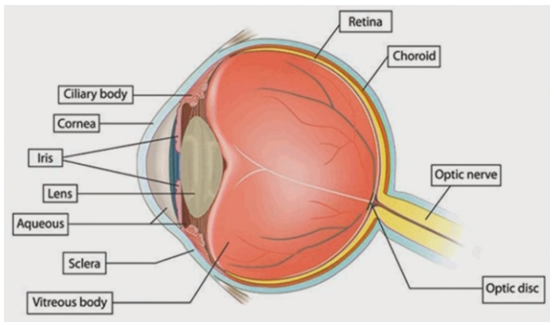
1.4 What Makes a Problem Hard, Part 1

[00:00] One of the great advances that has been made in understanding the mind because of the comparisons between computational methods and the problems that human beings have to face - in other words, one of the great achievements that has occurred over the last half century or so because of the way in which computers have informed our thinking about the mind is in this general notion of what makes a problem hard or difficult. Our intuitions about what is hard or easy for us proved to be pretty interestingly myopic. That is, we're not always very good about understanding what things are actually hard or easy for us to do until we try and get a computer to do them.

[01:07] It's interesting to note in the article that Alan Turing wrote on the Turing test, he finished up that article, he was talking about getting a computer to mimic or imitate human intelligence. And he finished up that article by saying, what would be a good problem for us to tackle as computer scientists in trying to look at this general notion of, he wouldn't use the term artificial intelligence but essentially that's what he was talking about, what would be a good problem for a computer to tackle to illuminate human intelligence. And the problem he chose was chess. After all, chess is the kind of thing that intelligent people play: it's a hard game. So if you can get a computer to play chess well, that's a step forward in intelligence.

[02:04] It turns out that there are other things that are extraordinarily hard, and in some sense harder than chess. But they don't feel that hard to us. So let's take an example: how hard is it to see? When you open your eyes, you seem to automatically know where all the objects are in the world. You look around and you see an object there and an object over there and you know where you are. It seems a kind of automatic thing. It doesn't seem like it ought to be that hard to do, at least as far as our intuition is concerned. It turns out that in some sense seeing is an impossible problem. We do it and yet in a mathematical sense the most formal statement of the problem is impossible. Let me give you the explanation for that.

[03:04] Here is a diagram of the structure of the eye, and let us just assume that what we're dealing with is monocular vision: seeing with one eye. Of course we have the added advantage of generally seeing with two eyes, but people who only have sight in one eye can also see very well. They can see pretty well and make out the objects in the world. So it's possible to solve the vision problem with only one eye. Now the eye is of course a highly complex organic structure, but we can abstract all of the complication out of it and say that it's basically like a pinhole camera, where light comes in through the pupil and it gets projected onto what is essentially a screen like surface at the back of the eye: the retina. The retina is lined with sensory cells that enable the eye to then interpret the patterns of light falling on the retina and then do a great deal of processing from that point on and understands where objects are in the world. But for our purposes right now the structure of the eye is a pinhole camera



where light comes in through the pupil and hits the retina. From that information alone, that is from the information of patterns of light hitting the retina, we are going to determine where objects are in the world.

[04:45] Now in the way that I've just phrased it, this is a mathematical impossibility. The reason being that you cannot recover three-dimensions from two-dimensions. To put it another way, an infinite number of patterns of three-dimensional objects or an infinite number of three-dimensional structures could give rise to the same pattern of light on the retina. So there is no way that we could in general know from the pattern of light on the retina what objects there are out in the world. That's a mathematical impossibility and yet we manage the task. So this is a very interesting question.

[05:36] It has in fact been a huge and interesting challenge to get computers to see with anything like the accuracy and reliability with which people see. This is the field of computer vision. It's a marvelous and interesting field. But essentially this is computer scientists trying to solve the problem of how to get from a camera view to an understanding of objects in the world.

[06:10] We're able to do this relatively early in life. Vision seems like an easy problem to us. In fact, it's a highly difficult problem and in order to solve it in the way that we managed to do, we can't solve the impossible mathematical problem, but we can make lots and lots of approximations and smart guesses and so forth. And we can integrate other techniques to try and figure out where objects are.

[06:40] Here's another very hard problem; one that we all seem to learn how to do by the time we're five, six, seven years old. How do we learn language? Again, in some sense, this seems intuitively like a highly easy problem. We were born. We don't know when we're born, whether we're born in Athens circa 400 BC or modern day Helsinki or Beijing 100 years from now. We don't know that at the time that we're born, and we don't know what language community we're being inducted into. All we have is, I mean we have a great deal of internal structure and the information that we have is listening to people speaking around us along with other things that they might be doing. How hard is it to learn one's native language? Well, it seems pretty easy to us in the sense that we all pretty much succeeded doing it. You can say it takes us four, five, six years of ongoing work, nonetheless, we all succeed. Nobody has yet made a computer program that can essentially sit on the desk and have people talk to it for five or six years and from that learn a natural language. That seems to be a very difficult problem.

[08:13] So going back to Turing's instincts about this, he felt that chess would be a difficult problem because people find it effortful to do. The things that people don't find that effortful to do or we don't remember the effort we put into it like learning how to see or learning how to speak, those proved to be much harder problems than chess. It turns out that over time, computer scientists have learned how to program computers to play an exceptionally good game of chess, but the things that we seem to know how to do by the time we're five or six are the things that prove to be most difficult for computers to do.

[08:58] Let's take an example of a reason that it seems to be hard to get a computer to pass the Turing test. So you may remember in our conversation about the Turing test that the judge who is sitting outside these two rooms and asking questions of the person and the computer, the judge is essentially holding conversations with each of the entities inside these doors. And if the Turing test is successful, if the computer is a really good mimic of human intelligence, then it should be possible to have human-like conversations with the computer that's inside there. Why is that difficult? Why is that a difficult task for computer scientists to solve? There are a number of different theories and hypotheses and explanations.

[10:04] One thing that is exceptionally interesting is not just problems like learning how to speak or learning how to see, but all of the common sense knowledge that we seem to have about the world in the course of our conversations. So here's an example on this slide. These are four consecutive sentences taken from apparently a real children's book. It doesn't sound like the most interesting children's book to me, but these are four consecutive sentences taken from a children's book. Now there is some cultural knowledge involved in understanding these sentences, but I think it's fair to say that within the particular culture for which this kind of story is written, namely an American or Western culture, that any child of the age of six, maybe five, could listen to this part of the story and answer questions cogently about what's going on.

[11:13] Let's look at these sentences one by one and think about, especially if you happen to be a computer programmer, think about what it would take to program a machine to understand these sentences. What kind of knowledge it would take. So the first sentence is “Jane was invited to Jack's birthday party.” Okay, conceivably we could have a computer program that can understand enough of grammatical English to know what that means - "Jane was invited to Jack's birthday party."

[11:49] Second sentence: "She wondered if he would like a kite." Now, if you're a computer program trying to make sense of this, there's something of a leap between those two sentences. Jane was invited to Jack's birthday party - why is she even wondering about whether he would like a kite or not? In order to understand the connection between those two sentences you have to know something about the cultural structure of birthday parties and also things like what's appropriate as birthday parties involve gifts and what's appropriate as a gift. You notice that you would respond very differently if the sentences were, "Jane was invited to Jack's birthday party. She wondered if he would like a yacht." That would imply an entirely different kind of story than the one we're talking about here. So there's a fair amount of information required just going from sentence one to sentence two.

[13:08] Now let's go from sentence two to sentence three. “She wondered if he would like a kite. She went to her room and shook her piggy bank.” Now again, think about the huge gap. She wondered if he would like a kite so for that reason or in some connection with that, she goes to her room and shakes her piggy bank. You understand the connection between sentence two and sentence three. Think of all the common sense knowledge that goes into that. To a computer that could look like a non-sequitur. It'd be like, she wondered if he would like a kite - she went to her room and shook her leg. It seems if you don't know a great deal about children and gifts and piggy banks and so forth, then you would really be puzzled at why sentence two leads to sentence three. And then what's the significance of sentence four? “It made no sound.” What does that mean? Again, because we know something about everyday physics, the fact that a shaken piggy bank makes no sound tells us a great deal.

[14:29] Getting all of this kind of information into a computer program proves to be a really interesting problem. And in general, I don't want to make this a hard and fast rule, but I think it's fair to say overall that since the time that Turing's article was written, the things that seem easiest to get computers to do are the things that we are able to learn how to do after say the age of 17 or 18. That is, we can get computers to play chess. We can get computers to interpret molecular spectra. We can get computers to do mathematical integration. We can get computers to invert matrices. All these things are in fact pretty easy or at least approachable to get computers to do. However, the things that we know how to do when we're five or six like walking around the room without bumping into things, to interact and converse, to understand stories like this one; those proved to be very difficult.

[15:49] So in this talk and a couple of following talks, what I'd like to do is try to unpack this notion of difficulty a little bit in the light of what we've learned from computer science. So in effect, by trying to mimic minds with machines, we have learned a great deal about the nuance of what makes a problem easier or difficult. Let's start with one type of example. For some types of problems, we think of them as difficult because as computer scientists, we would have no idea really or very little idea about how to even begin to program an answer to these problems. For all we know, maybe sometimes the problems are ill-defined or there's some reason that we couldn't achieve them. But we're not really sure what that reason is. In other words, they feel like very profound problems. In any event, they're problems that we find it difficult to get some purchase on.

[17:02] So an example is write a program that will pass the Turing test. It's hard to know where to start. We might have some ideas, but it's awfully hard to know where to start. Plato's dialogue, *Meno*, starts with somebody, a young man, going up to Plato's mentor Socrates in the streets of Athens and saying, "Socrates, can you teach virtue? Is virtue teachable?" From that they immediately get to the question of what is virtue. If we know what virtue is, maybe we know whether it can be taught. Well, in some sense, that's a very important problem. Every parent wants to be able to teach their child, if possible, to be a good and virtuous person. So it's not an unimportant problem, but it's hard to know where to begin to define virtue and to think about what it would mean to teach it. It's a problem that we look at and we say, this is so large and it may be ill-defined as far as we know, but there's no apparent easy way to get started with it. Or to take a kind of classic almost facetious example, to explain the meaning of life. It's hard to do.

[18:26] Another kind of difficult problem is one where you're asked to do something and you know that it is impossible. It can't be done. In human affairs, this is actually a tremendous knowledge usually. When you know that something, a well-defined problem, is impossible to solve, you know a great deal and often these impossibility proofs represent major landmarks of human thinking.

[19:00] So for example, the angle trisection problem. Using a straightedge and compass, and given an angle theta, construct an angle of magnitude theta over three. Take an arbitrary angle and create an angle one-third that size, an arbitrary angle. Now, it turns out that given a straightedge and compass, this can't be done. But it was an extremely provocative problem to the ancient geometers, and they did not know that it was impossible. In fact, the proof that this problem is impossible wasn't established until the 1800s, until the 19th century. So it took thousands of years for people to come to the realization, to be able to prove, that this is not only just a hard problem: it’s an impossible problem. You might as well not even try to do it because we know that it can't be done.

[20:07] Similarly, another impossibility is to write a computer program, and I’ll give this in the kind of more or less formal description, to write a computer program which given any other program P and input N will determine whether the program P halts on N. Let me say that again. What we want is a program that will take as its input another computer program P and an input value N and will run on its own and then come back and tell us whether the program P ever halts on the input N. It turns out that that is impossible. That's called the halting problem, and Alan Turing showed that that was impossible in a marvelous and much discussed and anthologized paper written in 1936-1937. So again, this is something that, in fact, it may sound like an abstruse mathematical problem, but essentially, it's the debugging problem. Could you write a program which could infallibly debug other programs? Informally is the way we could put that. The fact is you can't; it's impossible. Knowing that it's impossible is again a huge achievement in human thought. The perpetual motion problem in Physics and so forth - knowing that something is impossible is a very big deal. So we'll continue with this discussion, but these are two types. We've just introduced two types of ways in which a particular problem can be difficult, and we'll see some other ways in the next talk.

1.5 What Makes a Problem Hard, Part 2

[00:00] In the previous presentation, we talked about what it means for a problem to be difficult, to be hard, and we looked at a couple of examples of ways in which a problem could be hard. A problem might be hard because we don't even know how to begin to solve it, and we're not even sure that it's a well-defined problem in some cases; or a problem may be hard because it's provably impossible, which in most cases is a very good thing to know. We might as well not even start trying to solve this problem because we know it's impossible. So in this talk, what I would like to do is expand a little bit on this notion of difficulty or hardness. There are other ways in which a problem can be hard, and I'll mention a couple of them. Just genres of problems that are difficult, and why they're difficult.

[01:15] So sometimes problems are difficult for reasons that have to do with the nature of space and time. We don't know about things that, for example, are going to happen in the future, and sometimes we don't know about things that might have happened in the past. Perhaps we'll find out at some future time. So here's some examples. These are problems that involve some notion of uncertainty. We might be able to take good or bad guesses. There might be ways of approaching the problem. They're not necessarily ill-defined like the first type of problem. There might be ways we could go about answering in a better or worse way. But the kind of answer we can give is always tinged with a certain amount of uncertainty or levels of confidence or other notions that come up often in computer programming, in ways that involve probability or other kinds of formalisms to deal with uncertainty.

[02:31] Here are some examples. Was there life on Mars at some past time? Now at the moment, at the time that I'm speaking now, we don't quite have sufficient information to answer that problem yes or no. We can't give it a totally certain answer one way or the other given the evidence that we have in hand now. We could. Perhaps we could solve the problem or partially solve the problem by saying here's the evidence that we have that suggests that there might have been or that there could have been life on Mars at some past-time. And we might even come back with an answer that says in our view, it is probable that there was life on Mars at some past time. Again, we would have to be much clearer about the notion of probable, and we'd have to explain what we mean by saying that the answer to this question is probably yes or probably no. So we'd have to be much clearer about the notion of probability, but we could begin to answer a question like this. The first question about Mars is a question about the past. It's information that we have in hand, and maybe someday we'll get more information that will actually resolve the problem one way or the other.

[04:02] The second question has to do with a future event. If you, and of course computers in fact, do tackle questions like this - betting or making predictions about future events. Let's take a prediction from the realm of baseball. Will the Red Sox win the pennant this year? It's a source of endless conversations at the beginning of every baseball season, at least in Boston and the similar conversations go on all over the country at the beginning of the baseball season. Well, why do we even talk about this? We don't know, and it didn't happen in the past. So this is all a matter of prediction, but there are better and worse predictions. Some programs and some people are much more thoughtful about the predictions that they make, and their predictions have higher quality than others. So again, it's not unreasonable to ask a question like this and say how would we go about answering it. It's inherently difficult because it involves future events. We can't say the Red Sox will, yes, win the pennant this year or, no, they will not. It hasn't happened yet. Again we can talk in terms of probability or levels of confidence or things like that to talk about future events. It's another way a problem can be difficult.

[05:32] Sometimes, a problem is difficult because we have to make a judgment and perhaps in real time. Think about that third question. Is this object a threat? Is this person a threat? Is this animal a threat? It's a question that we as human beings might have to answer in extraordinary circumstances. We run across somebody or perhaps a stray dog in the street. Is this stray dog dangerous or is he friendly? That's a question that we might have to answer, and we might have reason to answer it one way or another, and we could give reasons for that. But again the answer is always going to have a certain level of uncertainty. First of all, even the notion of threat has some uncertainty. Maybe this person or animal is a mild threat or a severe threat. Terms like mild and severe are qualifications that make a problem like this, again, inherently difficult because we can't answer with certainty, but we can answer with some description of uncertainty or probability.

[06:49] And finally, the sentence at the bottom there, is that shape, which I typed with the zero on the keyboard, is that shape an ellipse? Is it close to an ellipse? Again, it's an interesting question. In some sense, the answer has to be no. An ellipse is a one-dimensional figure, and even if we think of that zero shape on the screen, it's a two-dimensional thing. It has thickness to it so in some sense the answer is absolutely no. That shape is not an ellipse. But then again if we take that point of view seriously, we've never actually seen an ellipse because you've never seen a one dimensional figure. So intuitively, we look at that question and think, well yeah, for our purposes, it's an ellipse. Even if mathematically it doesn't follow the perfect form of an ellipse, it looks close enough so that we could answer yes. Again this proves to be in some ways a difficult problem. It's difficult, because to answer yes or no requires additional language that we have to invent or formalize or elaborate on. So all of these are problems that can be thought of as difficult, because we need additional tools to talk about them.

[08:31] Let's go back. In the previous talk, we mentioned a couple of problems that we as people seem to solve in a fairly natural way, and by the time we're young children, we are able to solve them reliably: the vision problem and the language learning problem. Now expressed in formal terms, and these may not entirely reflect the problems that we as humans have to solve, but expressed in formal terms both these problems are in fact impossible. So I'm going to express these problems in terms that may be a little unfair to human beings; it’s not quite the way we encounter these problems. But let's just put them in the most stark terms. In the case of vision, if you recall, the question was can you recover three-dimensions from two? Can you take a pattern of light on the retina and recover from that, reliably, the arrangement of three-dimensional objects and the outside world? The answer is no. That's impossible because you can't recover three-dimensions from two. In fact, in general in mathematics, you can't recover n dimensions from n minus one dimensions. So this is just a special case of that. We can't get three-dimensional structure from a two-dimensional array of light.

[10:03] Similarly, let’s take a very strong and, again, almost certainly unfair version of the language problem. But given a set of sentences - the baby is born, he or she is listening to their native language - given a set of sentences can we determine the structure of the language that produced those sentences? Even here I'm going to be fairly specific about it. Let us say that the language that we're seeking has the structure of what's called a context-free grammar. Now English, there are all kinds of arguments about this. I'm not arguing that English or any natural language does in fact have that structure. But for the sake of argument, for now, let's say that we are trying to learn a language which has the structure of what in formal linguistics is called a context-free grammar. It's a relatively simple structure of a natural language. So given a set of sentences, can we determine the context-free grammar that produced those sentences? Again, the answer is, this is impossible. There could be an infinite number. Given a finite number of sentences, and that's all we can ever hear, there could be an infinite number of context-free grammars that produced those sentences. Now, again, this is not in fact a perfect reflection. Maybe it's not even a very good reflection of the language learning problem as we encounter it. But from a formal computer science standpoint, it's a start. Could you get a computer to sit on the desk, listen to a bunch of sentences, and find out from that the grammar that generated those sentences? The answer is no, it's impossible.

[12:09] Well, so are we back to something like the perpetual motion problem? These are impossibilities, but they're problems that are of such importance to us, such evolutionary importance to us as animals, that we have to solve them as best we can. So to take the vision problem as an example, we may not be able to, with other certainty, recover three-dimensional structure from two. We may not be able to do that. But we can make very good guesses. In fact, the human vision system is extraordinarily good. Animal vision systems in general are extraordinarily good at recovering three-dimensional structure from 2D patterns of light. So in order to solve the problem, we have to accept that we're not going to solve it with 100 percent certainty, but we can solve it to the best of our ability using all kinds of additional heuristics. Those are rules of thumb, and guesswork, and past knowledge, and a whole variety of other things that come into play in order to see reliably. We know that we can't do it with 100 percent certainty because of things like optical illusions.

[13:37] Optical illusions are fun and interesting precisely because they indicate to us the limitations of our own ability to see. We see some pattern, and we interpret it in one way, and then through other information we find out, no, we have misinterpreted this image. It is in fact something else. That's interesting. The reason we're able to misinterpret images is because our vision systems make smart guesses and use heuristics and so forth. If they somehow could solve the problem with 100 percent certainty, we would experience no optical illusions, but we do precisely because of the algorithmic limitations of our own vision systems.

[14:36] There are some problems from the computer scientists standpoint which aren't hard. They're easy problems. Computer scientists think of them as easy problems, and they're not always the easiest problems for people for one reason or another, but sometimes there's an overlap. So here are some examples. Given a number, a positive number, find its square root. Well, there are algorithms to do that. We can learn some of those algorithms, and we ourselves can find the square root of a number. It's a little more tedious for us than it is for a computer which can do this task very fast and very reliably. So that's an easy problem for a computer, and to some extent it's an easy problem for a person, although it feels like it would take some training and a little bit of work. Given a context-free grammar, produce a sentence using that grammar. In other words, this is not taking sentences and inferring backward to a grammar that produced them. It's you're given a grammar: can you produce a sentence from it? That's very easy.

[15:51] The third example is one that most people would find very difficult in practice. Because using pencil and paper this would be an extremely hard problem, but the technique to solve it is easily programmed into a computer. We can understand the technique to solve it. So this is one of these cases where the computer finds the job easier than people do. But the reason that that's true is because we as programmers know how to write programs to do this kind of thing. Given 100 linear equations with 100 variables, determine whether those equations have a solution, and if so, what it is. It would be a tremendously difficult problem for human beings to solve sitting at a desk, but it's not that hard a problem for a computer to solve.

[16:48] So we've looked at a variety of ways in which problems can be difficult. What I'd like to go on and talk about now is some of the language that has made a transition, has influenced our discussions of difficulty or easiness of problems that originated in computer programming. So I'm going to spend some time both in this talk and the following one describing a little bit of some of these ideas about what makes problems easy or difficult.

[17:25] We're going to start with this notion, in computer science it goes by the name of order of growth. Here's what I'm thinking about. Imagine a style of problem like, for example, solving a Rubik's cube. Now as you know, Rubik's cubes come in a variety of sizes these days. You can get a two by two by two Rubik's cube, or three by three by three, or four by four by four. I think they market a five by five by five. In our minds, we could imagine 1,000 by 1,000 by 1,000 Rubik's cube. No one would ever market or buy such a thing, but we could imagine such a thing in our mind. The idea of order of growth notation is to say how much harder does a certain kind of problem become to solve? How much harder is it to solve a certain type of problem as the problem grows in scale in some natural way? So applying this to Rubik's cube we could ask, how much harder is it in some formal sense to solve a three by three by three Rubik's cube than a two by two by two Rubik's cube? How much harder is it still to solve a four by four by four, or for that matter, 1,000 by 1,000 by 1,000? What we're really interested in when we talk about order of growth is as a problem gets large, as it becomes large-scale, how much more difficult time-wise does it take to solve the problem?

[19:06] Let me give you, just to start, an example of a very favorable order of growth, a favorable problem for this kind of order of growth notation. It's the guess a number problem. What I do is I tell you I'm thinking of a number between, let's say, 1 and 10, and you have to guess that number. If you guess wrong, I will tell you whether your guess is too low or too high, and we're limited to integers here. So I'm thinking of an integer in the range 1-10, and you can make a guess. If you guess too low, I'll tell you, and if you guess too high, I'll tell you, and if you make the correct guess I'll tell you that. So it's pretty obvious or it should be pretty obvious how you would approach a game like this. You might as well start by guessing the middle number: 5. If I tell you that that is too high, all of a sudden now you know you have some information that the number I'm thinking of is somewhere in the range from 1-4. If I tell you it's too low, you know that the number is from 6-10. If I tell you just right, you've solved the problem. Let's say I tell you it's too low so now you know the number is between 6 and 10: 6, 7, 8, 9 or 10. What would be the sensible guess to make? Again, guess the middle of the range that you have left. So your second guess would be an eight. If I tell you that's too low, you know that the number is nine or 10. If I tell you it's too high, you know the number is six or seven.

[20:52] Let's step back and say what's the idea here? The idea is if I give you a range, and I tell you I'm thinking of a number in that range, and I'll tell you if it's too low or too high, you already should be intuiting that there's a kind of algorithm for addressing this problem. First, you guess the middle number. If it's too low, you've cut the size of the problem in half, and now you have the upper half of the numbers in which to guess. If it's too high, same thing. You've cut the size of the problem in half. If it's exactly right, you win.

[21:32] Now, let's think about this in order of growth terms. It only takes a few guesses to get the correct answer for 1-10. How many guesses would it take for 1-100? Well, your first guess would be 50. If I tell you that's too high, you guess 25. If I tell you that's too low, you guess 37 and so forth. Each time you're cutting the size of the problem in half. It takes perhaps seven, eight guesses to get the right answer. Really seven guesses, I guess. It should take seven guesses to get the correct answer to the problem when the range is from 1-100. What about if the range is from 1-1,000? It should take about 10 guesses. What about if the range is from one to a million? That's a large range. I'm telling you I'm thinking of a number between one and a million. How many guesses will it take you using this algorithm to figure out what number I'm thinking of? The answer is approximately 20. Not that many guesses. So this is a case where the length of time to solve the problem grows at a favorable rate as the problem gets very large. Even when the problem is guess a number from one to a million, the amount of time that it takes to solve that problem hasn't increased that much. In point of fact, we would say that this is a problem that exhibits logarithmic order of growth, which is a slow order of growth, meaning that this is a problem that is relatively easy to solve even when the size of the problem gets harder.

[23:37] Let's be a little more specific about what order of growth means. So I'm going to give you a possibly not the most formal definition, but something close to a formal definition about what order of growth means. So the way in which computer scientists talk about this is they'll say, for example, they might say that a problem is O(n^2). You'll see occasional descriptions of that form. What does that mean? First, I'll give you a very informal definition. It means that as the problem gets larger at where we measure the size of the problem, the growth of the problem itself with some number n, the amount of time that it takes to solve that problem grows as pretty much some constant times n^2 or lower. It really means that we know that the problem won't get harder by more than a factor of some number times n^2. Let's be a little more specific. So here's the idea: imagine that we have a graph, and I'll put n on this axis. So this n indicates that as n gets larger, the measure of the size of the problem itself gets larger. Now, I'm going to take some constant, which we'll call k, and I'm going to graph kn^2 square. So we'll make this the time taken to solve the problem, and this is going to be some function of n. So there is a y-axis here with numbers on it, but this is going to be some function of n that I'm going to draw on this axis. Now, first, I can take the function kn^2, and it'll kind of look like this. I have left unspecified what k is, but I'm just saying, I can choose a k. So suppose this happens to be the graph of $4n^2$: I've chosen 4 for my value of k. When we say that a problem is O(n^2), what we mean is that if we graph the time to solve the problem against n, it means that at some point, like here, the graph of the time to solve the problem is going to drop below our kn^2 graph, and it's going to stay below that.

[26:54] So let me step back and explain again what's going on here. What we're saying is that as the measure of the problem size gets large, the time taken to solve the problem by our particular method is eventually forever lower than some kn^2. This is actually a very informative description because, for example, a problem that takes n^3time to solve, if I were trying to draw a graph and the graph happens to be expressed by n^3, then it would not fit under the graph of $4n^2$ forever after a certain point. Another way we can say this is that a problem that is O(n^3) is harder, has a larger order of growth than a problem that is O(n^2). A problem that's O(n^2) is a harder problem than a problem that's O(n). A problem that's O(n) is a harder problem than a problem that grows logarithmically like the guess a number problem that I showed you before.

[28:18] So what I really want you to get out of this description is that when you see a term like O(n^2), what that's giving you is a rough notion, but an informative notion of how difficult a problem is to solve as the problem gets large. People who have written programs understand in general that an O(n^2) problem is to be preferred, or if a method will solve the problem in O(n^2), then that's to be preferred to a method that requires O(n^3) or O(n^5).

[29:01] Now, in this second bullet point here, the crucial for computer scientists, the large-scale most important divide in talking about order of growth, is the difference between problems that are polynomial in their order of growth, meaning they grow as n^2, n^5, n^{100}, and problems that grow as order, say, 2^n. A problem that grows exponentially, a problem that grows not as a polynomial n^2, but as the exponent, the n is in the exponent of the expression here, so order 2^n, or 3^n, or 10^n, or 1.5^n, in fact, any number above one. A problem that grows exponentially is always worse in the large-scale than a problem that grows polynomially. Now, there are many specific cases where if we have to solve a very small problem that happens to grow as 2^n, then that may not be difficult for us, it may not require a lot of time. But ultimately, as a problem gets large, the distinction between polynomial-time problems and exponential-time problems is profound. To say it in a kind of general way, computer programmers when they're given something to do, they tend to hate exponential-time problems, because those mean or if a problem seems to only be solvable in exponential-time, then only very small versions of the problem can in fact be approached with a computer. We can't deal with the problem as it gets large, as it gets industrial scale. This isn't the only division, because some problems, in fact, are harder than exponential-time problems. These are often problems of theoretical interest only. Nonetheless, there's an entire range of mathematical difficulty of problems. For computer scientists, the first main divide to be concerned with is that between polynomial and exponential-time problems.

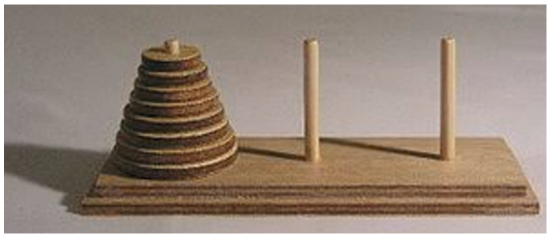
[31:37] I'll spend a little more time talking about variants of this, but for now, let's just say that there are other notions that arise from this distinction between polynomial and exponential-time problems. One of them is this idea that you might have heard about of NP-Completeness. For now, I'm not going to go into detail about that, but I'll just say that an NP-Complete problem tends to be a very interesting type of problem. It's one which to the best of our current knowledge in order to solve this problem reliably would take exponential time. However, an NP-Complete problem also has an interesting special property, which is that you could guess an answer in polynomial time, and you could check whether your guess is correct in polynomial time. So if you're very lucky, you could guess the correct answer, and check that it is the correct answer and do that in a reasonable amount of time. But in order to infallibly or reliably get the correct answer seems to take exponential time. That may seem like a fine distinction but a lot of really interesting hard problems turn out to be, and not unrealistic problems, turn out to be NP-Complete problems.

[33:07] Finally, this will be a topic that comes up as in this linkage between machines and minds because the brain, as we know, is a highly parallel structure. If you think of the individual neurons as tiny computational elements, then there are billions of these computational elements working all at once. This is an idea of parallel computation, and in the computer science world, again linking machines and minds, sometimes you have parallel computational structures, computers with multiple processors, or sometimes we think of the many computers, thousands of computers, all trying to solve a problem simultaneously and solving parts of it and sharing answers. That's often a very good strategy for reducing the time needed to solve a problem. In just a matter of theory, it does not really solve the difficulty of going from a polynomial to an exponential-time problem. That is to say, parallelism as a computational strategy can reduce the time needed to solve a problem, but it can't change an exponential-time problem to a polynomial problem. So this distinction that we began with is still in every case a very important one. When we return to this topic, we'll talk about some other ways of thinking about the difficulty of problems for minds and machines.

1.6 What Makes a Problem Hard, Part 3

[00:00] In the previous two discussions, we've been talking about what makes a problem hard or easy. And some of the new ideas that have come up because of the way in which we've started in the last half century, because of the way in which we've started to model minds with machines, we've been able to develop a more nuanced and expressive language about what makes problems difficult or easy. And so we spent the last couple of sessions talking about that; there is much more to say about it. But at least for this session, where it's kind of a final session on this topic (but we'll return to the topic as we discuss more about mind to machines) - in this particular session, what I'd like to do is talk about how computer scientists think about a certain kind of difficulty of problems. This ultimately is a rather challenging and perhaps technical subject, but for our purposes, I think we can offer an explanation here. We can look at an explanation, and we can explore these ideas without getting too much into the sort of mathematical depth of the subject. So what I'd like to do is describe one particular type of difficult problem, and use that to introduce another kind of difficult problem, at least as far as computer scientists are concerned.

[01:43] So let's begin with a classic puzzle. I believe it goes back to at least the beginning of the 20th century and the American puzzles. Sam Loyd wrote about this puzzle. I'm not sure if he invented it, but regardless of whether he invented it or not, he popularized it.



[Editor's note: The Tower of Hanoi was invented by a French mathematician, Édouard Lucas, in the 19th century.] This is called the Tower of Hanoi puzzle, and you see a picture of it here. There are three pegs and, in this case, there are eight discs. So you see a tower of eight discs. Each of the discs has a hole in the center, and they've been stacked on the left most peg. And you see that they go from largest disc at the bottom to smallest disc at the top. So what we now want to do, the goal of the puzzle, is to transfer these eight discs from the left most peg to the right most peg. And we're going to do that subject to certain rules. The rule is you can only take the top disc off a stack and move it onto another peg. And secondly, and quite important, you can never move a disc to a stack so that it's resting on a smaller disc than itself. So you gotta transfer these eight discs. You can take a disc off the top of the stack and put it on another peg. But you can't move a disc onto the top of a smaller disc.

[03:22] So if you think about how you might start this particular puzzle - I was considering actually bringing the physical puzzle and working it for you, but I decided it would be kind of unwieldy, and if I got the thing wrong on camera, I would be embarrassed beyond measure. So I'm not going to do that. But here's how you might start this particular puzzle. Take the smallest disc off the top of that stack at the left and place it on the middle peg. Now take the second disc off the top of that left most deck because that's now at the top, and move it to the right peg. Now as a third move, you could take the small disc which we placed on the center peg, and move it to the right peg, and so forth. In other words, what we're going to be doing in order to solve this puzzle is to move the discs one by one from peg to peg subject to the constraint that we can never move a disc onto a smaller disc.

[04:25] Now how hard is this puzzle? This is an interesting question. In one sense, the puzzle is actually quite easy, and here is what I mean. The strategy for the puzzle is easily expressed, and you can find examples, for instance, on YouTube where you see people solving versions of the puzzle. So here's the strategy - I've expressed it on this one slide. We want to move a tower of N discs from peg 1 to peg 3, using the center peg as a kind of spare, as a place where we can occasionally move discs if we need them, okay? So we want to move a tower of N discs from peg 1 to peg 3, using peg 2 as a spare. Now think about how we could do this; and I'm going to express a solution, and if you consider it, you'll see that this would work. In order to move eight discs from the left peg to the right peg using the second peg as a spare, first what we're going to do is move seven discs. We're going to solve the puzzle for seven discs, and we're going to move seven discs from peg 1 to peg 2, that is the center peg, using peg 3 as a spare. Now, think about what happens when we do that. We're going to move the top seven discs; we're never going to touch the bottom disc. After all, who cares? If we're moving only the top seven discs, we don't need the eighth at all, and it'll always be bigger than anything else that we're moving. So for the time being we'll ignore the eighth disc, the one at the bottom. We'll just work with the top seven, and we'll move those from peg 1 to peg 2, using peg 3 as a spare. Once we've done that, think about what we have at the end of that process. We have one - the bottom eighth disc is still there on the left peg. We have seven discs on the center peg, and the third peg is empty. Now what we do is take the big disc from peg 1 and move it over to peg 3. That's just a simple move. And then we solve the seven disc tower a second time: moving things from peg 2 to peg 3, using peg 1 as a spare.

[07:07] This is inherently, the term that computer scientists use for this kind of process is recursive, this a beautifully recursive process. I've expressed it in an abstract way. To solve the tower of N disc going from peg 1 to peg 3, first solve the N-1 tower going from peg 1 to peg 2, move the bottom disc, and then solve the N-1 tower again. And you'll see, you could try using this strategy. It's trivial on a tower of size one disc. There you just move the disc from peg 1 to peg 3. For a peg 1 to peg 3 move up two discs, you first move the small disc to Peg 2, the second disc to Peg 3, and then the small disc to Peg 3. So we can solve a puzzle of size one. We can solve a puzzle of size two. Using our knowledge of how to solve a puzzle of size two, we can solve a puzzle of size three, and so forth. So in some sense, the Tower of Hanoi is an easy puzzle. We can express the idea of solving it on one slide. That's all it takes. So is this an easy puzzle?

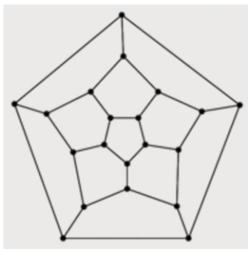
[08:31] Well, in another respect, this is a very hard puzzle. To solve the tower puzzle with 1 disc, it takes 1 move, just one move. Move the 1 disc from Peg 1 to Peg 3. If you think about what I was saying before, solving the tower of size two takes 3 moves. Solving a tower of 3 discs takes 7 moves. You have to do the 'two solution' twice with one additional move in between. Overall, to solve a tower of n discs requires twice the time to solve the n minus 1 disc puzzle, plus 1 additional move. So as it turns out, solving a tower of 8 discs requires 255 moves. Solving a tower of 40 discs, and 40 is not a whole lot of discs - it's just a big tower, but you could engineer such a puzzle; solving that would take a trillion moves. Solving a tower of 60 discs would take a million trillion moves.

[09:44] Now consider what we've just said. What we've just said is that expressing the idea of the solution is easy. It is easy to say what the solution path would be. But actually enacting that solution, actually implementing it, takes, in fact, exponential time. The time to enact the solution grows as approximately 2^n, where n is the number of discs in the original tower. So solving a tower of size 60 takes about a million trillion moves. In other words, we, as human beings, and for that matter, machines, computers, would take a long time to actually print out a solution to the Tower of Hanoi problem. And if the tower starts out with, say, 100 discs, we're talking about a problem that would take too long for even the fastest computer to solve in anything like a meaningful time. So even though the problem is easy conceptually, it takes an extraordinary amount of time to implement that solution.

[11:04] We might think, well, if we were more clever, we could come up with a faster solution. Maybe we came up with this easy solution, but there's a more complex solution to express, but it'll take a shorter time. No, this is the best we can do. This is the best solution you could have. And I haven't shown you that, but that is the truth. So there is no way of improving on the situation that we've got. If you have a tower of size n, it takes approximately 2^n moves, even to print out a solution. And that becomes, well, I mean, just impractical at least for either a person or a computer, even the fastest computer. So is this a hard or an easy problem? Well, I don't know how to quite answer that question without sort of going into this sort of detail. In one sense, it's an easy problem. In another sense, it's a very hard problem.

[12:12] Now I mentioned when we were talking about the difficulty of problems earlier, yet a different kind of difficult problem. This is called, you might have heard this term, if you're a computer scientist, you've definitely heard this term, if not, you might have heard this term in the sort of popular culture, the idea of an NP-complete problem. And while I won't go into complete detail, fine detail, about what this means, here's what it means in essence: NP stands for "non-deterministic polynomial". And a NP-complete problem is one where you can guess a solution quickly in polynomial time. And you can check that you've got a correct solution quickly in polynomial time. But in general, there's no way of actually being sure to find that solution, except in exponential time. Now that may seem like a fine detail.

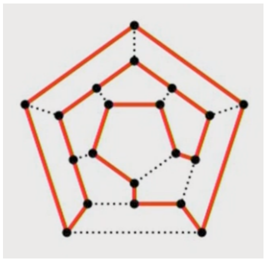
[13:19] Let me show you a very famous example of an NP-complete problem. It's called the Hamiltonian cycle problem. Here's the idea. You have a graph, think of the graph at the top here in this slide. As it happens, it has 20 vertices and edges between those vertices, okay? The question of whether this graph has a Hamiltonian cycle is whether we could draw a red line starting at one vertex and moving over edges from to vertex to vertex, never visiting the same vertex twice, and end up finally at the end back where we started. I've shown a Hamiltonian cycle in the graph below this graph. So in other words, we have a graph of 20 vertices, and the question is does there exist a path that will close on itself, which visits each vertex exactly once and only traverses



edges that are in the graph? You might have heard of a variant of this kind of problem under the name of the traveling salesman problem. The traveling salesman problem is one where you have n cities to visit, and you want to visit them each once, and you want to minimize the costs, either in mileage or expenditure, in making a complete tour of all the cities. It's a similar problem. This happens to be a slightly more abstract problem.

[15:03] Now, in general, if somebody hands you a graph, does it have a Hamiltonian cycle in it? Again, the sort of obvious way to answer a question like that is to list all the possible orderings of vertices in the graph. List all possible cycles in the graph, that is, list all the possible orderings of, in this case, 20 vertices. See if any of them correspond to valid edges that can move between the vertices. If you find such a thing, the graph has a Hamiltonian cycle. If you can't find such a thing, the graph doesn't have a Hamiltonian cycle. That's a very straightforward path to a solution. The only problem is that first step: listing all 20 possible cycles. We don't know when we make a list of all 20 possible, it turns out to be 20 factorial. So if you list all the possible ways of ordering 20 vertices, you have a list whose length is 20 factorial. Many of the elements of that list will not correspond to a cycle because there's no edge between neighboring vertices in that ordering. So you have to actually make a list, it's 20 factorial elements long. And then you have to check each and every one of the elements in that list.

[16:43] Now 20 factorial is a very big number. And as the graph gets bigger and bigger, say I give you a graph of 100 vertices, then my simple solution requires a first step, which involves making a list of 100 factorial elements. Again, a hundred factorial is the kind of number that, it's a beyond astronomical number. There is no way that a computer program could write such a list or go over such a list element by element in anything like meaningful time. So what have we got here? We have a problem that looks superficially like it might be kind of like the Tower of Hanoi problem. In order to solve the problem, we have to do a step that is exponential time. But there is one important difference here, and it's illustrated by these two graphs on this slide. Somebody drew on this graph a red line that corresponds, in fact, to a valid Hamiltonian cycle. It didn't take an exponential time to draw that red line. Unlike the Tower of Hanoi problem, it doesn't take exponential time to actually express a solution. In this case, it took a very short time to guess a solution. And how long does it take to check that solution? Not long at all. Just look at the second graph, follow the red line and you'll see, yep, that's a good Hamiltonian cycle. So guessing a solution is brief polynomial time. Checking a solution is brief. But as far as anybody knows, and this is an unsolved problem at present, as far as anybody knows, there is no surefire way to find a Hamiltonian cycle that does not need exponential time.



[18:46] NP-complete problems, you might think from this description that I've just given, NP-complete problems are kind of a mathematical oddity, but it turns out they're not. They're all over the place, and they often come up in situations where you have to optimize a solution. They're often in situations of ordering and optimization, and there are many, many np-complete problems that are quite naturally related to things that we might want to write computer programs to do. So when a computer scientist runs across an NP-complete problem, usually what he or she does is decide that they're not going to be able to solve the problem with certainty. But maybe they can come up with a solution that is sometimes called satisficing: it'll satisfy the situation, and it'll suffice for the current situation. So computer scientists, when they run into NP-complete problems, they often start looking for ways to come up with an approximate solution. NP complete problems are extremely interesting, and they've only been discovered over the last half century or so.

[20:08] Okay, so where does all this leave us at this point? We've now talked about different ways of thinking about difficult problems. First of all, the main thing, the whole point of this discussion, has been there are many ways in which a problem can be hard or easy. Some of these have profound implications. Some of them involve the idea that stating a solution would not be difficult, but actually implementing the solution would be difficult. In the case of, we've been talking about the link between machines and minds, in the case of the problems that we looked at earlier - learning one's native language or opening one's eyes and seeing objects in the world - those proved to be extremely difficult problems even though in some sense they feel easy for us. We open our eyes and we see objects. But it took millions of years of evolution to get to the point where we would have eyes that were capable of doing that. We learn our native language without direct instruction, but it takes us years to do it. And again, we have a great deal of evolutionary equipment built in that allows us to do this, that gives us the tools to do it. At present, we do not understand entirely the mechanisms either by which we learn our native language or by which we accomplish the vision task. These proved to be very hard problems, even though they feel easy for us. And as we continue to talk about minds and machines, this sort of gap in the notion of difficulty - things that are easy for machines but hard for people, things that seem to be easy for people but are hard to program - these will come to the fore over and over again.