ERIK DEMAINE: Welcome to the final week of 6.046.

Are you excited?

[CHEERING] Yeah, today-- AUDIENCE: Oh.

ERIK DEMAINE: Well, and sad, I know.

It's tough.

But we've got two more lectures.

They're on that one topic, which is cache oblivious algorithms.

And this is a really cool concept.

It was actually originally developed in the context of 6.046, as sort of an interesting way to teach cache-efficient algorithms.

But it turned into a whole research program in the late '90s, and now it's its own thing.

It's kind of funny to bring it back to 6.046.

The whole idea is in all of the algorithms we have seen, except maybe distributed algorithms, we've had this view that all of the data that we can access is the same cost.

If we have an array, like a hash table, accessing anything in a hash table is equally costly.

If we have a binary search tree, every node costs the same to access.

But this is not real.

Let me give you some idea of what a real computer looks like.

You probably know this, but we've not yet thought about it in an algorithmic context.

These are caches, what are typically called caches, in your computer.

Then you have what we've mostly been thinking about, which is main memory, your RAM.

And then there's probably more stuff.

These days you probably have some big flash.

If you have a fancier computer, you have flash, which is maybe caching your disk, which is huge.

And then maybe there's the internet at the end, if you like.

So the point is all the data in the world is not on your CPU.

And there's this big thing which is called the memory hierarchy, which dictates which things are fast and which things are slow, not exactly which data items; that's up to you.

But the idea is that on board your CPU you have probably, these days, up to four levels of cache.

As I've tried to draw them, they get increasingly big.

Typical values-- a level one cache is something on the order of 10, 32 K, whatever.

Level four cache these days, as introduced by like Haswell Architectures, has about 100 megabytes.

Main memory you know; this is the thing you usually think.

About.

It's in the gigabytes.

These days you can buy computers with a terabyte of RAM.

It's not crazy.

Flash gets bigger.

Disk-- these days you can buy 4-terabyte single disk, but if you have a whole RAID of disks, you can have petabytes of data on one computer.

So things are getting bigger as we go farther to the right.

But they're also getting slower. .

And the point of cache efficient algorithms is to deal with the fact that things get slow when they get far away.

And this makes sense from a physics standpoint.

If you think about how much data can you store in a cubic inch or something and how much could possibly be near your CPU, at some point, you're just going to run out of space, and you've got to go farther away.

And to go farther away is going to take more time.

So you can think of it-- I mean, there's the speed of light argument, that things that are farther away in your computer are going to take longer.

Typical computers are not anywhere near the speed of light, so there's a more real issue, which is how long are your traces.

And then when you have physical moving parts, like a disk, I don't know if you know, but disks actually spin, and there's a head, and it has to move around.

And that's called seek time.

Moving a head around on the disk is really slow, on the order of milliseconds.

Whereas reading from on-chip cache, that's on the order of nanoseconds, whatever your clock rate is, so a few billion times a second.

So there's a big spread of like a factor of a million or 10 million from level one cache to disk speed.

That sucks.

And so you might think, well, if your data's big, you're just screwed.

You've got to deal with disk, and disk is slow.

But that's not true.

Life is not so bad.

So, in general, there's two notions of speed, and I've been kind of vague on them.

One notion is latency, which is if right now I have the idea that I really need to fetch memory location 2 billion and

73, how long does it take for that data-- say, one word of data-- to come back?

That's latency.

But there's another issue, which is bandwidth; how fat are these pipes?

What's my rate of information that I could pump?

If I said, please give me all of main memory in order, how fast could it pump it back?

And that's actually really good.

So latency is like your start up cost.

When I ask for something, how long does it take for that one thing to come?

But then there's a data rate.

And bandwidth you can generally make really large.

For example, in disk, bandwidth of a disk is pretty big.

But even if it weren't big, you could just add 100 more disks.

And then when you ask for some data, all 100 disks could give you data at the same speed, and provided you don't overload your bus, so you've got to also make more buses and so on.

You can actually really huge amount of data per second, but still the time to get there and the time for all the disks to seek their heads, that's slow.

It doesn't add up, actually, because they're all doing it in parallel.

So you can't reduced latency, but you can increase bandwidth.

And let's say-- it doesn't match physics, but we can get pretty close to arbitrarily high bandwidth.

And so in a well-designed computer, the fatnesses of these pipes are going to increase, or could increase, if you want.

So you can move lots of data around.

But latency we can't get rid of, and this is annoying because from an algorithmic standpoint, when we ask for something, we'd like it immediately.

In a sequential logarithm, we can't do anything until that date arrives.

So cache efficiency is going to fix this by blocking.

This is an old idea, since caches were introduced.

There's the idea of blocking.

So when you ask for a single word in main memory, you don't get one word.

You get maybe 32 kilobytes of information, not just 4 bytes or 8 bytes.

And we're kind of free to choose these block sizes however we want, when we designed the system.

So we can set them, in a certain sense, to hide latency.

So if you think of amortizing the cost over the block, then you have something like amortized cost over block.

This is per word.

Essentially, we divide the latency by the block size.

And we have to pay one over bandwidth.

Bandwidth is how many words a second you can read, say, from your memory.

So one over bandwidth is going to be your cost.

So this we can't change but.

By adding enough disks or adding enough things at making these pipes fat enough, you can basically make this big.

Latency is the thing we can't control.

But if this block is sort of useful, then we're paying the initial start up time, say, hey, give me this block, and then waiting for the response.

That latency we only pay once for the entire block.

So if there are block-size words in that block, per item, we're effectively dividing latency by block size.

This is kind of rough, but this is the idea of how to reduce latency.

Now, for this actually work, we need better algorithms.

Pretty much every algorithm you see in the class so far works horribly in this model.

So that's the point of today and next class is to fix that.

For this kind of amortization to work, I'm using "use" in a vague sense so far.

We'll make it formal in a moment.

When I fetch an entire block, all of the elements in that block should be useful.

We should be able to compute something on them that we needed to compute.

Otherwise, if I if I only needed the one item that I read out of the block, that's not going to help me so much.

So I really want to structure my data in such a way that when I access one element, I'm also going to access the elements nearby it.

Then this blocking will actually be useful.

This is a property normally called spatial locality.

And the other thing we'd like-- these caches have some size, so I can store more than just one block.

It's not like I read one block, and I just finish processing it, and then I read the next block and go on.

Some of these caches are actually pretty big.

If you think of main memory as a cache to your disk, that can be really big.

So ideally, the blocks that I'm using here relate to each other in some way, or when I access the block, I'm going to access it for awhile, along with other blocks.

So the way this is usually said is that we'd like to reuse the existing blocks in the cache as much as possible.

And this you can think of as temporal locality.

When I access a particular block, I'm going to access it again fairly soon.

That way it's actually useful to bring it into my cache, and then I use it many times.

That would be even better.

I don't have to have both of these, and exactly to what extent I have them is going to dictate what the overall time it's going to take to run my algorithm.

But these are so the ideal properties you want in a very informal sense.

Now, in the rest today, we're going to make this formal, and then we're going to develop some algorithms for this model.

But this is the motivation.

In reality, we're free to choose block size in the system.

Though, in a moment, I'm going to assume that it's given to us.

You'd normally set the block size so that these two terms come out roughly equal.

Because if you're spending the latency time to go and get something, you might as well get a whole chunk of something, according to whatever your bandwidth is.

If it only cost you, say, twice as much to fetch an entire block than to fetch one word, that seems like a pretty good block size.

So for something like disk, that block size is on the order of megabytes, maybe even bigger-- hundreds of megabytes.

So think of the block sizes as really big.

We really want all that data to be useful in some way.

Now it's really hard to think about a memory hierarchy with so many levels.

So we're going to focus on two levels at a time-- the sort of the cheap and small cache versus the huge thing, which I'll call disk, just for emphasis.

So I'm going to call this two-level model the external memory model.

It was originally introduced as a model for main memory versus disk.

But you could apply it to any pair of levels.

In general, you have your problem size N, choose the smallest level that fits N. Typically that's main memory.

Maybe it's disk.

And just think of the level between that and the previous, so the last level and the next to last level.

Often that's what matters.

Like if you run a program, and you run out of RAM, and you start swapping the disks, that's when everything just slows to a crawl.

You can see that difference at each of these levels, but it's probably most dramatic at disk just because it's so slow-- a million times slower than RAM, or at least 1,000 times slower than RAM, I should say.

Anyway, so we have just two levels.

So let me draw a more precise picture.

We have the CPU.

This is where all of our operations are doing this, where we add numbers and so on.

We'll think of it as having a constant number of registers.

Each register is one word.

And then we have a really fat pipe, low latency pipe, to the cache.

Cache is going to be divided into blocks.

So let's say there's B words per blocks.

Instead of writing block size, I'll just write capital B. And the number of blocks.

I'm going to call M over B. So the total size of your cache is capital M. And then there is a relatively thin and slow connection-- this one's fast.

This one's slow-- to your disk.

Disk we'll think of as huge, essentially infinite size.

It's also divided into blocks of size B, so same block size.

So this is the picture.

And so, initially, all of the input is over here, all of your end data items, whatever.

So you want to sort those items.

And in order to access those items, you first have to bring them into cache.

That's going to be slow, but it's done in a blocked manner.

So when I can't access an individual item here, I have to request the entire block.

When I request that block, it gets sent over here.

It takes a while.

And then I get to choose where to store it.

Maybe I'll put it here.

And then maybe I'll grab this block and then store it here and so on.

Each of those is a block read, so these are new instructions the CPU can do.

And eventually, this cache will get full.

And then before I bring in a new block, I have to kick out an old lock.

Meaning I need to take one these blocks and write it to some position, maybe to the same place.

I think, in fact, we will always assume that you write to the same place, overwrite what was on the disk.

You made some changes here, send it back.

And, in general, what we're going to do is count how many times we read and write blocks.

Question?

AUDIENCE: When you talked about how fast the connection is, you're just talking about latency, right?

ERIK DEMAINE: Yes, sorry, this is latency.

AUDIENCE: Yeah, so like the [INAUDIBLE] connections [? just don't have ?] [INAUDIBLE]?

ERIK DEMAINE: Right, this could have huge bandwidth.

So in this model, we're assuming the block size is fixed, and then the latency versus bandwidth is not-- we're not going to think about bandwidth.

We'll assume the block size has been chosen in some reasonable way.

And then all we need to do is count the number of blocks.

But underneath, yeah, you have some kind of bandwidth.

Presumably you set the block size to make these two things roughly equal, and so then latency and bandwidth are kind of the same thing.

That's the idea.

But really, we're just going to think about counting latency, which is how many times do I have to request to block and wait for it to come over, and how much does it cost to write a block?

How many times do I write a block?

I'm not going to worry about how much physical time it takes me to do either of those things; I'm just going to count them and assume that that is what I need to minimize.

So I'm going to count-- we call these memory transfers-- transfers of blocks between levels, between these two levels.

This is the number of blocks read from or written to disk.

We're going to view accesses to the cache as free.

I'm not going to count those.

You don't need to worry about that so much because we can still count the number of operations that we do on the computer, on the CPU.

We still can think about how much time, regular time, it takes to do the computation-- how many comparisons, how many additions, things like that.

And that would include things like reading and writing elements from cache-- individual things.

But we're going to view this connection-- let's say, these are on the same ship.

So reading cache is just as fast as reading from registers.

So we're not going to worry about that time.

What we're focusing on, for the purpose of this model, is between these two levels.

So these are essentially one level combined.

I'll change that in a little bit.

But for now, just think about the two levels.

And we're counting how many memory transfers do we have between these two levels, cache and disk.

So we want to minimize that.

Now, just like before, we want to minimize the running time in the usual traditional measure.

And we want to minimize space and all the usual things we minimize.

But now we have a new measure, which is number of memory transfers, and we want our algorithm to minimize that too, for a given block size and for a given cache size.

And at this point-- I'm going to change this in a moment-- the algorithm that we would write in this external memory model explicitly manages the blocks.

It has to explicitly read and write blocks.

And there's a software system that implements this model, particularly for disk, and lets you do this in a nice controlled way, maintain your memory, maintain reading and writing disk.

The operating system tries to do this, but it usually does a really bad job with swapping.

But there are software systems that let you take control and do much better.

So that's a good model.

External memory model is especially good for disk.

It's not going to capture the finesse of all these other levels, and it's a little bit annoying to write algorithms in this way-- explicitly reading and writing blocks.

Today I will not write any such algorithms.

Although, you could think about them.

I personally love this other model, which is cache obviousness.

It's going to lead to, in some sense, cleaner algorithm.

Although, it's more of a magic trick to get them to work.

But writing the algorithms is very simple.

Analyzing them is more work.

And it will capture, in some sense, all of these levels.

But, in fact, it is basically exactly this model, almost the same.

We're going to change one thing, which is where the oblivious comes from.

We're going to say that the algorithm doesn't know the cache parameters.

It doesn't know B or M. So this is a little weird.

We're going to have to make some other changes to make it work.

From an analysis perspective, I want to count memory transfers and analyze my algorithm with respect to this memory hierarchy.

But the algorithm itself isn't allowed to know what that member hierarchy looks like.

Another way to say this is that the algorithm has to work simultaneously for all values of B and all values of M. As you might imagine, this is not so easy.

But there are some simple things where this is easy, and more complicated things where this is possible.

And it gives you all sorts of cool things.

Let me first formalize the model a little bit.

The other nice thing about cache oblivious algorithms is it corresponds much more closely to how these caches work.

When you write code on your CPU, you may have noticed you don't usually do block reads and block writes, unless you're dealing with flash or disk.

All of this is taking care for you.

It's all done internal to the processor.

When you access a word, behind the scenes, magically, the system, the computer, finds which word to read or which block to read.

It moves the entire block into a higher level cache, and then it's just serving you words out of that block.

And you don't have explicit control over that.

So the way that works is when you access a word in memory-- and I'm going to think of memory as everything; this is what's stored in the disk, say.

This is the entire memory system, the entire memory hierarchy.

And, as usual in this class, we're going to think of the entire memory as a giant array of words.

Each of these squares is one word.

But then also, the memory is now divided into blocks.

So let's say every four.

Let's say B equals 4.

Every four words is a block boundary, just for the sake of drawing a figure.

So this is B equals 4.

When you access a single word, like this one, you get the entire block containing the word.

Let's say, to emphasize, it's not you personally; the system somehow fetches the block containing that word.

It has to do this automatically.

We can't explicitly read and write blocks in this model because we don't know how big the blocks are.

So it couldn't even name them.

But internally, on the real system and in your analysis, you're going to think of whenever you touch something, you actually get all this into the cache.

So you hope that you will use things nearby because you've already read them in.

Ideally, they're useful.

But you don't know how many you've read in.

You've read in B, and you don't what B is.

The algorithm doesn't now.

One more detail-- the cache is going to get full pretty quickly.

And so then, whenever you read something, you have to kick something out.

In steady state, cache might as well always stay full-- no reason to leave anything empty.

So which block do you kick out?

Any suggestions?

Which block should I kick out?

If I've been reading and writing some blocks, reading and writing to words within these blocks.

Yeah?

AUDIENCE: [INAUDIBLE].

ERIK DEMAINE: The block that was fetched farthest in the past?

Yeah that is usually called First In, First Out.

That's FIFO.

And that is a good strategy.

Any other suggestions?

Yeah.

AUDIENCE: [INAUDIBLE].

ERIK DEMAINE: The block has been least recently used.

So maybe you fetched it a long time ago, but you use it every clock cycle.

That one you should probably not throw away because you use it a lot.

That's called LRU, and that is also a good strategy.

Other suggestions?

Those are two good ones.

If you go beyond that, I'm worried I won't know.

But there are some bad strategies.

Yeah?

AUDIENCE: Just random.

ERIK DEMAINE: Random-- yeah, random is probably pretty good.

I don't know offhand.

There are some randomized strategies that beat both of those.

But from this perspective, both are good.

We've got lots of Frisbees to go through, so.

That's a good answer.

Random is definitely a good idea.

I know there's a randomized strategy called [? bit, ?] that in certain senses is a little bit better.

But from my perspective, I think all of those are good.

Random, I have to double check whether you lose a log factor.

And expectation should be fine.

So all of those strategies will work.

You could define this model with any of them.

I think it would work fine, except randomize, you'd get an expectation bound.

So the system evicts, let's say, the least recently used page.

The least recently loaded page would also work fine.

That's FIFO.

Sorry I'm switching to page, but I've been calling them blocks.

Blocks and pages are the same thing for this lecture.

And either at the end of this lecture or beginning of next, I'll tell you why that's an OK thing.

But let's not worry about it at this point.

So now we have a model-- cache flow oblivious.

We have two models, actually.

But I think now that the cache flow oblivious model is complete, we're going to analyze.

Again, we're still counting the number of memory transfers in this thing.

The algorithm's just not allowed know B and M, and so we had to change the model to make the reading and writing of blocks automatic because the algorithm's not allowed to do it.

So someone's got to.

The cool thing about cache oblivious model is every algorithm you see in this class, or most of the algorithms you see in this class, are in a certain sense cache oblivious algorithms.

They weren't aware of B and M before, still not.

What changes is now you can analyze them in this new way, in this new model.

Now, as I said, all the algorithms we've seen are not going to perform well in this model-- almost all.

But that makes things interesting, and that's why we have some work to do.

I have some reasons why cache obliviousness-- why would you tie your hands behind your back and not know B or M?

Reason one, it's cool.

I think it's pretty amazing you can actually do this.

I guess that's reason two is you can actually do it for a lot of problems we care about.

Cache oblivious algorithms exist that are just as good.

So, I mean, of course they exist.

But there are ones that are optimal.

They're within a constant factor of the best algorithm when you know B or M. So that's surprising.

That's the cool part.

In general, the algorithms are easier to write down because we can use pseudo code just like before.

We don't need to worry about blocking in the algorithm.

The analysis is going to be harder, but that's unavoidable.

In some sense, it makes it easier to write code.

And it's also a little easier to distribute your code because every computer has different block sizes that matter.

Also, as you change your value of N, a different level in the memory hierarchy's going to matter.

And so it's annoying-- each of these levels, I didn't mention, has a different block size and, of course, has a different cache size.

So tuning your code every time to a different B or M is annoying.

The big gain here, though, I think, is that you capture the entire hierarchy, in a sense.

So in the real world, each of these pipes has its own latency.

And let's just think about latency.

And you'd like to minimize the number of block transfers between here and here.

You'd like to minimize the number block answers here here.

Well, OK, I can't minimize all of them.

That's a multi-dimensional problem.

What I'd like to minimize is some weighted average of those things-- latency times number of blocks here, plus the latency times the number of blocks here, plus latency times the number of blocks here, and so on.

If you can find an optimal cache oblivious algorithm and analyze it just with respect to two levels, because the algorithm's not allowed to know B and M, it has to work for all levels.

It has to minimize the number of block transfers between all these levels, and so, in particular, will minimize the weighted sum of them.

It's a bit hand wavy.

You have to prove something there.

But you can prove it.

So there's a paper about this from 1999 by Frigo, Leiserson, Prokop, and Ramachandran.

It's old enough that I remember all the names.

After about 2001, when I became a professor, I can't remember anything.

But before that, I can remember everything.

So Frigo, we've talked about him in the context of FFTW.

That was the fastest Fourier Transform in the West.

So he was a student here.

And FFTW uses a cache oblivious Fast Fourier Transform algorithm.

Leiserson, you've probably seen on the cover of your textbook or walking around Stata.

Professor Leiserson here at MIT.

And Prokop, this is actually his [? M Enge ?] thesis.

So pretty awesome [? M Enge ?] thesis.

All right, so cool, I think I said all the things I wanted to say.

So if you want to see the proof that you can solve the entire memory hierarchy, you can read their paper.

You have to make a couple of assumptions, but it's intuitive.

Cache oblivious has to work for all B and M, so it's going to optimize all the levels simultaneously.

Doing that explicitly, with all the different B's and M's, that would be really messy code, probably also slower.

Cache oblivious is just going to do it for free with the same code.

All right, let's do some algorithms.

There's one easy algorithm which works great from a cache oblivious perspective, which is scanning.

Let we give you some Python code.

For historical reasons, in this field, N is written with a capital letter.

Don't ask, or don't worry about it.

So here's some very simple code.

Suppose you want to accumulate an array.

You want to add up all of the items in the array or multiply them or take them in or whatever.

This is a typical kind of thing.

Again, an array, we're going to think of-- so here was my memory.

We're going to think of the array as being stored as some contiguous segment of that array, let's say, this segment.

So this is important.

Assume array is stored contiguously, no holes, relative to how it's mapped on to memory.

And this is a realistic assumption.

When you allocate a block of memory, the promise by the system is that it's essentially a contiguous chunk of memory or disk, or whatever.

And when Python makes an array, it does this.

It guarantees that these things will be stored contiguously.

If you use a dictionary, this would not be true.

But for regular [? array's ?] list, this is true.

So I'm accessing the items in the array in order, and so I start here at item zero.

I end up with item N minus 1.

That seems good because I read this one.

I get the whole block.

Then I read this one.

I already had that block.

It's free.

This one's free.

This one's free.

Here I have to read a new block.

But then this one's free.

So the first item I access in each block costs one, but as long as my cache store's at least one block, that's enough.

And let's say the sum is a register; that's enough to remember that block so that the next operation I do will be free.

So the cost is going to be-- actually, be a little more precise-- ceiling of N over B almost.

Without the big O here, this is right in the external memory model, but not quite right in the cache oblivious model.

Can someone tell me why?

Yeah?

AUDIENCE: If N is two, you could have it beyond a border [INAUDIBLE].

ERIK DEMAINE: Good, N could be two.

But it could span a block boundary.

Remember, the algorithm has no idea where the block boundaries are.

And again, in reality, there are block boundaries all over the place, and there's no way to know.

You can't request that when you allocate an array it always begins in a block boundary.

So great, you can span block boundaries in-- oh, way off.

I just spanned a block boundary, sorry.

So it's going to be, at most, ceiling over N over B plus 1 cache obviously.

So it's just going to hurt you by one.

But I want to point out, there's a slight difference between the two models, even with this very simple algorithm.

In general, I'm just going to think of this as big O N over B plus 1.

There's some additive constant.

I guess you could even say it's N over B plus big O 1, but we won't worry about constant factors today.

So that's scanning, cache oblivious external memory, both great.

Slightly more interesting-- AUDIENCE: [INAUDIBLE]?

ERIK DEMAINE: Yeah, in the external memory algorithm, because you're explicitly controlling the blocks, you're explicitly reading and writing them.

And you know where the block boundaries are.

You could, if you wanted to, you don't have to, but you could choose the array to be aligned, to be starting at a block boundary.

So that's the distinction.

In the cache oblivious, you can't control that, so you have to worry about the worst case.

External memory you could control it, and you could do better, and maybe you'd want to.

It will hurt you buy a constant factor.

And in disks, for example, you want things to be track aligned because if you have to go to an adjacent track, it's a lot more expensive.

You've got to move the head.

Track is a circle, what you can read without moving the head, so great.

So slightly more interesting is you can do a constant number of parallel scans.

So that was one scan.

Here's an example of two scans.

Again, we have one array of size N. Python notation, that would be the whole thing.

And what I want to do is swap Ai with-- this is not Python, but it's, I think, textbook notation.

But you know what swap means.

What does this do, assuming I got my minus ones right?

Yeah?

AUDIENCE: It reverses the array.

ERIK DEMAINE: It reverses the array, good.

We'll just run through these Frisbees.

So this is a very simple algorithm for reversing the array.

It was originally by John Bentley, who was Charles Leiserson's adviser-- PhD adviser-- back in the day.

So very simple, but what's cool about it, if you think about the array and the order in which you're accessing things, it's like I have two fingers-- and I should have made this smaller.

So here, we'll go down here.

I start at the very beginning of the array and the very end of the array.

Then I go to the second element, next to last element, and I advance like this.

So as long as your cache M, the number of blocks in the cache is at least two, which is totally reasonable.

You can assume this is at least 100, typically.

You've got at least 100 blocks, say.

So for any fixed constant, we're going to assume N over B is bigger than a constant.

We'll only need like two or three or four for the algorithms we cover.

Then great, when I access this item, I will load in the block that contains it.

I don't know how it's aligned, but don't care so much.

And then I load in the block that contains this item.

And then the next accesses are free until I advance to the next block.

But once I advance to the next block on the left or the right, I'll never have to access the old ones.

And so again, the cost here is just going to be equal to the number of blocks, which is big O of N over B plus 1.

So a constant number of parallel scans is going to be basically the number of blocks in the array.

So N is smaller than B, this is a bad idea or not so hot.

But when N is bigger than B, this is just N over B. That's how much it takes to read in the data-- big deal.

So these are boring cache oblivious algorithms.

Let's do interesting ones.

And I would say the central idea in cache oblivious algorithms is to use divide-and-conquer.

This goes back to the first few lectures in this class.

And so we will go back to examples from there.

Today we're going to do the median finding, in particular, which we did in lecture two, so really a blast from the past.

But it's good review because the final covers everything, so you've got to remember that.

Matrix multiplication, we've talked about, but not usually-- well, I guess we did actually use divide-and-conquer for Strassen's algorithm.

We're going to use -and-conquer even for the boring algorithm today.

And then next class, we're going to go back to van Emde Boas, but in a completely different way.

So if you don't like van Emde Boas, don't worry; it's much simpler.

So let's do median finding.

Or actually, sorry, let me first talk about divide-and-conquer in general.

You know what divide-and-conquer is.

You take your problem.

You split it into non-overlapping subproblems, recursively solve them, combine them.

But what I want to stress here is what it's going to look like in a cache oblivious context.

So the algorithm is going to look like a regular divide-and-conquer algorithm.

So, in particular, the algorithm will recurse all the way to, let's say, constant size problems, whatever the base case is.

So same as usual, but what's different is the analysis.

When we analyze a cache oblivious algorithm, then we get to know what B and M are.

In some sense, we're analyzing for all B an M.

But let's suppose B and M is given to us, then will tell you how many memory transfers you need.

This kind of bound, you need to know what B is to know what the value of this bound is.

But you learn it as a function of B and, in general, a function of B and M, and that's the best you could hope for as a complete characterization.

So in the analysis, let's just look at one value of B and one value of M. So analysis knows B and M, and it's going to look at, let's say, the recursive level, where one of two things happens.

Either the problem size fits in order one blocks.

So meaning it's order B size.

That's an interesting level.

Another interesting level, the more obvious one probably, is that it fits in cache.

So that means that the size is less than or equal to capital M. Everything here is counted in terms of words.

This is the more obvious one.

For a lot of problems, the cache size isn't so relevant.

What really matters is the block size.

For example, scanning, you're only looking through the data once.

So it doesn't matter how big your cache is, as long as it's not super tiny.

As long as it has a few blocks, then it's just a function of B and N, no M involved.

So for that kind of problem this would be more useful-- constant number of blocks.

Because I think of the cache M as being larger than any constant times B, this is strictly smaller, or this is smaller or equal to problem fitting in cache.

So when M is relevant, we'll look at this level and maybe the adjacent levels in the recursion.

So the algorithm doesn't know what B and M are, so it's got to recurse all the way down-- turtles all the way down.

But the analysis, because we're only thinking about one value B and M at a time, we can afford to just consider that one level, and that will be like the critical place where all the cost is.

Because once things fit in cache and you've loaded things in, the cost will be zero.

So below that, the base case is kind of trivial.

So basically what this is going to do is make our base cases larger.

Instead of our base case being constant, it's going to be order B or M.

What don't I need?

So now let's going to median finding.

Median finding, you're given an unsorted array.

You want to find the median.

And in lecture two, we had a linear time worst case algorithm for this.

And so my goal today is to make it this running time.

This is what you might call linear time in the cache oblivious model because that's how long it takes just to read the data.

It turns out basically the same algorithm works.

First, you've got to remember the algorithm.

So let me write it down quickly.

This is the sort of five by in N array.

So think of the array as being partitioned into, I'll call them, five columns.

So this picture of five dots by N over 5 dots-- this is dot, dot, dot.

So this is five.

Now, we didn't talk about it then, and there's a few different ways you could actually implement it, but let's say these-- the actual array is one-dimensional.

Let's say these are the first five items.

These are the next five items.

So, in other words, this matrix is stored column-by-column.

This is just a conceptual view.

So we can define it either way, however we want.

So I'm going to view it that way.

And then what the rest of the algorithm did was for sort each column, it's only five items, so you can sort it in constant time, each one.

But, in particular, what we care about is the median of those five items.

Then we recursively found the median of the medians.

This is the step we're going to have to change a little bit.

Then we-- leave a little bit of space.

Then we partition the array by x.

Meaning we split the array into items less than or equal to x and things greater than x.

We probably assumed there was only one value equal to x, but it doesn't matter.

And finally, we recurse on one of those two halves.

So this is a pretty crazy divide-and-conquer algorithm, one of the more sophisticated ones.

You don't need to know all the details here, just that it worked and it ran in linear time.

What's crazy about it is there are two recursive calls.

Usually, like in merge sort, where you do two recursive calls and spend linear time to do the stuff, like this partition, you get n log n time, like merge sort.

Here, because this array is a lot smaller, this is a size N over 5.

And this one was reasonably small; it was like [? M of ?] 7/10 N. Because 7/10 plus 1/5 is strictly less than 1, this ends up being linear time instead of n log n.

That's just review.

Now, what I'd like to do is the same thing, same analysis, or same algorithm, but now I want to analyze it in this two-level model.

So actually, I will erase this board.

So now my array has been partitioned into blocks of size B, like this picture.

In fact, it's quite similar.

Here, we're partitioning things into blocks, but they're size five.

That's different.

Now someone has partitioned my array into blocks of size B.

I need to count how many things I access.

Well, let's just look line-by-line at this code and see what we do.

Step one, we do absolutely nothing.

This is a conceptual picture, so zero cost, great.

Step one is zero, my favorite answer.

Step two, we sort each column.

How long does this take?

What am I doing?

It's right above me.

AUDIENCE: N over B.

ERIK DEMAINE: N over B because this is a scan.

It's a little bit weird of a scan.

We look at five items, and then we look at the next five items, and then the next five items.

But it's basically a scan.

You could think of it as almost five parallel scans, I suppose, or you could just break into the case where maybe if B is a constant, then it doesn't matter what you do.

But if B bigger than a constant, then reading five items, those are all probably going to be in one block, except the ones that straddle the block boundaries.

So in all cases, for step two-- maybe I should rewrite step one-- zero cost.

Step two, is order N over B plus 1, to be careful.

That's a scan.

Actually, it's two parallel scans because we have to write out these medians somewhere, so we'll have to.

Step three is recursively find the medians.

Now, before, we had in T of N is T of N over 5 plus T of 7/10 N plus linear.

In this new world-- this is regular running time.

In this new world, I'm going to use a different notation for the recurrence, MT of N for memory transfers.

This is a good old-fashioned time, and this is our new modern notion of time-- how many block transfers do I need to do for problem size N.

So this is a recursion, and should be MT of N over 5.

But, and this is important, for this to be a same problem of the same type, I need to know that the array that recursing on is stored contiguously.

Before, I didn't need to do that.

I could say, well, let's put the medians in the middle.

So now every fifth item in this array is my new subarray.

And so I could recursively call this thing and say, OK, here's my array, but really only think about every fifth item.

That's like a stride in the array.

And then the next recursive level, oh, only worry about every 25th item.

And every 5-cubed item, I'm going to stop computing, and so on.

And that would be fine for regular running time.

But when I get my stride gets bigger and bigger, at some point, every item is going to be in a different block.

That's bad.

I don't want to do that.

So when I find these medians, or when I recurse, I need that the medians that I'm recursing on are stored in a contiguous array.

Now, this is easy to do.

But we didn't have to do before.

That's the key difference.

Make sure they are stored contiguously.

I can do that because when I sort each column in one scan, I can have a second scan which is the output, which is the array of medians.

So as I'm scanning through the input, I'm going to output the median.

It's going to be 1/5 the size.

Then I've got all the medians nicely stored in a contiguous array.

So with order-one parallel scans, same time here, this is actually a legitimate recursive call.

Then we partition.

Partition, again, is a bunch of parallel scans, I think, three.

You've got one reading scan, which is you're reading through the array, and you've got to writing scans.

You're writing out the elements less than or equal to x, and you're writing out the elements greater than x.

But again, all of those are scans.

You're always writing the next element right after the previous one.

So if you already have that block in memory and if you assume that the number of blocks in cache is at least three, then three parallel scans is fine.

It's different from the CLRS partition algorithm.

That one was fancy to be in place.

We're not trying to be in place or fancy at all.

Let's just do it with a bunch of scans.

So now we have two arrays-- the element less than x, the elements greater than x.

Then we recurse on one of them, and those elements are consecutive already, so good.

This is a regular recursive call.

Again, we're maintaining the variant that the array is stored contiguously.

And by the old analysis, that array is sized at most 7/10 N.

So I get a new recurrence, which is MT of N is MT of N over 5 plus MT-- this analysis feels very "empty--" plus N over B-- sorry, bad joke-- N over B plus 1.

So basically the same recurrence, but now N over B plus 1 for what we're doing here.

But I had to change the algorithm a little bit for this recurrence to be correct, for it to correctly reflect the number of memory transfers.

Now all we need to do is solve the recurrence.

And actually, in some sense, more importantly, we need to figure out what the base case is.

Because we could say, all right, here's the usual base case.

If I have a constant-sized problem, well, that's going to be constant.

This is our base case for every recurrence we've ever done.

And that's enough usually.

It's going to give us a really bad answer here.

So let's go off to the side here and solve that recurrence.

So if that's my base case, well, in particular-- so this is some recursion tree.

It's very uneven, so it's kind of annoying to draw.

But what I know with this base case, this overall MT event is going to at least the number of leaves in the recursion tree.

So let's say MT of N is at least L of N, number of leaves in the recursion.

So this is really if I run the algorithm, how many base cases of constant size do I get?

And that satisfies-- so it's not obvious what that is.

There's no plus here.

Number of leaves is just how many leaves are over here, how many leaves are over here, and L of 1 equals 1, say, or some constant equals constant.

I happen to know, because I saw lots of recurrences, this solves to some N to the alpha.

I claim that L of N is N to the alpha for some constant alpha.

Why?

I'll just prove that it works.

So this is now N over 5 to the alpha, and this is 7/10 N to the alpha.

If it's going to work, this recurrence should be satisfied.

And now, if you look at this equation, there's a lot of N to the alphas, and they all cancel.

So I get 1 equals 1/5 to the alpha plus 7/10 to the alpha.

It's confusing because I was just watching the TV show Alphas, but no relation.

So this is now something purely in terms of alpha.

You just need to check that there is a real solution.

There is one.

You have to plug it into Wolfram Alpha or something, no pun intended.

Wow, they're just coming out today.

And then alpha is... next page...

I can't do this by hand.

Something like .83978.

So we get L of N is say at least N to the 0.8th bigger.

It's sublinear and that was enough when we cared about time but now it's bad news because N over B...

our goal was to get N over B+1.

If B is huge, if B is bigger than N to the 0.2, then we are not achieving this bound.

Right. We are always are paying at least N to the 0.8.

For example B is roughly N. We are way off!

But that's because we used the wrong base case.

Turns out if you use a better base case, things just work.

So let's do that. I think its going to be smaller.

So... the next base... I mean...

When you are doing cache full release analysis you never use this base case.

The first one you should think about is this one.

If you have a problem of size that fits in a constant number of blocks.

Well of course that's going to take... once they are read into the cache, you are not going to pay anything.

How long does it take to read a constant number of blocks into cache?

Constant number of memory transfers.

Okay, this is obviously a strictly better base case than this one.

Because we have the same thing on the right hand side as a constant but we've solved a larger problem.

So clearly you should cut here, instead of there.

Then the number of leaves in this recursion...

So same recurrence- different base case.

So we'd stop recursing conceptually in the analysis, the algorithm goes all the way down, but in the analysis we stop recursing when we reach a problem of size B.

The number of leaves in that new recursion tree will be N over B to the alpha.

That's good! That's smaller than N over B.

OK, now I'm going to wave my hands a little bit and say, MT of N is order N over B plus 1.

I guess to do that, you want to prove it the same way we did before when we solved this recurrence, which is by substitution.

You assume this is true, you plug it in, verify it can actually be done with some constants.

The intuition of what's going on is, in general, this recurrence is dominated by the root.

The root cost for this recursion is N over B plus 1.

So this is the root cost.

I claim that, up to constant factors, that is the overall cost.

Roughly because, as you go down the recursion tree, the cost is decreasing geometrically.

But that's not obvious for this recurrence because it's so uneven.

But it's kind of like the master method, a little fancier.

Intuitively, this should be obvious.

There's the root cost and then there's the other ones.

But to actually prove it, you should do substitution method.

I want to go to more interesting algorithms instead, but any questions before we continue?

All right.

So next algorithm, that was median, now we're going to do matrix multiplication via divide and conquer.

So what we just saw was an example where, in divide and conquer, in the analysis we think about the case where things fit in a constant number of blocks.

That was sort of case one.

The next example, matrix multiplication, will be the other case.

So you get to see both types.

So multiplying matrices, something we've done many times.

For example, in the FFT lecture and in the Strassen's algorithm, just to remind you.

I'm just thinking about the square case, although this generalizes.

We have two square matrices, N by N.

Normally, I would say C equals A times B, but I realized we used B for block side.

So this is going to be s equals x times y.

Hopefully that doesn't conflict with anything else, but no B's.

All right, so standard matrix.

Let's start with the standard algorithm.

Let's start by analyzing that.

Because if you're reasonably clever, this the standard algorithm is not so bad.

So in general, this won't matter too much.

Let's suppose we're doing z row-by-row, and let's say we're currently computing this product cell.

So that product cell is the dot product this ZIJ here is the dot product of this row with this column.

How do I compute dot products?

Two parallel scans.

Right?

I scan through this row and I parallel scan through this column.

Now, it depends the order in which you store x and y, but let's suppose we can store x in row major order, meaning row-by-row, and we store y in column major order, meaning column-by-column.

Then this will be an honest-to-goodness scan of a contiguous array.

Again, the order we store things in memory really matters.

So let's make our life ideal.

Let's say that this is row-by-row and this one is column-by-column, then hey, this is two parallel scans so order N over B to compute this cell.

OK, I claim that computing ZIJ costs N over B, so maybe plus 1.

Again, these are end-by-end matrices, so total size N squared, which means the total cost is what?

N cubed over B plus N squared, I guess.

Seems pretty good.

I mean, we had a running time of N cubed before and we divided by B. How could you possibly do better?

Well, by being smarter.

This is not optimal, you can do better.

It's not obvious, but let me just spend a little more time convincing you this is the right answer.

Not only is this big O, but for appropriate settings-- in the worst case this is going to be theta.

Because if you think of the order in which we're-- see, we look at these rows several times.

And if you look at, when I compute this cell and this cell and this cell of the z matrix, or the product matrix, each of them uses the same row of x.

So maybe you could reuse that.

You could reuse that row of x.

That might actually be free, depending on how B and N relate.

But the columns of y, those are different every time.

When I compute this one, I use the first column of y, when I compute this one I use the second column of y.

Unless the cache is so big that it can store all of y, which is like, you could store the entire problem in cache that's unrealistic.

So unless M is bigger than N squared, in this algorithm at least, you have to read a new column of y every single time.

So that's why it's theta N over B plus 1.

You need to spend N over B, assuming M is less than N squared.

OK.

And I claim this is not the best you can do because we're going to do better.

And we're going to do better by divide and conquer.

Now, you've already seen divide and conquer used for matrix multiplication to get Strassen's algorithm, and the idea there is to use blocks.

So this is sort of an algorithm you've already seen.

I'm going to divide the matrix z into N over 2 by N over 2 sub-matrices.

Each of these ZIJs is an N over 2 by N over 2 matrix.

And I do the same thing for x and y.

Numbers are right.

1, 2, y, 2, 1, and so on.

And you can write this out explicitly.

I prefer not to do all of it, but let's do one of them.

You can just think of these as two-by-two matrices, because matrix multiplication is associative and good things happen.

I can just take these two elements-- but they're actually matrices, sorry.

I might take these two and dot product with these two.

And I get x1,1 y1,1 plus x1,2 y2,1, and that's what I should set z1,1 to.

So this is a formula, but it's also a recursive algorithm.

It says, if I want to compute z I'm going to say, well, there are four sub-problems.

The first one is to compute z1,1, and I'm going to do that by recursively computing the product of x1,1 and y1,1, recursively computing the product of x1,2 y2,1 and then adding them together.

This is not recursive.

Addition is easy.

OK.

And there's two products here, two products here, two products here, two products here, a total of eight products, so we're going to have eight recursive calls in size N over 2.

If we look at the number of memory transfers, this is 8 times recursive call on N over 2 by N over 2 sub-matrices plus the cost of addition.

And I claim the cost of addition is at most N squared over B plus 1, because addition is basically parallel scans.

I can scan through x, scan through y.

As long as they're stored in the same order, I just am adding them element by element, and there's a third scan, which is writing out the z vector once things are linearized.

Now, for this to work, for this to be a true recursion, I need that, say, x1,1 and y1,1 are stored as contiguous things in memory.

So this means that the layout of a matrix, let's consider the matrix z, is going to be like the following.

I'm going to recursively lay out 1,1-- so when I say lay out, I mean what order do I store the elements in memory?

What order do I store the cells in memory?

And what I'm going to say is, recursively lay out the pieces-- there's four pieces-- recursively call layout of those and then concatenate them together.

That's my layout.

So I'm going to store all of these items, then I'm going to store all of these items, and then all of these items, then all these items.

How do I store these items, in what order?

Recursively.

So I'm going to divide them like this, store these before these before these before these, how do I store these?

Recursively.

OK, same recursion.

So it's a really weird order, it's a divide and conquer order.

There's only four things here.

In what order should I combine the four things?

Doesn't matter.

All that matters is that this is consecutive, this is consecutive, and this is consecutive, so that when I recurse, I'm recursing on consecutive chunks of memory.

Otherwise the analysis just won't work.

So for this to be right, got to have this layout.

OK.

Now we just need to solve the recurrence, and we're done.

I already told you, the base case we're going to use is this one.

We're going to use this one because it's stronger and better, and we'll need it, in this case, to get a better analysis.

You could solve it using the weaker base cases, you'll get larger numbers.

But if you use the strongest base case, MT of-- it's not M. Got to be a little careful.

Because N here is actually just one side length.

This is an end-by-end matrix, so the total size is N squared-- actually the total size is 3N squared, so this is going to be the square root of M over 3, at some constant, times the square root of N. It actually doesn't matter what the constant is.

But this is the size of-- this is the value of N for which all three matrices will fit in cache.

So I claim we know this costs at most M over B memory transfers, because we were kind of stroke here because we know that all of these guys fit in cache and because we know that they can store it consecutively in memory, well three consecutive chunks.

Once, no matter what I do, there are only M over B blocks there, and so at worst I read them all in.

But once the cache is filled with them, for the duration of this recursion, I won't be reading any other blocks, and so the cache will just stay full with the problem.

And so I never pay more than this.

So that's the base case.

Easy, but you have to think about it for a second.

Cool.

Now we have a recurrence and a base case, and now we have a good old fashioned recursion tree.

This one I can actually draw, because it's-- well, partly because it's nice and uniform.

It just explodes rather fast.

So at the top we have a cost of N squared over B plus 1, and we have eight recursive calls.

And the recursive calls are to something in size N over 2 squared over B, also known as N squared over 4B.

OK, so if I add up everything on this level, I get N squared over B, and if I add up everything on this level I'm going to get 8 times N over 4-- is that right?

Yeah.

So 2 times N squared over B.

OK.

I did that in order to verify that the cost per level is increasing geometrically, so all that will matter is the leaf level.

This is the proof of the master theorem.

When things are doubling at every step-- and this was just a special case, but every level would look the same-- every level of recursion, if you add them all up, you're getting twice as much as you had at the previous level.

So all that will matter is the leaf level.

OK, the leaf level.

Actually, maybe I'll do it over here.

First question is how many leaves are there?

The leaves are this thing.

So the way I would think about this is, because everything is nice and uniform, is 8 to the power of the number of levels.

What's the number of levels?

Well, we're dividing by 2 each time, so it's going to be log of something, but it's no longer log N because we're stopping early.

We're stopping when N reaches this value.

So it turns out that is N divided by that value.

This is, how many times do I have to multiply by 2 before I get to this, which is the same thing as how many times do I have to divide N by 2 before I get that?

Think about it.

OK, but 8 to the log.

This is 2 to the 3 times log.

2 to the log is just the thing.

So this is N over root M over B-- so many overs-- to the third power.

OK, this is starting to look familiar.

This is N cubed, that should appear somewhere, divided by square root of M over B.

This is the number of leaves.

Now, for each leaf we're paying this cost, so the overall cost of MT of N is going to be this times this.

So let's do that and simplify.

So MT of N is going to be big O, because we're taking the leaf level but there's some other things that's just going to lose us a factor of 2.

We have this thing multiplied by this thing.

So we've got N cubed over square root of M over B times M over B.

AUDIENCE: You-- PROFESSOR: I made a mistake.

Yea, thank you.

This was supposed to be cubed.

So this was M over B to the 1/2, so now we have, down here, M over B to the 3/2.

Thank you, thought that looked weird.

All right.

M over B to the 3/2.

OK.

AUDIENCE: [INAUDIBLE] PROFESSOR: Yeah.

What was I doing here?

This is supposed to be M over 3.

I was not missing a stroke, thank you.

M over 3, this is supposed to be M over 3.

Wow.

OK, so this is M over 3.

I'm just going to drop the-- well, I'll put it here.

But then I'm just going to write theta so I can forget about the 3, because that's just a square root of 3 factor.

So now this is going to be M to the 3/2.

That makes me much happier.

Did I get it right this time?

Let's double-check.

So this is square root of M to the 3 power, so that's M to the 1/2 cubed M to the 3/2.

I think that's good, this base case was square root of M.

OK, get it right.

So now this is M to the 3/2.

There is a square root that's going to come back, there's M to the 3/2 and there's an M upstairs, so the one cancels.

We're going to be left with N cubed over square root of M times B. OK.

There was a lower order term because I dropped this plus 1, but let's not worry about that right now.

Here we had N cubed divided by B, that was the standard algorithm.

Now we've got M cubed divided by B divided by square root of M.

That's big.

I mean, this is basically, you're dividing by-- well, square root of your cache size.

Wow.

So who knows how big that is, but say, between memory and disk, we're talking gigabytes.

So this is like billions.

Square root of a billion is still pretty big, like 10 to 100,000, so this is a huge amount faster than the standard algorithm.

You can do way better than scans.

Basically because we're reusing the same rows and columns over and over.

Now, this is standard matrix multiplication.

You might ask, what about Strassen's algorithm?

Well, same thing works.

You can do the same analysis Strassen, of course.

You get a similar improvement over Strassen.

You can do this for non-square matrices and all those good things.

And one minute left.

And it's going to be enough, I think, to cover LRU block replacement.

So here's what I want to say about LRU block replacement.

So in the beginning, we said the model is LRU, or it could have been FIFO.

Remember that?

And this algorithm will work just fine from an LRU perspective or a FIFO perspective if you think about it, but how do we know that LRU is as good as anything?

I claim, if you look at some sequence of block axises-- so suppose you know what B is-- and you count, for a cache of size M, how many memory transfers does LRU do, it's going to be within a factor of 2 of the optimal.

But not the optimal for a cache of size M, the optimal for a cache of size M over 2.

This is a bit of a weird statement.

I have a factor of 2 here and a factor of 2 here.

This is a cool idea called resource augmentation, fancy word for a simple idea.

This we're used to.

This is approximation algorithms.

OK, but this is an approximation in cost.

Here we're approximating the resources available to the algorithm.

We're changing the machine model, dividing M by 2, and we get a nice result.

Why is this OK?

Because, if you look at a bound like this, if you change M by a factor of 2, it will not change the bound by more than a factor of square root of 2.

So as long as you have at most, say, a linear or polynomial dependence on M, changing M by a constant factor will not change the overall running time of the cache for the previous algorithm.

This is why we can assume it's LRU.

The same is true for FIFO, it's probably true in expectation for random sequences.

And I will leave it at that.

If you want to see the-- do you want to see the proof of this theorem?

Tomorrow?

Or, Thursday?

Yes.

OK, we'll cover it on Thursday.