get some kind of a pattern sensitivity to these things, without the idea that we're going to use them a lot. You ought to have a general idea of waht the results are. We're not going to spend a lot of time on this because, as I said before, the only thing I want you to recognize is that if you ever want to model a source, this, in fact, gives you a general way of doing it.

So this general entropy then is using what we had before. It's the sum over all the states, and the sum over all of the source outputs of these log pmf probabilities. This general entropy of both a symbol and a state is equal to this combined thing. Which is equal, not surprisingly, to the entropy of the state. Namely, first you want to know what the state is that has this entropy. And then given the state, this is the entropy of the next letter, conditional on the state.

Since this joint entropy is less than or equal to H of s plus H of x, I think you just proved that in the homework, didn't you? I hope so. That says that the entropy of x conditional on s, is less than or equal to the entropy of x, which is not surprising. It says that if you use the previous state in trying to do source encoding, you're going to do better than if you don't use it. I mean the whole theory would be pretty stupid if you didn't. That's what that says.

As I told you before, the only way we can make all of this work is to use n-to-variable-length codes for each state. In other words, you encode n letters at the same time. If you look at the entropy of the first n states given a starting state, turns out to be n times H of x given s. By the same kind of rule that you were using before.

The same argument that you used to show that this is equal to that, you can use to show that this is equal to H of S 1, given S zero, plus H of S 2, given S 1, plus H of S 3, given S 2 and so forth. And by the stationarity that we have here, these are all equal, so you wind up with n times times this conditional entropy.

And since the source outputs specified the states, and the states specified the source outputs, you can then convince yourself that this entropy is also equal to n times H of X, given S. And once you do that, you're back in the same position we were in when we looked at n-to-variable-length coding when we were looking at discrete memoryless sources.

Namely, the only thing that happens when you're looking at n-to-variable length coding, is it that one fudge factor becomes a 1 over n. When you have a small symbol space by going two blocks, you get rid of this, you can make the expected length close to H of X, given S. Which means, in fact, that all of the memory is taking into account and it still is this one parameter, the entropy, that says everything.

The AEP holds -- I mean if you want to, you can sit down and just see that it holds, once you see what these entropies are, I mean the entropy you're using log pmf's, you're just looking at products of probabilities, which are sums of log pmf's, and everything is the same as before. And again, if you're using n-to-variable-length codes, you just can't achieve an expected length less than H of X, given S. So H of X, given S gives the whole story.

You should read those notes, because I've gone through that very, very fast, partly because some of you are already very familiar with Markov chains some of you are probably less familiar with it, so you should check that out a little bit on your own.

I want to talk about the Lempel Ziv universal algorithm, which was rather surprising to many people. Jacob Ziv is one of the great theorists of information theory. Before this time, he wrote a lot of very, very powerful papers. Which were quite hard to read in many cases. So it was a real surprise to people when he came up with this beautiful idea, which was a lovely, simple algorithm. Some people, because of that, thought it was Abe Lempel, who spends part of his year at Brandeis, who was really the genius behind it.

In fact, it wasn't Abe Lempel, it was Jacob Ziv who was the genius behind it. Abe Lempel was pretty much the one who really implemented it, because once you see what the algorithm is, it's still not trivial to try to find how to implement it in a simple, easy way. If you look at all the articles about it, the authors are Ziv and Lempel, instead of Lempel and Ziv, so why it got called Lempel Ziv is a mystery to everyone.

Anyway, they came up with two algorithms. One which they came up with in 1977, and people looked at their 1977 algorithm and said, oh that's much too complicated to implement. So they went back to the drawing board, came up with another one in 1978, which people said, ah, we can implement that. So people started implementing the LZ78 and of course, by that time, all the technology was much better. You could do things faster and cheaper then you could before, and what happened then is that a few years after that people were implementing LZ77, which turned out to work much better.

Which is often the way this field works. People do something interesting theoretically, people say, no you can't do it, so they simplify it, thereby destroying some of its best characteristics. And then a few years later people are doing the more sophisticated thing, which they should have started out doing at the beginning.

What is a Universal Data Compression algorithm? A Universal Data Compression algorithm, is an algorithm which doesn't have any probabilities tucked into it. In other words, the algorithm itself simply looks at a sequence of letters from an alphabet, and encodes it in some way. And what you would like to be able to do is somehow measure what the statistics are, and at the same time as you're measuring the statistics, you want to encode the digits.

You don't care too much about delay. In fact, one way to do this -- I mean, if you're surprised that you can build a good universal encoder, you shouldn't be. Because you could just take the first million letters out of the source, go through all the statistical analysis you want to, model the source in whatever way makes the best sense to you, and then build a Huffman encoder, which, in fact, encodes things according to that model that you have.

Of course you then have to send the decoder the first million digits, and the decoder goes through the same statistical analysis, and therefore finds out what code you're going to use, and then the encoder encodes, the decoder decodes and you have this million symbols of overhead in the algorithm, that if you use the algorithm for a billion letters instead of a million letters, then it all works pretty well.

So there's a little bit of that flavor here, but the other part of it is, it's a neat algorithm. And the algorithm measures things in a faster way then you would believe. And as you look at it later you say, gee, this makes a great deal of sense even if there isn't much statistical structure here.

In other words, you can show that if the source really is a Markov source, then this algorithm will behave just as well, asymptotically, as the best algorithm you can design for that Markov source. Namely, it's so good that it will in fact measure the statistics in that Markov model and implement them.

But it does something better than that. And the thing which is better is that, if you're going to look at this first million symbols and your objective then is to build a Markov model, after you build the Markov model for that million symbols, one of the things that you always question about is, should I have used a Markov model or should I have used some other kind of model? And that's a difficult question to ask. You go through all of the different possibilities, and one of the nice things about the Lempel Ziv algorithm is, in a sense, it just does this automatically.

If there's some kind of statistical structure there, it's going to find it. If it's not Markov, if it's some other kind of structure, it will find it. The question is how does it find this statistical structure without knowing what kind of model you should use to start with? And that's the genius of things which are universal, because they don't assume that you have to measure particular things in some model that you believe in. It just does the whole thing all at once.

If it's running along and the statistics change, bing, it changes, too. And suddenly it will start producing more binary digits per source symbol, or fewer, because that's what it has to do. And that's just the way it works. But it does have all these nice properties.

It has instantaneous decodability. In a sense, it is a prefix-free code, although you have to interpret pretty carefully what you mean by that. We'll understand that in a little bit. But in fact, it does do all of these neat things. And there are better algorithms out there now, whether they're better in terms of the trade-off between complexity and compressability, I don't know.

But anyway, the people who do research on these things have to have something to keep them busy. And they have to have some kind of results to get money for, and therefore they claim that the new algorithms are better than the old algorithms. And they probably are, but I'm not sure. Anyway, this is a very cute algorithm.

So what you're trying to do here, the objective, one objective which is achieved, is if you observe the output from the given probability model, say a Markov source, and I build the best code I can for that Markov source, then we know how many bits we need per symbol. Number of bits we need per symbol is this entropy of a letter of a symbol given the state. That's the best we can do.

How well does the Lempel Ziv algorithm do? Asymptotically, when you make everything large it will encode using a number of bits per symbol, which is H of X, given S. So it'll do just as well as the best thing does which happens to know the model to start with. As I said before the algorithm also compresses in the absence of any ordinary kind of statistical structure. Whatever kind of structure is there, this algorithm sorts it out.

It should deal with gradually changing statistics. It does that also, but perhaps not in the best way. We'll talk about that later.

Let's describe it a little bit. If we let x 1, x 2 blah blah blah, be the output of the source, and the alphabet is some alphabet capital X, which has size m, let's just as notation, let x sub m super n denote the string xm, xm plus 1, up to xn. In other words, in describing this algorithm we are, all the time talking about strings of letters taken out of this infinite length string that comes out of the source.

We want to have a nice notation for talking about a sub string of the actual sequence. We're going to use a window in this algorithm. We want the window to have a size with the power of 2. Typical values for the window range from about a thousand up to about a million. Maybe they're even bigger now, I don't know. But as we'll see later, there's some constraints there.

What the Lempel Ziv algorithm does, this LZ77 algorithm, is it matches the longest string of yet unencoded -- this is unencoded also, isn't it, that's simple enough -- of yet unencoded symbols by using strings starting in the window. So it takes this sequence of stuff we haven't observed yet, it tries to find the longest string starting there which it can match with something that's already in the window. If it can find something which matches with something in the window, what does it do? It's going to first say how long the match was, and then it's going to say where in the window it found it. And the decoder is sitting there, the decoder has this window which it observes also, so the decoder can find the same match which is in the window.

Why does it work? Well, it works because with all of these AEP properties that we're thinking of, you tend to have typical sequences sitting there in the window. And you tend to have typical sequences which come out of the source. So the thing we're trying to encode is some typical sequence -- well you can you can think of short typical sequences and longer typical sequences. We try to find the longest typical sequence that we can. And we're looking back into this window, and there are enormous number of typical sequences there. If we make the typical sequences short enough, there aren't too many of them, and most of them are sitting there in the window. This'll become clearer as we go.

Let's go on and actually explain what the algorithm does. So here's the algorithm. First, you take this large W, this large window size, and we're going to encode the first thought W symbols. We're not even going to use any compression, that's just lost stuff. So we encode this first million symbols, we grin and bear it, and then the decoder has this window of a million symbols. We at the encoder have this window of a million symbols, and we proceed from there.

So it gets amortized, so we don't care. So we then have a pointer, and we set the pointer to W.

So the pointer is the last thing that we encoded. So we have all this encoded stuff starting at time P, everything beyond there is as yet unencoded. That's the first step in the algorithm. So far, so good.

The next step is to find the largest n, greater than or equal to 2, I'll explain why greater than or equal to 2 later, such that the string x sub p plus 1 up to p plus n, what is that? It's the string which starts right beyond the pointer, namely the string that starts here, what we're trying to do is find the largest n, in other words the longest string starting here, which we can match with something that's in the window. Now we look at a, a is in the window, we look at a b, a b as in the window. We look at a b a, a b a as in the window. a b a b, a b a b is not in the window. At least I hope it's not in the window or I screwed up. Yeah, it's not in the window.

So the longest thing we can find which matches with what's in the window is this match of length three. So this is finding the longest match which matches with something here.

This next example, I think the only way I can regard that is as a hack. It's a kind of hack that programmers like. It's very mysterious, but it's also the kind of hack that mathematicians like because in this case this particular hack makes the analysis much easier. So this is another kind of match. It's looking for the longest string here, starting at this pointer. a b a b so forth, which matches things starting here. Starting somewhere in the window. So it finds a match a b here. a b a here. But now it looks for a b a b, a match of four. Where do we find it? We can start back here, which is still in the window, and what we see is a b a b. So these four digits match these four digits.

Well you might say foul ball, because if I tell you there's a match of four and I tell you where it is, in fact, all the poor decoder knows is this. If I tell you there's a match of four and it starts here, what's the poor decoder going to do? The poor decoder says, ok, so a is that digit two digits ago, so that gives me the a. So I know there's an a there. b is the next digit, so b is there. And then I know the first two digits beyond the window, and therefore this third digit is a, so that must be that digit. The fourth digit is this digit, which must be that digit. OK?

If you didn't catch that, you can just think about it, it'll become clear. I mean it really is a hack. It's not very important. It won't change the way this thing bahaves. But it does change the way you analyze it

So that's the first thing you do, you look for these matches. Next thing we're going to do is we're going to try to encode the matches. Namely, we're going to try to encode the thing the we found in the window. How do we encode what we found in the window? Well the first thing we have to do -- yeah?

AUDIENCE: What if you don't find any matches?

PROFESSOR: I'm going to talk about that later. If you don't find any matches, I mean what I was looking for was matches of two or more. If you don't find any matches of two or more, what you do is you just take the first letter in the window and you encode that without any compression.

I mean our strategy here is to always send the length of the match first. If you say the length of the match is one, than the decoder knows to look for uncompressed symbols, instead of looking for something in the window. So it takes care of the case where there haven't been any occurrences of symbol anywhere in the window. So you only look for matches of length two or more.

So then you use something called a unary-binary code. Theoriticians always copy everybody else's work. The unary-binary code was due to Peter Elias, who was the head of this department for a long time. He just died about six months ago. He was here up until his death. He used to organize department colloquia. He was so essential that since he died, nobody's taken over the department colloquia. He was my thesis adviser, so I tend to think very kindly of him And he was lots of other things.

But anyway, he invented this unary-binary code, which is a way of encoding the integers, which has a lot of nice properties. And they're universal properties, as you will see. The idea is to encode the integers, there are infinite number of integers. What you'd like to do, somehow or other, is have shorter code words for lower integers, and longer code words for longer integers. In this particular Lempel Ziv algorithm, it's particularly important to have the lenght of the code words growing as the logarithm of n.

Because then anytime you find a really long match, and you got a very large n, you're encoding a whole lot of letters, and therefore you don't care if there's an overhead which is proportional to log n. So you don't mind that. And if there's a very small number of letters encoded, you want something very efficient then. So it does that.

And the way it does it is, first you generate a prefix, and then you have a representation in base 2. Namely base 2 expansion. So the number n, the prefix here, I think I said it here, the positive integer n is encoded into the binary representation of n, preceded by a prefix of integer part of log to the base 2 of n zero. Now what's the integer part of log to the base 2 of 1? Log to the base 2 of 1 is zero. It was a prefix of zero zeros. So zero zeros is nothing. So the prefix is nothing, the expansion of 1, in a base 2 expansion or any other expansion, is 1. So the code word for 1 is 1.

If you have the number 2, log to the base 2 of 2 is 1. The integer part of 1 is 1, so you start out with a single zero and then you have the expansion, 2 is expanded as 1 zero. And so forth. Oh and then 3 is expanded as 1 1, again with a prefix of 1. Four is encoded as 1 zero zero, blah blah blah and so forth.

Why don't you just leave the prefix out? Anybody figure out why I need the prefix there?

AUDIENCE: If without those prefixes, you don't a prefix-free code.

PROFESSOR: Yeah Right. If I left them out, everything would start with 1. I would get to the 1, I would say, gee, is that the end of it or isn't it the end of it? I wouldn't know. But with this, if I see 1, the only code word that starts with 1 is this one. If it's 2 or 3, it starts with zero and then there's a 1, which says it's on that next branch which has probably 1/4. I have a 1 0, 1 1, a prefix of 0 0, followed by a 1, put me off on another branch. And so forth.

So, yes, this is a prefix-free code. And it's a prefix-free code which has this nice property that the number of digits in the code word is approximately 2 times log n. Namely, it goes up both ways here. The number of zeros I need is log to the base 2 of n. The number of digits in the base 2 expansion, is also the integer part of log to the base 2 of n. So it works both ways. And there's always this 1 in the middle.

Again, it's a hack. It's a hack that works very nicely when you try to analyze this.

OK so if the size of the match is bigger than one, we're going to encode the positive integer u. u was where the match occurred. How far back do you have to count before you find this match? You're going to encode that integer u into a fixed length code of length log of w bits.

In other words, you have a window of size 2 to the twentieth. You can encode any point in there with 20 binary digits. The 20 binary digits say how far do you have to go back to find this code word.

So first we're encoding n, by this unary-binary code, then we're encoding w just with this simple minded way of encoding log w bits. And that tells us where the match is. The decoder goes back, there finds the match, pumps it out if n is equal to 1, here's the answer to your question, you encode the single letter without compression. And that takes care of the case, either where you have a match to that single letter, or there isn't any match to the single letter.

The next thing, as you might imagine, is you set the pointer to P plus n, because you've encoded n digits, and you go to step two. Namely, you keep iterating forever. Until the source wears out, or until the encoder wears out, or until the decoder wears out. You just keep going. That's all the algorithm is. Yeah?

AUDIENCE: Can you throw out the first n bits, when you reset the pointer, because you only have w bits that say where n was for the next iteration?

PROFESSOR: No, I throw out the n oldest bits in the window.

AUDIENCE: Well, those are the first n bits.

PROFESSOR: Yes the first n bits out of the window and I keep all of the more recent bits. I tend to think of the first ones as the things closest to the pointer, but you think of it either way, which is fine. So as you do it, the window keeps sliding along. That's what it does.

Why do you think this works? / There's a half analysis in the notes. i'd like to say a little bit about how that analysis is cheating, because it's not quite a fair analysis. If you look at the window, there are w different starting points in the window. So let's write this down. w starting points. So for any given n, there are there are w springs of length n. We don't know how long this match is going to be, but what I would like to do, if I'm thinking of a Markov source, is to say, OK, let's make n large enough so that the size of the typical set is about w. ok

So choose n to be about w divided by H of X given S. And the size of the typical set is then going to be 2 to the n, wait a minute.

n is equal to log w. So the size of the typical set then is T sub epsilon, is going to be roughly from what we said 2 to the n times H of X given S. So what I'm going to do is to set w equal to T of epsilon.

I'm going to focus on a match length which I'm hoping to achieve, of log w over H of X given S. The typical set, then, is a size 2 to the n times H of x give S. And if the typical set is of this size, and I look at these w strings in the window, yeah, I'm going to have some duplicates but roughly I'm going to have a large enough number of things in the window to represent all of these typical strings. Or most of them.

If I try to choose an n which is a little bigger than that, let's call this n star. If I try to make n a little bit bigger than this typical match size, I don't have a prayer of a chance, because the typical set then is just very much larger than w, so I'd be very, very lucky if I found anything in the window. So that can't work. If I make n a good deal smaller, then I'm going to succeed with great probability it seems, because I'm even allowing for many, many duplicates of each of these typical sets to be in the window.

So what this is saying is there ought to be some critical length when the window was very large, critical match length, and most of the time the match is going to be somewhere around this value here. And as w becomes truly humongous, and as the match size becomes large -- you remember for these typical sets to make any sense, this number has to be large. And when this number gets large, the size of the typical set it humongous. Which says, that for this asymptotic analysis, a window of 2 to the twentieth, probably isn't nearly big enough. So the asymptotic analysis is really saying, when you have really humongous windows this is going to work.

You don't make windows that large, so you have to have some faith that this theoretical argument is going to work here. But that tells you roughly what the size of these matches is going to be. If the size of the matches is that, and you use log w bits plus 2 log n bits, to encode each match, what happens?

Encode match. You use log w, plus 2 log n star. That's the number of bits it takes you, this is the number of bits it takes you to encode what the match size is. You still have to encode that. This is the number of bits it takes you to encode where the match occurs. Now how big is this relative to this?

Well n star is on the order of log w, so we're taking log w plus 2 times log of log of w. So in an approximate analysis, you say, I don't even care about that. You wind up with encoding a match with log w bits.

So you encode n star bits, you use log w bits to do it, how many bits are you using per symbol? H of X given S. That's roughly the idea of why the Lempel Ziv algorithm works. Can anybody spot any problems with that analysis? Yeah.

AUDIENCE: You don't know the probabilities beforehand, so how do you pick w?

PROFESSOR: Good one. You picked w by saying, I have a computer which will go at a certain speed. My data rate is coming in at a certain speed, and I'm going to pick w as large as I can keep up with. With the best algorithm I can think of for doing string matching. And string matching a hard thing to do, but it's not a terribly easy thing to do either. So you make w as large as you can. And if it's not large enough, tough. You got matches which are somewhat smaller -- all this argument about typical sets still work except for the epsilon and deltas that are tucked into there.

So it's just that the epsilons and the deltas get too big when you're strings are not long enough. Yeah?

AUDIENCE: So your w is just make your processing time equal the time [UNINTELLIGIBLE]?

PROFESSOR: Yeah. That's what the determines w. It's how fast you can do a string search over this long, long window. You're not going to just search everything one by one, you're going to build some kind of data structure there that makes these searches run fast.

Can anybody think of why you might not want to make w too large? This isn't a theoretical reason, this is more practical thing.

AUDIENCE: Is it if the probabilities change, it's slow to react to those changes?

PROFESSOR: If the probabilities change, it's slow to react to them, because it's got this humongous window here, and it's not until the window fills up with all of this new stuff, that it starts to work well. And before it fills up, you're using an effective small window, but you're using a number of bits which is proportionate to log of a large window, and therefore you're wasting bits.

So another thing that that determines how big you want w to be -- I mean the main thing that's determines it is just that you can't run that fast. Because you'd like to make it pretty big.

Another question. How about this matter of wasting w symbols at the beginning to fill up the window? what do you do about that? I mean, that's a pretty stupid thing, right? Anybody suggest a solution to that? If you're building this yourself, how would you handle it? It's the same argument as if the statistics change in mid-stream. I mean, you don't measure that the statistics have changed, and throw out what's in the window. What? AUDIENCE: Can you not assume that you already have a typical sequence?

PROFESSOR: Yes, and you don't care whether it's right or wrong. You could assume that the typical sequence is all zeros, so you fill up the window with all zeros. The decoder also fills it up with all zeros, because this is the way you always start. And then you just start running a log, looking for matches and encoding things. And as w builds up, you start matching things.

You could even be smarter, and know that you're window wasn't very big and let your window grow also. If you wanted to really be fancy about this. So if you want to encode this you can have a lot of fun, and a lot of people over the years have had a lot of fun trying to encode these things. It's a neat thing to do.