[SQUEAKING]

[RUSTLING]

[CLICKING]

PETER SHOR: OK, so I'm going to start with a timeline. So Shannon, 1948, discovered the noisy coding theorem. Hamming discovered Hamming codes in 1950. Reed and Solomon discovered Reed-Solomon codes in 1960. Concatenated codes were discovered in 1966. And Berlekamp-Massey decoding algorithm for Reed-Solomon codes was discovered in 1969.

So this is a lot of steps. And it turns out that the Berlekamp-Massey decoding algorithm for Reed-Solomon codes, along with concatenated codes, was what made using error-correcting codes practical.

And in Voyager, deep space was 1977. And one of the Voyager aircrafts is still transmitting signals to Earth. And they're doing that using Reed-Solomon codes, which is what I'm going to tell you about later in the lecture.

And after all this stuff, Reed-Solomon codes were still used until-- oh, I don't know-- around mid-1990s. Actually, they were still used until probably around 2000, but in the mid-1990s, these three French engineers-- so Reed-Solomon codes do not actually meet Shannon's bound from the noisy coding theorem.

Reed-Solomon codes come within a constant factor of it, but between 1948, for the next 40-something years, coding theorists have struggled to find a technique for actually meeting Shannon's bound. And in 1990-something, this paper appeared by three French engineers. They weren't even coding theorists. They were three people at a French telecom company.

And they claimed to meet Shannon's bound with a completely different technique. And none of the coding theorist researchers believed it until they programmed up, and it worked. And even that-- it took many years for them to explain why it worked. And in fact, they still don't understand why it worked, why turbo codes work. They do understand why a similar type of codes, called LDPC codes, work really well.

But so now, after this discovery, Reed-Solomon codes gradually faded out of use in new applications. And turbo codes and LDPC codes and polar codes, which were discovered in the 2000s, took over. But I find it amazing that you needed something like three discoveries to actually make Shannon's theorem practical.

So what is a Reed-Solomon code? OK, well, it's based on an ancient theorem from mathematics, a polynomial of degree d over a finite field that has, at most, d roots.

OK, so a Reed-Solomon code-- you have a message, m0 through mk. Well, mk minus 1 because we're starting with 0. Associate the polynomial p of x equals m0 plus m1 x plus m2 x squared plus mk minus 1 x to the k minus 1.

So this is the message. And once you have the message, you construct this polynomial, although we're not going to actually transmit the polynomial. What we're going to transmit is the code word, which is p of 0, p of 1, p of 2, all the way up through p of n minus 1.

Now, you note, to do this, we need n is less than or equal to-- OK, so this is over a finite field. And for this lecture, we'll assume the finite field is the integers mod p. In real life, Reed-Solomon codes-- they actually use the polynomial z over the finite field with 2 to the k elements, but that's because they're using binary. And 2 to the k is very convenient for binary.

But the theory is exactly the same if we take z sub p. And so for this, well, we need n minus 1 different elements in the field-- or n different elements in the field. And for that, we need n is, at most, p.

And in fact, as an aside, you don't have to use 0, 1, 2 through n minus 1 for these points. You could just use any n distinct points, but the lecture notes start out by saying choose n points. And then they say later, we'll use the point 0 through n minus 1. So I'm just going to start out with 0 through n minus 1.

OK. And note-- Reed-Solomon's codes are linear. And the generator matrix G is equal to 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 2, 4, 8, et cetera, 2 to the k minus 1, 1, 3, 9, 27, et cetera, and 3 to the k minus 1, all the way up to-- well, how many values of the polynomial are there?

There are-- how many numbers do you plug into the polynomial? There are n minus 1, so this last column is 1 n minus 1, n minus 1 squared through n minus 1 to the k minus 1. And I should note that these numbers are all mod z sub p. So if p was 13, this entry would not be 27, but 1. Oh. Well, I mean 27 is 1 mod 13, but yeah.

So this is the generator matrix G. And if you put in m0, m1 through mk, the i-th entry and the code word is just 1 plus m1 times 3 to the 1st plus m2 times 3 to the 2nd plus m3 times 3 cubed, et cetera, which is the value of p of x at 3. So they're linear codes, which means that we want to find how many errors they can correct.

So remember-- OK. So need to find minimum weight nonzero code word. If weight-- let's give this a name, c. If weight of c is equal to d, we can correct d minus 1 over 2 errors.

Theorem-- and that's because the minimum distance between any two code words is d minus 1 over 2. Minimum weight non-zero code word has Hamming weight n minus k plus 1.

Proof-- the code word-- well, we know what the code word is. It's just p of 0, p of 1 through p of n minus 1 where p equals m0 plus m1 x plus plus mk minus 1 x to the k minus 1 is a polynomial of degree k. And remember that this polynomial-- actually, the coefficients are the message. So it could be any polynomial of degree k minus 1.

And now, we use this theorem that I reminded you a little while ago-- a polynomial of degree d has, at most, d roots. So p of x has, at most, k minus 1 0s. So at most, k minus 1 of these entries are 0, which means at least n minus k plus 1 values of p sub i are nonzero.

Because you have n values in the code word, k minus 1 of them or fewer are 0, which means this many have to be nonzero. So the minimum Hamming weight is n minus k plus 1.

Suppose we want to correct e errors. Well, we need minimum distance to be greater than or equal to 2e plus 1. And that means we need n minus k plus 1 greater than or equal to 2e plus 1 or n minus k is greater than or equal to 2e.

OK. So that are our Reed-Solomon codes. Now, remember that to have a usable code, we not only need to be able to encode, but we also need to be able to correct errors and decode. So the question is how do you decode?

So this is what I will spend probably the next half hour telling you. OK. And I am not going to give you the Berlekamp-Massey decoding algorithm because that is conceptually very-- or I shouldn't say "very"-- that's conceptually fairly difficult. And I'm going to give you a much easier decoding algorithm to explain, which is not as efficient as the Berlekamp-Massey decoding algorithm.

And there's another decoding algorithm, which is probably equally efficient, which is based on the Euclidean algorithm for finding greatest common divisors, but I'm not going to give you that one, either, because again, that's fairly difficult, conceptually.

How do we decode? So this decoding algorithm is based on a theorem. So theorem-- suppose we have a code word, c equals p of 0, p of 1 through p of n minus 1.

We get the transmission, or rather, we receive the code word with some errors in it. c tilde equals r0, r1, r2 through rn minus 1. And these guys are not going to have a polynomial where ri equals p of i. For some of these, we have ri equals p of i, but for some others, we won't.

Assume there are i less than e errors. Then there are positions i1, i2 through il, where p of i sub l is not equal to r-- p sub i sub k is not equal to r sub i sub k.

So we have, at most, i errors. And we'll assume they're in positions i1 through il. And l is less than or equal to e because we're assuming there are fewer errors than the code is able to decode. So this is the hypothesis of the theorem. And I need to put the theorem itself on a different board.

Then-- continued-- there are nonzero polynomials f and q and q of x of degree less than or equal to k plus e minus 1 such that q of i is equal to r sub i times f of i.

OK. So this looks like a totally crazy theorem. What does it have to do with decoding? Well, you will see what it has to do with decoding when I give you the decoding algorithm, but first, let's prove the theorem.

Let f of x equal x minus i1 x minus i2 through x minus i sub l is equal to the product of x minus i sub k, k equals 1 through l. So this is-- you just take all the positions which are in error.

Of course, you don't know which positions are in error when you receive this word, which is the code word with the most e errors, but we'll let f of x be that. And we will let q of x equals p of x times f of x.

OK. Oh, I think we-- ah. I should have put this down. q of i equals r sub i, f sub i for i equals 0, 1 through n minus 1. So this is only true at the positions which give you values in the code.

And of course, this r sub i is not defined for any position other than 0, 1 through n minus 1. So this condition is probably a little bit unnecessary, but still, I'm going to write it. So if i is in i0 i1 through i sub l, then f of i equals 0 because f of x is just the product of x minus i sub k, where i is in this set.

And q of x equals p of x, f of x equals 0, or q of i equals p of I times f of i equals 0. So we wanted to show that q of i equals r sub i times f of i. q of i equals r sub i times f of i equals 0. Because remember-- q of x was p of x times f of x.

OK? So the other alternative is if i is not in the set of positions where things are in error, then, well, if there are no errors, then r sub i equals p of i. And q of i is equal to p of i times r sub i.

q of i equals p of i times f of i, which is equal to-- since ri equals p of i, it's r sub i times f of i. And that's what we wanted-- q of i equals r sub i times f of i. So whether or not i is in the position containing the errors, q of i equals p of i times f of i.

OK, so that's the theorem. We have these two polynomials, f of x and q of x, which have this relation between them. How do we decode? Linear algebra. So the lecture notes give you an explicit example of how to decode, but I am not going to go through it in class because it's really fairly tedious to go through. And I will just explain the theory.

So we have q of i equals r sub i times f of i. So this is a whole bunch of equations. q of 0 equals r 0 times f of 0. q of 1 equals r1 times f of 1. q of 2 equals r2 times f of 2 all the way down to q of n minus 1 equals r sub n minus 1 times f of n minus 1. So you can think of these equations as linear equations in the coefficients of the polynomials.

And OK. So what do they look like in terms of the linear equations? Well, q of 0-- well, I should probably say what the coefficients are. q of x equals q0 plus q1x plus plus Q sub k plus e minus 1 x to the k plus e minus 1. And f of x equals f0 plus f1 x plus plus. So this has degree e. So it's f sub e x to the e.

OK. So what do these equations look like? Well, the first one, q of 0, is just q0. So it's q0 equals r0 f0. The next one is q of 1. So it's just q0 plus q1 plus q2 plus dot, dot, dot plus q-- and what was the degree? k plus e minus 1 is equal to r1 f0 plus f1 plus dot, dot, dot plus f sub e.

And the next one is f of 2. So it's q0 plus 4 plus 2 q1 plus 4 q2 plus dot, dot, dot plus 2 to the k plus e minus 1 q sub k plus e minus 1. Of course, this is mod p, so the numbers don't actually grow huge. And this equals r sub 2 f of 0 plus 2 f1 plus dot, dot, dot plus 2 to the e f sub e.

OK. I'm going to claim this is a linear equation in coefficients of q and r. Well, what are the coefficients of q and r? Well, there's q0, q1, q2, 2k plus e minus 1, and f0, f1 through f sub e. And it's clearly a linear equation in the coefficients.

So linear algebra-- OK. We have a bunch of linear equations and a bunch of variables. How many solutions do they have?

Well, let's see. How many variables? Well, there are e plus 1 variables in f of x because there are e plus 1 coefficients. And there are k plus e minus 1 plus 1 variables in q of x because there are k plus e coefficients. So how many variables? There are k plus e plus E plus 1.

How many equations? Well, there are n equations because we have one for q0, q1, all the way up to q of n minus 1. n equations, but somewhere, we have n minus k equals 2e.

And up there we said we needed n minus k greater than or equal to 2e to correct e errors, but let's assume that n minus k equals 2e because that's the minimum we need to correct e errors. So n minus k equals 2e.

So we have k plus 2e plus 1 variables and k plus 2e equations. And these equations are all linearly independent. And how do you see that? OK, I'm not going to really explain this in full detail.

But remember G is equal to 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 2, 4, 8, 2 to the n minus 1, 1, 3, 9, 27 through 3 to the n minus 1, et cetera. So this is a Vandermonde matrix. And Vandermonde matrices have full rank.

And this is a theorem which we could easily have proved at some point in our course, but we didn't. And I'm not going to prove it today, but this is the generating matrix for our code. And it has full rank.

And by using that fact and a similar fact on the right-hand side for the f's, you can show that these linear equations have full rank, which means that we have n minus k equals-- we have k plus 2e linearly independent equations and k plus 2e plus 1 variables. And that means there exists a nonzero solution.

And in fact, you can use the fact that s sub e equals 1 because f was-- yeah, f was this. And so f sub e equals 1. Use f sub e equals 1 to get a unique solution, assuming l equals e.

OK, I don't know what happens when l is-- the number of actual errors is smaller than e, the maximum number of errors you can correct. In other error correction algorithms for Reed-Solomon codes, you basically have to go through all possibilities of how big l is and check each one out, except there are usually much simpler ways to do that than going through all of these possibilities.

OK. So we have one solution to all of these equations. And that means we can figure out what q of x and f of x are because we know that q of x and-- sorry, q of x and p of x are because we know that q of x and p of x satisfy these-- I'm sorry. We can figure out what f of x and q of x are because we had linear equations for the coefficients of q of x and f of x. So we can deduce q of x and f of x.

So linear algebra gives q of x and f of x. And we had p of x-- this was the message-- times f of x equals q of x. So p of x equals q of x over f of x. And we have gotten the message back.

OK. So that's how to decode Reed-Solomon codes. The last thing I want to do is tell you about concatenated codes and why these make Reed-Solomon codes truly practical.

So we have n equals k plus 2e for Reed-Solomon codes. And let's assume the Reed-Solomon codes over the finite field with 2 to the 8th elements.

So I don't know whether Professor Moitra mentioned it before, but there's a finite field for every power of a prime. And one of the very common finite fields used for Reed-Solomon codes in practice is 2 to the 8th.

So let's say n equals 256. That's 2 to the 8th. k equals 150. And that implies that e equals n minus k over 2 because we had n minus k equals 2e. And 256 negative 150 over 2 is 53.

Can encode-- let's see, k was 150-- 150 bytes, if we assume a byte is 8 bits, into 256 bytes, correct 53 errors, which means 53 is the number of bytes that can have errors in them.

And if you think about this, in general, cannot correct an error rate greater than n over 2 because the number of errors you can correct is n minus k over 2. And even if you use the smallest possible code which encodes a single byte-- that's 8 bits-- you cannot correct more than n over 2 errors. OK.

Let's assume the bit error rate is 10%. So you're transmitting 0s and 1s. And you're grouping them into blocks of 8 bits, but each of your bit is likely to have a 10% error.

Probability of byte-- that's a group of 8 bits-- is in error-- is 1 minus 1 minus 1/10 to the 8th, OK, which equals 57%. This is the probability that each of your bits is not an error.

So you raise it to the 8th. That's the probability that all of your bits do not have an error. And the probability of an error is 1 minus this. That's 57%. So if you're trying to use a Reed-Solomon code with blocks of size 8, you cannot find parameters that will let it correct the error. And I want to say what do you do? You use concatenated codes.

You can look in code tables. And these are available on the web. Find that there is a 19, 8, 7 code. This encodes 8 bits into 19 bits. And the distance is equal to 7, which means it corrects 3 errors.

OK. So we should call this code something. So why don't we call it C inner? OK. And use Reed-Solomon code. And somewhere, I had this example of 150 bytes into 256 bytes and correct 53 errors. So I guess that is the 250. That's n and 150. And corrects 53 errors because the distance is 107. So how do you concatenate two codes?

So the first thing you do is apply the outer code to your message. Wait. Yeah. What does that look like? Well, your message, you break into blocks of 8 bits.

And 8 bits can be represented as numbers between 0 and 256. So your message might look like 317, 251, 109, 113, 75, et cetera. And it has 150 of these, 150 of these things.

So now, you encode it using your Reed-Solomon code. And you get 250. code word length, 250. Looks pretty much the same-- 113, 11, 17, 257, et cetera. And there are 250 of these numbers.

Take each of these numbers and encode using C inner. And now, to decode, you first decode C inner. And you get this code word. And now, you decode it using the decoding algorithm for Reed-Solomon codes, and you get your message.

And you have taken your message. Take, I guess, 8 times 150 is equal to, I think, 1,200 bits into what was our 19, 8, 7 bit codes into 19 times 256. OK, I did not do this multiplication before I class. So 250 times 20 is around 5,000 bits.

So you've taken 1,200 bits to 5,000 bits. And you can ask what is the probability you decode successfully with a 10% bit error rate? OK, so I used Mathematica for this.

But at a 10% error rate, if you have 8 bits, 10% error rate, probability of greater than or equal to 3 errors is 11%. And now, the chance-- and what do we have? We have an 11% error rate in the bytes.

We have 256 bytes. And if there are fewer than 53 of them in error, you can decode with high probability. And from probability theory, you know how to figure out the probability that you cannot decode, but the probability you cannot decode is approximately 10 to the minus 60th, which is really incredibly small.

So that means that if you take concatenated code and use this code for the outer code and this code for the inner code, and you have a 10% error rate on your bits, your information almost always gets through completely intact.

And actually, so this 10% error rate-- we've taken 1,200 bits to 5,000 bits. We've multiplied the number of bits by 4. Now, if you look at Shannon's theorem, if you have 1,200 bits and a 10% error rate, you should be able to get by with multiplying the number of bits by 2.

So the ideal code would give us 2,400 bits. And we've used twice that many-- 5,000 bits. So that's not that horrible. If we use turbo codes, or LDPC codes, or Polar codes, which were invented much later than Reed-Solomon codes, we would reduce that to something very close to Shannon's bound.

But this was good enough for-- let's see. Yeah, so this was good enough for 30 years. All the machinery for using Reed-Solomon codes had been discovered by 1970, and they remained in widespread use-- well, they remain in widespread use today. Anytime you play a CD-- of course, very few people are playing CDs anymore, but anytime you play a CD, you're using Reed-Solomon codes.

OK. There's one last thing I want to say. Well, I've told you how to decode the outer code. How can you decode the inner code?

It's a 19, 8, 7 code. There are 2 to the 19th, which is approximately 500,000, code words. And can use table lookup for these. You have, in your computer, a complete table of all-- there are 500,000 possible.

Yeah, there are only 256 code words, but those get mapped into 19-bit words. And there are 500,000 19-bit words. And you can use table lookup to decode. You can just list all possible 500,000 words you might receive. And to decode them, you look up in the table to find the nearest code word. And that's the decoding.

And nowadays, this is incredibly easy. In 1977, when Voyager did it, I expect that the Voyager spacecraft did not have enough memory for 500,000 table lookup words. So they must have done something cleverer.

Well, one thing you could do is, instead of using Reed-Solomon codes with 8 bits in each block, use ones with smaller number of bits in each block. And then you will get a smaller code word.

Or another thing you could do is, for the inner code, you could use a Hamming code word. And Hamming codes are very easy to decode, although they really do not get quite as good performance as you know tree error correcting codes. And there are other things you can do, too.

So I don't actually know what they did for Voyager, but it's probably somewhere on the line. And I could look it up. OK, so that's all I have on error correcting codes. And in fact, this is my last lecture of the course because Professor Moitra will be giving the next two non-exam lectures. And we have one exam for you. OK.