PROFESSOR: All right, welcome to the final lecture of 6046. Today we continue our theme of cache oblivious algorithms. We're going to look at two of the most basic problems in computer science-- searching and sorting, a little bit of each. And then I'll tell you a little bit about what class you might take after this one.

So brief recap of the model, we introduced two models of computation although one was just a variation of the other. The base model is an external memory model. This is a two-level memory hierarchy. CPU and cache, we view as one. So there's instant communication between them, which means what you're computing on can involve this cache of size m-- total size of the cache is m-- words. The cache is divided into these blocks of size b each. So they're m over b blocks.

And your problem doesn't fit here, presumably, or the problem's not very interesting. So your problem size n is going to require storing your information on disk. So the input is really provided over here. Disk is basically infinite in size. It's also partitioned into blocks. And you can't access individual items here. You can only access entire blocks.

So the model is you say, I want to read this block and put it here. I want to write this block out and put it here. That's what you're allowed to do in the external memory model. And what we count is how many block-memory transfers we do. We call those memory transfers. So you want to minimize that. And usually you don't worry too much about what happens in here, although you could also minimize regular running time as we usually do.

The cache oblivious variation is that the algorithm is not allowed to know the cache parameters. It's not allowed to know the block size. Sorry, it's also block size b in the disk. So they match. And you're not allowed to know the cache size, m. Because of that, all the block reads and writes are done automatically. So the model is, whenever you access an item, you view the disk as just written row by row-- sequentially block by block. So in linear eyes, it looks like this, partitioned into blocks.

And so whenever you touch an item, the system automatically loads that block. If it's not already in cache, it loads it in. If it's already in cache, it's free. When you load a block in, you probably already have something there. So if the cache is already full, you have to decide which one to evict. And we had a couple of strategies. But the one I defined was the least-recently-used block. So whichever one in the cache that's least recently been used by the CPU, that's the one that gets written out, back to disk where it originally came from. And that's it. That's the model.

OK, this is a pretty good model of how real caches work. Although this last part is not how all real caches work. It's close. And at the very end, I mentioned this theorem that Why LRU is good. And if you take the number of block evictions-- the number of block reads, equivalently-- then LRU has to do on a cache of size m. Then that's going to be, at most, twice whatever the best possible thing you could do is given a cache of size m over 2.

So we're restricting OPT. We're kind of tying OPT's hands behind his back a little bit by decreasing m by a factor of 2. But then we get a factor of 2 approximation, basically. So this was the resource augmentation. And this is regular approximation algorithms. In general, this is a world called online algorithms, which is a whole field. I'm just going to mention it briefly here.

The distinction here is LRU, or whatever we implement in a real system, has to make a decision based only on the past of what's happened. The system, we're assuming, doesn't know the future. So in a compiler, maybe you could try to predict the future and do something. But on a CPU, it doesn't know what instruction's going to come next, 10 steps in the future. So you just have to make a decision now, sort of your best guess. And least recently used is a good best guess.

OPT, on the other hand, we're giving a lot of power. This is what we call an offline algorithm. It's like the Q in *Star Trek: Next Generation* or some other mythical being. It lives outside of the timeline. It can see all of time and say, I think I'll evict this block. This is like the waste of Q's resources. But I'll evict this block because I know it's going to be used for this in the future.

LRU is evicting the thing that was used farthest in the past. There's a difference there. And it could be a big difference. But it turns out they're related in this way. So this is what we call an online algorithm, meaning you have to make decisions as you go. The offline algorithm gets to see the future and optimize accordingly. Both are computable, but this one's only computable

if you know the future, which we don't.

What I haven't done is prove this theorem. It's actually really easy proof. So let's do it. I want to take the timeline and divide it into phases. Phases sounds cool. So this is going to be an analysis. And in an analysis, we're allowed to know the future because we're trying to imagine what OPT could do relative to LRU. So we're fixing the algorithm. It's obviously not using the future. When we analyze it, we're assuming we do know the future. We know the entire timeline.

So all the algorithms we covered last time, all the ones we covered today you can think of as just making a sequence of accesses. They're making sequences of accesses to elements. But if we assume we know what b is, that's just a sequence of accesses to blocks. OK so you can just think of the timeline as a sequence of block IDs. And if you access a block that's currently stored in cache, it's free. Otherwise, you pay 1.

All right, so I'm just going to look at the timeline of all these accesses and say, well, take a prefix of the accesses until I get to m over b distinct blocks, block IDs. Keep going until, if I went one more, I'd have m over b plus 1 distinct blocks. So it's a maximal prefix of m over b distinct blocks. Cut there. And then repeat. So start over. Start counting at zero. Extend until I have m over b distinct block accesses. And if I went one more, I'd have m over b plus 1 and so on.

So the timeline gets divided. Who knows? It could be totally irregular. If you access the same blocks many times, you could get along for a very long time and only access m over b distinct blocks. Who knows? The algorithm definitely doesn't know because it doesn't know m or b. But from an analysis perspective, we can just count these things. So each of these has exactly m over b distinct accesses, distinct block IDs.

So I have two claims about such a phase. First claim is that LRU with a cache of size m on one phase is, at most, what? It's easy.

STUDENT: M over b.

PROFESSOR: M over b. The claim is, at most, m over b basically because the LRU is not brain dead. Well, you're accessing these blocks. And they've all been accessed more recently. I mean, let's look at this phase. All the blocks that you touch here have been accessed more recently than whatever came before. That's the definition of this timeline. This is an order by time. So

anything you load in here, you will keep preferentially over the things that are not in the phase because everything in the phase has been accessed more recently.

So maybe, eventually, you load all m over b blocks that are in the phase. Everything else you touch, by definition of a phase, are the same blocks. So they will remain in cache. And that's all it will cost, m over b memory transfers per phase. So this is basically ignoring any carry over from phase to phase. This is a conservative upper bounds. But it's an upper bounds.

And then the other question is, what could OPT do? So OPT-- remember, we're tying its hands behind its back. It only has a cache of size m over 2. And then we're evaluating on a phase. I want to claim that OPT is, well, at least half of that if I want to get a factor of 2. So I claim it's at least 1/2 m over b.

Why? Now we have to think about carry over. So OPT did something in this phase. And then we're wondering what happens in the very next phase. So some of these blocks may be shared with these blocks. We don't know. I mean, there's some set of blocks. We know that this very first block was not in the set, otherwise the phase would have been longer. But maybe some later block happens to repeat some block that's over there. We don't really know. There could be some carry over.

So how lucky could OPT be? At this moment in time, at the beginning of the phase we're looking at, it could be the entire cache has things that we want, has blocks that appear in this phase. That's the maximum carry over, the entire cache. So sort of the best case for OPT is that the entire cache is useful, meaning it contains blocks in the phase that we're interested in-- the phase we're looking at-- at the start of the phase. That's the best case.

But because we gave up only m over 2, that means, at most, one half m over b blocks. This was cache size. This is the number of blocks in the cache. At most, this many blocks will be free, won't cost anything for OPT. But by definition, the phase has m over b distinct blocks. So half of them will be free. The other half, OPT is going to have to load in.

So it's a kind of trivial analysis. It's amazing this proof is so simple. It's all about setting things up right. If you define phases that are good for LRU, then they're also bad for OPT when it has cache at half the size. And so OPT has to pay at least half what LRU is definitely paying. Here we can forget about carry over. Here we're bounding the carry over just by making the cache smaller. That's it. So this is a most twice that. And so we get the theorem.

I mean, changing the cache size could dramatically change the number of cache reads that you have to do or disk reads it you have to do into cache. But in all of the algorithms we will cover, we're giving some bound in terms of m. That bound will always be, at most, some polynomial dependence in m. Usually it's like a 1 over m, 1 over square root of m, 1 over log m, something like that. All of those bounds will only be affected by a constant factor when you change m by a constant factor. So this is good enough for cache oblivious algorithms.

All right, so that's sort of review of why this model is reasonable. LRU is good. So now we're going to talk about two basic problems-- searching for stuff in array, sorting an array in both of these models. We won't be able to do everything cache obliviously today. But they're are all possible. It just takes more time than we have. We'll give you more of a flavor of how these things work. Again, the theme is going to be divide and conquer, my glass class.

So let's say we have n elements. Let's say, for simplicity, we're in the comparison models. So all we can really do with those elements is compare them-- less than, greater than, equal. And let's say we want to do search in the comparison model, which I'll think of as a predecessor search.

So given a new element x, I want to find, what is the previous element? What's the largest element smaller than x in my set? I'm thinking of these n elements as static, let's say. You can generalize everything I say to have insertions and deletions. But let's not worry about that for now. I just want to store them somehow in order to enable search. So any suggestions in external memory model or cache oblivious model? How would you do this?

[STUDENT COUGHS]

This may sound easy, but it's not. But that's OK. You know, I like easy answers, simple answers. There's two simple answers. One is correct, one is wrong. But I like both, so I want to analyze both. Yeah?

STUDENT: Store them sorted in order?

PROFESSOR: Store them sorted in order, good. That's how we'd normally solve this problem. So let's see how it does. I thought I had solution, too, here. But that's OK. Binary search in a sorted array, sort the elements in order. And then to do a query, binary search on it.

So you remember binary search. You've got an array. You start in the middle. Then let's say

the element looking for is way over here. So then we'll go over this way and go there, this way, there, this way, log n time. I mean, binary search is, in a certain sense, a divide and conquer algorithm. You only recurse on one side, but it's divide and conquer. So divide and conquer is good. Surely binary search is good. If only it were that simple.

So sort of orthogonal to this picture-- maybe I'll just draw it on one side-- there's a division into blocks. And in a cache oblivious setting, we don't know where that falls. But the point is, for the most part, every one of these accesses we do as we go farther and farther to the right-- almost all of them will be in a different block. The middle one is very far away from the 3/4 mark.

It is very far away from the 7/8 mark, and so on, up until the very end. Let's say we're searching for the max. So this will hold for all of them. At the end, once we're within a problem of size order b, then there's only a constant number of blocks that we're touching. And so from then on, everything will be free, basically.

So if you think about it carefully, the obvious upper bound-- this is our usual recurrence for binary search-- would be constant. And what we hope to gain here is, basically, a better base case. And I claim that all you get in terms of base case here is t of b equals order 1. And, if you think about it, this just solves to log n minus log b, which is the same thing as log n over b, which is a small improvement over just regular log n but not a big improvement.

I claim we can do better. You've actually seen how to do better. But maybe we didn't tell you. So it's a data structure you've seen already-- b tree, yeah. So because we weren't thinking about this memory hierarchy business when we said b tree, we meant like 2-4 trees or 5-10 trees or some constant bound on the degree of each node.

But if you make the degree of the node b-- or some theta b, b approximate-- so you allow a big branching factor. It's got to be somewhere, let's say, between b over 2 and b. Then we can store all of these pointers and all of these keys in a constant number of blocks. And so if we're doing just search, as we navigate down the b tree we'll spend order 1 block reads to load in this node and then figure out which way to go.

And then let's say it's this way. And then we'll spend order 1 memory transfers to read this node then figure out which way to go. So the cost is going to be proportional to the height of the tree, which is just log base b of n up to the constant factors because we're between b over 2 and b. But that will affect this by a factor of 2.

So we can do search in a b tree in the log base b of n memory transfers. OK, remember, log base b of n is log n divided by log b. So this is a lot better. Here we had log n minus log b. Now we have log n divided by log b. And this turns out to be optimal. In the comparison model, this is the best you can hope to do, so good news.

The bad news is we kind of critically needed to know what b was. B trees really only make sense if you what b is. You need to know the branching factor. So this is not a cache oblivious data structure. But it has other nice things. We can actually do inserts and deletes, as well. So I said static, but if you want dynamic insert and deleting elements, you can also do those in log base b of n memory transfers using exactly the algorithms we've seen with splits and merges.

So all that's good. But I want to do it cache obviously-- just the search for now. And this is not obvious. But it's our good friend van Emde Boas. So despite the name, this is not a data structure that van Emde Boas. But it's inspired by the data structure that we covered.

And it's actually a solution by Harold [INAUDIBLE], who did the m-edge thesis on this work. In the conclusion, it's like, oh, by the way, here's how you do search. It seems like the best page of that thesis. And then I think we called it van Emde Boas because we thought it was reminiscent.

So here's the idea. Take all the items you want to store. And you're really tempted to store them in sorted order, but I'm not going to do that. I'm going to use some other divide and conquer order. First thing I'm going to do is take those elements, put them in a perfectly balanced binary search tree. So this is a BSTT-- not a b tree, just a binary tree because I don't know what b is. So then maybe the median's up here. And then there's two children and so on. OK, the mean's over here. The max is over here, a regular BST.

Now we know how to search in a tree. You just walk down. The big question is, in what order should I store these nodes? If I just store them in a random order, this is going to be super bad-- log n memory transfers. But I claim, if I do a clever order, I can achieve log base b of n, which is optimal.

So van Emde Boas suggests cutting this tree in the middle. Why in the middle? This was n nodes over here. And we're breaking it up into a square root of n nodes at the top because the height of this overall tree is log n. If we split it in half, the height of the tree is half log n. 2 to the half log n is root n. I'm losing some constant factors, but let's just call it root n.

Then we've got, at the bottom, everything looks the same. We're going to have a whole bunch of trees of size square root of n, hopefully. OK, that's what happens when I cut in the middle level. Then I recurse. And what am I recursing? What am I doing? This is a layout.

Last time, we did a very similar thing with matrices. We had an n by n matrix. We divided it into four n over 2 by n over 2 matrices. We recursively laid out the 1/4, wrote those out in order so it was consecutive. Then we laid out the next quarter, next quarter, next quarter. The order of the quarters didn't matter. What mattered is that each quarter of the matrix was stored as a consecutive unit so when recursed, good things happened.

Same thing here, except now I have roughly square root of n plus 1. Chunks, little triangles-- I'm going to recursively lay them out. And then I'm going to concatenate those layouts. So this one, I'm going to recursively figure out what order to store those nodes and then put those all as consecutive in the array. And then this one goes here. This one goes here.

Actually, the order doesn't matter. But you might as well preserve the order. So do the top one, then the bottom ones in order. And so, recursively, each of these ones is going to get cut in the middle. Recursively lay out the top, then the next one. Let's do an example. Let's do an actual tree.

This is actually my favorite diagram to draw or something. My most frequently drawn diagram, complete binary tree on eight children, eight leaves. So this is 15 nodes. It happens to have a height that's a power of 2, so this algorithm works especially well. So I'm going to split it in half, then recursively lay out the top. To lay out the top, I'm going to split it in half. Then I'm going to recursively lay out the top.

Well, single node-- it's pretty clear what order to put it in with respect to itself. So that goes first, then this, then this. Then I finish the first recursion. Next, I'm going to recursively lay out this thing by cutting it in half, laying out the top, then the bottom parts. OK, then I'm going to recursively layout this-- 7, 8, 9, 10, 11, 12, 13, 14, 15.

It gets even more exciting the next level up. But it would take a long time to draw this. But just imagine this repeated. So that would be just the top half of some tree. Cut here, and then you do the same thing here and here and here. This is very different from in-order traversal or any other order that we've seen. This is the van Emde Boas order.

And this numbering is supposed to be the order that I store the nodes. So when I write this into

memory, it's going to look like this-- just the nodes in order. And when I'm drawing a circle-- wow, this is going to get tedious. And then there's pointers here. Every time I draw a circle, there's a left pointer and a right pointer. So 1's going to point to 2 and 3. 2 is going to point to 4 and 7. So just take the regular binary search tree, but store it in this really weird order. I claim this will work really well, log base b of n search.

Let's analyze it. Good first time, this is a cache oblivious layout. I didn't use b at all. There's no b here. Start with a binary tree. And I just do this recursion. It gives me a linear order to put the nodes in. I'm just going to store them in that order. It's linear size, all that.

Now in the analysis, again, I'm allowed to know b. So let's say b is b. And let's consider the level of recursion. Let's say the first level of recursion, where the triangles have less than or equal to b nodes. So I'm thinking of this picture. I cut in the middle.

Then I recursively cut in the middle of all the pieces. Then I recursively cut in the middle. I started out with a height of log n and size n. I keep cutting, basically square rooting the size. At some point, when I cut, I get triangles that are size, at most, square root of b.

So the tree now will look-- actually, let me draw a bigger picture. Let's start down here. So I've got triangle less than or equal to b, triangle less than or equal to b. This is some attempt to draw a general tree.

And first we cut in the middle level. Then we cut in the middle levels. And let's say, at that moment, all of the leftover trees have, at most, b nodes in them. It's going to happen at some point. It's going to happen after roughly log n minus log b levels of recursion. The heights here will be roughly log b. We keep cutting in half, still with height log b. Then we know the size of it's b.

OK, so this is a picture that exists in some sense. What we know is that each of these triangles is stored consecutively. By this recursive layout, we guarantee that, at any level of recursion, each chunk is stored consecutively. So, in particular, this level-- level b-- is nice. So what that tells us is that each triangle with, at most, b elements is consecutive, which means it occupies at most two blocks.

If we're lucky, it's one. But if we're unlucky in terms of-- here's memory. Here's how it's split into blocks. Maybe it's consecutive, but it crosses a block boundary. But the distance between these two lines is b and b. And the length of the blue thing is b. So you can only cross one line.

So you fit in two blocks. Each of these triangles fits in two blocks.

Now, let's think about search algorithm. We're going to do regular binary search in a binary search tree. We start at the root. We compare to x. We go left to right. Then we go left to right, left to right. Eventually we find the predecessor or the successor or, ideally, the element we're actually searching for. And so what we're doing is following some root-to-node path in the tree. Maybe we stop early. In the worst case, we go down to a leaf. But it's a vertical path. You only go down.

Over here, same thing. Let's say, because these are the ones I drew, you go here somewhere. But in general, you're following some root-to-node path. And you're visiting some sequence of triangles. Each triangle fits in, basically, one block. Let's assume, as usual, m over b is at least two. So you can store at least two blocks, which means, once you start touching a triangle, all further touches are free. The first one, you have to pay the load in, maybe these two blocks. Every subsequent touch as you go down this path is free.

Then you go to a new triangle. That could be somewhere completely different. We don't really now, but it's some other two blocks. And as long as you stay within the triangle, it's free. So the cost is going to be, at most, twice the number of triangles that you visit. MTN is going to be, at most, twice the number of triangles visited by a root-to-node path, a downward path in the binary search tree.

OK, now to figure that out we need not only an upper bound on how big the triangles are but also a lower bound. I said the height of the tree is about log b. It's close. Maybe you have triangles of size b plus 1, which is a little bit too big. So let's think about that case. You have b plus 1 nodes. And then you end up cutting in the middle level.

So before, you had a height of almost log b-- slightly more than log b. Then, when you cut it in half, the new heights will be half log b. And then you'll have only square root of b items in the triangle. So that may seem problematic. These things are, at most, b. They're also at least square root b. The height of a triangle at this level is somewhere between half log b and log b.

Basically, we're binary searching on height. We're stopping when we divide it by 2. And we get something less than log B in height. Luckily, we only care about heights. We don't care that there's only root b items here. That may seem inefficient, but because everything's in a log here-- because we only care about log b in the running time, and we're basically approximating log b within a factor of 2-- everything's going to work up to constant factors.

In other words, if you look at this path, we know the length of the path is log n. We know the height of each of these triangles is at least half log b. That means the number of triangles you visit is log n divided by half log b. And the length of the path is log n. So the number of triangles on the path is, at most, log n divided by how much progress we make for each triangle, which is half log b-- also known as 2 times log base b of n.

And then we get the number of memory transfers is, at most, twice that. So the number of memory transfers is going to be, at most, 4 times log base b of n. And that's order log base b of n, which is optimal. Now we don't need to know b. How's that for a cheat?

So we get optimal running time, except for the constant factor. Admittedly, this is not perfect. B trees get basically 1 times log base b of n. This cache oblivious binary search gives you 4 times log base b of n. But this was a rough analysis. You can actually get that down to like 1.4 times log base b of n. And that's tight. So you can't do quite as well with cache oblivious as external memory but close.

And that's sort of the story here. If you ignore constant factors, all is good. In practice, where you potentially win is that, if you designed a b tree for specific b, you're going to do really great for that level of the memory hierarchy. But in reality, there's many levels to the memory hierarchy. They all matter. Cache oblivious is going to win a lot because it's optimal at all levels simultaneously. It's really hard to build a b tree that's optimal for many values of b simultaneously.

OK so that is search. Any questions before we go on to source?

[STUDENTS COUGHING]

One obvious question is, what about dynamic? Again, I said static. Obviously the elements aren't changing here. Just doing search in log base b of n, it turns out you can do insert, delete, and search in log base b of n memory transfers per operation. This was my first result in cache oblivious land. It's when I met Charles Leiserson, actually.

But I'm not going to cover it. If you want to know how, you should take 6851, Advanced Data Structures, which talks about all sorts of things like this but dynamic. It turns out there's a lot more to say about this universe. And I want to go in to sorting instead of talking about how to make that dynamic because, oh, OK, search log base b of n, that was optimal.

I said, oh, you can also do insert and delete in log base b of n. It turns out that's not optimal. It's as good as b trees. But you can do better. B trees are not good at updates. And if you've ever worked with a database, you may know this. If you have a lot of updates, b trees are really slow. They're good for searches, not good for updates. You can do a lot better. And that will be exhibited by sorting.

So sorting-- I think you know the problem. You're given n elements in an array in some arbitrary order. You want to put them into sorted order. Or, equivalently, you want to put them into a van Emde Boas order. Once their sorted, it's not too hard to transfer into this order. So you can do search fast or whatever.

Sorting is a very basic thing we like to do. And the obvious way to sort when you have, basically, a-- let's pretend we have this b tree data structure, cache oblivious even. Or we just use regular b trees. Let's use regular b trees. Forget about cache oblivious. External memory, we know how to do b trees. We know how to insert into a b tree.

So the obvious way to sort is to do n inserts into, if you want, a cache oblivious b tree or just a regular b tree. How long does that take? N times log base b of n. It sounds OK. But it's not optimal. It's actually really slow compared to what you can do. You can do, roughly, a factor of b faster.

But it's the best we know how to do so far. So the goal is to do better. And, basically, what's going on is we can do inserts. In this universe, we can do inserts and deletes faster than we can do searches, which is a little weird. It will become clearer as we go through.

So what's another natural way to sort? What means to sorting algorithm that we've covered are optimal in the comparison model?

STUDENT: Merge sort.

PROFESSOR: Merge sort, that's a good one. We could do quick sort, too, I guess. I'll stick to merge sort. Merge sort's nice because A, it's divide and conquer. And we like divide and conquer. It seems to work, if we do it right. And it's also cache oblivious. There's no b in merge sort. We didn't even know what b was. So great, merge sort is divide and conquer and cache oblivious.

So how much does it cost? Well, let's think about merge sort. You take an array. You divide it in half. That takes zero time. That's just a conceptual thing. You recursively sort this part. You

recursively sort this part. That looks good because those items are consecutive. So that recursion is going to be an honest to goodness recursion on an array. So we can write a recurrence.

And then we have to merge the two parts. So in merge, we take the first element of each guy. We compare them, output one of them, advance that one, compare, output one of them, advance that guy. That's three parallel scans. We're scanning in this array. We're scanning in this array. We're always advancing forward, which means as long as we store the first block of this guy and the first block of this guy who knows how it's aligned--

But we'll read these items one by one until we finish that block. Then we'll just read the next block, read those one by one. And similarly for the output array, we first start filling a block. Once it's filled, we can kick that one out and read the next one. As long as m over b is at least 3, we can afford this three-parallel scan. It's not really parallel. It's more like inter-leaf scans. But we're basically scanning in here while we're also scanning in here and scanning in the output array.

And we can merge two sorted arrays into a new sorted array in scan time, n over be plus 1. So that means the number of memory transfers is 2 times the number of memory transfers for half the size, like regular merge sort, plus n over b plus 1. That's our recurrence. Now we just need to solve it.

Well, before we solve it, in this case we always have to be careful with the base case. Base case is MT of m. This is the best base case we could use. Let's use it. When I reach an array of size m, I read the whole thing. And then that's all I can pay. So I won't incur any more cost as long as I stay within that region of size m. Maybe I should put some constant times m because this is not in place algorithm, so maybe 1/3 m something. As long as I'm not too close to the cache size, I will only pay m over b memory transfers. So far so good.

Now we just solve the recurrence. This is a nice recurrence, very similar to the old merge-sort recurrence. We just have a different thing in the additive term. And we have a different base case. The way I like to solve nice recurrences is with recursion trees. This is actually a trick I learned by teaching this class. Before this, cache oblivious was really painful to me because I could never solve the currencies. Then I thought the class and was like, oh, this is easy. I hope the same transformation happens to you.

You'll see how easy it is once we do this example. OK, this is merge sort. Remember recursion

tree, in every node you put the additive cost so that, if you added up the cost of all of these nodes, you would get the total value of this expands to because we're basically making two children of size n over 2.

And then we're putting, at the root, this cost, which means, if you add up all of these nodes, you're getting all of these costs. And that's the total cost. So it's n over b at the top. Then it's going to be n over 2 divided by b and so on. I'm omitting the plus 1 just for cleanliness. You'd actually have to count.

And this keeps going until we hit the base case. This is where things are a little different from regular merge sort, other than the divided by b. We stop when we reach something at size m. So at the leaf level, we have something of size m, which means we basically have m over b in each leaf. And then we should think about how many leaves there are. This is just n over m leaves, I guess.

There's lots of ways to see that. One way to think about it is we're conserving mass. We started with n items. Split it in half, split it in half. So the number of items is remaining fixed. Then at the bottom we have m items. And so the number of leaves has to be exactly n over m because the total should be n.

You can also think of it as 2 to the power log of that. We have, usually, log n levels. But we're cutting off a log m at the bottom. So it's log n minus log m as the height. I'll actually need that. The height of this tree is log n minus log m, also known as log n/m.

OK, so we've drawn this tree. Now, what we usually do is add up level by level. That usually gives a very clean answer. So we add up the top level. That's n over b. We add up the second level. That's n over b, by conservation of mass again and because this was a linear function. So each level, in fact, is going to be exactly n over b cost.

We should be a little careful about the bottom because the base case-- I mean, it happens that the base case matches this. But it's always good practice to think about the leaf level separately. But the leaf level is just m over b times n over m The m's cancel, so m over b times n over m. This is n over b. So every level is n over b. The number of levels is log of n over m. Cool. So the number of memory transfers is just the product of those two things. It's n over b times that log, log n over m.

Now let's compare. That's sorting. Over here, we had a running time of n times log base b of

n. So this is n log n divided by log b. Log base b is the same as dividing by log b. So n log n divided by log-- we had regular sorting time. And then we divided by log b.

Over here, we have basically regular sorting time. But now we're dividing by b. That's a huge improvement-- a b divided by log b improvement. I mean, think of the b being like a million. So before we were dividing by 20, which is OK. But now we're dividing by a million. That's better. So this way of sorting is so much better than this way of sorting. It's still not optimal, but we're getting better.

We can actually get sort of the best of both worlds-- divide by b and divide by log b, I claim. But we need a new algorithm. Any suggestions for another algorithm?

STUDENT: Divide into block size b.

PROFESSOR: I want to divide into block size b. So, you mean a merge sort?

STUDENT: Yes.

PROFESSOR: So merge sort, I take my array. I divide it into blocks the size b. I could sort each one in one memory transfer. And then I need to merge them. So then I've got n divided by b sorted arrays. I don't know how to merge them. It's going to be hard, but very close.

So the answer is indeed merge sort. What we covered before is binary merge sort. You split into two groups. What I want to do now is split into some other number of groups. So that was n over b groups. That's too many because merging n over b arrays is hard. Merging two arrays was easy. Assuming m over b was at least 3, I could merge these guys just by parallel scans. So you have the right bound?

STUDENT: B way.

PROFESSOR: B way, maybe.

STUDENT: Square root of b.

PROFESSOR: Square root of b? That's what I like to call root beer.

[LAUGHTER]

Nope. I do call it that. Yeah?

STUDENT: M over b?

PROFESSOR: M over b, that's what I'm looking for! Why m over b?

STUDENT: I was just thinking of the bottom layer of the [INAUDIBLE] binary merge sort.

PROFESSOR: Because m over b is up here? Nice.

[LAUGHTER]

Not the right reason, but you get a Frisbee anyway. All right, let's see if I can do this. Would you like another one? Add to your collection.

All right, so m over b is the right answer-- wrong reason, but that's OK. It all comes down to this merge step. So m over b way means I take my problem of size n. Let's draw it out. I divide into chunks. I want the number of chunks that I divide into to be m over b, meaning each of these has size n over m over b. That's weird. This is natural because this is how many blocks I can have in cache.

I care about that because, if I want to do a multi-way merge, you can mimic the same binary merge. You look at the first item of each of the sorted arrays. You compare them. In this model, comparisons are free. Let's not even worry about it. In reality, use a priority queue, but all right. So you find the minimum of these. Let's say it's this one. And you output that, and then you advance.

Same algorithm, that will merge however many arrays you have. The issue is, for this to be efficient like it was here, we need to be able to store the first block of each of these arrays. How many blocks we have room for? M over b. This is maxing out merge sort. This is exactly the number of blocks that we can store. And so if we do m over b way merge sort, merge remains cheap.

An m over b way merge costs n over b plus 1, just like before. It's m over b parallel scans. M over b is exactly the number of scans we can handle. OK, technically we have, with this picture, m over b plus 1 scans. So I need to write m over b minus 1. But it won't make a difference.

OK, so let's write down the recurrence. It's pretty similar.

Memory transfer's size m. We have m over b sub problems of size n divided by m over b. It's Still conservation of mass. And then we have plus the same thing as before, n over b plus 1. So it's exactly the same recurrence we had before. We're splitting into more problems. But the sums are going to be the same. It's still going to add up to n over b at each step because conservation of mass. And we didn't change this. So level by level looks the same. The only thing that changes is the number of levels.

Now we're taking n. We're dividing by m over b in each step. So the height of the tree, the number of levels of the recursion tree now is-- before it was log base 2 of n over n. Now it's going to be log base m over b of n over m. If you're careful, I guess there's a plus 1 for the leaf level.

I actually want to mention this plus 1. Unlike the other plus 1's, I've got to mention this one because this is not how I usually think of the number of levels. I'll show you why. If you just change it by one, you get a slightly cleaner formula. This has got m's all over the place. So I just want to rewrite n over m here. Then we'll see how good this is. This is just pedantics. Log base m over b of n-- I really want n over b. To make this n over b, I need to multiply by b, divide by m. OK, these are the same thing. M over m, b's cancel.

But I have a log of a product. I can separate that out. Let's go over here. This is log base m over b of n over b-- this is what I like-- and then, basically, minus log base m over b of m over b.

STUDENT: It's b over m.

PROFESSOR: I put a minus, so it's m over b. If I put a plus, it would be b over m. But, in fact, m is bigger than b. So I want it this way. And now it's obvious this is 1. So these cancel. So that's why I wanted the 1, just to get rid of that. It doesn't really matter, just a plus 1. But it's a cooler way to see that, in some sense, this is the right answer of the height of the tree.

Now, we're paying n over b at each recursive level. So the total cost is what's called the sorting bound. This is optimal, n over b times log base m over b of n over b. Oh my gosh, what a mouthful. But every person who does external memory algorithms and cache oblivious algorithms knows this. It is the truth, it turns out. There's a matching lower bound.

It's a weird bound. But let's compare it to what we know. So we started out with n log n divided by log b. Then we got n log n divided by b. Let's ignore-- I mean, this has almost no effect, the

part in here. Now we have n log n divided by b and divided by log m over b. It's not quite dividing by log b. But it turns out it's almost always the same.

In some sense, this could be better. If you're cache is big, now you're dividing by log m, roughly. Before, you were only dividing by log b. And it turns out this is the best you can do. So this is going to be a little bit better than merge sort. If your cache is 16 gigabytes, like your RAM caching your disk, then log m is pretty big. It's going to be 32 or something, 34 I guess log m.

OK, I have to divide by b. So it's not that good. But still, I'm getting an improvement over regular binary merge sort. And you would see that improvement. These are big factors. The big thing, of course, is dividing it by b. But dividing by log of m over b is also nice and the best you can do.

OK, obviously I needed to know what m and b were here. So the natural question next is cache oblivious sorting. And that would take another lecture to cover. So I'm not going to do it here. But it can be done. Cache obliviously, you can achieve the same thing. And I'll give you the intuition.

There's one catch. Let me mention the catch. So cache oblivious sorting-- to do optimal cache oblivious sorting like that bound, it turns out you need an assumption called the tall-cache assumption. Simple form of the tall-cache assumption is that m is at least b squared. What that means is m over b is at least b.

In other words, the cache is taller than it is wide, the way I've been drawing it. That's why it's called the tall-cache assumption. And if you look at real caches, this is usually the case. I don't know of a great reason why it should be the case. But it usually is, so all is well. You can do cache oblivious sorting. It turns out, if you don't have this assumption, you cannot achieve this bound. We don't know what bound you can achieve. But we just know this one is not possible. You can get a contradiction if you achieve that without tall cache.

So it's a little bit weird. You have to make one bonus assumption. You can make a somewhat weaker form of it, which is m is omega b to the 1.000000001. That will do. In general, 1 plus epsilon. Any epsilon will be fine. We just mean that the number of blocks is at least some b to the epsilon, where epsilon's a constant bigger than zero.

OK, then you can do cache oblivious sorting. Let me tell you how. We want to do m over b way

merge sort. But we don't know how to do-- we don't know what m over b is. So instead, we're going to do something like n to the epsilon way merge sort. That's a so-so interpretation. This is back to your idea roughly. We're dividing into a lot of chunks.

And then we don't know how to merge them anymore because we can't do regular merge with n to the epsilon chunks it could be n to the epsilon's too big. So how do we do it? We do a divide and conquer merge. This is actually called funnel sort because the way you do a divide and conquer merge looks kind of like a funnel.

Actually, it looks a lot like the triangles we were drawing earlier. It's just a lot messier to analyze. So I'm not going to do it here. It would take another 40 minutes or so. But that's some intuition of how you do cache oblivious merge sort. That's what I want to say cache oblivious stuff. Oh, one more thing!

One more cool thing you can do-- I'm a data structures guy. So sorting is nice. But what I really like are priority queues because they're more general than sorting. We started out by saying, hey, look. If you want to sort and you use a b tree, you get a really bad running time. That's weird because usually BST sort is good in a regular comparison model. It's n log n.

So b trees are clearly not what we want. Is there some other thing we want? And it turns out, yes. You can build a priority queue, which supports insert and delete min and a bunch of other operations. Each of those operations costs 1 over b log base m over b of n over b amortized memory transfers-- a bit of a mouthful again.

But if you compare this bound with this bound, it's exactly the same. But I divided by n, which means if I insert with this cost n times, I pay the sorting bound. If I delete min with this bound n times, I get the sorting bound. So if I insert n times and then delete all the items out, I've sorted the items in sorting bound time. So this is the data structure generalization of that sorting algorithm.

Now, this is even harder to do. Originally, it was done external memory. It's called buffer trees. Then we did it cache obliviously. It's called cache oblivious priority queues. We weren't very creative. But it can be done. And, again, if you want to learn more, you should take 6851, Advanced Data Structures, which leads us into the next topic, what class you should take next-- classes, that's what I mean to say.

So a lot of bias here. And well I'm just going to give a lot of classes. There's a lot of them. I

believe this is in roughly numerical order almost. It changed a little bit-- so many classes. Are you OK with numbers, or do you want titles?

The obvious follow-on course to this class is 6854, which is Advanced Algorithms. It's the first graduate algorithms class. This is the last undergraduate class, roughly speaking, with the exception of 6047. But in terms of straight, general algorithms, this would be the natural class. It's only in the fall-- sadly, not next fall. But in general, it's a cool class. It's a very broad overview of algorithms but much more hard core, I guess. It's an intense class but covers a lot of fields, a lot of areas of algorithms.

Then all the other ones I'm going to list are more specialized. So 6047 is Computational Biology. So if you're interested in biology, you want algorithms applied to biology. That's a cool class. It's also an undergrad class. Everything else here-- I mean, you know the story. You take grad classes all the time, or you will soon if you want to do more algorithms.

So 6850 is computational geometry. I think it's called Geometric Algorithms. So we've seen a couple examples, like the convex hull divide-and-conquer algorithm and the range trees. Those are two examples of geometric algorithms where you have points and lines and stuff-- maybe in two dimensions, maybe in three dimensions, maybe log n dimensions. If you like that stuff, you should take computational geometry. This is the devil that lad me into algorithms in the first place. Cool stuff.

6849 is my class on folding algorithms. This is a special type of geometric algorithms where we think about paper folding and robotic arm folding and protein folding and things like that. So that's a bit of a specialized class. 6851, I've mentioned three times now-- Advanced Data Structures. Then we've got 6852, its neighbor. This is Nancy's Distributed Algorithms class. So if you liked the week of distributed algorithms, there's a whole class on it. She wrote the textbook for it.

Then there's 6853. This is Algorithmic Game Theory. If you care about algorithms involving multiple players-- and the players are each selfish. And they have no reason to tell you the truth. And still you want to compute something like minimum spanning tree, or pick your favorite thing. Everyone's lying about the edge weights. And still you want to figure out how to design a mechanism like an auction so that you actually end up buying a minimum spanning tree. You can do that. And if you want to know how, you should take 6853.

What else do we have? 6855 is Network Optimization. So this is like the natural follow on of

network flows. If you like network flows and things like that, there's a whole universe called network optimization. It has lots of fancy, basically, graph algorithms where you're minimizing or maximizing something. OK, this is fortuitous alignment. 6856 is kind of a friend of 6854. These are both taught by David Carter.

This is Randomized Algorithms. So this is a more specialized approach. I don't think you need one to take the other. But this is the usual starting class. And this is specifically about how randomization makes algorithms faster or simpler. Usually they're harder to analyze. But you get very simple algorithms that run just as well as their deterministic versions. Sometimes you can do even better than the deterministic versions.

Then there's the security universe. This is a great numerical coincidence-- probably not a coincidence. But there's 6857 and 6875. I have to remember which is which. 6857 is Applied Cryptography. 6875 is Theoretical Cryptography, at least as I read it. So they have similar topics. But this is more thinking about how you really achieve security and crypto systems and things like that. And this one is more algorithm based. And what kind of theoretical assumptions do you need to prove certain things? This is more proof based. And this is more connecting to systems, both great topics.

And I have one more out of order, I guess just because it's a recent addition. 6816 is Multicore Programming. That has a lot of algorithms, too. And this is all about parallel computation. When you have multiple cores on your computer, how can you compute things like these things faster than everything we've done? It's yet another universe that we haven't even touched on in this class. But it's cool stuff, and you might consider it.

Then we move on to other theory classes. That was algorithms. Some more obvious candidates, if you like pure theory, are 6045 6840. This is the undergrad version. This is the grad version. Although, by now the classes are quite different. So they cover different things. Some of you are already taking 6045. It's right before this lecture.

These are general theory of computation classes, atomita, complexity, things like that. If you like the brief NP completeness lecture, then you might like this stuff. There's so many more complexity classes and other cool things you can do.

If you really like it, there's advanced complexity theory. There's, basically, randomized complexity theory-- how randomness affects just the complexity side, not algorithms. Then there's quantum complexity theory if you care about quantum computers. As Scott says, it's

proving things you can't do with computers we don't have.

[LAUGHTER]

It's complexity. It's all about lower bounds. And then there's coding theory, which is another universe. It's actually closely related to-- it comes out of signals and systems and electrical engineering. But by now it's closely related to complexity theory. You can use bounds on codes to prove things about complexity theory. Anyway, choose your own adventure.

Now I have one last topic, which was not on the outline. This is a bit of a surprise. It's a boring surprise. I want to remind you to fill out student evaluations.

[LAUGHTER]

Because, you know, we want to know how we did and how we can continue to improve the class. But really we want to know who's the better teacher. But more importantly than who is the better teacher, I think we all have a dying question, which is who is the better Frisbee thrower? So I want to invite Srini Devadas, our co-lecturer here, to a duel.

[LAUGHTER AND APPLAUSE]

SRINI DEVADAS: I think you mean, no contest.

[LAUGHTER]

PROFESSOR: Not so sure. Maybe-- actually, I'm pretty sure.

[LAUGHTER]

I want to take you on, man. Blue or purple?

SRINI DEVADAS: Blue.

PROFESSOR: Good choice.

SRINI DEVADAS: Blue's better, remember?

[LAUGHTER]

PROFESSOR: Purple's better, remember?

[LAUGHTER]

All right, so how are we going to do this?

SRINI DEVADAS: So, you guys get to cheer and bet.

PROFESSOR: Bet? I don't think we can condone them betting money. I think maybe they can bet their Frisbees. Anyone got a Frisbee on them? We can bet those.

SRINI DEVADAS: Yeah, all right.

PROFESSOR: All right, maybe not.

SRINI DEVADAS: Put your Frisbees on me here.

PROFESSOR: All right.

SRINI DEVADAS: All right, so some rules here-- we actually talked about this. So the way this is going to work-- I mean, it's going to be algorithmic, obviously. And we get to choose our algorithm, maybe with a little game theory here. We're going to toss the coin. And we're going to decide who goes first. So won't spin the Frisbee. Remember what happened with that? So you get to call heads or tails while it's spinning.

PROFESSOR: Oh, while it's spinning.

SRINI DEVADAS: While it's spinning. This is our Super Bowl.

PROFESSOR: Heads! Oh, a trick. Tails.

SRINI DEVADAS: Tails, all right. You're going to throw first.

PROFESSOR: OK, that's your choice.

SRINI DEVADAS: And I don't know if you've heard the legend of William Tell. How many of you have heard the legend of William Tell? All right. So that was a 14th century Swiss legend where there was this archer who was renowned for his skill. And he was forced by this villainous king to shoot an

apple off the top of his son's head.

PROFESSOR: Yikes.

SRINI DEVADAS: So we're going to reenact that.

[LAUGHTER]

PROFESSOR: Did you bring your daughter?

[LAUGHTER AND APPLAUSE]

SRINI DEVADAS: I was thinking TAs.

PROFESSOR: Our "sons."

SRINI DEVADAS: But there's a big difference between the 21st century and the 14th century. What is that?

STUDENT: You get sued.

[INTERPOSING VOICES]

PROFESSOR: You get sued, yeah.

SRINI DEVADAS: Now there's many more lawsuits in the 21st century. So we want to avoid lawsuits.

STUDENT: Genetically modified apples.

PROFESSOR: And genetically modified apples, also.

SRINI DEVADAS: Electronically modified Apples, yeah that's going to be another difference. So we decided we'd just throw Frisbees at each other.

PROFESSOR: So I'm going to throw to you and try to hit an apple off of your head.

SRINI DEVADAS: Yeah, well you might want to tell them what we decided about the apple.

PROFESSOR: Well, I brought an easy-to-hit apple, a nice big apple, the cowboy hat.

SRINI DEVADAS: Cowboy hat.

PROFESSOR: That should be a little easier.

SRINI DEVADAS: So I get to wear that had first because you're going to throw first.

PROFESSOR: OK.

SRINI DEVADAS: And this is really simple, guys. Knock the hat off, I guess from the furthest distance, and win. In your case, lose but yeah.

PROFESSOR: Now for the PETA people in the audience, I want your assure no humans will be harmed during this performance, only professors.

[LAUGHTER]

And maybe egos, pride.

SRINI DEVADAS: Seven. I think seven is a good number.

PROFESSOR: Seven for--

SRINI DEVADAS: You get to--

PROFESSOR: I'm going to grab purple.

SRINI DEVADAS: You get to pick. You can stand right here. That's probably what's good for you.

[LAUGHTER]

Or you can go all the way up there. And after you knock this hat off, I'm going to have to match you.

PROFESSOR: All right.

SRINI DEVADAS: So furthest away wins.

PROFESSOR: I think I'll try from about here.

SRINI DEVADAS: Right. Ah! I've got to look good here, man.

PROFESSOR: I have to do it without looking.

SRINI DEVADAS: I'm going to stand right here.

PROFESSOR: OK.

SRINI DEVADAS: OK. Ah!

[LAUGHTER]

I've got to gear up for this.

PROFESSOR: OK.

SRINI DEVADAS: Look, I know how well you throw.

PROFESSOR: Are you scared? That's embarrassing.

SRINI DEVADAS: Ah! I can't deal with this. I just have no confidence in the way you throw. So I borrowed this.

[LAUGHTER]

Since this is mine, it's going to cost you a throw to wear this.

PROFESSOR: Hey!

SRINI DEVADAS: No, all right. Whatever, fair. All right. Now I'm feeling much better about this. I won't claim to have a pretty face. But I like it just the way it is, just the way it is.

PROFESSOR: All right. Well, since we have a little more protection, maybe I'll start from farther back.

SRINI DEVADAS: All right. I think I'll hold this up here like that.

PROFESSOR: OK. So I just have to hit the hat off, right? Easy.

SRINI DEVADAS: Yup.

[LAUGHTER]

PROFESSOR: You can keep that. Oh right, so I'm going to get closer. One step closer. Luckily, my steps are much bigger than your steps. OK, throw 2.

[LAUGHTER]

Throw three.

[LAUGHTER]

Throw four.

SRINI DEVADAS: That didn't even hurt. Come on, man! Throw a little harder here!

PROFESSOR: Oh!

[CHEERING AND APPLAUSE]

SRINI DEVADAS: All right, mark it.

PROFESSOR: All right, I marked my spot.

SRINI DEVADAS: You've got a couple more throws here to do better.

PROFESSOR: More throws?

SRINI DEVADAS: You can do better.

PROFESSOR: Wait, to go back.

SRINI DEVADAS: You've got a couple more, yeah.

PROFESSOR: You almost hit me.

[LAUGHTER]

[GROANING]

I'm getting better! Not much-- can I go back now?

SRINI DEVADAS: Yeah, sure.

PROFESSOR: I don't know the rules.

[APPLAUSE]

SRINI DEVADAS: Goodness.

PROFESSOR: No contest, right?

SRINI DEVADAS: I'm getting a little worried here.

PROFESSOR: One more Frisbee. Two more Frisbees. Hey, that was your Frisbee.

SRINI DEVADAS: One more. All right.

PROFESSOR: I think we've done binary search here.

[APPLAUSE]

SRINI DEVADAS: You need this. And you don't really need this, but I'll give it to you.

PROFESSOR: So much confidence. Well, I have so much confidence in you I brought some extra Frisbees.

[LAUGHTER]

SRINI DEVADAS: I get to use all of them, huh?

PROFESSOR: You need it, man.

SRINI DEVADAS: All right. No, we're going to be fair. You threw seven. I'm going to throw seven. 1, 2, 3, 4, 5, 6, 7. You've got a bit of a big head here.

PROFESSOR: Do this. All right, so I put the hat on the head. OK. Where were you standing, by the way? Way back here, right?

SRINI DEVADAS: No, nope. I was right there.

PROFESSOR: OK.

SRINI DEVADAS: Right there.

PROFESSOR: All right, right here. Can I hold onto the hat?

SRINI DEVADAS: No! You can hold on to your helmet!

PROFESSOR: All right.

SRINI DEVADAS: Wow.

PROFESSOR: Ah!

[LAUGHTER]

How many throws--

SRINI DEVADAS: Maybe I should start from right here.

[GROANING]

PROFESSOR: Phew. That was close.

SRINI DEVADAS: Oh, I grazed it! But it's supposed to fall off.

PROFESSOR: What, my head?

[LAUGHTER]

SRINI DEVADAS: Getting kind of tight here, guys. Wow.

[YELLING AND GROANING]

Does that count?

STUDENT: No!

SRINI DEVADAS: It does count.

STUDENT: No!

SRINI DEVADAS: It's a tie. So far, it's a tie. So far, it's a tie! All right, if I knock it off, I win.

[LAUGHTER]

[GROANING]

There you you.

PROFESSOR: Is that it?

[APPLAUSE]

SRINI DEVADAS: This was fair and square. We want the world to know that we did not deflate these Frisbees.

[LAUGHTER]

[APPLAUSE]

So not only did we do a bad job of throwing Frisbee to you guys, we didn't throw enough Frisbees, as you can see, through the term. So if you want a Frisbee, pick one up. And if you're embarrassed about throwing Frisbees with this lettering on it, I've got two words for you-- paint remover. All right, have a good summer. And have fun on the final exam.

[APPLAUSE]