[SQUEAKING]

[RUSTLING]

[CLICKING]

JASON KU: Welcome, everybody, to the second-to-last lecture of 6.006. In this lecture, we've mostly covered all of the testable material that we're going to have on the final, or on quiz 3. Today, really what we're talking about is putting into context all the material that we've learned over the course of the term at a high level and talk about where we can go from here in terms of other theory classes and other classes in the department that are related to this material.

Now, most things in the department are in some way related to this material. And so that's why there's a foundational course. But we're going to try to talk about it from a high level and talk about how some future things that you might be interested relate.

OK, so we started out the term, in lecture one, talking about 6.006, and we had four main goals that we had for our course-- really three main goals Does anyone remember what those goals were?

So you got to the last one first. The first one was to solve hard computational problems, to be able to solve problems. So this is kind of like the "let's make an algorithm" part of the course. 1, solve hard computational problems.

I guess "hard" here maybe should be in quotes because we saw in the last lecture what hard means in a technical sense. Hard could mean that there's no efficient algorithm that we know how to solve a problem on. That's getting a little bit of ahead of ourselves. Computational problems with algorithms is really the key part about this goal.

It's kind of the same goal that you have in a class like 6.0001 or 6.009. You're trying to convince a computer that you solved a problem on a finite set of inputs. But really what this class is about is two other things, which is more about communication to people rather than computers. Your algorithm might be correct or efficient, but you need to be able to communicate that to humans. And that's what the other two goals are.

So second one is argue correctness. Basically, the thing that I'm doing to my inputs is always going to lead me to a correct output. No matter what input I give it, any valid input-- there could be an infinite space of possible inputs, and in this class, that's the case, because we want our input size to grow arbitrarily large-- we need to be able to argue correctness that it's going to return me the correct thing no matter what my inputs are.

And in order to do that, that's essentially-- that's 6.042. This whole class has basically been applied 6.042. I've given you some procedures, and you have to prove things about these procedures. Or most of the time, we proved it for you, and then you've used them as black boxes. But that's a lot of what this class is about.

And the third one is efficiency-- argue that it's "good," for lack of a better thing. This is efficiency. What does "good" mean? Well, that was hard to know at the beginning of our class. And so we set up this model of computation, a framework, through which we could determine how good or bad our algorithms were by saying-- by defining a model of computation, saying what things we can do in constant time, and then just building off of that.

So this is basically our model plus some asymptotics or something like that. Ran out of space. What?

AUDIENCE: It's about scalability.

JASON KU: Yeah, this is about scalability. A model of computation tells us how much time we can spend, but it's compared to our input size. This is always about, how does our algorithm perform relative to the rate that our problem size grows? And so that's what we mean by "good."

And in this class, we don't tend to talk about constant size problems. It's about how algorithms can scale as you have arbitrarily large inputs. That's why we need recursion and induction to be able to prove things about our algorithms, because they're for arbitrary n. And that's why we need this relative-to-input size, the growth factor of our algorithm's performance, relative to the input.

OK, and then the last thing is, to me, one of the most important things, is communicating these things to another human. So communication is key here. If you can always write good code that's always right, good for you. I can't do that all the time. But that might mean that you can be very-- a competent, independent computer programmer. But you are going to be limited in what you can do if you're only able to rely on yourself.

A lot about computer science is working with others to solve computational problems. And when you're working with others to solve computational problems, you need to be able to communicate with them, and you need to be able to communicate them both what it is you're doing and why it is you're doing it-- that you're doing the correct thing and that it's efficient.

And so that's a big part about what this course is. At the end of the day, on your quiz, if you write down Python script for a correct algorithm, but we don't know what it's doing, but it's correct, we're not going to give you full points on that, because you're not satisfying the conditions of this class. It's really about the communication here.

OK, so just to review, since we've not discussed how the most recent lecture fits into your problem sets. We didn't have any problem sets that covered complexity, so how does that fit in? Well, argue that the ways that we're solving our problems are good. What we proved in the last lecture was that most problems cannot be solved good. They can't be solved in polynomial time with respect to the size of your input.

However, most of the problems that we think about, in a sense, I can prove to you that it's a yes solution. I can show you a simple path in this graph that has a certain length. Or I can show you a subset that sums to a certain value in a particular problem. I can give you a certificate that I can prove to you in a reasonable amount of time that, yes, I can prove to you that this is-- the answer to this thing is correct. And that's what we talked about in the last lecture.

So not always "good" algorithms to solve problems, but many problems we think about can be either checked in polynomial time-- this is the concept of having a certificate that I could give you of polynomial size that could be checked in polynomial time-- in a sense, that's a way-- check-- checked in polynomial time. This leads to our class of decision problems, NP. Or it can be solved by brute force in exponential time.

Most of the things that we've talked about in this class fall into one of these two categories. We can just brute force over the combinatorial space of possible outputs and check to see if they're correct. Or I can give you a certificate basically saying, look, I can solve-- actually, anything that's of this form can be checked in this form, because there's only a polynomial number of things to check-- or, sorry, an exponential possible number of certificates of polynomial length to check.

But basically, this is saying that the problems that we think of mostly fall into these two categories. And so there usually are algorithms to solve the problems that we care about, even if most random problems in terms of bit strings that we gave an analysis in the last lecture actually prove that most random problems are not solvable. In a sense, the problems we think about are not random. They kind of have this structure that they can be checked pretty quickly.

OK, so that's what we mean when we are talking about complexity. For the purposes of the final, you'll be able to see on your final exam practice problems that we're going to give you, most of what we cover in terms of material on the final that will be testing the lecture 19 material will be in terms of the definitions. Do you understand what the decision problem class NP is? EXP is? Do you know how these relate to each other? EXP is definitely a superset of NP here. NP nestles inside here. They could be equal-- probably not. Those are the types of things that we would address.

Knowing a directionality of a reduction. If you have a problem A and a problem B, and I know that this one is difficult by some measure-- I already happened to know that it's very hard, like NP-hard or something like that. If this is a problem that I'm interested in knowing the complexity about, and I can prove that I can solve it if I had a black box to solve B-- any black box to solve B, and I could make this reduction in polynomial time, and if this is hard, that means this can't be-- that means that if this is hard, then I better not be able to solve this in polynomial time, because then I would be able to solve this in polynomial time.

So that's basically the type of argument usually in a true/false question we might have on the final exam for you to kind of understand the basic high-level definitions involved in what was talked about in lecture 19. Hardness-- the very most difficult problems of these classes-- and completeness-- sorry, anything harder than things in these classes. Whereas completeness is the ones that are in this set, but at least as hard as anything in those classes. So that's just to give you a brief overview of the only material that hasn't been tested but might be tested on the final.

So when we don't have a good algorithm, we can actually prove that it probably doesn't have a good algorithm. And that's a problem that you'll be able to solve in future classes, if you continue along this track.

OK, so what's the actual content that we talked about? This is a very high-level overview of why we're taking this class, why you're taking this class. But what is the content we actually covered?

I like to break it up into three units and, in a sense, two subunits. So quiz 1 material and quiz 2 material was about showing you some nice black boxes. Basically, if I'm going to have inputs of non-constant size, it's going to be useful for me to be able to find things among those elements. So that's really what quiz 1 is all about-- data structures for finding things in non-constant size database.

Sure. And when we were storing these things, we want to support maybe two different types of queries-- ones that were intrinsic to the items, what the items were, and ones based on what-- an extrinsic order was placed on these items. And that was a way in which we broke down, how should I approach looking at this problem?

I want to be able to support queries and maintain an extrinsic order on these things. I might want a sequence. This is a sequence extrinsic order. Or I want to be able to look up, is this thing in my set, by a key that we identify, with a unique key. So this is some intrinsic queries, and often order.

A hash table doesn't maintain any order on my keys. But it does support intrinsic queries-- is this thing in my set or not? But we did show you other set data structures that do support an intrinsic order that allows me to see what the next larger and the next previous-- the next larger and the next smaller item is in my set.

So here's a summary of those data structures that we had. I'm not going to go into how to use these things or how to choose from among them here. That's what your quiz 1 review lecture was all about. But basically, the idea here is, if we have a sequence, most of the time when you're programming, being able to push and pop at the end of a list is pretty good. Which is why Python, the most fundamental data structure that you have, is a list, because it's a super useful thing.

I just want to store a bunch of things, have random access to the, say, 10th element to my thing, but I'm not necessarily having to dynamically update the order of these things dynamically. I don't necessarily have to insert something in the middle of the list. But most of the time, what I can do is put it at the end of the list and maybe swap it down into place if I need to.

So that's why a list is super useful. A sequence AVL tree, useful, but not as ubiquitous as a linked list-- I mean, as a dynamic array, sorry. I said linked list, I meant Python list, which is a dynamic array. So the dynamic array tended to be, in your coding practice, your most common sequence data structure here. Though, we can get pretty good for this insert in the middle operation with the sequence AVL.

OK, then on the set data structure side, I categorize these things into a couple different categories here in terms of the operations we can support on these things. These are all intrinsic operations-- finding things, inserting, deleting things. I think of the first three as being dictionary operations. I want to just look up whether something's there. Whereas the last two are order-preserving operations, where it matters what the order of these things are stored in.

And so as you can see from the asymptotic complexity of the various operations here, the hash table is actually super good if you want the dictionary oper-- if you just want to support dictionary operations. But in the cases where you need to maintain order dynamically, a set AVL is the way to go. But if you don't need it dynamic, but you still need those order operations, a sorted array is just good enough if you don't need to change what they are.

So that's a quick overview of quiz 1-type data structures material. But then we used most of these data structures to get faster sorting algorithms in different contexts. Basically, everything on this list involved making a data structure and exploiting that data structure to get a better running time, all except for merge sort, really.

The first two we presented in terms of a priority queue, whether we used a sorted array or an array. We represented it at the end of lecture eight to get n-squared running time. We generalized that down to n log n by using a heap. That was a nice optimization. But we also got interesting data structures using an-- I mean, interesting sorting algorithms using an AVL tree because of the power of maintaining a dynamic order over time. But then exploiting a direct access array to be able to sort in linear time for small-bounded-- bounded in terms of the input, polynomially bounded in terms of the input-- ranges of numbers.

So we leverage that direct axis array to get counting sort. And then we kind of amplified that effect by sorting on a bunch of digits multiple times to get basically polynomial blow-up in terms of the numbers that we could sort in linear time.

So that's an overview of the content of quiz 1. In quiz 2, we were kind of like, OK, now you know how to find things within a set of just a flat list of things, you can put it in a data structure. But in a sense, a graph is a special kind of data structure that relates the different things in your input.

So if you've got a bunch of vertices, there's a relation now between those vertices that are your edges. And this is a super useful framework in talking about discrete systems, because you can think of a vertex as a state of your system, and then connect these transitions as a graph.

That's the reason why-- I mean, graphs are awesome, but they're awesome because they can be used to model so many different things within our world. It's not just about road networks. It can also be about playing your favorite turn-based game, like Tilt.

OK, so we talked about a lot of different types of problems that you could solve, various algorithms, with a focus on a bunch of different ways of solving single-source shortest paths. And again, just like the sorting algorithms and just like the data structures, we presented multiple of them, because we had this trade-off of generality of the graph that they apply to contrasted with the running time.

So I guess, in particular, the top line there is, in some sense, the most restrictive. We don't have any cycles in our graph. That's a very special type of graph, and we're able to get linear time. But then even if we do have cycles in our graph, we can do better if we have a bound on the weights in our thing, whether they be-- there's an easy conversion to a linear time algorithm via an unweighted process, or whether these things are non-negative, so there can't be negative weight cycles, and we don't have to deal with that.

OK, so that's quiz 2 material. And then quiz 3 material was kind of applying this graph material to a recursive framework. What was our recursive framework? Everyone say it with me.

AUDIENCE: Dynamic programming.

JASON KU: Dynamic programming, and the framework was SRT BOT, right? Missing a letter, but SORT BOT, right? You can actually think of the quiz 3 material as really an application of the graph material. What are we doing in SORT BOT? We're defining a set of subproblems. These are a set of vertices in a graph. What is the relationship doing? It's saying, what are the relation between the subproblems, essentially defining the edges of a graph?

And then this topological order and the base cases, all of these things are just saying, what is the problem that I want to solve on this graph? And how do I compute that for things that don't have any outgoing edges? I need to start writing on the board again. This is graphs. There was sorting in here, too. This is basically an application.

OK, graphs was basically a relationship on these non-constant things. So this was kind of like useful black boxes that you can just bundle up and stick in some inputs, stick out some outputs, and you're golden. Whereas quiz 3 was very different, the material in quiz 3 is very different.

Dynamic programming, while it was, in some sense, related to this graph material-- I'm constructing a graph-- I have to construct that graph. There's a creative process in trying to construct that graph. I don't give you a set of vertices. Usually what I give you are a set of-- a sequence or something like that. And you have to construct vertices, subproblems, that will be able to be related in a recursive way so you can solve the problem. This is a very much more difficult thing than these other things, I think, because there's a lot more creativity in this.

In the same way that just applying-- reducing to the graph algorithms we have is fairly easy. But actually doing some graph transformations to change the shape of the graph so that you can apply these algorithms, that's a harder thing to do. The difficulty with these two sets of materials is very similar. Figuring out what the graph should be, figuring out what the subproblems should be and how they relate, is really the entire part of the-- the entire difficulty with solving problems recursively.

And we've only given you a taste of solving problems recursively. In future classes, like 6.046, which is the follow-on to this one in the undergraduate curriculum, this is all about introduction to algorithms. The next one's about design and analysis of algorithms. It's quite a bit more difficult, because we've mostly left it to you to use the things that we gave you or make your own algorithms based on this very nice cookbook-like framework that you can plug in a recursive algorithm to.

Now actually, that cookbook is super nice for any way of looking at a problem recursively, but while in dynamic programming, the inductive hypothesis of combining your subproblems is almost trivial, in other types of recursive algorithms, that's not necessarily the case.

Especially when instead of looking at all possible choices, for example, in a greedy algorithm where you're just looking at one of the choices, the locally best thing, and recursing forward, you're not doing all the work. You're not locally brute-forcing. Your locally picking an optimal thing locally and hoping that will lead you to good thing. That's a much harder algorithmic paradigm to operate under. And so that's more like the material that you'll be talking about in 6.046.

So that's 006, a very quick overview of the content of this class. And we really like the structure of how this class is laid out, because it gives you a fundamental idea of the things people use to store information on a computer and a sense of how you solve problems computationally and how to argue that they're correct and efficient. That's really what this problem-- this course is about.

And if you feel like you enjoy this kind of stuff, that's where you go to take 6.046. And 6.046 was actually the first algorithms class I ever took here at MIT, as a grad student actually. This was hard for me. It's actually hard to look at these problems, these types, and think in a computational way, especially having not taken this class, 6.006. So hopefully you guys are all in a better position than I was when I took it.

There's two ways I like to think of the content in 6.046. One is kind of just as an extension of 006. It's the natural follow-on to the things that we do in this class. They still talk about data structures. This isn't the core part of 046, but they do touch on data structures for more complicated-- that have more complicated analyses involved in them.

It's really about-- usually in 046, stating what the algorithm is doing is not so hard. Basically, giving you the algorithm, number one here, is not so difficult, to state what's happening in the algorithm. But the number two and number three here, arguing that that thing is correct and arguing that thing is efficient, that's where the complexity comes in in 046. The analysis part is quite a bit more complicated in 046 than in 006.

So they solve a problem called union-find and give a much-- we talked a little bit about amortization. This goes into a much better-- a much more formal way of proving things run in amortized time. So this is basically amortization via what we call a potential analysis.

It's basically making that notion that we talked about when we were talking about dynamic arrays of, we're not doing this expensive thing too often. Basically what we do is we keep track of the cost of all sequence of operations and prove that the average cost is small. That's kind of what this potential analysis is doing. It's a little bit more formal process for making that argument a little more formal. Right. OK.

So then on the graph side, this is kind of an extension of quiz 1-type material. This is, what is this union-find data structure? It's basically-- it's a set type thing, where I can make a set of just a single element, I can take two sets, merge them together, make them their union, and then given an object, I say, which set am I part of, essentially by electing a leader within a set and saying, return me a pointer to that one.

And so this can be useful in dynamically maintaining, say, the connected components in a dynamically changing graph supporting the query of, am I in the same component as this other guy? That could be a very useful thing to know about a graph as it's changing. So that's an application of this problem. And they get near-constant performance for a lot of these queries. It's not quite, but pretty close.

OK, on the graph side, they solve a number of very useful problems on graphs. Minimum Spanning Tree-- so I'm trying to find a tree connecting all of the vertices in a connected component of my graph. And I'm trying to find-- in a weighted graph, I'm trying to find the spanning tree that has minimum total weight. So that's a problem-- a fundamental problem in weighted-graph algorithms. They've solved this via a greedy algorithm.

And network flows and I guess cuts. So this is-- what is this? This is, I'm given a weighted graph. Basically, each of the weights correspond to a capacity. I could push water through along this edge. And I may be given a source vertex and a sink vertex. And I want to say, I want to shove water through the source vertex along the edges with their various capacities. And I'll get some amount of water on the other end in the source.

So the question is, what's the most amount of water that I can push through this? Well, I could build that pipe network with the different things and just do this experimentally. I just stick a bunch of-- maybe-- I'm a mechanical engineer, so that maybe makes sense to me. But you want to be able to just look at those numbers and be able to tell me how much water can I push through.

That's what the max flow in a network is talking about. And we give you some polynomial time algorithms in this class, basically incremental algorithms that, kind of like Dijkstra, or kind of like Bellman-Ford, will incrementally update estimates to-- of a max flow and improve them over time.

Then on the, basically, design paradigms, you've got more involved making your own divide-and-conquer algorithms, dynamic programming algorithms, greedy algorithms. Basically, they go a lot more in depth in terms of how to design these algorithms and these paradigms than we do in this class.

And then the last thing is-- we only touched on complexity. And in a sense, 046 is only going to touch on complexity. It's a very big field. But it will give you the tools to be able to prove that something is NP-hard, whereas we just kind of say that, oh, there's this thing called a reduction. We didn't give you any problems in which you actually had to reduce one problem to another. And you'll do a lot more of that here. So, reductions.

So in a big sense, 046 is really just a natural extension to the 006 material, plus some additional stuff, which I'm going to get to in a second. Yeah, question?

AUDIENCE: Do you want to add randomization for time paradigms?

JASON KU: I'm going to talk about that slightly in a separate-- I'll get to your question in just a second. I like to think of it as a separate topic, which I will go into right now. The separate topic I like to think of it as, instead of being the natural extension to the things in the 006 units, what I'm going to do is kind of relax either what it means to have a correct algorithm or relax what it means to-- what my model of computation is.

So 006, this is kind of as an extension of 006. And this is kind of like 6.046 as change my definition of what it means to be correct or efficient.

So we've already kind of done this a little bit in 006. Basically, one of the things that we can do, which is what the question that a student asked a question about, was about randomized algorithms, which is a big part of 046 actually-- randomized analysis of algorithms that are not deterministic. It's not guaranteed that it'll give you the same output every time or not guaranteed that it will do the same computations over the course of the algorithm every time. But it exploits some randomization.

And in 006, this is-- we've mostly not touched on this, except in one area. Where did we use randomization? In hashing, right? When we used hashing, what were we doing? We changed the definition of correct versus efficient. We didn't really change the definition, what we did was we said that it was OK that sometimes our algorithm was slower than we-- than on-- in expectation. That's what we meant there. We're relaxing the idea of efficient, but we're still saying it's good, because most of the time it is good.

So there's two types of randomized algorithms. They have these weird names based on betting regions of the world, shall we say? There are-- this is L-O? Los Vegas? It is Las, OK, Vegas algorithms. These are always correct, but probably efficient.

In a sense, that's what hashing is. I'm always going to give you the right thing, whether this thing is in my set or not. But some of the time, it's inefficient. I have to look through a chain of length of-- that's linear in the size of the things that I'm storing.

And this is in contrast to a Monte Carlo algorithm, which is always efficient for some definition of efficient, but only probably correct. And I mean, I could define you a hash table that has Monte Carlo semantics instead. Say, for example, I say that I'm going-- it's going to be exactly the same as a hash table, except instead of storing all the things that collide in a place, I just store the first two, say.

Well, actually, that's actually going to be always efficient. I'm going to look through the things and see if it's in there. And the chains that I'm storing there only have two things. It's going to be always efficient. It's always going to give me constant time. But some of the time, it's going to be the wrong thing, because I'm not storing everything in that chain.

So there's some probability that that's not going to be correct. And so that's a different kind of-- maybe I want my hash tables to always be fast, but I can afford to be wrong some of the time. I don't know. In practice, this is actually sometimes a good trade-off in real systems. Sometimes it's OK to be wrong some of the times, if we get good performance.

OK, but generally can do better if you allow randomization. And by better I mean, usually we can get faster bounds on a lot of problems if we allow randomization and things aren't necessarily always correct or always efficient. So this is a big area in 046 that requires a lot more analysis using randomness and probability. So if you need some primers on that-- we didn't have a lot of this in 006, but if you go on to 046, that's going to be a really important thing for you to brush up on.

The next part on 006 is kind of changing what our definition of correct or efficient means. I mean, we've restricted ourselves in this class to a class of problems where we only talk about integers. But there's tons of problems in this world, especially in scientific computing, where I want to be able to find out what this real number is. And I can't even store a real number on my computer. So what the hell, Jason? What are you talking about?

I can't do that on a computer. But what I can do is basically compute things in a numerical sense-- numerical algorithms. And in 046, a lot of times we put this in the context of continuous optimization, continuous being the opportune word here, not discrete systems. You have a continuum of possible solutions, real numbers essentially. How do we do this on a computer that's a discrete system?

Basically, in 046 what you can do, and in other numerical methods classes, what you can say is, well, I know that you can't return me a real number. I got that. Or you can maybe have a model of computation that allows integers to represent other kinds of real numbers, like radicals or rationals or something like that. And I can do manipulations on those. But really what these algorithms are usually about is computing real numbers not completely, but to some bounded precision. And I pay for that precision.

The more bits of precision I want on my number, I have to pay for them. So this is basic-- I think of these as an approximation-- approximation of real number to some precision, and I pay for precision with time.

So let's say I wanted to compute the square root of a number. I could have an algorithm just like the algorithms-- or I guess division, right, long division. You all know the algorithm of long division. You put the quotient under here with these-- an AB and you get the C on top or whatever. That's an algorithm. That's a procedure using essentially small numbers.

I'm only talking about the digits zero to nine here when I'm doing that algorithm. So it's a procedure that only uses small integers to compute arbitrary precision of a division. So that's an algorithm, and I have to pay time to get more digits. So that's an example of this kind of-- how we live in the world of real numbers when all we have is a discrete system.

And then the last category I'd like to talk about here is really approximation algorithms. Whereas this is kind of an approximation algorithm, I'm approximating my outputs, this is an approximation algorithm from the standpoint of, well, there's a lot of problems that I can't solve efficiently. They're NP-hard. They're in EXP or even harder problems. But maybe I'm OK with not getting the optimal solution.

So this is in the domain of optimization problems. So most of the dynamic programming problems that we gave you were optimization problems. They're the shortest paths problems. Those are optimization problems. Basically, the possible outputs are ranked in some way-- the distance of a path that you return or something like that. They're ranked in some way. There is an optimal one-- the one with the smallest metric or something like that.

Well, in an approximation algorithm what I do is, OK, I get that it's computationally difficult for you to give me the longest simple path in this graph, or the shortest possible route for my traveling salesman, but maybe that's OK. I mean, my engineering Spidey-sense tells me that within 10% is fine. So maybe instead of giving me the most optimal thing, can I give you an algorithm that's guaranteed to be within a certain distance from the optimal thing?

Usually, we're looking for constant factor approximations which have low constant, or maybe even have to do for worse if such things don't exist. OK, so that's approximation algorithms. Can we get close to an optimal solution in polynomial time? OK.

And then the last way we could change things in, especially future classes, though sometimes they talk about this in 046 as well, is we could change the model of computation. We could basically change something about our computer to be put in some other weird paradigm of solving problems with more power essentially, or you're in a situation where there's less power. OK, so change in the model of computation.

So what we've been talking to you in terms of model of computation is our word-RAM-- word-RAM. And that essentially says I can do arithmetic operations, and I can look up stuff in my memory in constant time. And but if I allocate a certain amount, I have to pay that amount and that kind of thing. So that's this word-RAM model.

But in actuality, all of your computers, it's a lot easier for me to figure-- to find and read memory that's on my CPU in a register than it is for me to go out to the hard disk, ask this-- well, in my day, it used to be this movable mechanical head that had to go and scan over a bit on a CD-ROM drive and actually read what that thing was.

So we can add complexity to our model to better account for the costs of operations on my machine. One of those models is called the cache model-- cache model. It's basically a hierarchy of memory. I have my registers on board my CPU. I have maybe an L1 cache that's close to my CPU. Then I have another set of caches and another set of caches maybe out to RAM.

And reading from a hard disk, a solid state drive of some kind, that's the slowest thing to access. And I can put a cost associated with each of those things. And instead of having to-- having all of our operations be said to be constant, the constants are actually different, and I have to pay for that difference. And so that's extending our model to be a little bit more realistic to our machine.

Another one is we have computers right now that operate in classical physics, that exploit things in classical physics. But in actuality, our world allows for even more complicated types of operations, like quantum operations, where you're exploiting entanglement and superposition of different atoms to potentially get operations that I can act on my data that are actually provably stronger than the classical models in some sense.

So this is a huge reason why there's a lot of work being done in, say, lots of industry research facilities in figuring out these models. Because maybe if you can make a big enough quantum computer, you can break encryption and stuff in polynomial time. And that's something that maybe the NSA is interested in. And I'm not going to go into that. But, you know.

I mean, some people-- you look at artificial intelligence and things like discussions around artificial intelligence, my brain might be doing things that a classical computer cannot. It could be using quantum superposition in some way. And our computers that are in your phone and your laptop and things like that aren't exploiting those operations, so how could we ever get intelligence, because in some sense, our brains are more powerful.

And so a lot of what AI should be looking into is, what is the actual model of computation of our brains that can give us the power to have sentience? OK, so that's kind of quantum computing. I don't know much about it actually. And then there's things like, maybe I have more than one CPU. I mean, most computers-- all the computers you have, even the ones in your phone, probably have multiple cores. In a sense, you have lots of CPUs running in parallel.

So this is like par-- there's one R in parallel? Parallel computing basically says, it's cheap for me to make another computer potentially. If I have two computers running on the same problem, maybe I can get a two-fold speed-up on my-- on the time in which it takes to solve my problem.

Now, suppose I had then 100 CPUs running on a machine. Maybe I can get 100-fold speed-up. And actually, in real life, 100-fold speed-up makes a difference. It's, am I waiting for this for 10 minutes? Or am I waiting for this for 1,000 minutes? That's, like, all day. I don't want to do that. Maybe it's on weeks. I don't even remember.

But parallel computing, if I can get a 100-fold speed-up, that might be a huge win. But for some problems, it's not possible-- if I have k CPUs, can I get a k-factor speed-up? It's not always possible to do. And so parallel computing is another paradigm in which there's a lot of interesting theory going on.

There's a lot of complications there, because there are a couple different models. You can have multicore set-up, where you have a lot of computers that are accessing the same bank of memory. And then you don't want them all to be reading and writing from them at different times, because you don't necessarily know what their state is, and you get these collisions, which are something that you really have to think about in this world.

Or you have situations where maybe I have a bunch of nano-flies or something that are going around, and they have very small computer brains themselves. But they can talk to each other and pass information to each other, but they don't have access to one central network repository of information. That's what we call a distributed parallel system, where all of the CPUs that you have can interact with each other maybe locally, but don't have access to the same memory system. So they have to work together to learn information about the system.

OK, so that's a brief overview of the different directions this class, 6.006, and theory in general, could lead you-- into a huge array of different branches theory and different problems that you could address with different types of computers. So I know this is a very high-level lecture and maybe less-applied than some of you might like. But I hope this gives you a good understanding of the directions you can go after this class that I think are really excited in terms of how to solve problems computationally. So with that, I'd like to end there.