[SQUEAKING]

[RUSTLING]

[CLICKING]

ANKUR MOITRA: So today we're going to be talking about Huffman codes, which is a pretty elegant topic. So I just want to give you a reminder from last time. So what did we talk about last time was we talked about this notion of a first order source that defines a distribution on an alphabet of size k. So you think about the entries in the set A as being all the symbols that could arise. And then what we wanted was we wanted a way to compress the outputs of our first order channel.

So we did last time was we had this crucial definition, which was called a coding function. So we talked about a function which maps from length n sequences of the alphabet into some binary code of some varying lengths. So just the sequence of zeros and ones. And we want the crucial property that it can be decoded. So we want the property that for any two sequences that our channel could generate, as long as they're different, they map to different outputs.

And last time what we talked about was we talked about Shannon's theorem that gave us both upper and lower bounds on how well we could do. So in particular, the main thing we proved was this upper bound that we defined this quantity called the binary entropy of the channel. And we showed that the expected length of our code is at most n times the per symbol entropy plus little o of n.

So that was the context of what we did last time, and today we're going to drastically improve upon this. So first of all, you can wonder how small we can make this little o of n. But there's another point, which is a little bit more subtle. See, this scheme is not really so computationally efficient, because if I think about what I have to do, I might, in principle, have to write down one code word for each possible output of my channel.

So as the length of my channel becomes larger and larger, now I have k to the n possible strings. So I'm writing down this giant table of what are all the possible outputs, and that might be a very inefficient way to do these things, because maybe I don't want to come up with my coding scheme by brute force, by writing down everything that could possibly output from the channel and mapping it to what the coding function would send it to.

So we're going to do a lot better today with something called Huffman coding. In fact, let me tell you a bit of the history of Huffman coding. So Huffman coding was invented back in 1951 by Huffman. And it was invented right here at MIT, in fact, not by a professor, but by a grad student. So what happened here was that the grad student was taking a graduate seminar. And as is often the case with these types of things, students don't want to take final exams. I'm sure you guys don't.

And so his professor, Bob Fano, who's still here, he offered him the choice of whether he could write a term paper or do the final. So he chose, of course, writing a term paper. And the term paper had to be about improving something in the state of the art in coding theory. Coding theory was a topic that was just developing around that time. He was pretty stuck. And then he thought, OK, I better cut my losses and start studying for the final.

And then at the very last minute, he came up with this amazing new algorithm, which we teach in a lot of our undergraduate classes, as a neat example and a powerful example. So that's what we'll be covering today is some of the work that went into Huffman's PhD thesis. So as I mentioned, we'll come back to the comparison to Shannon later on. But let me give you a crucial definition. So we're going to be interested in a stronger notion of an encoding. We'll define what's called a prefix code.

Now in comparison to the coding function, that took an entire sequence of symbols of length n and it outputs some binary string. A prefix code is just going to take one single symbol from our alphabet, and it's going to output some binary string of variable length. But now we have a stronger condition on what we want out of this code.

So we want it to satisfy the condition that for any two input symbols ai and aj that are different, we want that phi of ai is not a prefix of phi of aj. So this is a stronger condition than them merely being different in terms of their binary strings. We want that whatever ai is mapped to, we want that that binary string doesn't occur in the front and the beginning of what aj is mapped to and vice versa. Of course, we want this to be true for all pairs of distinct symbols.

So in fact, what I claim is that there's a nice way to think about a prefix code as a binary tree. So let's develop a way to think about binary codes, prefix free codes, as-- sorry, prefix free codes. Let's develop a way to visualize them as binary trees. So first let me give you an example of a prefix free code. So let's imagine-- and this will be our running example today as we see the algorithm under the hood of Huffman coding.

So let's imagine that we have symbols a, b, C, d, e, and f. And I'm going to map these to the code words 1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, and 1, 1, 1. So you can check that this is a prefix free code, because you can check for each of the possible outputs for your encoding function. 1, 0 doesn't appear at the start of any of these strings, and that's true for all of the other encodings for each of these symbols.

And one of the ways that we can think about this is we can think about it as a binary tree in which each branch in the binary tree tells us whether the next symbol is either a 0 or 1. So in this case, let's arrange these into a binary tree where the leaves are going to represent what the actual symbols are. So when I have a symbol like d that my prefix free code is mapping to the string 0, 1, then I get that just by tracing from my root to that leaf, I concatenate these symbols in order and I get 0, 1.

And let me do this for all of the different symbols in my prefix free code. And every time going left down the tree is going to correspond to concatenating with a 0, and every time going right is going to correspond to concatenating with a 1. So when I put this all together I'll get a binary tree that looks like this.

And you can check me on the details that indeed each symbol is mapped to the position in the binary tree, where its associated word comes from just following the root to leaf path. And now what is the condition that this code is prefix free? Well, if I had one leaf, if I had one symbol appear right here, right in the middle of this tree, I would have another symbol that's a descendant of it. And that would correspond to a code that's not prefix free, because in that case, the way that I would reach this symbol would be a prefix of how I reach this later symbol.

So in fact, what this constraint on prefix freeness is, another way to think about it is that when I arrange all of the symbols into this binary tree describing their encoding, I want the property that each position in the tree that describes the encoding of a symbol has no other descendants that are symbols beneath it. So that's an equivalent way to think about prefix free. So any questions about this key definition? Does this make sense? Yeah? All right. So now we can talk about what's the important thing about prefix free codes. So what I claim is that it actually makes decoding much, much easier. So let's see why that is.

So when I talked about Shannon, I had to specify how the entire string of length n was mapped to a binary output. But now what I can do is I have a much simpler code that operates on a symbol by symbol basis. So all I'm going to do is I'm going to take my sequence of symbols, and I'm just going to map them to 0, 1 strings using my prefix free encoding.

And then I'm going to concatenate them all together. So let's do an example. What if I tell you I started off with a word and I ended up with this binary string. How would I figure out what was the sequence of symbols that I started off from? So we're going to have to use the fact that it's prefix free crucially. So how can I go about figuring out what the first letter is? Any guesses for the first letter? f.

So in fact, what I can do is I can see that if I chop off these first three symbols, that's the encoding of f. What's my next symbol? a. So you can see what's tricky here is that because my output has variable length, I don't know where the breaks are. I smushed everything together when I did this, but how did I know that the next break should be distance two? How did you guys figure that out? Yeah?

AUDIENCE: You traverse down the tree.

ANKUR MOITRA: That's right. That's right. And so if I traverse down the tree and hit a leaf, that's right. So we can continue this, we'll get the next letter is d, and last but not least, we'll get e. So I started off with the word fate. But now here's the crucial question. Why couldn't there be some other way, hypothetically, to break up the string into pieces in a different way that results in a totally different decoding of what my original string was.

You see, when I started off with Shannon's encoding theorem, I was doing it on the entire string, and the property was that two different strings mapped to different outputs. Now I'm doing this on a symbol by symbol basis. Why am I guaranteed, with my prefix free encoding, that there's no other way to break this up that makes sense? It's really because of the prefix freeness. So what if the break point had really happened at this yellow line, so that I pulled out the first two symbols?

Well, if that really mapped to another symbol, if there was a symbol whose associated code word was 1, 1, then the trouble is that my code word, my encoding scheme, would not be prefix free, because that symbol would have an encoding that's a prefix of that of f. So prefix freeness is just the constraint that you can sequentially do decoding, symbol by symbol, in an unambiguous way. I always know where the breaks are, even when I erase them. So does that make sense?

All right. So prefix freeness is an awesome property. It's not obvious how it compares to what the fundamental limits were that we proved last time using Shannon. We also don't know what's the best way to-- what's the way to choose the prefix free code that minimizes the expected length. So that'll be our goal today is we're going to be interested in the following very basic algorithmic problem. We want to design a prefix free code that minimizes expected length.

And just to be clear, what is our expected length here? It's just on a by symbol basis, we're thinking about outputs of our first order source that are just length 1 for simplicity. And what I want is I want that the sum over all of the symbols in my alphabet, the probability that the symbol is chosen times la, where la is the length of the encoding of phi of a.

I want to minimize this quantity, because intuitively, now, when I really care about the case where n is large and is not 1, well, my expected length is just going to be n times whatever I can achieve for a single symbol. And we know how encoding and decoding work, so our goal is just to solve this algorithmic problem. And then we're going to compare it back with Shannon. So are there any questions? Makes sense?

All right. So as I told you, Huffman, in lieu of taking a final exam at MIT, he designed a simple greedy algorithm for optimally solving this problem. And that's what we're going to do. We'll explain how the approach works through a running example. And our running example will be the same one that I wrote up there. Our running example-- let me just write it here so that we have it.

We'll consider the problem with symbols a, b, c, d, e, and f and their associated probabilities, because, of course, the optimal encoding depends crucially on these probabilities, will be 0.4, 0.05, 0.18, 0.07, 0.20, 0.10. And if I did this right, they all sum up to 1. So what we want to do is we want to understand what is the optimal prefix free encoding for this particular first order source. And we'll see that in solving this problem, we're going to come up with a general algorithm that works really for any channel.

OK. So now let's get started. So let's imagine, let me tell you what our strategy is going to be. We're going to start off with some encoding, which we remember we think about as a binary tree. And we're going to pretend that this encoding is optimal and we're going to derive structural properties about this optimal encoding that will be enough to figure out what the encoding is. So that's our game plan. So I'm going to pretend that someone has given me already an optimal prefix free code.

And what I'm going to do is I'm going to reverse engineer properties of this binary tree. Now, just for simplicity, I'm going to assume that there are no ties. What I mean by that is when I start off with all of these symbols, none of the symbols have exactly the same probability. It'll just be a bit more of a pain to talk about what the structural properties are when we allow ties, even though the same algorithm is essentially going to work as is.

So I'll assume that none of these symbols have probabilities that are tied. And we're going to create new synthetic symbols along the way. And I'm going to assume that there are no ties even after we make those operations. So now let's state the first key Lemma that's going to get us started in terms of structural properties of this tree t. So what I claim is that the smallest probability symbol, so whichever symbol has the smallest probability in our first order source, has the property that it's at least tied for the longest encoding.

So what I'm claiming is that whatever is the symbol with the smallest probability-- in this case, it's the symbol b here. I claim that symbol had better be in one of the deepest nodes in my binary tree that corresponds to my prefix free code. OK? So let's prove this fact. And then we're going to prove a related fact about the symbol with the second smallest probability. So this thing is very easy to prove once you know what you're trying to prove. Does anyone have any intuition for how we could try and prove this lemma? Why should it be true? Yeah.

AUDIENCE: If it is not tied for the longest, then you could swap it with something with higher probability.

ANKUR MOITRA: That's right. That's right. So that's exactly how our proof is going to go, because you can see the way that this proof is structured, it makes it a lot easier to have the algorithm fall out at the end because I'm not, in one shot, trying to tell you what the optimal tree is. I'm just trying to say, if you gave me the optimal tree, I could find some interesting structural properties that constrain something about the tree that I'm looking for.

So that's exactly right. Our proof is going to be, if not, we're going to swap and contradict the optimality of t to begin with. So let's do a proof by contradiction following exactly that logic. So suppose that this lemma is not true. Well, let's let b be the symbol with the lowest probability, like in our example. And let's let c be the symbol which is tied for longest encoding. Now the crucial thing is that if we swap the encodings of b and c.

Well, first of all that's still a valid prefix free code, because the property of being a prefix free code doesn't actually depend on how the symbols are mapped to the binary strings. It's purely a property of the binary strings I've selected. When you think about graphically how we visualize the prefix free code, it just had this property that all of these nodes, which were boxes that correspond to symbols being mapped to them, we need the property that no box had another box as its descendant.

And now if I interchange the contents of these boxes, if I swap b and c, then it doesn't actually change this combinatorial condition of being prefix free. So this is a valid move. We're allowed to swap the encodings. And what we can do is if we swap these encodings, we can track how the expected length changes. So let's in particular compute how the expected length changes.

So we have originally-- well, our new encoding is going to have the probability of the symbol b times the length of the encoding for symbol c plus the probability of the symbol c times the length of the encoding for b. That will be some of contribution to the new expected length, because every time I sample the symbol b, its encoding is what the encoding of c previously was. And every time I sample the symbol c, its length will be the length of the encoding of what b previously was.

But now I have to subtract off what it was before. So I'm going to get Pb times the length of the encoding for b minus Pc times the length of the encoding for c. So in particular, when I have my formula for the expected length, which is right here, notice that for none of the other symbols am I changing their probability or the length of their encoding. The only changes are happening between b and c. So when I look at the difference in the expected length of my old prefix free code and my new one, the change in the length is this quantity right here because all of the other things cancel out.

So let's figure out what's going on here. We can write this out in a more convenient way. This is just Pb times delta minus Pc times delta, where delta is equal to the length of the encoding of c minus the length of the encoding of b. And we know by assumption, because we're trying to prove something by contradiction, that by assumption, that c is the symbol with the longest encoding. So b is not tied for the same length of the encoding.

So we know that the difference between these encoding lengths is at least 1. So it's strictly positive. And we also know that Pb is less than Pc. Why is that true? Because of the way that I defined Pb. Pb was the symbol with the smallest probability, and I assume that there were no ties. So whatever Pb is, Pc is strictly larger than that. So if you look at these two quantities together, then what this means is that you actually have a change that's strictly negative.

So I've actually decreased the expected length of my encoding. So that's the proof of the first key lemma. It tells us this really nice structural property. And you can see that the proof is a very simple proof by contradiction, where I just have to figure out that I want to use the swap and make the calculation of how much I've improved the length, and argue that it really is an improvement. So are there any questions so far? Good?

All right. So I claim that there's a companion lemma that the same lemma holds also for the symbol with the second smallest probability. So let me state what I mean exactly by a version of this lemma that holds for the second smallest probability. And then we'll prove it. The proof will build on our proof of lemma one, and then we'll be off to the races. So what I claim is that this lemma holds for second smallest probability too. So let's formally state what this lemma is.

The second smallest symbol, I claim, is second smallest probability symbol, because that's how we're measuring their contribution, is tied for the longest encoding. And so the proof for this lemma is going to very closely mirror the proof for lemma 1, but it's just going to build on it. So let's prove this fact. Actually, in my example, what is my second smallest probability? Let's double check. So it's d.

So now what I'm claiming between lemma one and lemma two is that these two symbols both had better be tied for the longest encoding. So let's follow our lemma. So let's let b be the symbol with the lowest probability. Let's let d be the symbol with the second lowest probability.

And now what we know from lemma one, what we already proved, is that if we look in our optimal tree we start off at the root, we follow some path, and then we reach our symbol b, which is the symbol with the smallest probability. And let's imagine that we call P the parent. So whoever comes before. So this is not the image of a symbol, because we have this prefix free constraint, but it's still a node in our binary tree. There is some parent of wherever b gets mapped to.

And now what I claim is that but P, the way that I've drawn it, I claim must have another child. I claim that there must be some node in my binary tree that's the sibling of b. So let's make sure we're on the same page. So why is this claim true. Why can I assert that P has another child? If it didn't, what would it contradict? Yeah?

AUDIENCE: Would it not be a binary tree, then or cannot be represented?

ANKUR MOITRA: Well, it would still be a binary tree. So we'll allow 0, 1, or 2 children. But there's something else that's really crucial. So think back to my proof strategy. What am I trying to do? I'm trying to say that t is an optimal prefix free code. And then my first lemma was a structural statement about some property that must be true of that optimal binary tree. Lemma two is an even stronger structural property about that optimal binary tree. So what would happen if P really didn't have another child right here? Yeah?

AUDIENCE: No longer be optimal code.

ANKUR MOITRA: Yes. It would no longer be optimal. Why?

AUDIENCE: Because you have an empty space where you could have just--

ANKUR MOITRA: Right. But another way to think about it is that I know that there's no symbol that's mapped to P because it's prefix free. If P didn't have another child, I could just promote this node and delete everything underneath. So in this case, I would be strictly improving the expected length of my prefix free code because I'm not changing anything except for the length of the encoding of b, which is decreasing by 1.

So what I claim is that P must have another child, otherwise t is definitely not optimal. Does that make sense? So now I know that this child really must be there. And let's call this child c, which is just the sibling. And so what I claim now is that if I look in this tree and I have some path that goes to my other symbol, which is d, my second smallest probability, well, what could I do if d was not tied for the second-- not tied for the longest encoding?

We know that b has the longest encoding, so d better have an encoding length that's less than or equal to it. But what if it was strictly less than the length of the encoding of b? We know that there's another child c that has the same length encoding as b, so what should I do if d wasn't tied for the second longest encoding? I can do the same trick as before, right? I can swap.

So what I claim is that now, if d has a strictly shorter encoding, what I claim is that I can swap c and d and improve the expected length. So this is the same proof as before. So this is a bit subtle, but it's a purely combinatorial proof. Are there any questions? So the way that this worked logically, from lemma one, we know that b is tied for the longest encoding. So nothing can have an encoding length that's strictly longer.

We argued that b has a parent P and that P had better have another child. And if P has another child, this would better be some other symbol in terms of the encoding. And if d were strictly off this line and the length of its coding were shorter than the length of the encoding of b and c, what we could do is we could just swap d and c. And then the same type of computation that we did in the proof of lemma one will show us that we've strictly improved. Why?

Because the probability of d is strictly less than the probability of c. So these are our two key structural lemmas. And now we're actually almost in good shape. So we're almost ready to talk about Huffman's algorithm. So let me do one more claim. This last structural property is the thing that's going to make Huffman's greedy algorithm work. So I claim building off of lemma one and lemma two, we can say something even stronger.

What I claim is that without loss of generality, these two symbols, b and d, which are the smallest and second smallest probabilities, respectively, I claim without loss of generality their siblings in the tree. So why is this true? Well, from lemma one and lemma two, what do we know? We can draw it out pictorially. We know that we start off from the root. We take some path to the parent P and then we reach our symbol d.

And we know that P has some other child. Maybe it's c. And then we know that there's a path that reaches the other symbol b which has the second smallest probability according to our channel. And we know that all of these things are tied for the same length. So what could I do in this case is I could just swap the positions of b and c. That doesn't change anything about the length of the encoding, but it has the property now that b and d are siblings of each other.

So the proof for all of these structural statements, lemma one, lemma two, and this claim is really just, if not, swap. Lemma one was if b isn't the longest, then swap and you'll improve the length. Lemma two was, well, we know where b is, but if d is also not the same length, then swap and improve the length. And finally, even knowing that they're tied for the same length of the encoding, swap so that they're siblings. So this is the key structural property that we're going to use. Without loss of generality, b and d are siblings.

And now we can come back to our running example and we can build up what the optimal tree is. And we'll be able to-- Huffman's algorithm is just going to fall out as a consequence. So let's do the same intuition on this. So we don't know what our optimal tree looks like. I'll just draw it as a big blob. But we know from our claim that without loss of generality b and d are siblings of each other. And so now here's the idea.

What I'm going to do is I'm going to take these two symbols, and I'm going to replace them with a new symbol. So I'm going to write down a new symbol alpha. And so here's the step in my Huffman coding algorithm. I'm going to replace these symbols b and d with a new symbol, which I'm going to call alpha. And I'm going to have the property that my probability of this new symbol is going to be the sum of the probabilities of the symbols that I've merged.

So that's the key idea. So I'm going to set its probability to be equal to the sum of the two old probabilities. This way when I get rid of these two symbols from my list and instead add my symbol alpha at the bottom, I'm still going to have a valid distribution because I've just merged those two probabilities. But now what we're going to do, the main idea behind Huffman, is we're going to solve the optimal problem on the smaller set of symbols.

And then from the optimal binary tree for this new problem, we're going to derive what the optimal binary tree was for our original problem. So are there any questions about the strategy? Does this make sense? All right. So let's do that and let's see what Huffman would do. All right. So the corresponding transformation on the tree.

We started off with this tree t and now we're going to have this tree t prime. And what's going to happen is I've replaced the parent of b and d instead with this new symbol alpha that occurs in the same exact place in the tree. But then I've changed nothing else about the tree. So now we get to the key lemma. What I claim is that this new tree t prime is also optimal for our new problem, where our new alphabet is A prime.

So we take our old alphabet A, we add in this new element alpha. We subtract off the two symbols b and d. And we've already talked about how we update the probabilities. So this is the key property that I was assuming that someone gave me the optimal tree t for my original problem.

Now I said something structurally about without loss of generality, where b and d have to occur in relation to each other. And now I've created a subproblem on a smaller number of symbols. But I claim that just this new tree, t prime that I get, had actually be an optimal answer to the new problem that I've constructed. So does this make sense intuitively? OK. So let's prove this lemma.

So just for notation, let's let L be the expected length of T. I'm going to let L prime be the expected length of T prime, this new tree that I've constructed. And let's prove this lemma that T prime is optimal. So to make sure you guys are following, so does anyone have any intuition for how I can prove this lemma that T prime is again optimal? There's basically only one proof we have today. Yeah?

AUDIENCE: I was thinking you could swap it, but I was a little bit concerned because if you swap the probabilities together, maybe it gets big. Yeah.

ANKUR MOITRA: So let's think about what we're trying to prove. We're trying to prove that T prime is an optimal prefix free code for this new subproblem I've constructed on a smaller alphabet. So what proof strategy do you think makes sense to prove that T prime is optimal. Greedy, induction, some other proof strategy? Proof by contradiction? Yeah?

AUDIENCE: Proof by contradiction.

ANKUR MOITRA: OK. That's right. Because in fact, the easiest way to prove this is what if T prime is not optimal? Then what I can do is I could hope to construct-- I could take whatever tree is optimal for this subproblem. And then what would I want to do with that tree? Let's say that proof by contradiction is the right way to do this.

So suppose not. Suppose there's a better tree. Let's call this tree, I don't know, S. What should I do with my better tree? How am I trying to prove a proof by contradiction? So you can either answer the question or ask me a question about what we covered so far. So it's your choice, but someone's got to answer. Yeah?

AUDIENCE: I presume you can show that S is actually just deeper.

ANKUR MOITRA: S is actually T prime? That's true. But it's harder than I want. So some of these things, you have to have a little bit of intuition about which proof technique would be the easiest. So we already agreed it's proof by contradiction. But the question is what am I trying to contradict and how? Because that's how proof by contradiction works. So what can I do with S? Yeah?

AUDIENCE: Can you backtrack to make another version of T?

ANKUR MOITRA: Yeah. OK. So let's think about S. I like that. So S is some other blob. And it's better. What are the symbols that occur in S? Well, a, c, e, f, and alpha, my new symbol. So alpha appears somewhere. What should I do with alpha now?

Well, maybe I can try and construct a better encoding for my original problem. So that's the idea is that there's a proof by contradiction. So we suppose that there's a better tree S. And this better tree is a better tree for this particular problem on this new alphabet A prime. And what I want to do is I want to use this S to construct a better answer to my original problem that contradicts the optimality of T.

So think about how the logic of this proof worked. Someone gave me an optimal tree T. I proved a structural property about where b and d occur. And now what I'm doing is I'm saying that I want-- that when I contract b and d, the result I get is optimal for the corresponding subproblem. Because if it weren't optimal for the corresponding subproblem, then my original tree T could not have been optimal to begin with. So this is the key point for the logic of the proof.

Even though the proof is very simple, it takes a little bit to wrap your head around the logic. So let's imagine that there's a better prefix code S than the associated tree. And let's say that it has expected length, which is L double prime. OK? Remember, L prime was the expected length of T prime. So we're assuming for the purpose of contradiction that L double prime is strictly less than L prime, that it does strictly better. So in this case, what we can do is we can convert. We can invert the operation that we just did.

So we can invert the operation, which took us from the problem on the full alphabet A to the subproblem on alphabet A prime. We can replace the symbol alpha that was our new synthetic symbol by this gadget right here. And so when we do that, let's track what the new expected length is. So the new expected length is L double prime, because I still pay the expected length, until I-- for this answer S.

But now by replacing alpha with b and d, every time I incur-- encounter either b or d, I'm going to pay one extra unit of length, because previously I stopped at the symbol alpha. And now I have to continue. Wherever alpha was, that's not the end. I have to go further down to either b or d. That happens with probability Pb plus Pd, respectively.

So this is my new expected length. When I take my new better answer for this subproblem and replace it with inverting this operation, that's my new expected length. But when I think about how this compares to my old expected length, see, if I had the original tree T prime here, then I pay expected length L prime. And the way that I get from T prime to T was I do that same operation, where I take alpha and replace it with b and d.

So my old expected length was actually L prime plus Pd. Sorry, plus Pb plus Pd. And this is just equal to L. So does everyone see how I got this? This is the crucial point in the proof. If you don't understand this, you don't understand Huffman coding. And so the crucial thing right here is that now I've come up with a better solution for my original problem.

And this contradicts the optimality of T. OK. So this is the proof. And I'm not going to go on further until you convince me you understand the proof. So who can explain to me how this proof works in your own words? The one that's on the board. Or ask me a question. Those are your two options. I guess the third option is try and avoid eye contact, but that won't be as effective. Yeah, go for it.

AUDIENCE: So basically, we're trying to do it by contradiction, by saying that there's a more optimal tree S, and we're going to show that the expected-- that when we make S back into the same alphabet with T that S finds that another shape is more optimal.

ANKUR MOITRA: Yeah. Yeah.

AUDIENCE: Contradiction [INAUDIBLE].

ANKUR MOITRA: That's right.

AUDIENCE: So to do that, we calculated the expected length for S, and then given that S is more optimal than t Prime, you can know that optimal length is less than L prime plus the reverting L prime, which is equal to T.

ANKUR MOITRA: OK. Yeah. Very good. That was perfect. Let's stress test our understanding. I think the right way to understand proofs is to think about them like an engineer, to look at each individual piece. So we proved a bunch of things today, all getting to this key lemma. And one of the key things we had was this claim, that without loss of generality, in the optimal tree b and d, the symbols with the smallest and second smallest probability are siblings. Where in the proof of this, or in the setup of this lemma three did I use this claim? Yeah?

AUDIENCE: They have to be siblings in order for you to remove b and d.

ANKUR MOITRA: That's exactly right. Because I used it even in the setup the definition. So how did I define T prime? T prime was what I got from taking an optimal tree T and then taking the symbols with the smallest and second smallest probability and replacing them with alpha. And in the tree, the only way that that can make sense is if they're actually siblings of each other.

So I invoked this claim implicitly when I said that let T be an optimal tree without loss of generality, the two symbols b and d are siblings of each other. And then this operation makes sense. This contraction operation where I replaced b and d with the new symbol alpha. And that was what allowed me to proceed with this proof, because once I had that that operation made sense.

The key point was exactly as you said, that if this new tree T prime weren't optimal, then I could do better by inverting my operation, calculating the expected length, and that would contradict the optimality of T. OK? So now are we all on the same page? We're good? Happy? OK. All right.

When you're trying to learn things yourself, I think this is a good way to try and do it because you should stress test yourself. All right. But I claim that now we're basically done. So we're going to have Huffman's main result, because it'll just be a lot of pretty pictures. So let's do the pretty pictures. And don't fear, I still have more questions for the audience as we do this. So let's just keep track of applying this lemma again and again.

What I know is that when I replace the symbols with the smallest and second smallest probabilities with a new symbol, I know that my finding the optimal solution to that problem will help me solve my original problem. So let's just track it through and figure out what happens to our original problem. So our original problem that we cared about was just we have a, b, c, d, e, and f, and they had associated probabilities 0.4, 0.05, 0.18, 0.07, 0.20, 0.10.

And then what I'm going to do in my next problem is I'm going to have symbols a, c, b went away, e, f, and alpha. And their associated probabilities. Well, a stays the same. We haven't touched it. c stays the same, e stays the same, f stays the same. And alpha is now 0.12, because I added up the probabilities of b and c.

What's my next step in this procedure? Can anyone help me out? What should I do? I'm trying to reduce my original problem down to smaller problems in such a way that I can back out what the optimal solution is. What should I do now? I've got this new problem here on this alphabet A prime. Yeah?

AUDIENCE: You do a similar operation with f and alpha.

ANKUR MOITRA: f and alpha. That's right. So these become the two symbols with the smallest and second smallest probability. And I can replace them with the new symbol beta. So I get a, c, e, and beta. And now their associated probabilities are still 0.4 for a. I haven't touched it. 0.18, 0.2, and 0.22. And after this, my symbols with the smallest probabilities are c and e.

So I'm going to replace them with a new symbol gamma. So I have a gamma beta, and they have associated probability 0.4, 0.38, 0.22. And last but not least, I take the two ones with the smallest probabilities right here, and I replace some of the new symbol delta. So what's my optimal prefix code for this?

If I gave you a problem with just an alphabet of size two, and the two symbols had probability 0.4 and 0.6, my goal is to try and take my original problem, which I didn't know the optimal solution for, and hopefully reduce it to a problem that's so simple that I can solve it. So I'm done reducing it. I ran out of board space. What's the optimal solution to this last problem at the very end of the chain? What's my optimal prefix free code? Yeah?

AUDIENCE: Could you say just, like, gamma is like 1 and then like A01?

ANKUR MOITRA: That's right. Perfect. Because remember, what's going on? These prefix free codes, I can think about them as binary trees. There's only one binary tree I can do on these two things. So my optimal code is just this guy right here. I have the root and then my two symbols get encoded with either a 0 or 1.

Doesn't matter which one is which. And now for the question for the audience. OK, my goal wasn't to get down to a trivial problem that I can solve. If we give you on your exam three a question like this, you can't just replace it with your own favorite trivial question and solve it. I care about the solution to my original problem.

So how can I try and get the solution to my original problem? Maybe let's start simple. What's the optimal solution to this problem right here? Slightly harder. There is an alphabet of size three. We know what the optimal solution is on the associated subproblem. So how can I get my next optimal solution trying to go the other direction? Or what would the optimal solution be? Yeah?

AUDIENCE: Adding a branch off of delta.

ANKUR MOITRA: That's right. That's right. Because how do we get delta? It was merging gamma and beta. What we did was we invoked the claim that says that the optimal tree for this had better have the symbol with the smallest and second smallest probabilities be siblings. And now I can undo my operation where I merge them together into a new symbol delta, and what I'll get is I'll get this new binary tree.

A stays where it is. That's still encoded as 0. But now we get a deeper binary tree where I replace delta with beta and gamma. So beta gets mapped to 1, 0 and gamma gets mapped to 1, 1. OK? Just to make sure that we're really on the same page, what should I do next? What's my next optimal solution? What should I do? Yeah?

AUDIENCE: Split up gamma.

ANKUR MOITRA: Split up gamma. Perfect. So now I think you guys see how this works. So my new solution right here would be a is over here. I still have beta where it was before I split up gamma into c and e. And then I go backwards once more. I'm going to run out of space soon. So this will be the last one that I'll do. Actually, maybe I'll do one more. That's fine.

And my last one, which I'll need a bit of space to draw. OK. I guess I'll draw it over here. So my last one is this binary tree. So this is a valid prefix free encoding. What can I say about this tree T? It's optimal.

Why is it optimal? I claim that this is an optimal solution to the original running example I gave you at the beginning of class. Why is this tree optimal? Because in each step, all we did was we reasoned about any tree that's optimal without loss of generality had the property that b and d were siblings. And then for the new problem on my alphabet A prime, without loss of generality, it had the property that f and alpha were siblings.

And for my new subproblem, it had the property with loss of generality and so on. So in fact, we argued in each step that without loss of generality, our optimal tree could take this form. Now, there could be other ways of doing it because I could swap f and c together. But we argued that whatever we get out of this procedure had actually better be optimal. So this was Huffman coding.

In fact, we're not done. There's one more crucial question that we want to ask and answer about it. But right now, what we did was we gave you a simple greedy algorithm that finds the optimal prefix free code. It's not like Shannon where I say it's approximately optimal up to some additive terms. It is the optimal prefix free code. But there's one more crucial question, which is, how good is this code?

So when we talked about Shannon, I have it conveniently up here. It's not just that we argue that there's a coding function and we constructed it. It's that we actually argued about how good it is. We bounded its performance, and we saw that the binary entropy fell out from the end of that. So our last step is to bound the performance of the Huffman code and show that it's very good. In fact, it'll really beat what we proved in Shannon last time. So that's the last part of today's lecture.

So let me state one helper lemma, which you're going to give an alternative proof for in recitation and talk about what are figures to illustrate this proof. This proof is called the Kraft inequality. And once we have this Kraft inequality, we'll be able to get a very nice bound on the performance of Huffman codes. So this is a purely combinatorial lemma. And it's called the Kraft inequality.

So what I claim is that for any natural numbers l1, l2 all the way up to lk. So of course, natural number just means non-negative integers. So any non-negative integers. I claim that there is a binary tree with leaves at depths given by these values. So at depths l1, l2, all the way up to lq, if and only if the following condition holds. The sum of i equals 1 to k of 2 to the minus li is at most 1.

So this is a little bit of a mouthful. So what I'm asking you is some of packing constraint. Obviously you can't have three leaves at depth two because it's a binary tree. And what this Kraft inequality tells you is it tells you a necessary and sufficient condition. If you give me the sequence of depths, whether or not there is a tree with leaves at those depths. So let's do a simple example just to make sure that we're on the same page.

So let's start off with some depths of the leaves that satisfy the Kraft inequality. Let's say l1 equals 2, l2 equals 3, l3 equals 3. You can check when you add up the Kraft inequality that you're good. Actually, sorry this should be depth 1, 2, 2. Depth 1, 2, 2. And so what would be the associated tree here?

Well, here's one example of a binary tree that satisfies these conditions as promised by Kraft inequality. So I take this associated binary tree. It has one leaf of depth 1 because we're going down distance 1 from the root. It has one leaf of depth 2. And its third leaf is also of depth 2. And you can check that 1/2 plus 1/4 plus 1/4 is less than or equal to 1. So this is an example which both satisfies this Kraft condition and has an associated binary tree.

So this is just one illustration. But what if instead I take l3 and replace it by l3 is equal to-- let's see. 3. And l4 is equal to 3. This still satisfies the Kraft inequality. What would I do is I would just replace this part with a deeper binary tree with a3 here and a4 here. So the proof for the Kraft inequality essentially goes the way you'd expect it to is you just build up a binary tree.

The key point is that if I had all of the same lengths, like I had l1 equals 1 and l2 equals 1, then it's just a regular binary tree. And every time what's going on is that when I have larger depths, it really corresponds to replacing the leaf with some deeper binary tree, exactly as I did in this example. So you're going to prove this in recitation. So I'll just leave the proof as to be continued. You're going to work on both understanding the proof of this and drawing diagrams to illustrate the proof.

But for us, we're going to take this for granted, and we're going to prove our main theorem about Huffman codes, which is not just that there's an algorithm for building Huffman codes, but we're going to give a bound on their performance, and we're going to relate it back to what we proved on Thursday using Shannon. So here's our last statement for today that we're going to prove, and we're going to build on this Kraft inequality. What I claim is that the Huffman code has expected length l, which is between h and h plus 1.

So this is an incredibly strong statement because I'm saying that the binary entropy really tightly controls what the expected length of our optimal prefix free code is. I'm not going to prove this part. This part, in effect, we already proved through Shannon, but I'll argue about why we already proved that. Let's prove the other side, that the Huffman code is good.

OK. So here's our idea. Let's let la be the length of the encoding for symbol a, and we're going to very cleverly choose it to be the ceiling of log of Pa. So what's going on here is that the depths of my leaves had better be natural numbers. They're non-negative integers. And Pa is the associated probability of the symbol a. So I take log of Pa, I'm going to get a negative number. I multiply it by minus 1, I'm going to get a positive number. And I'm just going to round up that number.

And those are going to be my depths la. And so the first thing I have to argue about is that these are valid depths for the leaves, because that's where I want to put each of these symbols. So how do I argue that these are valid depths is I appeal to the Kraft inequality. So the Kraft inequality tells me I just have to check a particular inequality. So let's verify Kraft, that the conditions of Kraft inequality hold.

So I want that the sum over all symbols in my alphabet of 2 to the minus la, I want to show that this is at most 1. So let's write out what this thing is. Well, by definition, the way we've constructed la, this is still the sum over all symbols in our alphabet of 2 to the minus ceiling of minus log Pa. And I claim that this is upper bounded by the sum over all the symbols in our alphabet of 2 to the log Pa.

So what's going on here is that remember inside I have a positive number, because it's minus log Pa. Log Pa is negative. So when I'm rounding it up and then I'm multiplying by minus 1 I'm making this associated number smaller. So I can make the number only bigger by removing the ceiling function. When I remove the ceiling function, the two minuses cancel and I get log Pa, and now 2 to the log Pa is very simple. It's just Pa.

So I get sum over all of the symbols in my alphabet of Pa, which is 1. So this verifies the Kraft inequality, because all of these lengths la that I've conveniently guessed, they really do correspond to a binary tree with those associated depths of the leaves. And now the last part I have to do is I just have to analyze how good this particular construction is. And then because the Huffman code is optimal, it'll only ever be better than whatever I get for this particular code that follows by appealing to Kraft.

So now let's calculate the performance of the code. So we know, by lemma four, we know that there is a binary tree with these depths, the depths that I've guessed. And let's calculate its expected length. So I know that l, the expected length of my Huffman code, is only ever better than whatever expected lengths I'm going to get for this tree that I'm guessing from Kraft. And this right here is the sum over all symbols in my alphabet of Pa times la, the length of the associated symbol.

Well, I can replace this again. This is the sum over all symbols in my alphabet of Pa times the ceiling of minus log Pa. And now I claim that this is upper bounded just by adding one instead of taking the ceiling, because at most I bump it up by 1. So this is the sum over all symbols in my alphabet of Pa times minus log Pa plus 1. But now I'm in good shape, because this term should look familiar. This is the binary entropy. And this term right here, when I'm summing up all the Pa's, I get 1.

So in total the thing that I get is my upper bound is the binary entropy plus 1. So that's the proof. The Huffman code is very, very good because it's optimal and because there exists a good code by the Kraft inequality whose performance is H plus 1.

Now, in the last couple minutes, I want to take this way further and connect it back to Shannon. So we proved a bunch of things. We proved that we found an algorithm, Huffman's coding algorithm for finding the optimal prefix free code. We analyzed its performance in terms of its expected length. But now what I want to argue about is how we can do much better than Shannon.

So we talked about how Shannon-- I think I still have it here. It gets a bound when you apply it to length n sequences instead of a single length single symbol. It gets n times entropy plus some other term.

So if I naively use my Huffman code, it'll sound like I'll get n times entropy plus n because I'm paying that plus 1 every time. But there's a trick where we can replace that. So that's the last thing I want to show you. And this is one of the really cool punch lines. So let's imagine, and then the length n case will follow immediately from this. What if we want to encode outputs of length 2? OK.

So what if our source outputs two symbols instead? Well, we can still think about them as a single symbol, just on a larger alphabet, because originally we had six symbols, but now we'll have 36 in our alphabet. Each of these pairs of characters is a new symbol in my alphabet, aa, ab, ac. We still have a first order source which just has a larger value for k. What I can do is I can compute the entropy.

Sp what's the entropy of this source? Well, now it's minus the sum over all i, the sum over all j, because I'm taking two symbols, i and j, the probability that our first symbol is i, the probability that our second symbol is j, times log of Pi, Pj. But now something amazing happens, which is that I can just algebraically manipulate this statement. I'll get minus the sum over i, the sum over j of Pi Pj log Pi plus Pi Pj log Pj. This is just using properties of the log.

And now look at this term right here. I can sum up over j, and then I can replace the Pj with 1 because the Pj's sum to 1. So I'll get the sum over i of Pi log Pi. And the same way here I can sum up over i because i doesn't appear inside the log. So I can replace this Pi with one and get rid of the sum. So this entire thing is then equal to minus the sum over i of Pi log Pi minus the sum over j Pj log Pj. All this is is 2 times the entropy.

It's 2 times the entropy because each of these terms is computing the entropy. So this is one of the amazing facts about entropy is that when you look at the entropy of the output at length 2, it's just twice the entropy of the per symbol entropy that you started with H. And now something amazing happens, which is if I think about this theorem, and instead of encoding my output for my channel symbol by symbol, what if I encode it as a pair?

So instead, when I look at the output as being two symbols, my new entropy is 2H. So I'll have that my expected length is at most 2H, but still plus 1. When I do it for an output that's size n, it'll be n times H, but still plus 1. That plus 1 is just the fixed thing that we add independently of how large our alphabet is. So this result that we had from Shannon, where there was a bound and the suboptimality that's some little o of n that's like n over log n or root n, it turns out that we can replace this with plus 1.

So this is an amazingly sharp thing that follows immediately from our proof of Huffman codes. So we improved upon two things here. First of all, we have a more explicit coding function because it follows just from this greedy algorithm. And second of all, we drastically improve this additive loss in our original bound. So Huffman is really one of the crown jewels in coding theory, and it was proven, really, by a grad student, which is pretty amazing, who wanted to get out of taking a final exam.

So with that, I'll leave you as an invitation. If you also want to get out of your exam three, just prove something else that we're going to teach in lecture next semester and I will happily give you an A for the class. If not, come to office hours and we'll help you study. All right. So we'll continue next time with Shannon's noisy coding theorem. That'll probably be the hardest lecture in the coding theory unit, but come prepared. Be awake and ready to answer questions. See you then.