ERIK DEMAINE: Yeah. I'm going to talk about I/O models. Just to get a sense, how many people know a model called I/O model? And how many people don't?

It doesn't matter. I'm just curious. As some of you may know, I/O models have a really rich history. And they're pretty fascinating.

They are all central to this problem of modeling the memory hierarchy in a computer. We have things like RAM model of computation where you can access anything at the same price in your memory. But the reality of computers is you have things that are very close to you that are very cheap to access, and you have things that are very far from you that are big. You can get 3 terabyte disks these days, but are very slow to access.

And one of the big costs there is latency. Because here, the head has to move to the right position, and then you can read lots of data really fast. The disk actually can give you data very fast, but the hard part is getting started in reading stuff.

And so this is the sort of thing we want to model. These kinds of computers have been around for decades, as we'll see. And people have been trying to model them in as clean a way as possible that works well theoretically and matches practice in some ways.

I have just some fun additions to this slide. You can keep getting bigger, go to the internet, get to an exa- or a zettabyte. You have to look up all the words for these. In the universe, you've got about 10 to the 83 atoms, so maybe roughly that many bits. But I don't know if there's a letter for them.

So how do we model this? Well, there's a lot of models. This is a partial list. These are sort of the core models that were around, let's say, since this millennium.

So we start in 1972 and work our way forward. And I'm going to go through all of these in different levels of detail. There's a couple of key features in a cache that we want to model or maybe a few key features.

And then there's some measure of simplicity, which is a little hard to define. The goal is to get all four of these things at once. And we get that more or less by the end.

So first section is on this idealized two-level storage, which was introduced by Bob Floyd in 1972. This is what the first page of the paper looks like. It's probably typeset on a typewriter it looks like and underline, good old days of computer science, very early days of computer science.

And this was published in a conference called The Complexity of Computer Computations. How many people have heard of that conference? No one. Wow. There it is.

It's a kind of a classic, because it had Karp's original paper on NP-completeness. So you've definitely read this paper. But there are a lot of neat papers in there and a panel discussion including what should we call algorithms, which is kind of a fun read.

So this is in the day when one of the state of the art computers was the PDP-11. This is what PDP-11, or one of them, looks like by probably owned by Bell Labs. But Dennis Ritchie and Ken Thompson's the inventors of C and Unix, working away there.

It has disks, each of which is about 2 megabytes in capacity. And it has internal memory which was core memory at the time. So each of these is a little circular magnetic core. And it stores 1 bit. And in total, there are 8 kilobytes.

So you get a sense of already this being an issue. And this is why they wrote their paper. So here's the model they introduced, a very simple model, maybe the simplest we'll see.

You have your CPU, which can do local computation. And then you have your memory, which is very big. But in particular, it's divided into these blocks of size B. So each block can have up to B items.

And what you're allowed to do in one block operation is read two of the blocks. You can read all the items in the block. So let's say you read these two items. You pick some subset of those items to pick up.

And then what you're allowed to do is store them somewhere else. So you can pick some other target block like this one and copy those elements to overwrite that block. I mean, there's no computation in this model, because he was just interested in how you can permute items in that world.

So simple model, but you get the idea. You can read two blocks, take up to B items out of them, stick them in here. Here, we just ignore what the order is within a block, because we're

assuming you can just rearrange once you read them in and spit them out.

So don't worry about the order within the block. It's more for every item, which block is it in? And we're assuming here items are indivisible.

So here's the main theorem of that paper. If you're given N items and you want to permute them into N over B blocks, which means each of those blocks is going to be full-- let's say that's sort of the most interesting case-- then you need to use N over B log B block operations even for a random permutation on average with high probability. So this is kind of nice or kind of interesting, because just to touch those blocks requires N over B block operations.

But there's an extra log factor that starts to creep up, which is maybe a little bit surprising, less surprising to people who are familiar with I/O models, but at the time, very new. And I'm making a particular assumption here, but just a small thing. I thought I'd go through the proof of this theorem, because it's fairly simple.

It's going to use a slightly simplified model where, instead of copying items, you actually move items. So these guys would disappear after you put them in this new block. Because we're thinking about permutation problems, again, that doesn't really change anything. You can just, for every item, see what path it follows to ultimately get to its target location, throw away all the extra copies and just keep that one set of copies. And that will still be a valid solution in this model.

So how does the lower bound go? It's a simple potential argument. You look at for every pair of blocks, how many items are there in block i that are destined for block j?

You want to move from block i to block j. This is going to be changing over time. This is where they currently are.

So that's nij. You take and nij, log nij, and sum that up over all i's and j's. That's the potential function. And our goal is to maximize that potential. Because it's going to be-- for those familiar with entropy-- negative entropy.

So it's going to be maximized when all the items are where they need to be. This is when everything is as clustered as possible. You can only have a cluster of size B, because items can only be up to B in the same place.

One way to see this, in the target configuration, nii is B for all i. Everyone's where they're

supposed to be. And so that potential gives you the number of items times log B. And this is always, at most, log B. And so that's the biggest this could ever hope to get.

So our goal is to increase entropy as much as possible. And we're starting with low entropy. If you take a random permutation, you're trying to get the expected number of guys that are where they're supposed to be. It's very small, because most of them are going to be destined for some other block.

So we're starting with the potential of linear. We need to get to N log B. And then the claim is that each block operation we do can only increase potential by, at most, B. And so that gives us this bound of the potential we need to get to minus the potential we had divided by how much we can decrease potential in each step, which is basically N over B log B minus a little O.

Why is this claim true? I'll just sketch. The idea is this fun fact, the x plus y log x plus y is, at most, x log x plus y log y plus x plus y. What this means is if you have two clusters, our goal is to sort of cluster things together and make bigger groups that are in the same place or in the correct place.

So if you have two clusters x log x and y log y contributing to this thing and you merge them, then you now have this potential. And the claim is that could have only gone up by x plus y. And when you're moving B items, the total number of things you're moving is B. So you can only increase things by B. So it was a quick sketch of this old paper. It's a fun read, quite clear, easy argument.

So we proved this theorem that you need at least N over B log B. But what is the right answer? There's actually not a matching upper bound. Of course, for B at constant, this is the right answer. It's N, but that's not so exciting.

On the upper bound side, this paper has almost matching lower bound. It's another log, but not quite the same log, N over B log N over B instead of log B. And the rough idea of how to do that--

AUDIENCE: [INAUDIBLE]

ERIK DEMAINE: Yeah, question.

AUDIENCE: [INAUDIBLE]

ERIK DEMAINE: I said a tall disk assumption. I'm assuming N over B is greater than B. The number of blocks in your disk is at least the size of a block.

AUDIENCE: You needed that in the proof?

ERIK DEMAINE: I needed that in the proof I think. Good question, Where N over B log B.

AUDIENCE: [INAUDIBLE]

ERIK DEMAINE: Yeah. Exactly. Yeah, that's where I'm using it. Thanks. Otherwise this expectation doesn't work out. I mean, if you have one block, for example, this will fail, because you need zero operations. So there has to be some trade off at the very small regime.

OK. So the way to get N over B log N over B is basically a radix sort. In one pass through the data, you can rewrite everything to have the lower order bits of 0 before all the lower order bits of 1. So in N over B, you can sort by each bit in the target block ID of every item.

And so you do log of N over B things, because that's how many blocks there are. And so this is how many passes you need by a binary radix sort. You can achieve that bound.

And the paper actually claims that there's a lower bound. It's a little strange, because there's a careful proof given for this. And then this claim just says, "by information theoretic consideration--" this is also true. This is in the days when we didn't distinguish between big O and big omega before [INAUDIBLE] paper.

But this is not true. And we'll see that it's not true. It was settled about 14 years later. So we'll see the right answer.

This is almost the right answer, but it doesn't quite work when B is very small. And one way to see that is when B is 1. When B is 1, the right answer is N, not N log N. So when B is less than log N over B, then there's a slightly different answer which we'll get to later. But that was the early days.

There's some other fun quotes from this paper foreshadowing different things. One is the word RAM model, which is very common today, but not at the time. And it says, obviously, these results apply for distant drums, which was probably what they were thinking about originally, but also when the pages, the blocks, are words of internal memory and the records are the bits in those words.

So this is a word RAM model. Here, I said just ignore the permutation within each block. But you can actually do all the things you need to do for these algorithms using shifts and logical or, xor, and operations.

So all these algorithms work in the word RAM model, too, which is kind of nifty. Another thing is foreshadowing, what we call the I/O model, which we'll get to in a little bit. It says, "work is in progress." He got scooped, unfortunately.

"Work is in progress--" unless he meant by someone else-- "attempting to study the case where you can store more than two pages." Basically, this CPU can hold two of these blocks, and then write one back out, but has no bigger memory than that. or bigger cache. So that's where we were at the time.

Next, chapter in this story is 1981. It's a good year. It was when I was born. And this is Hong and Kung's paper. You've probably heard about the red-blue pebble game.

And it's also a two-level model, but now there's a cache in the middle. And you can remember stuff for a while. I mean, you can remember up to M things before you have to kick them out.

The difference here is there's no blocks anymore. It's just items. So let me tell you a little bit about the paper. This was the state of the art in computing at the time.

The personal computer revolution was happening. They had the Apple II, TRS-80, VIC-20. All of these originally had about 4 kilobytes of RAM. And the disks could store maybe, I don't know, 360 kilobytes or so. But you could also connect a tape and other crazy things.

So, again, this was relevant. And that's the setting they were writing this. They have this fun quote.

"When a large computation is performed on a small device--" at that point, small devices were becoming common-- "you must decompose those computations to subcomputations." This is going to require a lot of I/O. It's going to be slow. So how do we minimize I/O?

So their model-- before I get to this red-blue pebble game model, it's based on a vanilla single color pebble game model by a Hopcroft, Paul, and Valiant. This is the famous interrelation between the time hierarchy and space hierarchy paper. And what they said is, OK, let's think of the algorithm we're executing as a DAG.

We start with some things that are inputs. And we want to compute stuff that this computation depends on having these two values and so on. In the end, we want to compute some outputs. So you can rewrite computation in this kind of DAG form.

And we're going to model the execution of that by playing this pebble game. And so a node can have pebbles on it. And for example, we could put a pebble on this node.

In general, we are allowed to put a pebble on a node if all of its predecessors have a pebble. And pebble is going to correspond to being in memory. And we can also throw away a node, because we can just forget stuff. Unlike real life, you can just forget whatever you don't want to know any more.

So you add a pebble. Let's say, now we can add this pebble, because its predecessor has a pebble on it. We can add this pebble over here, add this pebble here. Now, we don't need this information anymore, because we've computed all the things out of it. So we can choose to remove that pebble.

And now, we can add this one, remove that one, add this one. You can check that I got all these right, add this one, remove that one, remove, add, remove, remove. In the end, we want pebbles on the outputs. We start with pebbles on the inputs.

And in this case, their goal was to minimize the maximum number of pebbles over time. Here, there's up to four pebbles at any one moment. That means you need memory of size four. And they ended up proving that any DAG can be executed using N over log N maximum pebbles, which gave this theorem time.

If you use t units of time, you can fit in t over log t units of space, which was a neat advance. But that's beside the point. This is where Hong and Kung were coming from. They had this pebble model. And they wanted to use two colors of pebbles, one to represent the shallower level of the memory hierarchy in cache, and the other to say that you're on disk somewhere.

So red pebble is going to be in cache. That's the hot stuff. And the blue pebbles are our disk. That's the cold stuff. And, basically, the same rules-- when you're initially placing a pebble, everything here has to be red. You can place a red pebble if your predecessors have red pebbles.

We start out with the inputs being blue, so there are no red pebbles. But for free-- or not for free. For unit cost, we can convert any red pebble to a blue pebble or any blue pebble to a red

pebble. So let's go through this. I can make that one red.

And now, I can make this one red. Great. Now, I don't need it right now. So I'm going to make it blue, meaning write it out to disk. I make this one red, make this one red.

Now, I can throw that one away. I don't need it on cache or disk. I can put that one on disk, because I don't need it right now. I can bring that one back in from cache, write this one out, put that one onto disk, put that onto a disk.

Now, we'll go over here, read this back in from disk, finish off this section over here. And now, I can throw that away, add this guy, throw that away. What do I need? Now, I can write this out to disk. I'm done with the output.

Now, I've got to read all these guys in, and then I can do this one. And so I needed a cache size here of four. The maximum number of red things at any moment was four. And I can get rid of those guys and write that one to disk.

And my goal is to get the outputs all blue. But the objective here is different. Before, we were minimizing, essentially, cache size.

Cache size now is given to us. We say we have a cache of size M. But now, what we count are the number of reads and writes, the number of switching colors of pebbles. That is the number I/Os.

And so you can think of this model as this picture I drew before. You have cache. You can store up to M items. You can take any blue item. You could throw them away, for example.

I could move a red item over here, turn it blue. That corresponds to writing out to disk. I can bring a blue item back in to fill that spot. That corresponds to reading from disk as long as, at all times, I have at most M red items. And these are the same model.

So what Hong and Kung did is look at a bunch of different algorithms, not problems, but specific algorithms, things that you could compute in the DAG form. The DAG form is, I guess you could say, a class of algorithms. There's many ways to execute this DAG. You could follow any topological sort of this DAG. That's an algorithm in some sense.

And so what he's finding is the best execution of these meta algorithms, if you will. So that doesn't mean it's the best way to do matrix vector multiplication. But it says if you're following

the standard algorithm, the standard DAG that you get from it or the standard FFT DAG-- I guess FFT is actually an algorithm-- then the minimum number of memory transfers is this number of red or blue recolorings.

And so you get a variety. Of course, the speed-ups, relative to the regular RAM analysis versus this analysis is going to be somewhere between 1 and M, I guess for most problems at least. And for some problems, like matrix vector multiplication, you get very good M odd even transpositions [INAUDIBLE] you get M. Matrix multiplication, not quite as good red M and FFT. Sorting was not analyzed here, because sorting is many different algorithms. Just one specific algorithm analyzed here, only log M.

So I don't want to go through these analyzes, because a lot of them will follow from other results that we'll get to. So at this point, we have two models. We have the idealized two-level storage of Floyd. We have the red-blue pebble game of Hong and Kung.

This one models caching, that you can store a bunch of things. But it does not have blocks. This one models blocking, but it does not have a cache, or it has a cache of constant size. So the idea is to merge these two models.

And this is the Aggarwal and Vitter paper many of you have heard of, I'm sure. It was in 1987, so six years after Hong and Kung. It has many names. I/O model is the original, I guess. External Memory Model is what I usually use and a bunch of people here use. Disk Access Model has the nice advantage of you can call it the DAM model.

And, again, our goal is to minimize number of I/Os. It's just a fusion of the two models. Now, our cache has blocks of size B. And you have M over B blocks. And your disk is also divided into blocks of size B. We imagine it being as large as you need it to be, probably about order N.

And what can you do? Well, you can pick up one of these blocks and read it in from disk to cache, so kicking out whatever used to be there. You can do computation internally, change whatever these items are for free, let's say. You could measure time, but usually you just measure a number of memory transfers.

And then you can take one of these blocks and write it back out to disk, kicking out whatever used to be there. So it's the obvious hybrid of these models. But this turns out to be a really good model.

Those other two models, they were interesting. They were toys. They were simple. This is basically as simple, but it spawned this whole field. And it's why we're here today.

So this is a really cool model, let's say, tons of results in this model. It's interesting to see-- I'm going to talk about a lot of models today. We're sort of in the middle of them at the moment. But only two have really caught on in a big way and have led to lots and lots of papers. This is one of them.

So let me tell you some basic results and how to do them. A simple approach algorithmic technique in external memory is to scan. So here's my data. If I just want to read items in order and stop at some point N, then that cost me order N over B memory transfers.

That's optimal. I've got to read the data in. I can accumulate, add them up, multiply them together, whatever. One thing to be careful with those is plus 1, or you could put a ceiling on that. If N is a lot less and B, then this is not a good strategy. But as long as N is at least order B, that's really efficient.

More generally, instead of just one scan, you can run up to M over B parallel scans. Because for a scan, you really just need to know what is my block currently. And we can fit M over B blocks in our cache. And so we can advance this scan a little bit, advance this scan a little bit, advanced this one, and go back and forth.

In any kind of interleaving we want of those M over B scans, some of them could be read scans. Some of them could be write scans. Some of them can go backwards. Some of them could go forwards, a lot of options here.

And in particular, you can do something like given a little bit less than M over be lists of total size N, you can merge them all together. If they're sorted lists, you can merge them into one sorted list in optimal N over B time. So that's good. We'll use that in a moment. Here

I have a little bit of a thought experiment, originally by Lars Arge who will be speaking later. You know, is this really a big deal? Factor B doesn't sound so big. Do I care?

For example, suppose I'm going to traverse a linked list in memory, but it's actually stored on disk. Is it really important that I sort that list and do a scan versus jumping around random access? And this is back of the envelope, just computing what things ought to be.

If you have about a gigabyte of data, a block size of 32 kilobytes, which is probably on the

small side, a 1 millisecond disk access time, which is really fast, usually at least 2 milliseconds, then if you do things in random order, on average every access is going to require a memory transfer. That'll take about 70 hours, three days. But if you do a scan, if you presorted everything and you do a scan, then it will only take you 32 seconds.

So it's just 8,000 in time space is a lot bigger than we conceptualize. And it makes things that were impractical to do, say, daily, very practical. So that's why we're here.

Let's do another problem. How about search? Suppose I have the items in sorted order, and I want to do binary search.

Well, the right thing is not binary search, but B-way search, so log base B of N. The plus 1 is to handle the case when B equals 1. Then you want log base 2.

So we have our items. We want to search, first, why is this the right bound? Why is this optimal? You can do an information theoretic argument in the comparison model, assuming you're just comparing items.

Then whenever you read in a block-- if the blocks have already been sorted, you read in some block-- what you learn from looking at those B items is where your query guy, x, fits among those B items. You already know everything about the B items, how they relate to each other. But you learn where x is.

So that gives you log of B plus 1 bits of information, because there are B plus 1 places where x could be. And you need to figure out log of N plus 1 bits.

You want to know where x fits among all the items. And so you divide log of N plus 1 by log of B plus 1. That's log base b plus 1 of N plus 1.

So that's the lower bound. And the upper bound is, you probably have guessed by now, is a B-tree. You just have B items and the node sort of uniformly distributed through the sorted list. And then once you get those items, you go to the appropriate subtree and recurse. And the height of such a tree is log base b plus 1 of N, and so it works.

B-trees have the nice thing, you can also do insertions and deletions in the same amount of time. Though, that's no longer so optimal. For searches, this is the right answer.

So, next thing you might want to do-- I keep saying, assume it's sorted-- I'd really like some

sorted data, please. So how do I sort my data? I think the Aggarwal and Vitter paper has this fun quote about, today, one fourth of all computation is sorting.

Some machines are devoted entirely to sorting. It's like the problem of the day. Everyone was sorting. I assume people still sort, but I'm guessing it's not the dominant feature anymore.

And it's a big deal, you know. Can I sort within one day, so that all the stuff that I learned today or all the transactions that happened today I could sort them. So it turns out the right answer for sorting bound is N over B log base M over B of N over B.

If you haven't seen that, it looks kind of like a big thing. But those of us in the know can recite that in our sleep. It comes up all over the place. Lots of problems are as hard as sorting, and can be solved in the sorting bound time.

To go back to the problem I was talking about with Floyd's model, the permutation problem, I know the permutation. I know where things are supposed to go. I just need to move them there physically.

Then it's slightly better. You have the sorting bound, which is essentially what we had before. But in some cases, just doing the naive thing is better. Sometimes it's better to just take every item and stick it where it belongs in completely random access.

So you could always do it, of course, in N memory transfers. And sometimes that is slightly better than the sorting bound, because you don't have the log term. And so that is the right answer to Floyd's problem.

He got the upper bound right. In his case, M over B is 3. So this is just log base 2. But he missed this one term.

OK. So why is the sorting bound correct? I won't go through the permutation bound. The upper bound's clear. Information, theoretically, it's very easy to see why you can't do better than the sorting bound.

Let's set up a little bit of ground rules. Let's suppose that whatever you have in cache, you sort it. Because why not? I mean, this is only going to help you. And everything you do in cache is free. So always keep the cache sorted.

And to clean up the information that's around, I'm going to first do a pass where I read a block,

sort the block, stick it back out, and repeat. So each block is presorted. So there's no sorting information inside a block. It's all about how blocks compare to each other here.

So when I read a block-- let's say this is my cache, and a new block comes in here-- what I learn is where those B items live among the M items that I already had. So it's just like the analysis before, except now I'm reading B items among M instead of one among B. And so the number of possible outcomes for that is M plus b choose B.

So you have M plus B things. And there's B of them that we're saying which of the B in the order came from the new block. You take log of that, and you get basically B log M over B bits that you learn from each step. And the total number of bits we need to learn is N log N, as you know.

But we knew a little bit of bits from this presorting step. This is to clean this up at the beginning. We already knew N log B bits, because each of those B things was presorted.

So we have B log B per block each of them. There's N over B of them. So it's N log B. So we need to learn N log N minus N log B bits. And in each step, which is a log of N over B N log N over B-- and in each step, we learn B log M over B.

So you divide those two things, and you get N over B log base M over B and N over B. It's a good exercise in log rules and information theory. But now, you see it's sort of the obvious bound once you check how many bits you're learning in each step.

OK. How do we achieve this bound? What's an upper bound? I'm going to show you two ways to do it. The easy one is mergesort. To me, the conceptually easiest is mergesort. They're actually kind of symmetric.

So you probably know binary mergesort. You take your items, split them in half, recursively sort, merge. But we know that we can merge M over B sorted lists in linear time as well in N over M time.

So instead of doing binary mergesort where we split in half, we're going to split into M over B equal sized pieces, recursively sort them all, and then merge. And the recurrence we get from that, there is-- did I get this right? Yeah. There's M over B sub-problems, each of size a factor of M over B smaller than N.

And then to do the merge, we pay N over B plus 1. That won't end up mattering. To make this

not matter, we need to use a base case for this recurrence that's not 1, but B. B will work. You could also do M, but it doesn't really help you.

Once we get down to a single block, of course, we can sort in constant time. We read it and sort it, write it back out. So you want to solve this recurrence. Easy way is to draw a recursion tree.

At the root, you have a problem of size N. We're paying N over B to solve it. We have branching factor M over B. And at the leaves, we have problems with size B. Each of them has constant cost.

I'm removing the big Os to make this diagram both more legible and more correct. Because you can't use big Os when you're using dot dot dot. So no big Os for you.

So then use sum these level by level, and you see we have conservation of mass. We have N things here. We still have N things. They just got distributed up. They're all being divided by B linearity.

You get N over B at every level, including the leaves. Leaves you have to check specially. But there are indeed N over B leaves, because we stop when we get to B.

So you add this up. We just need to know how many levels are there. One is log base M over B of N over B. Because there's N over B leaves branching factor M over B. So you multiply, done, easy.

So mergesort is pretty cool. And this works really well in practice. It revolutionized the world of sorting in 1988.

Here's a different approach, the inverse, more like quicksort, the one that you know is guaranteed to run [INAUDIBLE] log N usually. Here, you can't do binary quicksort. You do M over B root M over B-way quicksort.

The square root is necessary just to do step one. So step one is I need to split. Now, I'm not splitting my list into chunks. In the answer, in the sorted answer, I need to find things that are evenly spaced in the answer.

That's the hard part. Then, usually, you find the median to do this. But now, we have to find sort of square root of M over B median-like elements spread out through the answer. But we

don't know the answer, so it's a little tricky.

Then once we have those partition elements, we can just do it. This is the square root of M over B-way scan again. You scan through the data.

For each of them, you see how it compares to the partition elements. There aren't very many of them. And then you write it out to the corresponding list, and you get square root of M over B plus 1 lists.

And so that's efficient, because it's just a scan or parallel scans. And then you recurse, and there's no combination. There's no merging to do. Once you've got them set up there, you recursively sort, and you're done.

So the recurrence is exactly the same as mergesort. And the hard part is how do you do this partitioning? And I'll just quickly sketch that. This is probably the most complicated algorithm in these slides.

I'll tell you the algorithm. Exactly why it works is familiar to if you know the Bloom at all, linear time merging algorithm for regular internal memory. Here's what we're going to do. We're going to read in M items into our cache, sort them.

So that's a piece of the answer in some sense. But how it relates to the answer, which subset of the answer it is, we don't know. Sample that piece of the answer like this. Every root M over B items, take one guy.

Spit that in an output of samples. Do this over and over for all the items-- read in M, sort, sample, spit out-- you end up with this many items. This is basically a trick to shrink your input.

So now, we can do inefficient things on this many items, because there aren't that many of them. So what do we do? We just run the regular linear time selection algorithm that you know and love from algorithms class to find the right item.

So if you were splitting into four pieces, then you'd want the 25%, 50%, and 75%. You know how to do each of those in linear time. And it turns out if you re-analyze the regular linear time selection, indeed, it runs in N over B time in external memory. So that's great.

But now, we're doing this just repeatedly over and over. You find the 25%. You find the 50%. Each of them, you spend linear time. But you multiply it out. You're only finding root of M over

B of them. Linear time, it's not N over B, it's N divided by this mess.

You multiply them out, it disappears. You end up in regular linear time, N over B. You find a good set of partitions. Why this is a good set is not totally clear. I won't justify it here. But it is good, so don't worry.

OK. One embellishment to the external memory model before I go on is to distinguish not just saying, oh, well, every block is equally good. You want to count how many blocks you read. When you read one item, you get the whole block. And you better use that block.

But you can furthermore say, well, it would be really good if I read a whole bunch of blocks in sequence. There are lots of reasons for this in particular. Disks are really good at sequential access, because they're spinning.

It's very easy to seek to the thing right after you. First of all, it's easy to read the entire track, the whole circle of the desk. And it's easy to move that thing.

So here's a model that captures the idea that sequential block reads or writes are better than random. So here's the idea of sequential. If you read M items, so you read M over B blocks in sequence, then each of those is considered to be a sequential memory transfer.

If you break that sequence, then you're starting a new sequence. Or it's just random access if you don't fall into a big block like this. So there's a couple of results in this model. One is this harder version of external memory.

So one thing is what about sorting? We just covered sorting. It turns out those are pretty random access in the algorithms we saw. But if you use binary mergesort, it is sequential. As you binary merge, things are good.

And that's, essentially, the best you can do, surprisingly, in this model. If you want the number of random memory transfers to be little o of the sorting bound-- so you want more than a constant fraction to be sequential-- then you need to use at least this much total memory transfers. And so binary mergesort is optimal in this model, assuming you want a reasonable number of sequential axises.

And the main point of this paper was to solve suffix-tree construction in external memory. And what they prove is it reduces to sorting, essentially, and scans. And scans are good. So you get this exact same trade-off for suffix-tree construction, fair representation. I have to be

careful, because so many authors are in those room.

Cool. So let's move on to a different model. This is a model that did not catch on. But it's fun for historical reasons to see what it was about.

You can see in here two issues. One is, what about a deeper memory hierarchy? Two levels is nice. Yeah, in practice, two levels are all that matter. But we should really understand multiple levels.

Surely, there's a clean way to do that. And so there are a bunch of models that try to do this. And by the end, we get something that's reasonable. And HMM is probably one of my favorite weird models.

It's "particularly simple." This is a quote from their own paper, not that they're boastful. It is a simple model. This is true.

And it does model, in some sense, a larger hierarchy. But the way it's phrased initially doesn't look like this picture, but they're equivalent. So it's a RAM model. So your memory is an array.

If you want to access position x in the array, you pay f of x. And in the original definition, that's just log x. So what that corresponds to is the first item is free.

Second item costs 1. The next two items cost 2. The next four items cost 3. The next eight items cost 4, and so on.

So it's exactly this kind of memory hierarchy. And you can move items. You can copy. And you can do all the things you can do in a RAM. So this is a pretty good model of hierarchical memory. It's just a little hard.

So, originally, they defined it with log x based on this book, which is the classic reference of VLSI at the time by Mead and Conway. It sort of revolutionized teaching VLSI. And it has this particular construction of a hierarchical RAM. I Don't know if RAMs are actually built this way.

But they have a sketch of how to do it that achieves a logarithmic performance. The deeper you are, you pay log. The bigger your space is, you need to pay logarithmic to access it.

OK. So here are the results that they get in this model. I'm not going to prove them. Because, again, they follow from the results in some sense. But you've got matrix multiplication, FFT sorting, scanning, binary search, a lot of the usual problems.

You get kind of weird running times, log, log, and so on. Here, it's a matter of slow down versus speed up, because everything is going to cost more than constant now. So you want to minimize slowdowns. Sometimes you get constant.

The worst slow down you can get is log N, because everything you can access in, at most, log N time in this model. But I would say setting f of N to be log N doesn't really reveal what we care about. But in the same paper, they give a better perspective of their own work.

So they say, well, let's look at the general case. Maybe log x isn't the right thing. Let's look at an arbitrary f of x. Well, you could write an arbitrary f of x as a weighted sum of threshold functions.

I want to know is x bigger than xi. If so, I pay wi. Well, that is just like this picture. Any function can be written like that if it's a discrete function. But you can also think of it in this form if the xi's are sorted.

After you get beyond x0 items, you pay w0. After you get beyond x1 items total, you pay w1, and so on. So this gives you an arbitrary memory hierarchy even with growing and shrinking sizes, which you'd never see in practice.

But this is the general case. And we are going to assume here that f is polynomially bounded to make these functions reasonable. So when you double the input, you only change the output by a constant factor.

OK. Fine. So we have to solve this weighted sum. But let's just look at one of these. This is kind of the canonical function. The rest is just a weighted sum of them.

And if you assume this polynomial bounded property, really it suffices to look at this. So this is called f sub M. We pay 1 to access anything beyond M. And we pay 0 otherwise.

So they've taken general f with this deep hierarchy, and they've reduced to this model, the red-blue pebble game, which we've already seen. I don't know if they mentioned this explicitly, but it's the same model again.

And that's good, because a lot of problems-- well, they haven't been solved exactly. I would say, now, this paper is the first one to really say, OK, sorting, what's the best way I can sort in this model? And they get something. Do I have it here?

Yeah. They aim for a uniform optimality. This means there's one algorithm that works optimally for this threshold function no matter what M is. The algorithm doesn't get to know M. You might say the algorithm is oblivious to M. Sound familiar?

So this is a cool idea. Of course, it does not have blocking yet. But none of this model has blocking. But they prove that if you're uniformly optimal, if you work in the red-blue pebble game model for all M with one algorithm, then, in fact, you are optimal for all f of x, which means, in particular for the deep hierarchy, you also work.

And they achieve tight bounds for a bunch of problems here. You should recognize all of these bounds are now, in some sense, particular cases of the external memory bounds. So like sorting, you have this. Except there's no B. The B has disappeared, because there's no B in this model.

But, otherwise, it is N over B log base M over B of N over B and so on down the line. They said, oh, search here is really bad, because caching doesn't really help for search. But blocks help for search. So when there's no B, these are exactly the bounds you get for external memory.

So I mean, some of these were known. These were already known by Hong and Kung, because it's the same special case. And then the others followed from external memory.

But this is kind of neat. They're doing it in a somewhat stronger sense, because it's uniform without knowing M. So the uniformity doesn't follow from this. But they get uniformity. And therefore, it works for all f.

OK. They had another fun fact, which will look familiar to those of you who know the cache-oblivious model, which we'll get to. They have this observation that while we have these algorithms that are explicitly moving things around in our RAM, it would be nice if we didn't have to write that down explicitly in the algorithm. Could we just use least recently used replacement, so move things forward?

That works great if you know what M is. Then you say, OK, if I need to get something from out if here, I'll move it over here. And whatever was least recently used, I'll kick out. And at this point, this is just a couple of years prior to this paper.

Sleator and Tarjan did the first paper on competitive analysis. And they proved that LRU or

even first in, first out is good in the sense that if you just double the size of your cache-- oh, I got this backwards. TLRU of twice the cache is, at most, TOPT of 1 times the cache. So the 2 should be over here.

Great. And assuming you have a polynomially bounded growth function, then this is only losing a constant factor. OK. But we don't know what M is. This works for the threshold function f sub m. But it doesn't work for an arbitrary function f, or it doesn't work uniformly.

And we want a uniform solution. And they gave one. I'll just sketch it here. The idea is you have this arbitrary hierarchy. You don't really know. I'm going to assume I do know what f is.

So this is not uniform. It's achieved in a different way. But I'm going to basically rearrange the structure to be roughly exponential to say, well, I'm going to measure f of x as x increases. And whenever f of x doubles, I'll draw a line.

These are not where the real levels are. It's just a conceptual thing. And then I do LRU on this structure. So if I want to access something here, I pull it out. I stick it in here.

Whatever is least recently used gets kicked out here. And whatever is least recently used gets kicked out here, here, here. And you do a chain of LRUs. Then you can prove that is within a constant factor of optimal, but you do have to pay a startup cost. It's similar to move to front analysis from Sleator and Tarjan.

OK. Enough about HMM sort of. The next model is called BT. It's the same as HMM, but they add blocks. But not the blocks that we know from computer architecture, but a different kind of block thing. It's kind of similar. Probably, [INAUDIBLE] constant factors and not so different.

So you have the old thing accessing x costs f of x. But, now, you have a new operation, which is I can copy any interval, which would look something like this, from x minus delta to x. And I can copy it to y minus delta to y.

And I pay the time to seek there, f of max of x and y. Or you could do f of x plus f of y. It doesn't matter. And then you pay plus delta.

So you can move a big chunk relatively quickly. You just pay once to get there, and then you can move it. This is a lot more reasonable than HMM.

But it makes things a lot messier is the short answer. Because-- here's a block move-- these

are the sort of bounds you get. They depend now on f. And you don't get the same kind of uniformity as far as I can tell.

You can't just say, oh, it works for all f. For each of these problems, this is basically scanning or matrix multiplication. It doesn't matter much until f of x gets really big, and then something changes. You Dot product, you get log*, log, log, log, depending on whether your f of x is log or subpolynomial or linear.

So I find this kind of unsatisfying. So I'm just going to move on to MH, which is probably the messiest of the models. But in some sense, it's the most realistic of the models. Here's the picture which I would draw if someone asked me to draw a general memory hierarchy.

I have CPU connects to this cache for free. It has blocks of size B0. And to go to the next memory, it costs me some time, t0. And the blocks that I read here of size B0, I write of size B0. So the transfers here are size B0.

And one has potentially a different block size. It has a different cache size, M1. And you pay. So these blocks are subdivided into B0 sized blocks, which happen here.

This is a generic multi-level memory hierarchy picture. It's the obvious extension of the external memory model to arbitrarily many levels. And to make it so easy to program, all levels can be transferring at once. This is realistic, but hard to manipulate.

And they thought, oh, well, l parameters for an l-level hierarchy is too many. So let's reduce it to two parameters and one function. So assume that B [? does ?] grow exponentially, that these things grow roughly the same way. with some aspect ratio alpha.

And then the ti-- this is the part that's hard to guess-- it grows exponentially. And then there's some f of i, which we don't know, maybe it's log i. And because of that, this doesn't really clean the model enough.

You get bounds, which, it's interesting. You can say as long as f of i is, at most, something, then we get optimal bounds. But sometimes when f of i grows, things change. And it's interesting. These algorithms follow approaches that we will see in a moment, divide and conquer.

But it's hard to state what the answers are. What's B4? I think that's just a typo. That should be blank.

I mean, it's hard to beat an upper bound of 1. It also seems wrong. Ignore that row.

All right. Finally, we go to the cache-oblivious model by Frigo, et al. in 1999. This is another clean model. And this is another of the two models that really caught on.

It's motivated by all the models you've just seen. And in particular, it picks up on the other successful model, the External Memory Model and says, OK, let's take External Memory Model, exactly the same cost model. But suppose your algorithm doesn't know B or M.

And we're going to analyze it in this model knowing what B and M is. But, really, one algorithm has to work for all B and M. This is uniformity from the-- I can't even remember the model names-- not UMH, but the HMM model. So it's taking that idea, but applying it to a model that has blocking.

So for this to be meaningful, block transfers have to be automatic. Because you can't manually move between here and here. In HMM, you could manually move things around, because your memory is just a sequential thing.

But now, you don't know where the cutoff is between cache and disks. So you can't manually manage your memory. So you have to assume automatic block replacement. But we already know LRU or FIFOs only going to lose a constant factor.

So that's cool. I like this model, because it's clean. Also, in a certain sense, it's a little hard to formalize this. But it works for changing B, because it works for all B.

And so you can imagine even if B is not a uniform thing-- like the size of tracks on a disk are varying, because circles have different sizes-- it probably works well in that setting. It also works if your cache gets smaller, because you've got a competing process. It'll just adjust, because the analysis will work.

And the other fun thing is even though you're analyzing on a two-level memory hierarchy, it works on an arbitrary memory hierarchy, this MH thing. This is a clean way to tackle MH. You just need a cache-oblivious solution.

Cool. Because you can imagine the levels to the left of something and the levels to the right of some point. And the cache-oblivious analysis tells you that the number of transfers over this boundary is optimal. And if that's true for every boundary, then the overall thing will be optimal, just like for HMM uniformity.

OK. Quickly some techniques from cache-oblivious. I don't have much time, so I will just give you a couple sketches. Scanning is one that generalizes great from external memory. Of course, every cache-oblivious algorithm is external memory also. So we should first try all the external memory techniques.

You can scan. You can't really do M over B parallel scans, because you don't know what M over B is. But you can do a constant number of parallel scans. So you could at least merge two lists.

OK. Searching, so this is the analog of binary search. You'd like to achieve log base B of N query time. And you can do that. And this is in Harald Prokop's master's thesis.

So the idea is pretty cool. You imagine a binary search tree built on the items. We can't do a B-way, because we don't know what B is.

But then we cut it at the middle level, recursively store the top part, and then recursively store all the bottom parts, and get root N chunks of size root N. Do that recursively, you get some kind of lay out like this. And it turns out this works very well.

Because at some level of the recursion, whatever B is-- it doesn't know when you're doing the recursion. But B is something. And if you look at the level of recursion where you straddle B here, these things are size, at most, B. And the next level up is size bigger than B.

Then you look at a root to leaf path here. It's a matter of how many of these blue triangles do you visit. Well, the height of a blue triangle is going to be around half log B, because we're dividing in half until we hit log B. So we might overshoot by a factor of 2, but that's all.

And we only have to pay 2 memory transfers to visit these. Because we don't know how it's aligned with a block. but at most, it fits in 2 blocks, certainly. It's stored consecutively by the recursion.

And so you divide. I mean, the height of this thing, it's going to be log base B of N times 2. We pay 2 each. So we get an upper bound of 4.

Not as good as B-trees. B-trees get 1 times log base B of N. Here, we get 4 times log base B of N. This problem has been considered. The right answer is log of e plus little o. And that is tight. You can't do better than that bound.

So cache-oblivious loses a constant factor relative to external memory for that problem. You can also make this dynamic. This is where a bunch of us started getting involved in this world, in cache-oblivious world. And this is a sketch of one of the methods, I think this one. That's the one I usually teach.

You might have guessed these are from lecture notes, these handwritten things. I'll plug that in in a second. So sorting is trickier. There is an analog to mergesort. There is an analog to distribution sort.

They achieve the sorting bound. But they do need an assumption, this tall-cache assumption. It's a little different from the last one. This is a stronger assumption than before.

It says the cache is taller than it is wide, roughly, up to some epsilon exponent. So this is saying M over B is at least B to the epsilon. Most caches have that property, so it's not that big a deal.

But you can prove it's necessary. If you don't have it, you can't achieve the sorting bound. You could also prove you cannot achieve the permutation bound, because you can't do that min. You don't know which is better, same paper.

Finally, I wanted to plug this class. It just got released if you're interested. It's advanced data structures. There's video lectures for free streaming online.

There are three lectures about cache-oblivious stuff, mostly on the data structure side, because it's a data structures class. But if you're interested in data structures, you should check it out. That is the end of my summary of a zillion models.

The ones to keep in mind, of course, are external memory and cache-oblivious. But the others are kind of fun. And you really see the genesis of how this was the union of these two models. And this was sort of the culmination of this effort to do multilevel in a clean way.

So I learned a lot looking at all these papers. Hope you enjoyed it. Thanks.

[APPLAUSE]

PROFESSOR: Are there any questions?

AUDIENCE: So all these are order of magnitude bounds I'm wondering about the constant factors.

ERIK DEMAINE: Are you guys going to talk about that in your final talk? Or who knows? Or Lars maybe also? Some of these papers even evaluated that, especially these guys that had the messy models. They were getting the parameters of, at that time, [INAUDIBLE] 6,000 processor, which is something I've actually used, so not so old.

And they got very good matching even at that point. I'd say external memory does very good for modeling disk. I don't know if people use it a lot for cache. No, I'm told.

Cache-oblivious, it's a little harder to measure. Because you're not trying to tune to specific things. But in practice, it seems to do very well for many problems. That's the short answer.

AUDIENCE: [INAUDIBLE] it runs faster than [INAUDIBLE] cache aware.

ERIK DEMAINE: Yeah. It does better than our analysis said it should do in some sense, because it's so flexible. And the reality is very messy. In reality, M is changing, because there's all sorts of processes doing useless work.

And cache-oblivious will adjust to that. And it's especially the case in internal memory, in the cache world. Things are very messy and fussy. And the nice thing about cache-oblivious is because you're not specifically tuning, you have the potential to not die when you mess up.

AUDIENCE: I'd say that's especially the case in the disk world.

ERIK DEMAINE: Oh, interesting.

AUDIENCE: [INAUDIBLE] But--

ERIK DEMAINE: These are the guys who know.

AUDIENCE: [INAUDIBLE] people have different [INAUDIBLE]

ERIK DEMAINE: Yeah. They're both relevant.

AUDIENCE: What's the future? This is history.

ERIK DEMAINE: OK. Well, for the future, you should go to the other talks, I guess. There's still lots of open problems in both models. External memory, I guess, graph algorithms and geometry are still the main topics of ongoing research.

Cache-oblivious is similar. At this point, I think-- well, also geometry is a big one. There's some

external memory results that have not yet been cache-oblivified in geometry.

AUDIENCE: Multicore.

ERIK DEMAINE: Multicore. Oh, yeah, I forgot to say I'm not going to talk about parallel models here. Partly, because of lack of time. Also, that's probably the most active-- it's an interesting active area of research, something I'm interested in particular.

There are some results about parallel cache-oblivious. And all of these papers actually had parallelism. These had parallelism in a single disk. There's another model that has multiple disks. Those behave more or less the same. You basically divide everything by p.

These models also tried to introduce parallelism. Or there's a follow up to UMH by these guys. So there is work on parallel, but I think multicore or cache-oblivious is probably the most exciting unknown or still in progress stuff.

AUDIENCE: Thank the speaker again.

ERIK DEMAINE: Thanks.

[APPLAUSE]