[SQUEAKING]

[RUSTLING]

[CLICKING]

**JASON KU:** Good morning, everybody.

**STUDENT:** Morning--

**JASON KU:** My name's Jason Ku. I'm going to be teaching this class in Introduction to Algorithms with two other instructors here-- faculty in the department-- Eric Demaine and Justin Solomon. They're excellent people, and so they will be working on teaching this class with me. I will be teaching the first lecture, and we'll have each of them teach one of the next two lectures, and then we'll go from there.

This is Intro to Algorithms. OK, so we're going to start talking about this course content now. What is this course about? It's about algorithms-- introduction to algorithms. Really what the course is about is teaching you to solve computational problems. But it's more than that. It's not just about teaching you to solve computational problems.

Goal 1-- solve computational problems. But it's more than that. It's also about communicating those solutions to others and being able to communicate that your way of solving the problem is correct and efficient. So it's about two more things-- prove correctness, argue efficiency, and in general, it's about communication-- I can't spell, by the way-- communication of these ideas.

And you'll find that, over the course of this class, you'll be doing a lot more writing than you do in a lot of your other courses. It really should maybe be a CI kind of class, because you'll be doing a lot more writing than you will be coding, for sure. Of course, solving the computational problem is important, but really, the thing that you're getting out of this class and other theory classes that you're not getting in other classes in this department is that we really concentrate on being able to prove that the things you're doing are correct and better than other things, and being able to communicate those ideas to others, and not just to a computer-- to other people, convince them that it's correct.

OK, so that's what this class is about. So what do I mean when I say solve a computational problem? What is a problem? What is an algorithm? People make fun of me because I start with this question, but anyone want to answer that question? No? What's a problem, computationally? No? OK, so it's not such a stupid question. Yeah?

**STUDENT:** [INAUDIBLE]

**JASON KU:** Something you want to compute-- OK, yes, that's true. Right. But a little bit more abstractly, what I'm going to think of a computational problem being-- and this is where your prerequisite in discrete mathematics should come in-- a problem is-- you've got a set of inputs. Maybe I have one, two, three, four, five possible inputs I could have to my algorithm.

Then I have a space of outputs. I don't know. Maybe I have more of them than I do inputs, but these are the possible outputs to my problem. And what a problem is is a binary relation between these inputs and outputs. Essentially, for each input, I specify which of these outputs is correct. It doesn't necessarily have to be one. If I say, give me the index in an array containing the value 5, there could be multiple 5's in that array, and so any of those indices would be correct.

So maybe this guy maps to that output, and maybe this guy maps to-- I don't know-- two or three outputs. This input goes to one, two-- I don't know. There's some kind of mapping here. These edges represent a binary relation, and it's kind of a graph, a bipartite graph between these inputs and outputs. And these are specifying which of these outputs are correct for these inputs.

That's really the formal definition of what a problem is. Now, generally, if I have a problem-- a computational problem, I'm not going to specify the problem to you by saying, OK, for input 1, the correct answer is 0, and for input 2, the correct answer's 3, and so on and so forth. That would take forever, right?

Usually what we do when defining a problem is specify some kind of predicate, saying that, oh, we can check-- if I give you an input and an output, I can check whether that output is correct or not. That's usually how we define a problem is, if I am checking for whether this index contains a 5, I can just go to that array, look at index 5, and-- or the index you gave me, and see if it equals 5. So usually, we're putting it in terms of predicates because, in general, we don't really want to talk about small instances of problems.

So let's say I had the problem of, among the students in this classroom, do any pair of you have the same birthday? All right, well, probably, if there's more than 365 of you, the answer is yes. Right? By what? Pigeonhole principle-- two of you must have the same birthday.

So let's generalize it a little bit, say that-- I don't know-- I need a bigger space of birthdays for this question to be interesting. Maybe I tack on the year. Maybe I tack on the hour that you were born. And that's a bigger space of inputs, and I wouldn't necessarily expect that two of you would be born in the same year on the same day in the same hour.

That would be a little less likely. In fact, as long as that space is larger than something like the square of the number of you, then I'm less likely than even to have a pair of you. That's a birthday problem you may have seen in 042, potentially. But in general, I don't-- I'm not going to mess with probability so much here. I want a deterministic algorithm, right away of checking whether two of you have the same birth time, let's say.

OK, so in general, in this class, we're not going to concentrate on inputs such as, is there a pair of you in this class that have the same birthday? That's kind of boring. I could do a lot of different things, but what we do in this class-- this is for a fixed classroom of you. I want to make algorithms that are general to any classroom-- to go to your recitation.

I want an algorithm that will apply to your recitation. I want an algorithm that not only applies to this classroom, but also the machine learning class before you. I want an algorithm that can change its-- it can accept an arbitrarily sized input. Here we have a class of maybe 300, 400 students, but I want my algorithm to work for a billion students. Maybe I'm trying to check if there's a match of something in the Facebook database or something like that.

So in general, we are looking for general problems that have arbitrarily sized inputs. So these inputs could grow very large, but we want kind of a fixed size algorithm to solve those problems. So what is an algorithm, then?

I really can't spell-- told you. I didn't lie to you. So an algorithm is a little different than a problem. A problem specification-- I can tell you what this graph looks like. An algorithm is really-- I don't know what the outputs are. I don't know what these edges are. But I want a fixed size machine or procedure that, if I give it an input, it will generate an output. And if it generates an output, it better be one of these correct outputs.

So if I have an algorithm that takes in this input, I really want it to output this output, or else it's not a correct algorithm. Similarly, for this one, it could output any of these three outputs, but if it outputs this guy for this input, that would not be a correct algorithm. And so generally, what we want is an algorithm is a function. It takes inputs to outputs.

An algorithm is some kind of function that takes these inputs, maps it to a single output, and that output better be correct based on our problem. So that's what our algorithm is. It solves the problem if it returns a correct output for every problem input that is in our domain.

Does anyone have a possible algorithm for checking whether any two of you have the same birth time, as specified before? I'm going to let someone else have a try. Sure.

**STUDENT:** Just ask everyone one by one, and every time [INAUDIBLE]

**JASON KU:** Great-- so what your colleague has said is a great algorithm. Essentially, what it's going to do is I'm going to put you guys in some order, I'm going to give you each of you a number, one through however many number of students there are in this class. And I'm going to interview you one by one. I'm going to say, what's your birthday?

And I'm going to write it down. I'm going to put it in some kind of record. And then, as I keep interviewing you, I'm going to find out your birthday. I'm going to check the record. I'm going to look through all the birthdays in the record. If I find a match, then I return, yay-- I found a pair-- and I can stop. Otherwise, if I get through the record list, I don't-- and I don't find a match, I just stick you at the end of the record-- I add you to the record, and then I move on to the next person. I keep doing this.

OK, so that's a proposed algorithm for this birthday problem. For birthday problem, what's the algorithm here? Maintain a record. Interview students in some order. And what does interviewing a student mean? It means two things. It means check if birthday in record. And if it is, return a pair.

So return pair. Otherwise, add a new student to record. And then, at the very end, if I go through everybody and I haven't found a match yet, I'm going to return that there is none. OK, so that's a statement of an algorithm. That's kind of the level of description that we'll be looking for you in the three parts of this-- theory questions that we ask you on your problem sets.

It's a verbal description in words that-- it's maybe not enough for a computer to know what to do, but if you said this algorithm to any of your friends in this class, right they would at least understand what it is that you're doing. Yeah?

**STUDENT:** Does an algorithm have to be a pure function in a mathematical sense?

**JASON KU:** Does an algorithm have to be a pure function in a mathematical sense? As in it needs to map to a single output?

**STUDENT:** As in it can't modify some external state. It can't take in state and it can't do I/O.

**JASON KU:** So we're talking about kind of a functional programming definition of a function. I am talking about the mathematical-- I have a binary relation, and this thing has an output for every input, and there is exactly one output to every input. That's the mathematical definition of function that I'm using for when I'm defining an algorithm. Yeah?

**STUDENT:** Basically, is an algorithm like a plan?

**JASON KU:** Yeah. An algorithm's a procedure that somehow-- I can do whatever I want, but I have to take one of these inputs and I have to produce an output. And at the end, it better be correct. So it's just a procedure. You can think of it as like a recipe. It's just some kind of procedure.

It's a sequence of things that you should do, and then, at the end, you will return an output.

S here's a possible algorithm for solving this birthday problem. Now, I've given you-- what I argue to you, or I'm asserting to you, is a solution to this birthday problem. And maybe you guys agree with me, and maybe some of you don't.

So how do I convince you that this is correct? If I was just running this algorithm on, say, the four students in the front row here, I could argue it pretty well to you. I could assign these for people birthdays in various combinations of either their-- none of them have the same birthday, some two of them have the same birthday.

I could try all possibilities, and I could go through lots of different possibilities and I need to check that this algorithm returns the right answer in all such cases. But when I have-- I don't know-- 300 of you, that's going to be a little bit more difficult to argue. And so if I want to argue something is correct in-- I want to prove something to you for some large value, what kind of technique do I use to prove such things? Yeah?

Induction, right? And in general, what we do in this class, what we do is-- as a computer scientist is we write a constant sized piece of code that can take on any arbitrarily large size input. If the input can be arbitrarily large, but our code is small, then that code needs to loop, or recurse, or repeat some of these lines of code in order to just read that output.

And so that's another way you can arrive at this conclusion, that we're going to probably need to use recursion, induction. And that's part of the reason why we ask you to take a course on proofs, and inductive reasoning, and discrete mathematics before this class.

OK, so how do we prove that this thing is correct? We got to use induction. So how can we set up this induction? What do I need for an inductive proof? Sure.

**STUDENT:** [INAUDIBLE]

**JASON KU:** Base case-- we need a base case. We need some kind of a predicate. Yeah, but we need some kind of statement of a hypothesis of something that should be maintained. And then we need to have an inductive step, which basically says I take a small value of this thing, I use the inductive hypothesis, and I argue it for a larger value of my well-ordered set that I'm inducting over.

For this algorithm, if we're going to try to prove correctness, what I'm going to do is I'm going to-- what do I want to prove for this thing? That, at the end of interviewing all of you, that my algorithm has either already-- it has returned with a pair that match, or if we're in a case where there wasn't a pair somewhere in my set, that it returned none. Right?

That would be correct. So how can I generalize that concept to make it something I can induct on? What I'm going to do is I'm going to say-- let's say, after I've interviewed the first K students, if there was a match in those first K students, I want to be sure that I've returned a pair-- because if, after I interview all of you, I've maintained that property, then I'll be sure, at the end of the process, I will have returned a pair, if one exists.

So here's going to be my inductive hypothesis. If first K students contain a match, algorithm returns a match before interviewing, say, student K plus 1. So that's going to be my inductive hypothesis.

Now, if there's n students in this class, and at the end of my thing, I'm trying to interview a student n plus 1-- oh, student n plus 1's not there. If I have maintained this, then, if I replace K with n, then I will have returned a match before interviewing the last student-- when I have no more students left. And then this algorithm returns none, as it should.

OK, so this inductive hypothesis sets up a nice variable to induct on. This K I can have increasing, up to n, starting at some base case. So what's my base case here? My base case is-- the easiest thing I can do-- sure-- 2? That's an easy thing I could do. I could check those possibilities, but there's an even easier base case. Yeah? There's an even easier base case than 1.

**STUDENT:** 0--

**JASON KU:** 0, right? After interviewing 0 students, I haven't done any work, right? Certainly, the first 0 can't have a match. This inductive hypothesis this is true just because this initial predicate is false. So I can say, base case 0-- check. Definitely, this predicate holds for that.

OK. Now we got to go for the meat of this thing. Assume the inductive hypothesis true for K equals, say, some K prime. And we're considering K prime plus 1.

Then we have two cases. One of the nice things about abduction is that it isolates our problem to not consider everything all at once, but break it down into a smaller interface so I can do less work at each step. So there are two cases. Either the first K already had a match-- in which case, by our inductive hypothesis, we've already returned a correct answer.

The other case is the-- it doesn't have a match, and we interview the K plus 1th student-- the K prime plus 1th student. If there is a match in the first K prime plus 1 students, then it will include K plus-- the student K prime plus 1, because otherwise, there would have been a match in the things before it. So there are two cases.

If K contains match, K prime. If first K contains match-- already returned by induction. Else, if K prime plus 1 student's contains match, the algorithm checks all of the possibilities-- K prime checks against all students, essentially by brute force. It's a case analysis. I check all of the possibilities.

Check if birthday is in record-- I haven't told you how to do that yet, but if I'm able to do that, I'm going to check if it's in the record. If it's in the record, then there will be a match, and I can return it. Otherwise, I have-- re-establish the inductive hypothesis for the K prime plus 1 students. Does that makes sense, guys? Yeah.

OK, so that's how we prove correctness. This is a little bit more formal than we would ask you to do in this class all the time, but it's definitely sufficient for the levels of arguments that we will ask you to do. The bar that we're usually trying to set is, if you communicated to someone else taking this class what your algorithm was, they would be able to code it up and tell a stupid computer how to do that thing.

Any questions on induction? You're going to be using it throughout this class, and so if you are unfamiliar with this line of argument, then you should go review some of that. That would be good. OK, so that's correctness, being able to communicate that the problem-- the algorithm we stated was correct. Now we want to argue that it's efficient. What does efficiency mean?

Efficiency just means not only how fast does this algorithm run, but how fast does it compare to other possible ways of approaching this problem? So how could we measure how fast an algorithm runs? This is kind of a silly question. Yeah?

**STUDENT:** [INAUDIBLE]

**JASON KU:** Yeah. Well, just record the time it takes for a computer to do this thing. Now, there's a problem with just coding up an algorithm, telling a computer what to do, and timing how long it takes. Why? Yeah?

**STUDENT:** [INAUDIBLE]

**JASON KU:** It would depend on the size of your data set. OK, we expect that, but there's a bigger problem there. Yeah?

**STUDENT:** [INAUDIBLE]

**JASON KU:** It depends on the strength of your computer. So I would expect that, if I had a watch calculator and I programmed it to do something, that might take a lot longer to solve a problem than if I asked IBM's research computer to solve the same problem using the same algorithm, even with the same code, because its underlying operations are much faster. How it runs is much faster.

So I don't want to count how long it would take on a real machine. I want to abstract the time it takes the machine to do stuff out of the picture. What I want to say is, let's assume that each kind of fundamental operation that the computer can do takes some fixed amount of time.

How many of those kinds of fixed operations does the algorithm need to perform to be able to solve this problem? So here we don't measure time. Instead, count fundamental operations. OK? We'll get to what some of those fundamental operations are in a second, but the idea is we want a measure of how well an algorithm performs, not necessarily an implementation of that algorithm-- kind of an abstract notion of how well this algorithm does.

And so what we're going to use to measure time or efficiency is something called asymptotic analysis. Anyone here understand what asymptotic analysis is? Probably, since it's in both of your prerequisites, I think-- but we will go through a formal definition of asymptotic notation in recitation tomorrow, and you'll get a lot of practice in comparing functions using an asymptotic analysis.

But just to give you an idea, the idea here is we don't measure time. We instead measure ops. And like your colleague over here was saying before, we expect performance-- I'm going to use performance, instead of time here-- we expect that to depend on size of our input.

If we're trying to run an algorithm to find a birthday in this section, we expect the algorithm to run in a shorter amount of time than if I were to run the algorithm on all of you. So we expect it to perform differently, depending on the size of the input, and how differently is how we measure performance relative to that input.

Usually we use n as a variable for what the size of our input is, but that's not always the case. So for example, if we have an array that I give you-- an n-by-n array, that-- we're going to say n, but what's the size of our input? How much information do I need to convey to you to give you that information?

It's n squared. So that's the size of our input in that context. Or if I give you a graph, it's usually the number of vertices plus the number of edges. That's how big-- how much space I would need to convey to you that graph, that information.

We compare how fast an algorithm is with respect to the size of the input. We'll use the asymptotic notation. We have big O notation, which corresponds to upper bounds. We will have omega, which corresponds to lower bounds. And we have theta, which corresponds to both. This thing is tight. It is bounded from above and below by a function of this form.

We have a couple of common functions that relate an algorithm's input size to its performance, some things that we saw all the time. Can anyone give me some of those?

**STUDENT:** [INAUDIBLE]

**JASON KU:** Say again.

**STUDENT:** [INAUDIBLE]

**JASON KU:** Sorry. Sorry. I'm not asking this question well, but has anyone heard of a linear algorithm-- a linear time algorithm? That's basically saying that the running time of my algorithm-- performance of my algorithm is linear with respect to the size of my input. Right? Yeah?

**STUDENT:** [INAUDIBLE]

**JASON KU:** Say again.

**STUDENT:** Like putting something in a list--

**JASON KU:** Like putting something in a list-- OK. There's a lot behind that question that we'll go into later this week. But that's an example of, if I do it in a silly way, I stick something in the middle of a list and I have to move everything. That's an operation that could take linear time.

So linear time is a type of function. We've got a number of these. I'm going to start with this one. Does anyone know this one is? Constant time-- basically, no matter how I change the input, the amount of time this running time-- the performance of my algorithm takes, it doesn't really depend on that.

The next one up is something like this. This is logarithmic time. We have data n, which is linear, and log n. Sometimes we call this log linear, but we usually just say n log n. We have a quadratic running time. In general, if I have a constant power up here, it's n to the c for some constant. This is what we call polynomial time, as long as c is some constant.

And this right here is what we mean by efficient, in this class, usually. In other classes, when you have big data sets, maybe this is efficient. But in this class, generally what we mean is polynomial. And as you get down this thing, things are more and more efficient.

There's one class I'm going to talk to you about over here, which is something like-- let's do this-- 2 to the theta of n, exponential time. This is some constant to a function of n that's, let's say, super linear, that's going to be pretty bad. Why is it pretty bad?

If I were to plot some of these things as a function of n-- let's say I plot values of up to 1,000 on my n scale here. What does constant look like? Maybe this is 1,000 up here. What does a constant look like? Looks like a line-- it looks like a line over here somewhere. It could be as high as I want, but eventually, anything that's an increasing function will get bigger than this.

And on this scale, if I use log base 2 or some reasonable small constant, what does log look like? Well, let's do an easier one. What does linear look like? Yeah, this-- that's what I saw what a lot of you doing. That's linear. That's the kind of base that we're comparing everything against.

What does log look like? Like this-- OK, but at this scale, really, it's much closer to constant than linear. And actually, as n gets much, much larger this almost looks like a straight line. It almost looks like a constant. So log is almost just as good as constant.

What does exponential look like? It's the exact inverse of this thing. It's almost an exact straight line going up. So this is crap. This is really good. Almost anything in this region over here is better right. At least I'm gaining something. I'm able to not go up too high relative to my input size. So quadratic-- I don't know-- is something like this, and n log n is something like this.

n log n, after a long time, really starts just looking linear with a constant multiplied in front of it. OK, so these things good, that thing bad-- OK? That's what that's trying to convey. All right, so how do we measure these things if I don't know what my fundamental operations are that my computer can use?

So we need to define some kind of model of computation for what our computer is allowed to do in constant time, in a fixed amount of time. In general, what we use in this class is a machine called a word RAM, which we use for its theoretical brevity. Word RAM is kind of a loaded term. What do these things mean? Does someone know what RAM means?

**STUDENT:** [INAUDIBLE]

**JASON KU:** Random access memory-- it means that I can randomly access different places in memory in constant time. That's the assumption of random access memory. Basically, what our model of a computer is you have memory, which is essentially just a string of bits. It's just a bunch of 1's and 0's.

And we have a computer, like a CPU, which is really small. It can basically hold a small amount of information, but it can change that information. It can operate on that information, and it also has instructions to randomly access different places in memory, bring it into the CPU, act on it, and read it back.

Does that makes sense? But in general, we don't have an address for every bit in memory, every 0 and 1 in memory. Does anyone know how modern computers are addressed? Yeah?

**STUDENT:** [INAUDIBLE]

**JASON KU:** OK, so we're going to get there. Actually, what a modern computer is addressed in is bytes, collections of 8 bits. So there's an address I have for every 8 bits in memory-- consecutive 8 bits in memory. And so if I want to pull something in into the CPU, I give it an address. It'll take some chunk, and bring it into the CPU, operate on it, and spit it back.

How big is that chunk? This goes to the answer that you were asking, which-- or saying, which is it's some sequence of some fixed number of bits, which we call a word. A word is how big of a chunk that the CPU can take in from memory at a time and operate on.

In your computers, how big is that word size? 64 bits-- that's how much I can operate on at a time. When I was growing up, when I was your age, my word size was 32 bits. And that actually was a problem for my computer, because in order for me to be able to read to address in memory, I need to be able to store that address in my CPU, in a word.

But if I have 32 bits, how many different addresses can I address? I have a limitation on the memory addresses I can address, right? So how many different memory addresses can I address with 32 bits? 2 to the 32, right? That makes sense.

Well, if you do that calculation out, how big of a hard disk can I have to access? It's about 4 gigabytes. So in my day, all the hard drives were limited to being partitioned-- even if you had a bigger than 4 gigabyte hard drive, I had to partition it into these 4 gigabyte chunks, which the computer could then read onto. That was very limiting, actually. That's a restriction.

With 64 bits, what's my limitation on memory that I can address-- byte addressable? Turns out to be something like 20 exabytes-- to put this in context, all data that Google stores on their servers, on all drives throughout the world-- it's about 10. So we're not going to run out of this limitation very soon.

So what do we got we've got a CPU. It can address memory. What are the operations I can do in this CPU? Generally, I have binary operations. I can compare to words in memory, and I can either do integer arithmetic, logical operations, bitwise operations-- but we're not going to use those so much in this class. And I can write and write from an address in memory, a word in constant time.

Those are the operations that I have available to me on most CPUs. Some CPUs give you a little bit more power, but this is generally what we analyze algorithms with respect to. OK? But you'll notice that my CPU is only built to operate on a constant amount of information at once-- generally, two words in memory.

An operation produces a third one, and I spit it out. It takes a constant amount of time to operate on a constant amount of memory. If I want to operate on a linear amount of memory-- n things-- how long is that going to take? If I just want to read everything in that thing, it's going to take me linear time, because I have to read every part of that thing.

OK, so in general, what we're going to do for the first half of this class mostly-- first eight lectures, anyway-- is talk about data structures. And it's going to be concerned about not operating on constant amount of data at a time, like our CPU is doing, but instead, what it's going to do is operate on-- store a large amount of data and support different operations on that data.

So if I had a record that I want to maintain to store those birthdays that we had before, I might use something like a static array, which you guys maybe are not familiar with, if you have been working in Python is your only programming language. Python has a lot of really interesting data structures, like a list, and a set, and a dictionary, and all these kinds of things that are actually not in this model. There's actually a lot of code between you and the computer, and it's not always clear how much time that interface is taking.

And so what we're going to do starting on Thursday is talk about ways of storing a non-constant amount of information to make operations on that information faster. So just before you go, I just want to give you a quick overview of the class. To solve an algorithms class-- an algorithm problem in this class, we essentially have two different strategies. We can either reduced to using the solution to a problem we know how to solve, or we can design our own algorithm, which is going to be recursive in nature.

We're going to either put stuff in the data structure and solve a sorting problem, or search in a graph. And then, to design a recursive algorithm, we have various design paradigms. This is all in your notes, but this is essentially the structure of the class. We're going to spend quiz 1, the first eight lectures on data structures and sorting.

Second quiz will be on shortest paths, algorithms, and graphs, and then the last one will be on dynamic programming. OK, that's the end of the first lecture. Thanks for coming.