PROFESSOR: [? Diagonal ?] arguments are elegant, and infinite sets-- some people think-- are romantic.

But you could legitimately ask what is all this weird infinite stuff doing in a course that's math for computer science? And the reason is that diagonal arguments turn out to play a fundamental role in the theory of computing. And what we're going to talk about now is the application of diagonal arguments to show that there are noncomputable sets and examine a particular one.

So let's look at the class of infinite binary strings.

Now, we've seen that there are an uncountable number of infinite binary strings, and that's because there was a simple bijection between the infinite binary strings and the subsets of the natural numbers-- that is the power set of n.

Let's look at the infinite binary strings that we might think of and call computable strings. And what I mean by a computable string is that there's simply a procedure that will tell me what its digits are. So what I mean is that the procedure applied to argument n will return the n-th digit of the string s. That's the definition of what I mean by saying s is computable. I can compute its digits, whichever digits are needed.

Now, we saw that there were only a countable number of finite binary sequences, and I mention that now because I want to think about sequences over the slightly larger alphabet instead of 0 1, the 256 ASCII characters.

And by the same argument, the set of finite ASCII strings is also countable. You just list them in order of length-- same argument that we used for the binary strings.

Now, the point of looking at the ASCII strings-- the 256 keyboard characters-- is that every procedure that we enter into a computer, we type in an ASCII string. Every procedure can be represented by an ASCII string.

And since they're only countably many finite ASCII strings, it follows that there are only countably many computable procedures. Now, since in order to be a computable infinite string, there has to be a procedure which computes is digits, we can immediately conclude that there are only countably many infinite binary sequences that are computable-- only countably many computable infinite binary sequences.

But I already said there are an uncountable number of those infinite binary sequences. So it has to be that there are noncomputable sequences, noncomputable infinite binary strings, that exist.

So we can conclude that as a matter of fact, since the set of infinite binary strings is uncountable and the computable ones are a countable subset, there have to be an uncountable number of noncomputable infinite binary sequences. Most infinite binary sequences are actually noncomputable. OK.

That's kind of abstract thing to know. They're out there, and you can't get a hold of them computationally. But the reasonable question to ask is what do they look like?

And what we're going to see is that if you consider a sensible concrete computational problem of giving a procedure, figuring out whether it will run and turn a value successfully on some argument or not is called testing the halting property of procedures.

I want to know-- given a procedure and argument that I could apply it to, does it return a value or does something else bad happen and it runs forever and returns an error? We don't get a satisfactory value out. And if it does [? satisfactorily ?] return something, we say it halts.

And what I'm going to argue is that the halting problem is not decidable. That is, there's no procedure which given input that describes a procedure, the fixed procedure can figure out what its input is doing.

Let's look at that in more detail.

So let's think about string procedures because we're thinking about procedures being represented by ASCII strings. So let's think about procedures that take a string argument.

So an example of a procedure P, it might be that when you apply P to the string no, it returns 2. When you apply it to the string albert, it returns meyer. When you apply it to this string of weird symbols, that causes an error. And you apply it to the sequence of characters what now, and it actually runs forever.

These are just illustrations of the kind of behavior that some weird string procedure might exhibit.

So what I want to think about is-- suppose I have an ASCII string s, a finite ASCII string. That's

the one that defines this procedure P. When I'm trying to run P on the computer, I'll have to type in s in order to give the computer the definition of P to tell it what to do.

And I'm going to say that s HALTS-- the string has this property called halting or HALTS-- if and only if this procedure P that s describes returns successfully when it's applied to s.

This is where we're really doing a diagonal argument. We're taking the sth object-- the procedure that's described by s and applying it to s. And that's kind of going down the diagonal of s applied to s, where the n-th element of the n-th row in a pictorial diagonal argument. That's the idea that we're going here.

But let's to go back to the definition. A string is said to HALT if when you interpret it as the description of a procedure that takes a string argument and you apply that string procedure to that very same thing, s, you successfully return. That's the halting problem.

And what I want to argue is that it's impossible that there could be a procedure Q that decided the property HALTS of strings. That is to say, Q applied to a string returns yes if s does return successfully-- if s HALTS. And it returns no otherwise.

Q applied to s will say no if s runs forever or if s has a type error or s does anything other than successfully return a value.

Let's suppose, for the sake of contradiction, that there was this HALTS decider. I claim there can't be such a Q. For the sake of contradiction, let's assume there was one.

Then this is the trick that I'm going to do. I'm going to modify Q to act as though it was complementing the diagonal. More precisely, this is what I'm going to do with Q. I'm going to modify Q to be another procedure Q prime, which just behaves a little bit differently. Namely, Q prime of s returns yes when Q of s returns no, and Q prime of s returns nothing-- that is, it doesn't HALT-- if you Q of s returns yes.

So Q prime is like complementing the bits on the diagonal, but here's the precise definition. Q of s says no. Q of prime of s says yes. Q of s says yes. s HALTS successfully. Q prime then does not HALT successfully. It returns nothing at all. Let's go crank through the consequences of these definitions.

So s HALTS means Q prime of s returns nothing. That was the way we define Q prime of s.

Now, let's let t be the text for Q prime. If Q was a procedure, then surely we can tweak this procedure Q to get the procedure Q prime. So Q prime will have a text that describes. It'll be the ASCII string that defines Q prime. Let's let t be that ASCII string.

What do we have?

Then by definition of HALTS, t HALTS if and only if the procedure that t scribes-- namely Q prime applied to t-- returns a value successfully. OK?

Now by definition of Q prime however, Q prime was the thing that on t, it returned a value successfully if and only if it was not the case that t HALTS.

So if you put those two things together-- that is, we're looking at t HALTS if and only if Q prime of t returns, and Q prime of t returns a value successfully if only if not t HALTS-- then put the two together, and we have a contradiction. t HALTS if and only if t doesn't HALT.

And that contradiction says that our original hypothesis that we had a Q that would decide the halting problem can't be right. It's impossible to write a procedure that determines of strings whether they describe a procedure that HALTS when applied to itself. OK.

That at least gives us some concrete problem that we can say is not something that a computer can do. Even though it's a very well defined clear question, it's just not possible to get a computing procedure that will on an arbitrary string, figure out the right answer. Any program that applied to strings is trying to do this job, either it will give the wrong answer. Or if it never gives a wrong answer, it means it doesn't give an answer at all on some strings.

All right. Well, you could say that I don't really care very much about whether a program HALTS or not. So let's look at how do you apply the same reasoning-- or more precisely, as a corollary of the fact that the halting problem is not computable, let's talk about something that sounds closer to a practical interest, mainly type-checking.

So I want to think about the type-checking problem. And what I want to say is that in fact, there's no strict procedures that type-checks procedures perfectly.

So what I mean is that I want to be able to write a program that will look at a program text, an ASCII string that describes a procedure, and figure out whether that ASCII string, if you ran it, would cause a run-time type error.

That's what type-checkers are supposed to do. They're supposed to check your program, figure out whether the program will cause a run-time type error. If so, it reports it. If not, it says, this program is safe. Other things may go wrong, but it's not going to commit a run-time type error.

So let's suppose that I had such a type checking procedure C. So what that means is that for program text s, C of s returns yes if running s would cause a run-time type error. And C of s returns no-- the output string no-- otherwise if s would not cause a run-time type error. In other words, s is safe. All right.

Now, what I claim is that if you give me C-- if I have a procedure that's this perfect type-checker-- I can use C to build a tester for HALTS, which we said is impossible. So how would I use C to get a HALTS tester, H.

Here's how.

I'm going to tell you how to compute H of s.

I'm describing the procedure that this tester H carries out on argument s. And what it does is given argument s, it's going to construct a new program that's a small modification of s.

It's going to construct this new program s prime that acts like an interpreter for s. So s is a computer program or a procedure. I want to know whether if you just run it, it'll HALT or not. I'm going to tweak it a little bit so that s prime acts like s but in a slightly modified way. And here's how s prime works.

S prime is going to be an interpreter that's simulating step-by-step how s behaves. But at the moment that it discovers that s is about to commit a run-time type error-- that the next instruction that s prime would execute in simulating s was going to be a run-time type error-- s prime would just skip it. And who knows what the consequences of skipping it will be, but it'll skip it and just keep going. OK.

If s prime in simulating program s discovers that in fact s returns successfully-- those that is s HALTS-- then s prime will purposely make a type error. So let's think about what that means.

Well, actually let me just wrap up what the definition of H is. So the way H works is given input s, it constructs the program s prime and applies the type-checker C to s prime and returns the

same value that c does.

So what we can figure out by these definitions is the s HALTS-- the string s is a cloud halting string-- if and only if the string s prime makes a run-time type error. Because remember, the interpreter, which is what s prime was behaving like, was simulating what s did. And if s was going to HALT successfully, s prime makes a run-time type error.

That means that C is going to say yes to s prime-- yes, it has a run-time type error. And by definition of H, that means that H of s says yes because H of s constructed s prime to C. OK.

On the other hand, if s does not HALT, that means that something else goes wrong with s. It's not going to successfully return.

Then s prime-- when it's simulating s-- is never going to make a run-time type error because that's the way s prime goes. Whenever it detects that there would be about to be a run-time type error, it skips it.

So s prime is likely to keep running forever because it's simulating this program s that doesn't HALT, but it won't make a type error. And that means that C of s prime is going to say no-- no type error. And H of s is going to say no.

And that means that when s does not HALT, H of s properly says no. In other words, I've just walked through the argument that this procedure H that I've described actually is a decider for HALTS. And that's a contradiction.

The H that I gave you would solve the halting problem if there was a perfect type-checker, and there can't be a halting problem decider. So there can't be a perfect type checker.

C must make a mistake. It can't accurately predict run-time errors.

And that is an example of how you reason from this kind of contrived halting problem that's sort of self-referential whether the string procedure applied to its own definition HALTS or not. And we can apply it to all sorts of questions and properties of procedures that we really care about.

In fact, the same reasoning really shows that it's not just type-checking. That's a kind of arbitrary example, but there's more or less no perfect checker for any kind of property that procedure outcomes might exhibit.

Which is why theoretical computer scientists interested in the theory of computation have great respect and interest in diagonal arguments because they crystallize a whole set of absolutely logical, intrinsic limitations on the power of computation.