The following content is provided under a Creative Commons license. Your support will help MIT OpenCourseWare continue to offer high quality educational resources for free. To make a donation or view additional materials from hundreds of MIT courses, visit MIT OpenCourseWare at ocw.mit.edu.

PROFESSOR SRINI DEVADAS: Erik and I have been tag teaming this lecture in this class so we're going to split this lecture. So I get to do the first 2 minutes. No. I get to do the first 20 minutes, or so, talking about some of my research in parallel architecture. And Erik's going to talk about a bunch of things that he's been up to over the years in Algorithm Design and Analysis. So let's get started. When was the first PC built, anybody? Yeah.

AUDIENCE: In the 1950s.

PROFESSOR SRINI DEVADAS: No. The first personal computer was 1981-- not the first computer. So all of you know about Intel, and Microsoft, and IBM, and so on. Intel's gift to humankind is the x86 architecture. Though, some people would argue that point. And the x86 architecture was invented in 1981, and was part of the first PC-- that provided the horsepower for the first PC-- the IBM PC. And it ran at 5 megahertz.

And x86 has been around-- you still can buy x86 computers. The 80486, in 1989, ran at 25 megahertz. So you can see a trend here. And the 80486, as it turns out, ended up being called the I486 because there was a court ruling that said that you couldn't trademark numbers. And so Intel, at that point, decided to start naming their processors.

So the Pentium, which is one of the more famous Intel processors, was built and came out in 1993. And the clock speed went up to 66 megahertz, back in the early '90s. And since this is just such a cool name, Intel continued to call its processors Pentium. And the Pentium 4, in 2000, had this incredibly deep pipeline where you broke up the computation into a bunch of stages. In fact, it had a 30 stage pipeline. And so the clock speed went up all the way to 1.5 gigahertz.

The Pentium was famous for many things, including a couple of bugs in the floating

point pipeline where division, in particular corner cases, wasn't done correctly. And there was also this bug called the F00F bug, which allowed a malicious program to crash the entire system, regardless of whether it had administrative privileges or not.

But the Pentium was obviously very successful. A lot of machines sold. And it felt like it was only going to be a matter of time before we got to 10s of gigahertz, the way things were going. As you can see, this is a pretty steep growth from 5 megahertz to 25 to 1.5 gigahertz in the space of about 20 years.

As it turns out, after the Pentium D, which came out in 2005, where the clock speed peaked at about 3.2 gigahertz, clock frequency stopped increasing. And what you see now are things that correspond to multiple processors on a chip. So for example, the Quad Core Xeon came out in 2008. You can still buy it. Only runs at 3 gigahertz, which is basically about the same as the Pentium D ran. Each of these has a range of frequencies. And beyond about 2005, the clock speed of processors that you can buy is kind of saturated at about 3 gigahertz.

And the way you're getting performance is by putting multiple processors on the chip. And people use the term cores synonymously with processors. So a quad core means that they're, in effect, four x86 processors on the same silicon integrated circuit. And they're interconnected together. And they talk to memory. And you have, essentially, a parallel processor on a single chip. And the single user, potentially running many programs, is using this system. And you have dual core processors on your laptops.

And so the scale, now, is-- the metric now, I should say-- is how many cores do you have on a chip. And people are predicting that we're going to have 1,000 cores by 2020, on a chip. So this brings us to the problem of how do we use parallelism. So there's a lot of work in parallel algorithms. And there's also work in building hardware, such that algorithms can sort of automatically be parallelized while they're running in hardware, so they can run faster, and so on and so forth.

So some of my research is in parallel architecture. Some of it is in parallel

algorithms. I want to give you a sense of what the problems are in building parallel architectures. And in particular, I'll start with a canonical system that corresponds to, let's say, this quad core system.

And so you have 4 processors on this single integrated circuit. So that signifies that. And typically, you have a lot of fast, static random-access memory, SRAM, on the same chip. So typically, megabytes of the memory on the chip and gigabytes of memory in DRAM, which are separate modules that are connected via high speed bus, off the chip. So there are usually many DRAM modules. They're called DIMMS-- if you might have heard the term.

So the connection between the processors and the SRAM is typically very fast. It's on-chip. Things being clocked at gigahertz. And when you go off-chip, you're down to a few hundred megahertz. So typically, an order of magnitude less speed. But you're accessing much more memory. So this is really gigabytes and this is at the level of megabytes.

If you see this picture, here-- if you think about the number of processors increasing from four to eight to 16, all the way to, say, to hundreds of processors, you can see that there's going to be a bottleneck associated with accessing the memory. The big problem is you can't possibly build memory that serves hundreds of requests in parallel. If you try and make a large SRAM, which is megabytes long, the number of ports in the SRAM-- read ports-- is roughly of the order of four. And after that it's kind of hard to build. So this architecture isn't going to be sustainable beyond 4, 8, maybe 16 cores.

So typically, what people build is-- or people are trying to build in academia-- is something that corresponds to a distributed architecture on the chip, where you have processors and memory in tiles. So you have, essentially, something like this, where you can imagine having literally 100 processors on a chip that correspond to an implementation where you build tiles, where you have a processor that's doing the computation, and you have memory-- sometimes called cache memory. But there's multiple levels of caches, typically, that are attached to each of these

processors. And the space between the processor tiles is reserved for interconnect or for wires that connect these processors up.

And so there's research that goes on in routing algorithms. How you figure out if these processors want to talk to each other; what the best way of routing the messages are; you want to find the shortest path. In this case, the weight corresponds to the congestion that's associated with each of these channels that you have. And people actually use algorithms like weighted shortest paths, in hardware, to determine what the best way of getting from here to there is. It may not be this way. It may be going around the chip simply because that path-- the latter one is less congested.

The other issue that comes up has to do with how long it takes to go across the chip and come back. So if this processor wants to access its local memory-- that's typically pretty simple or fast. But if it wants to access remote memory-- and it's quite possible that it's sharing some data with a different thread running on a different processor. So typically, there's a program running on this processor, sometimes called a thread, and this program may share data with a different program, which is running on this processor. Or it may just require a lot more space. And what this program has to do is make a request all the way to this processor and this particular cache in this processor. And then it gets the data back.

So what you see here is a round trip access that goes across the chip. And this distance, if it's large, could take 10s of cycles. So typically, it's a single cycle to access local memory-- the fastest local memory, called the L1 cache. But it could take 10s of cycles to go send a message across the chip and 10s of cycles to get the data back. So the bottleneck, really, in parallel processing from a standpoint of communication is this routing of messages and getting the messages back.

One of the things that my research group is doing is looking at the notion of migrating computation as opposed to data. We call it execution migration, where you could say-- suppose I have a processor running a particular program, out here. And if this program wanted to access a remote memory, then, rather than doing

what I just showed you there-- send a message, get the data back-- you could imagine that you could migrate the program itself. And in particular, you think of it as migrating the context of the program from this processor to this one. And so what is the context?

For those of you who have taken 6.004 probably know what this is. But it's simply where you are in terms of executing your program. And that's typically given to you by our program counter, and your current state of your register file, and a few other things, including cache memory and so on and so forth. So the advantage with execution migration is that it's a one way trip, as opposed to a round trip.

You don't have to send a message and get the data back, which would be two messages, if you will-- one in the case of the address and the other for the data-- but you migrate your execution. Since you have computation out here, you can run on this remote processor. So that's one of the advantages of execution migration

One of the downsides of it is that this can be multiple kilobytes-- or kilobits. And it could be significantly more in terms of size, or in terms of bits, than the data that you want to access. So there's a trade-off here. And then, when any time you have a trade-off, you can think of an algorithm to try and find the optimal trade-off. So this is the context for the particular optimization problem that we need to solve, here, that corresponds to really deciding when you want to do data migration and when you want to do execution migration.

There's a choice. At the top level, it's a round trip to get the data. So you're really traveling longer-- twice as long. The distance is twice as much. But it's possible that the amount of state that you'd have to move, in terms of taking your context of your thread and moving across the chip, could be large enough that it offsets the advantage of the shorter distance.

So we set this up as an optimization problem. So now we're in the realm of-- we moved from 6.004 to 6.006, here, in the last couple of seconds. So assume we know or can predict the access pattern of a program. And you can do this-- people build these things in hardware-- prefetch engines, branch predictors, and so on.

They're in the x86 machines. And you can tell-- especially if you're going through a loop over and over-- you can make this prediction.

So you have some amount of look ahead. And you know that m1 through mn are the memory accesses that this program is going to make. And these other memory addresses. And I'm going to think about p of m1, p of m2, p of mn, as the processor caches for each mi.

So what might be the case, in a simple example, is you want to access memory in processor one. You're sitting there and you want to access memory in processor one. And then, the next one, you want to access memory in processor two. And so on and so forth. So you might see something like that. So the sequence of memory addressees-- if you're sitting on processor one-- this first one is local. And then, after that, you want to access processor two's memory because you're sharing data with it. Then you're back home, again, to processor one. And so on and so forth. So that's one example of a set up.

And we can think of about the cost of migration as-- if you want to go from s to d-- as being a function of the distance, s comma d, plus some constant, which is proportional to the context size. And that context size, we're going to assume is fixed for a particular architecture. It may change for different architectures, but if it's a few kilobits, then there's going to be some overhead associated with putting the context onto the network. And it's a sizable overhead that needs to be taken into account. That's the cost of migration.

The cost of an access, s comma d, is twice the distance between s and d. And it's typically just a word that you want to access-- 32 bits, 64 bits-- and so there's no additional overhead associated with a data access. So there you go. You have the formulation of the problem. You have the trade-off written, where the cost of migration has just the distance. But it has a constant factor. And you've got twice the distance, here, for the access.

Now if s equals d, and I want to write this down, you have a local access. And the cost is assumed to be zero. You could change that. We are in the realm of the

theory and symbols. So you can do whatever you want. But given those equations, our problem is decide when to migrate to minimize total memory access cost.

So in our example there, I suppose we had p1, p2, p2, et cetera. And let's say you start at p1. This first one would be a local access. And then, you may decide that you want to migrate to p2, over here. In this case, you get this as a local access, as well. So is this one. Right here, you might want to migrate to p1 back to be p1. So this becomes a local access. That's a local access. They're all, essentially, free. And then, if you just stay at p1, over here, you may end up doing remote accesses to p3 and p2, respectively. And so you have a cost of migration-- the cost of migration and the cost of two remote access.

So that's the set up. How are we going to solve this problem? Are we going Dijkstra? Are we going to use Bellman-Ford? Are we going to use balanced search trees? Are we going to use hash functions? What are we going to use?

AUDIENCE: Dynamic Programming.

PROFESSOR SRINI DEVADAS: Dynamic Programming. All together.

EVERYONE: Dynamic Programming.

PROFESSOR SRINI DEVADAS: Dynamic programming, all right. We're going to use dynamic programming to solve this problem. Good. So Erik taught you something.

AUDIENCE: Where are the erasers?

PROFESSOR SRINI DEVADAS: Yeah. Where are the erasers? I think they fluttered down here. All right. Let me bail out and use this while you find the erasers. So a program at p1, which is the processor, initially. I'm just going to set up this DP. Let's assume that the number of processors equals Q. Now, what are the subproblems?

You could do this many different ways. Let's go ahead and use prefixes. And so DP(k,p1) is the cost of the optimal solution for the prefix m1 through mk of memory accesses, when the program starts at p1 and ends at pi. So that's my subproblem. I

want to know, as I build this up, what is the optimal way that I'm going to choose between migrations and accesses for the first k memory access, assuming a starting point at p1 and ending at some pi. And I need to build up these subproblems. And I want to grow them.

Let's go ahead and set this up. What I want to do now is figure out DP(k plus 1, pj). And assuming I have all of the k, pi's computed-- and how many subproblems do I have? How many subproblems do I have? Total?

Look at this and tell me what the ranges of the possibilities are. So how many subproblems would I have? Someone? N times Q. So you have N times Q subproblems. So you've set this up for up until k and for all of the pi's.

Now, what you have to do is essentially say, well, DP of k plus 1, pj is going to be k, pj plus cost of access pj, p of mk plus 1 if pj is not equal to p of mk plus 1. So there's going to be two cases. I'll just write this out and I'll explain it. But the first case corresponds to if the new memory access is not in the processor cache corresponding to pj, then what you could do is use the optimum value, where you ended pj, and simply do a remote access that corresponds to accessing mk plus 1. So that's one case.

The case is to use the minimum solution-- optimum solution corresponding to ending at pi and do a migration. You have cost of migration from pi to pj. And you do this if you want to go do p of mk plus 1-- the processor corresponding to p of mk plus 1.

So that's the set up for this dynamic program. What you've done is created a sub problem, its optimum, and then you look at the two cases. You want to go migrate and do a local access-- that's this case over here. Migrate to the processor and do a local access there. That will be this case. And in this case, you stay where you are and do a remote access. In the case of migration, you could end up choosing different initial starting points corresponding to the pi's. And you have to run through all of those.

So what's the cost of a subproblem, or the running time of computing one of these things-- it's order? Q. And so the total cost is NQ squared. It's a little review of DP. I'm going to stop here and let Erik take over.

Just, in closing, while this makes some assumptions, It's actually fairly close to what we're building in hardware. This type of analysis is something that we have to do in hardware. My research group is building a 128 processor machine, that we call the Execution Migration Machine. And it does exactly what I've described to you, decide whether to do a remote access or to do a migration based on this kind of analysis. So hand it over to Erik.

PROFESSOR ERIK DEMAINE: I have a microphone.

PROFESSOR SRINI DEVADAS: All right. Good.

PROFESSOR ERIK DEMAINE: So I have a few things to tell you a little bit about. Srini talked about one topic in detail. I'm going to talk about many topics in less detail, as I said "shallowly." And these are my main areas of research. I do geometry, in particular, folding, and data structures, graphs, and recreational algorithms. That's the really fun stuff.

A lot of these have corresponding courses if you're interested in more about this stuff. Computational geometry, in general, is-- I'm not going to remember all numbers. 840? 50? 50. 6.850. That's a class I don't teach. Folding is 6.849. Data Structures is 6.851. And Graphs was being taught this semester, in parallel with this class. 6.889. And recreational algorithms isn't fully covered but you could check out SP.268, which was offered last semester.

And especially for those watching at home on MIT OpenCourseWare-- this class, all the video lectures are online for free. 6.851, we'll do that next semester. And 6.889 are all online, right now. And there's some lecture notes for SP.268 on OpenCourseWare.

There's a lot of material, here. And in particular, the obvious next class for you to be

taking is 6.046. But why should you be taking 6.046? Because then you can take all these exciting classes and many others about algorithms. There's a complete list of follow-on classes in the lecture notes, which are online. And there's a ton of-- there's so much research in algorithms. It's a really exciting area. This is just the beginning-- just a taste. And I want to show you various exciting places it can go. Let's do some algorithms.

So the first topic I'll tell you a little bit about-- maybe the most fun-- is geometric folding algorithms. That's the title of the textbook and the class 6.849. And in general-- well, there's a lot of different kinds of folding, in the world, but maybe the most accessible and fun is origami. So you have, on the one hand, a piece of paper. And you'd like to turn it into some crazy, three dimensional shape, which I'm not going to try to draw here. You want to fold a giraffe or you want to make some geometric sculpture. How do you do this?

So, usually, you put some creases into the piece of paper in some reasonable way. And one of the questions is what are the rules for putting creases into a piece of paper? When is that possible? And then you'd like to fold it into that shape.

So there are really two big problems here. One is I guess you could call it foldability. And this is what you do if you practice origami in the typical way. You get origami diagrams, and they say, "fold this." And you're like, oh, gosh. Takes you hours to figure out how to fold something. Especially, if they just gave you a crease pattern. Can you even tell does it fold into anything, first of all. And then, if so, how do I do it?

That problem-- folding increase pattern and understanding what crease patterns are valid-- unfortunately, is NP-complete. So there's no good way to really understand that. So origami is hard. In some sense, the more interesting direction, though, is the reverse direction, which I would call origami design.

I have an intended 3D shape I want to design. How can I come up with-- how can I, as an algorithm, convert that 3D shape into a crease pattern that does fold, that's guaranteed to fold into that 3D shape. And that's actually solvable. So design is

easier.

And there's all sorts of different versions of the design problem. Some of them, you could solve in polynomial time. Some of them, you can't. If you really want optimal design, that can be NP-complete again. But in particular, there's a way to fold any 3D shape you want.

So there's an algorithm-- the coolest one, right now, is called Origamizer. It's free software online, by Tomohiro Tachi. And you give it a 3D model of a polyhedron. And it outputs a giant crease pattern on a square piece of paper that folds into that 3D polyhedron. And it's reasonably practical. And he's folded tons of models in that way. Let's see.

I'll show you some other things. Here's a simple example of a geometric origami model. So this is folded from a square paper with concentric squares as creases. Alternating mountain and valley. So you see mountain valley, mountain valley. Also fold the diagonals. It's very easy to make. And what's funny-- what's cool about it is that when you put all those creases in, it pops into this 3D shape, which for many years people conjectured was a hyperbolic parabola.

This design is one of the earliest geometric origami designs. It goes back to late '20s in the Bauhaus School of Design. And it's very cool. People fold them a lot. I've personally folded thousands of them for sculpture and things. We also do a lot of algorithmic sculpture, which I won't talk about in detail here.

But we discovered, two years ago, that this does not exist. It is impossible to fold a square piece of paper with this crease pattern. That was a bit of a surprise. And it's kind of fun to make things that don't exist.

AUDIENCE: So what is that?

PROFESSOR ERIK DEMAINE: So what is this? Well, somehow, physical world is differing from the real world. Now, some ways it might be differing are that these creases might not be creases in the technical sense. A crease is a place that should be non-differentiable. So maybe they're kind of rounding it out. And then, who knows what's happening. Then, kind

of all bets are off.

Another possibility of what I think is happening is that their are extra creases, in here, that you don't see. They're very small. If you look, especially the raw edge, here, and that profile. It's a little bit wavy. And it's conceivable there's some points here that look non-differentiable to me. And I always thought I wasn't folding it well enough.

But in fact, something like that has to happen. And my conjecture is, if you look at this under a microscope, which we haven't done yet, there are little creases that are so shallow they're hard to see, but are there. And the theorem says some creases have to be there. It is possible to fold this with extra creases, but not with these. So get rid of that.

On the other hand, if you do the same thing with concentric circular creases-- this a little harder to unfold. It really wants to be in this kind of Pringles shape. This also is from about Bauhaus. It's a little harder to fold concentric circles. But this, we think, does exist. Can't prove it yet. So we've done a lot of sculpture based on these guys. What else do I want to say?

Another demo. So here's a fun problem. This is a magic trick. Goes back to Houdini and others. So imagine I take a rectangle of paper and then I fold it flat and take my scissors-- not strict origami, here-- and I make one complete straight cut. In this case, I get two pieces. And I unfold the pieces. And the question is what shapes can I get out of those pieces?

In this case, I get a swan. You're not impressed so I'll another one. Make one straight cut. These are on my web page if you want to impress all your friends. You could take the class if you want to know how it's done.

This example has a lot of symmetry. You get a little angelfish. I only have one more example. I hope you'll be impressed. This is very hard to fold. It was an MIT spotlight picture, at some point. And it's even harder to cut. Straight cut. This should

be the MIT logo.

[APPLAUSE]

So the theorem is there's an algorithm, given any set of polygons in the plane, you could fold, make one straight cut, and get exactly those polygons. There's some limits, in practice, because of paper thickness. But in theory, you can do everything. All right. Fun stuff.

I don't think I have time to talk about self-assembly. Let me talk a little bit about data structures because, conveniently, Srini drew this diagram for me. And I have the exact same diagram-- the left one, though. I'm old fashioned.

So the models of computation we've used, in this class, are pretty simple. We have, in particular, the Word RAM. You can read a word. You can add two words. Do whatever you want with a constant number words. Send them out to main memory. Everything's the same amount of time. It's all constant, anyway, so who cares?

Except there's this issue in real computers, and it gets even worse with parallel, but let's stick to sequential old fashioned computers. So you have this slow bottleneck between main memory and cache. Cache is really fast. Think of this as a really fat pipe. And this is a very thin pipe.

What do we do? We'd like to always work with things in cache, but that's kind of difficult. At some point, you run out of space. You've got to go to main memory. And maybe to disc, other levels of the memory hierarchy.

So what systems do is, when you fetch something from memory, you don't just get one word, you get an entire cache line. And cache lines are getting bigger and bigger. But memory transfers happen in blocks, when you're going to a big memory.

So let's say B is the size of a block. There is another model of computation that's more sophisticated than the Word RAM that says how should my running time depend on B. How many memory transfers do I need to do, as a function of B and n?

And so for example, if you want to do search-- normally, we think of doing binary search. That takes log(n) accesses if everything is uniform. But with asymmetry, and if you're reading in entire blocks, if you do it right, you can do it in log base B of n, instead of log base 2. This is counting memory transfers, not computation. Computation here is free. It's a little weird, but you get used to it.

Sorting. They're classic. Just to give you an idea of how this gets a little complicated. You get n divided by B times log base C of n divided by B. C is the number of blocks that fit in here. So there's C different blocks that fit in your cache. That's the optimal way to sort. Just upper and lower bounds in the comparison model.

Just to give you a flavor. And there's a whole study of algorithms to do this. What's really cool is you can achieve these bounds even if you don't know what B is. And if you don't know what C is. There's one algorithm, that whatever the architecture is underlying it, we'll still achieve the same bounds.

Those are called cache-oblivious algorithms, and they were invented, here, at MIT. I think I want to-- this is too much fun to pass up. On the Word RAM, there's this problem, which we've dealt with several times. What if you want to maintain a dynamic set of elements-- integers. I want to do insert, delete, predecessor, successor. This is what binary search trees do. But you can do better.

If we have integers-- n integers-- in the range 0 to u minus 1. So u is the size of the universe. Then, we already know how to do log(n). But you can do two bounds. One is log(log(u)). This is a data structure called [INAUDIBLE]. And it's in CLRS, if you're interested. You can also do log(log(n)) divided by log(log(u)).

This is a data structure called fusion trees. It's an advanced data structure. 6.851, if you're interested. And you can take the min of those two. That's, essentially, the best possible, the matching lower bound, that that that's all you can achieve. And so just to state it in terms that you know, which is normal n bounds. You take the min of those two things, there are always at most square root log(n) divided by log(log(n)).

Compare that with log(n). It's way better. A whole square root better. And a little tiny savings better. And this is optimal. It is a function of n. That's the best you can do for the predecessor problem. So pretty crazy stuff. It's a very complicated structure. It's probably completely impractical. But, hey. They're, theoretically, pretty cool.

I'll tell you a little bit about graph algorithms. We've seen a lot of graph algorithms in this class. One way to make them new and fun again is to suppose your graph is planar or almost planer. Meaning you can draw it in two dimensions without any crossings, as you might get from a graph that's drawn on the earth, like a road network or something with no or few overpasses. Then you can do things a lot better.

For example, you can do the equivalent of Dijkstra's algorithm. So non-negative weight shortest path, in linear time. That's not so impressive cause Dijkstra is number of edges. Here, I mean number of vertices. It doesn't really matter with planar graphs. And we had E log(V). You can write E, here, if you prefer. It's only a log savings.

More impressive, is you can do with negative weights-- the equivalent of Bellman-Ford-- in almost linear time. So some log factors. Log squared n divided by log(log(n)). It's the best bound known to date. That was a result from last year. So it's still a work in progress. And if you're interested in this kind of stuff, you should check out the videos for the class we just taught, 6.889.

And recreation algorithms. I've actually already told you about a lot of these-- like algorithms for solving a Rubik's cube in n squared divided by log(n) steps. That was a paper this year. Tetris is NP-complete. A whole bunch of NP-completeness, and x time completeness, and so on.

Results for games. Other fun stuff, like balloon twisting-- algorithms for designing how to balloon twist a given polyhedron, optimally, using the fewest balloons. Algorithmic magic tricks. There's tons of stuff out there. It's really fun. I should teach a class about some of those things, but I haven't yet. The last thing we wanted to do

is together. And it has to do with these

PROFESSOR SRINI DEVADAS: Getting rid of these--

PROFESSOR ERIK DEMAINE: These cushions. Getting rid of these damn cushions. We have so many of these cushions. Just gotta get rid of them. That's two freebies.

PROFESSOR SRINI DEVADAS: Now, you're going to have to pay for these cushions.

PROFESSOR ERIK DEMAINE: He's kidding. He's kidding. Actually we're having trouble. We're having trouble giving them away because-- I don't know-- some people seem to not like them very much. And neither do we. So we wanted to give you some motivation for why you really need some of these cushions. So we actually prepared a top 10 list.

PROFESSOR SRINI DEVADAS: This is the top 10 uses of 6.006 cushions. We're going to alternate here. Number 10.

PROFESSOR ERIK DEMAINE: You can sit on it and get guaranteed inspiration in constant time.

PROFESSOR SRINI DEVADAS: Don't forget to bring one for the final exam.

PROFESSOR ERIK DEMAINE: Highly recommended it. Number nine.

PROFESSOR SRINI DEVADAS: You can use it as a Frisbee. You've seen that before, except you cut it into a circle. You cut it into a circle. And it works really well.

PROFESSOR ERIK DEMAINE: We had fun with a Bandsaw, last night.

PROFESSOR SRINI DEVADAS: Number eight.

PROFESSOR ERIK DEMAINE: You can sell it as a limited edition collectible on eBay.

PROFESSOR SRINI DEVADAS: It's never ever going to be made, again. You can make money off this in 5 years-- 10 years.

PROFESSOR ERIK DEMAINE: At least $5. I don't know. Number seven.

PROFESSOR SRINI DEVADAS: Number seven. If you had two of these, you could stick them like this, and remove the branding, and use it as a regular cushion.

PROFESSOR ERIK DEMAINE: Now, no one will ever know you took this class. You just need two.

PROFESSOR SRINI DEVADAS: Number six.

PROFESSOR ERIK DEMAINE: Number six. It's a holiday conversation starter.

PROFESSOR SRINI DEVADAS: And conversation stopper.

PROFESSOR ERIK DEMAINE: Number five.

PROFESSOR SRINI DEVADAS: Asymptotically optimal-- we had to use that term, acoustic acoustic paneling.

PROFESSOR ERIK DEMAINE: That was a suggestion from a student. You just need a lot of them. This would be great for piano, guitar fingering practice. You know you're doing your DP.

PROFESSOR SRINI DEVADAS: Number four.

PROFESSOR ERIK DEMAINE: Number four. You can use it as target practice for your next larp session. Woah.

DEMAINE: Misfire. I'm missing.

PROFESSOR SRINI DEVADAS: You haven't hit me yet. All right. Finally, you got one.

[APPLAUSE]

PROFESSOR ERIK DEMAINE: Number three.

PROFESSOR SRINI DEVADAS: All right. 10 years from now, it might be all you remember about double 0 6.

PROFESSOR ERIK DEMAINE: In truth, you might also remember this top 10 list.

PROFESSOR SRINI DEVADAS: All right. Number two.

PROFESSOR ERIK DEMAINE: Number two. You can use it as your final exam cheat sheet. This is a new rule. Instead of 8 and 1/2 by 11, you could bring in the appropriate number of cushions. And the number one-- number one use for a double 0 6 cushion.

PROFESSOR SRINI DEVADAS: Three words. OK Cupid profile picture. Don't use this cheat sheet. But come to the final exam and good luck.

PROFESSOR ERIK DEMAINE: Thanks.

[APPLAUSE]