PROFESSOR: Good morning, everybody. We're on the home stretch here. Only two more lectures, no more recitations, and a small matter of a final exam. I should remind you the final exam is on Monday of finals week. And it will be a two-hour exam, not a three-hour exam.

Same kind of format as on the first two quizzes. Open book, open notes. The difference is that we'll be giving you some code, which we'll be posting today, which we're going to ask you to study in preparation for the exam. And then we'll ask you some questions about that code. For example, ask you to make a small modification to it on the exam.

And in fact, today's lecture, just as a special incentive to pay attention, will be about things related to the code we'll be posting for you to study for the exam. It covers one final class of simulation that we haven't looked at at all this term, but I think is important to at least know a little bit about it. And those are queueing network simulations.

From the time of conception until death, human beings find themselves all too often waiting for things-- waiting for events, waiting for conditions, waiting for TAs, et cetera. Queuing networks give us a formal way to study systems in which waiting plays a fundamental role. Some typical examples might be in a supermarket, waiting to check out. You get in a queue, wait your turn.

Computers all the time -- you may wonder sometimes why when you hit your computer, and ask it to do something, sometimes it's fast, sometimes it's slow. Well, it's because there are tasks running in the background that are ahead of your task in the queue, and you're waiting for them to finish. Computer networks, waiting for arrivals and things like that. If you look at the internal structure of the internet, as, say, an email hops from hither to yon between two sites, along the way, it enters queues and waits for its turn to get forwarded to the next hop.

A few of you have experienced this when you go to office hours, where there's a wait for a TA, and you get in a queue. Public transit, analysis of highway systems. All of these things depend upon understanding queues to understand how they work.

You might wonder why we have queues. The answer is pretty simple, in that it typically makes economic sense. As we'll see in our analysis, when resources are limited-- particularly, say, servers-- it makes economic sense to have jobs line up for service.

So if you think about a supermarket, ideally, when you go to check out, you'd like there to be no line. But think about how many cash registers they would need to ensure that all the time when someone was ready, there was no line. And the problem is if they did that, they would have idle servers. And so they would not get very good resource utilization of the people they would be hiring to do the checkout.

In designing a system with queues-- and that's what the simulations help us do-- we're aiming for a balance between, essentially, customer service and resource utilization. The organization paying for the resources would like them to be working a 100% of the time at full capacity, so there's no wasted expense.

On the other hand, it's easy to show in any reasonable system that if you have a 100% resource utilization, you're going to be providing terrible customer service. There is always a need for excess capacity in a complex system in order to provide adequate service most of the time. And that's because, as we'll see, the demand for service is probabilistic and unpredictable. And in order to absorb bursts of activity, you need excess capacity, because a lot of the time, you don't see those bursts.

All right. All that by way of introduction. If we think about how to model a system with queues, they're are all modeled essentially the same way.

You start with some jobs, work to be done. Those jobs enter a queue. They wait around for a while. They leave the queue one at a time and enter a server. And then they depart.

So that's what it looks like. And of course, it can be much more complicated than this. You can have multiple streams of jobs. You can have multiple queues. You can have multiple servers. But when we build an analysis of it, when we build a simulation, we typically break it down into components that look exactly like this and use that as the key.

As we do this, we have to look at a number of different things. We start with the arrival process. How do the jobs appear? And there are lots of questions we can ask. Do they arrive singly, or in groups?

If they arrive in groups, we say that it's a batch process. So if you think about a service for, say, processing students at MIT into the database, they typically arrive in a big batch in September, rather than straggling in one at a time. We have to ask, related to that, is how arrivals are distributed in time.

Assume for the moment that arrivals are one at a time-- for example, automobiles entering a highway. Do they come uniformly, at evenly-spaced intervals? Do they come at random times? If random, is there a time of day when we see most of them?

We typically model all of that in what's called an inter-arrival time distribution. The simplest process is where these things do arrive at regular intervals, constant intervals. We hardly ever see this in the real world.

Most of the time, we see something that looks much like our old friend, the Poisson process. Things arrive at a random interval. But they are typically exponentially distributed. We've seen this before. Ok. It is, as you will recall, memoryless. It is, as you will recall, described by a single parameter, the average arrival rate.

We can look at it just briefly to see what it looks like, just to remind ourselves, really. So I've just a small piece of code, where I've given the mean arrival. And then I'm going to just have 2000 things arrive at Poisson intervals.

Why is it not running? I cleverly rebooted before I got here, so it shouldn't have had any problem. There it is. All right, so that really should be-- anyone remember what this calls? Right.

And so we'll see that we have exponential arrival times, a rapid decay, only a small tail out at the right in this picture, and a big cluster near the mean. We can look at this in a different way if you look at the scatterplot. And you'll notice that we've given it a mean of 60, the dark blue line.

But we seem to have an awful lot of points below the mean compared to a very much smaller number of points above the mean. Why is that, in fact, inevitable in this situation? Yeah?

AUDIENCE: Because there's less space for them to--

PROFESSOR: Because?

AUDIENCE: There's just less space for them. We can go from 60--

PROFESSOR: Well, there's less space. And the reason that there's less space is we don't allow things to arrive yesterday, in negative time. So the smallest inter-arrival time is just epsilon more than 0. So the time between jobs can never be less than 0.

And so if we want-- and we're going to have a few way out here that take a lot longer, because that's the nature of an exponential. So we're going to have to have a bunch down here to counterbalance the few way up here. We can only get 60 below the mean, in this case. And you can see we here have over 500 above the mean.

So it's going to always look like that, which is why the histogram looks something like this. So you'll observe that as you look at these things. All right, that's the arrival process.

We also have to look at the service mechanism. How does the server work? Again, we can ask some question like, how long will it take to provide service? And this is called the service time distribution.

This depends on two things-- the amount of work the job to be served entails, and the speed of the server. So, again, you've experienced this, no doubt, at the supermarket, where if you get in the wrong line, where there's a very slow person running the cash register, even though the jobs are small, it takes forever, because there's a slow server. Or alternatively, you've gotten behind a line, not noticing that the person in front of you has two shopping carts full of junk. And so even though the server is fast, it'll take a long time to service that job.

The number of servers certainly matters. An important question is the number of queues. In particular, does each server have its own queue, which is what you see at the supermarket? Or is there a single queue for many servers, typically what you see if you are checking in, say, in coach class at the airport, or lining up for security at the airport?

Which of those is better, do you think? Which provides, on average, better service, multiple queues or a single queue? Single queue, right, is much better. In fact, it's easy to construct a proof why it's better. Which is why we now see them, typically, at places like banks and airports.

Didn't used to be. Initially, people didn't understand this was better, and they would have multiple queues. Why don't we always have a single queue? Because we may not have enough space. So you can, again, imagine at the supermarket there'd be no place to put the single queue for all of the cash registers, maybe.

And another question we have to ask is whether or not we allow preemption. In some queuing systems, a server can stop processing a customer to deal with another emergency customer. In others, once a job is started, it has to run to completion. And, again, you get very different performance, depending upon whether you allow preemption or not.

An important part of the architecture is, of course, the characteristics of the queue itself. The fundamental question is the policy. How from the set of customers waiting for service do we choose the customer to be served next?

So a very common policy is FIFO. It stands for First In First Out. That's the one we see most of the time in a lot of physical situations. Whoever gets in line first get served first.

There are some situations where you do Last In First Out. Whoever's arrived most recently gets served first. A very popular one in many computing applications is what's called SRPT, which is short for Shortest Remaining Processing Time. Take whichever job you can complete fastest.

So these are the three most popular policies. You could also imagine a random policy, where you just chose some random member of the queue. Then there are lots of other kinds of issues, which I won't go into in detail.

The choice of queuing discipline, as we'll see when we look at the code, can have a major effect on performance. In particular, it can be used to reduce congestion. We'll often see in practice this one. Now, why might that be a very useful practical method?

Well, It's got one obvious feature. If the available space for the queue is small, by taking the ones you can finish fastest, you reduce the average number of elements in the queue. Because you get to process more customers by taking the customers that you can process quickly. And so on average, there will be fewer customers in the queue. And so it reduces congestion.

As we'll see, it also improves expected service time. So it is an excellent method. When we talk about queuing networks, what we're going to see-- and I hope I've implied this already-- is that probability has to play a fundamental role in understanding them, because there's randomness. There's randomness in the inter-arrival time, and there's typically randomness in how long each job will take to process.

What are the questions we might want to ask about a queuing system to see whether we've designed it properly? An important question is the average waiting time. On average, how long does a job wait between the time it enters the queue and the time it leaves the queue? Now, imagine that you're waiting for the MIT Shuttle Bus. How long you wait before you get on the bus, on average?

You can also ask, once the job starts, how long to complete? We can also ask, is the waiting time bounded? So this is a different question than how long you should expect to wait on average. It's, what's the probability of your not having to wait longer than some upper bound?

So you can imagine that if you were working for some company that, for example, sold products over the telephone, you might want to put a bound on how long a customer would wait before their call got picked up. And you'd do some analysis, saying if people have to wait longer than three minutes, they hang up the phone. And so we'll have enough servers that it's only rare that someone has to wait more than three minutes. It's a different question than how long people have to wait on average.

We've already talked about the average queue length. And by analogy, we can also ask about the bound. on queue length. You can imagine if you're designing a router for the internet, you need to know how much memory to allocate for the queues of jobs so that you don't run out. It's not infinite, so there's an upper bound in how long you'll let the queues grow.

And finally, along a different dimension is something we've already talked about, server utilization. What is the expected utilization of each server, in the sense of, what is the expected amount of time it will be fully occupied? Remember, servers cost money, so keeping them busy is important.

If we can assign relative cost values to each of these things, we can then build a system that will let us do an analysis and build an optimal system-- decide how big the queues should be, how many servers, how fast they should be, do the analysis, and design a system that meets some optimization criteria. So now, we need to investigate these things.

Two basic methods, as we've discussed before, are analytic-- and, indeed, you can get books on analytic queuing theory, really thick, hard-to-read books, I might add, that give you complicated formulas for deriving these kinds of things. In practice, they're hardly ever used, because most systems are too complex to actually model analytically. So, in fact, today, most queuing network systems are analyzed using simulations, the kind of discrete event simulations we've looked at earlier in the term.

All right. Now, I want to look at a particular example, and see how we can build a simulation that will let us answer these kinds of questions about a particular example. The example is a -- and again, this is the one that you'll be asked to look at -- is a simplification of the MIT Shuttle Bus system. So as I'm sure you all know, it runs in a loop, picking up students, and dropping them off at some number of stops.

For legal reasons, each bus has a maximum capacity-- often exceeded I'm sure, in practice, but there is officially a maximum number of people allowed on a bus. And it takes a certain amount of time to go around the loop. The obvious question to ask is, how long should a student expect to have to wait for a bus at a particular stop to get picked up? And what should MIT do to, depending upon whether it's feeling benevolent or not, minimize or maximize the waiting time for the bus?

All right. Most of the code we're going to look at now is on your handout. I should say I've deleted the obvious parts so I could fit it on the handout. However, we'll be posting the code on the web today, a .py file that you'll be able to study, and more importantly, probably run it.

All right, let's start. So, as usual in designing a piece of code, I began with some useful classes. You won't be surprised that the first class I looked at was class job. So what is a job? A job is -- depending upon the mean arrival and the mean work involved. And I've chosen to model arrivals as exponential. And I've chosen to model work as a Gaussian.

Nothing magical about this. And in fact, I played with various other models. What would happen if the work were uniformly distributed or exponentially distributed? And, of course, you get different answers. I said next two attributes. There is actually only one attribute here. And this is going to keep track of when a job enters the queue, so that we can tell how long it had to wait. And then there's the inter-arrival, the work, the queue, the queue time, et cetera.

And then I'm going to model a passenger as a job. You'll note this is kind of a boring class. All I've done is say, pass. Because I'm not adding anything new. I am giving it a different type. And it gave me a place to put the comment here, that the arrival rate corresponds to passengers to arriving at a bus stop, and work is the time it takes a passenger to board the bus.

There are many other things I could have done for work. I could have made it multi-dimensional, having to do with how long to get on the bus, how many stops, et cetera. For simplicity, I'm just assuming that there are some agile people who just get on the bus quickly, and there's some people who are a little bit slower, who might take a longer time to get on the bus. And that's it for jobs. Pretty simple, but pretty typical of the kinds of things we see.

Then we come to the queues themselves. The root class is just the job queue. Jobs can arrive, and the queue can have a length. You'll note that I don't have any way for jobs to leave the queue in the base class. And that's because each subclass will be distinguished by the queueing discipline it's using. And the queueing discipline typically has to do with how jobs leave the queue.

So there's a FIFO discipline. Pretty simplistic. It just removes the first item from the queue and returns it. Raises an exception if you try and remove something from the queue when there's nothing there.

An SRPT is a little bit more involved, but not much. I've not tried to be clever. You could imagine a much more efficient implementation than the one I've got. But here, each time a job needs to leave the queue, I just search the queue to find one of the jobs with the least amount of work, and I remove that. That's why I'm searching with the index here, because pop lets me remove something from a list at a particular index.

And then finally, I have another simple class, bus stop. And the key thing here what it's a subclass of. Initially, I'm choosing it to be a subclass of FIFO. Later, we'll see I could make it a subclass of SRPT, or in fact, anything else I chose. But this gives me a convenient way to decide what queueing discipline I'm going to use at the bus stops.

So I mentioned earlier that SRPT is nice, because it allows us to reduce the size of the queue. I also mentioned that it's typically optimal in the sense of producing the shortest average waiting time. So let's think about why.

Imagine that we do the opposite. This is a good intellectual tool. Just think of the opposite. Suppose we always took the longest job first.

What would that do to the average waiting time? Say we had a job that took 10 minutes and a queue with 10 people in it. Well, while we were waiting for that 1 10-minute job to get on the bus, the other 9 people would each have to wait 10 minutes. 9 times 10, 90 minutes of waiting time. Not very good, right?

Now, suppose each of those 9 jobs only took 1 minute. Well, the first job would wait 0, the next job would wait 1 minute, the next job would wait 2 minutes. And even the longest job would only have to wait 9 minutes before it started. So the total amount of time spent waiting would be much less.

So we can see that by using shortest remaining processing time, the waiting times are reduced. Now, with the bus example, it's a little bit strange, in that, yes, you'd get on the bus, but maybe the bus wouldn't move at all while it was waiting for that last person to get on.

But most queueing disciplines-- imagine you're in the supermarket-- you don't care how long the guys behind you take. As soon as you get to the front, you get processed, you get to leave the store. So in fact, a supermarket would, on average, have happier customers if it followed shortest remaining processing time, and always took whoever had the fewest number of items first.

Why don't they do that? Or why is not SRPT always used in practice? It's got one glaring problem. What's that problem?

AUDIENCE: It doesn't seem fair to everyone else.

PROFESSOR: It's not fair. And this is actually a technical word. People talk about fairness of a queueing discipline. And in particular, it's so unfair that it allows starvation, another technical term.

Imagine that you're the person with the 10-minute job, and there's a continuing stream of people arriving with 1-minute jobs. You never to get your job processed at all. You just sit there, and you wait, and you wait, and you wait. And in fact, you wait forever.

In the bus situation, imagine you're the person who's in a wheelchair and takes 10 minutes to get on the bus. And people keep arriving at the bus stop. Well, before it gets to be your turn, the bus is full, and it leaves. And you have to wait for the next bus.

But people arrive while you're waiting. It arrives. Again, you don't get on. And you never get on.

This is technically called starvation. And it's a problem that SRPT has. So typically, if you're using that as your scheduling algorithm, you've got to put in some sort of a priority on top of it that gives the long jobs a chance to at least run once in a while. And there are lots of different disciplines for doing that.

All right, let's continue looking at the code. So now we've got bus stops, which are queues. We got jobs. Well, we're going to have our server. And in this case, our server is a bus.

Every bus has got a capacity, the number of people it's allowed to hold, and a speed. The speed has to do with how fast it goes from stop to stop. And for simplicity, we're going to assume that this depends only on the bus itself, not on the traffic. Not true in Cambridge and Boston, of course, but we can't get everything in a code that we get through in one lecture.

So it's got a capacity, it's got a speed. And then, we can keep track of how many people are on the bus. It's got an enter, which has to do with people getting on it. And does that until it's full, and then it raises an exception.

And things can leave the bus. And here's the interesting method. Unload takes a self and the number to be unloaded from the bus. And it just calls self.leave over and over again.

All right, those are our basic classes. Nothing too sophisticated. As is often the case with classes, the code is straightforward. And now comes the not-so-quite-straightforward code, which is the actual simulation itself.

Probably the most important thing to understand in looking at this code is the arguments. This will tell us what the simulation depends on. So there's going to be a bus. So notice we have only one bus.

So, again, I've simplified the simulation. Instead of having multiple buses running around the loop, we're going to assume you've just got to wait for that first bus to get back. Later, we can talk about what would be involved in having multiple buses. The number of stops in the loop, the loop length. And for simplicity, I'm going to assume that the stops are equally spaced along the length.

The mean arrival rate of jobs, the mean work per job, and then how long we're going to run the simulation. Roughly speaking, I'm using the same units for the simulation time and the loop length. So this would tell us, if we simulate for 30,000 times, and the loop is this 1,200 time units long, how many times around the loop we could get, and how many stops the bus would make.

All right. And then comes the body of the simulation. So the first thing-- actually, let me write up some pseudo-code here to basically show you the way it's structured. Yeah, I think we'll put it over here.

So the first thing we do is we create the stops, and we initialize some variables then we're going to use to keep track of performance. We then enter the main loop, which is essentially while time not expired. That has to do with how long we're going to run the simulation.

We move the bus. So the loop will go through one time unit each iteration. At least, it looks that way initially. So we'll move the bus that amount of time.

So then, while we're doing it, passengers will arrive at the stops. Then we'll see when a bus is at a stop. Has it driven long enough to get to the next stop?

And when it's there, we'll unload some passengers first, and then we'll load some passengers. And then, when we're done at the very end, we'll compute some statistics. So that's the overall structure of the simulation. Then we can put that in something outside it that tells us how many simulations to run and get more statistics about aggregate.

All right, now working our way through the code, the initialization, while time less than simTime, advance time. Move the bus. Now, how far we move it depends upon how fast it is. So we'll just add to the bus location however far it's allowed to move in one time unit. I don't care whether the time unit is seconds, minutes, hours, days, whatever. Just it has to be all consistent.

Then I'll see if there's a passenger waiting to enter the queue by checking the arrival time. And passengers can arrive simultaneously at each stop. I apologize for leaving the comments out of your handouts, but I couldn't squeeze it onto the page with the comments.

Then, here's the interesting piece. So we do this. We're just having passengers arrive based upon the characteristics of jobs. See if the bus is at a stop.

If the bus location divided by the inter-stop distance is equal to 0, so it's actually at a stop, I'll do my unloading, et cetera. Now, down here, you'll notice that up here, I had this assumption that the speed of the bus is small relative to the inter-stop distance.

This assumption is important because of this statement. If the bus were moving too fast. It might skip over the stops. Now, this is kind of a clumsy trick I've used here. It has to do with it being small. It has to do with each movement of the bus being, in fact, an integral of the inter-stop distance so that you don't skip over the stops.

In a real simulation, I would have done something much more careful to deal with not missing stops. But again, I wanted to make this relatively simple. So I'm going to jigger things so that the speed of the bus evenly divides the inter-stop distance, so when I advance the bus, it never skips over a stop. It's just a trick. Not very elegant trick, but it keeps things working.

All right. Then I'm going to unload some fraction of the passengers. Now, again, in a more realistic simulation, I'd be keeping track of what's the average distance a passenger wants to travel. And I would have, in fact, made work two-dimensional-- the amount of time to get on the bus, plus how far the passenger wanted to travel. Again, here, I've done something simple. I've just assumed that a fraction of all the passengers get off at each stop, and the fraction depends upon how many total stops there are.

We can still understand most of the interesting aspects of the problem, even with all of these simplifications. And that's an important lesson. Every simulation is a simplification of reality. No simulation is ever a perfect model of reality. No model is ever a perfect model of reality.

What we typically do in building the simulation is simplify it enough that we can get it to work, but not so much that we can't answer the questions we want to answer. And so I've made a lot of these simplifications, but they will still allow us, while maybe not to get precise answers about the impact of various things, to understand how things relate to each other, the relative impact of various design decisions.

All right. If the bus is at a stop, as I said, some passengers will get on, get off. Passengers that are already at the stop will try and get on the bus.

We'll have to advance time, because time starts to run out as people are getting on the bus, even though the bus is moving. But, of course, even though time is advancing, the bus is not. So we have to model that, the impact of loading times.

And in fact, in mass transit systems, it turns out that loading time is a very important parameter. In many urban systems, subways will spend more time at stations getting loaded than between stations, moving to the next one, if the stations are dense. You may have noticed this on the Green Line. So we have to make sure we model that.

And then, of course, since the bus could be at the stop for a while, passengers might arrive while it's at the stop. Then, we can keep track of the number of people left waiting when the simulation ends, and do things like that.

All right, finally, there's this program test, which I won't go over in detail. It's exactly like the drivers we've written for other simulations for at least half the semester, in which I run simbus multiple times, accumulate results from trials, and then plot them nicely.

All right, let's run it. So I'm going to run it with-- I've actually written it so that I can look at combinations of speeds and capacities. But first, I'll just look at small combinations-- i.e. each list will be one. So we'll test it with a capacity of 30 and a speed of 10, to 40 trials. Then, we'll test it with a capacity of 15 and the speed of 10 and a capacity of 15 and a speed of 20.

All right. We have some results. So what we can tell looking at Green Line is that I have a speed of 10 and a capacity of 15. The aggregate average wait time is long and apparently growing the longer I run the simulation. I've only plotted here up to the first 500 times if the bus stop at a stop. But we can imagine that if I went further, it would just get worse for a while. Then, it will eventually plateau, but it will take it quite a while to do that.

And here we can see the interesting thing, in that the Blue and the Red Lines are not very different. So what this tells us is there's a trade-off. I can afford to have smaller buses if I make them faster. Or conversely, if I want slow buses, I have to make them bigger. And I get more or less the same average wait time and not much difference in how many people are left behind at the end.

Clearly, here I have a big number. These are smallish numbers. Again, we'd have to run it longer to see whether these are stable, and with different seeds, or different random numbers, to see whether or not the fact that this is 80 and this is 40 is statistically significant, or just an accident of this run. Because we do have randomness.

But we did do 40 trials, so maybe there's a reason to believe that in fact, this is slightly worse. You'll notice also the Blue Line doesn't quite get all the way to 500, which means that slow bus didn't get to make enough stops. It took a long time to load up, because it had big capacity and didn't really get to all of the stops.

All right. Let's try another experiment. And let's go change the queueing discipline. Go way back up here. And let's make it SRPT and see what we get.

Well, we get something pretty different. In particular, you'll note that the average wait time has dropped rather significantly. It really has, as the theory predicted, led to the improved wait times. We'll also notice that it's got some interesting shapes here, where it jumps around, goes up, almost a knee here.

And you might try and understand why it's doing that as you look at the code. Is it random? Is it just bad luck? Is there something fundamental going on? Think it all through and decide what's going on.

All right, finally-- and, again, I suggest you play this with a lot of different combinations to get a sense of what the code is doing. So one last queuing network question to test your understanding-- different question.

Suppose you were given the job of hiring a 6.00 lab assistant, and you had a choice between hiring an MIT student at $15 an hour or two Harvard students at $7.50 an hour. Suppose further that the MIT student could deal with twice as many problems per hour as the Harvard students. Does it matter which you hire?

Well, you could build a simulation, and you would discover that in your queuing network, it didn't matter. On the other hand, when you ran it in real life, you would probably discover that it mattered a lot, because the MIT student would answer questions correctly. And the moral is, the queuing network never tells the whole story. You've really got to think about the thing in more detail.

All right. One more lecture. See you on Thursday.