-- week of 6.046. Woohoo!

The topic of this final week, among our advanced topics, is cache oblivious algorithms. This is a particularly fun area, one dear to my heart because I've done a lot of research in this area. This is an area co-founded by Professor Leiserson. So, in fact, the first context in which I met Professor Leiserson was him giving a talk about cache oblivious algorithms at WADS '99 in Vancouver I think. Yeah, that has to be an odd year. So, I learned about cache oblivious algorithms then, started working in the area, and it's been a fun place to play.

But this topic in some sense was also developed in the context of this class. I think there was one semester, probably also '98-'99 where all of the problem sets were about cache oblivious algorithms. And they were, in particular, working out the research ideas at the same time. So, it must have been fun semester. We consider doing that this semester, but we kept it to the simple.

We know a lot more about cache oblivious algorithms by now as you might expect. Right, I think that's all the setting. I mean, it was kind of developed also with a bunch of MIT students in particular, M.Eng. student, Harold Prokop.

It was his M.Eng. thesis.

There is all the citations I will give for now.

I haven't posted yet, but there are some lecture notes that are already on my webpage.

But I will link to them from the course website that gives all the references for all the results I'll be talking about.

They've all been done in the last five years or so, in particular, starting in '99 when the first paper was published. But I won't give the specific citations in lecture. And, this topic is related to the topic of last week, multithreaded algorithms, although at a somewhat high level.

And then it's also dealing with parallelism in modern machines.

And we've had throughout all of these last two lectures, we've had this very simple model of a computer where we have random access. You can access memory at a cost of one. You can read and write a word of memory. There is some details on how big a word can be and whatnot. It's pretty basic, simple, flat model. And, at the multithreaded algorithm is the idea that, well, maybe you have multiple threads of computation running at once, but you still have this very flat memory. Everyone can access anything in memory at a constant cost. We're going to change that model now. And we are going to realize that a real machine, the memory of a real machine is some hierarchy. You have some CPU, you have some cache, probably on the same chip, level 1 cache, you have some

level 2 cache, if you're lucky, maybe you have some level 3 cache, before you get to main memory.

And then, you probably have a really big disk and probably there's even some cache out here, but I won't even think about that. So, the point is, you have lots of different levels of memory and what's changing here is that things that are very close to the CPU are very fast to access. Usually level 1 cache you can access in one clock cycle or a few.

And then, things get slower and slower.

Memory still costs like 70 ns or so to access a chunk out of.

And that's a long time. 70 ns is, of course, a very long time. So, as we go out here, we get slower. But we also get bigger.

I mean, if we could put everything at level 1 cache, the problem would be solved. But what would be a flat memory. Accessing everything in here, we assumed takes the same amount of time.

But usually, we can't afford, it's not even possible to put everything in level 1 cache.

I mean, there's a reason why there is a memory hierarchy.

Does anyone have a suggestion on what that reason might be?

It's like one of these limits in life.

Yeah? Fast memory is expensive.

That's the practical limitations indeed, that you could try to build more and more at level 1 cache and maybe you could try to, well, yeah.

Expenses is a good reason, and practically, that's why they may be the sizes are what they are.

But suppose really fast memory were really cheap.

There is a physical limitation of what's going on, yeah? The speed of light.

Yeah, that's a bit of a problem, right?

No matter how much, let's suppose you can only fit so many bits in an atom. You can only fit so many bits in a particular amount of space. If you want more bits, and you need more space, and the more space you have, the longer it's going to take for a round-trip.

So, if you assume your CPU is like this point in space, so it's relatively small and it has to get the data in, the bigger the data, the farther it has to be away.

But, you can have these cores around the CPU that are, we don't usually live in 3-D, and chips were usually in 2-D, but never mind. You can have the sphere that's closer to the CPU that's a lot faster to access.

And as you get further away it costs more.

And that's essentially what this model is representing, although it's a bit approximated from the intrinsic physics and geometry and whatnot.

But that's the idea. The latency, the round-trip time to get some of this memory has to be big.

In general, the costs to access memory is made up of two things.

There's the latency, the round-trip time, which in particular is limited by the speed of light.

And, plus the round-trip time, you also have to get the data out. And depending on how much data you want, it could take longer. OK, so there's something.

There could be, get this right, let's say, the amount of data divided by the bandwidth.

OK, the bandwidth is at what rate can you get the data out?

And if you look at the bandwidth of these various levels of memory, it's all pretty much the same.

If you have a well-designed computer the bandwidths should all be the same. OK, as you can still get data off disc really, really fast, usually at about the speed of your bus, and that the bus gets the CPU hopefully as fast as everything else.

So, even though they're slower, they're really only slower in terms of latency. And so, this part is maybe reasonable. The bandwidth looks pretty much the same universally. It's the latency that's going up. So, if the latency is going up but we still get to divide by the same amount of bandwidth, what should we do to make the access cost at all these levels about the same? This is fixed.

Let's say this is increasing, but this is still staying big.

What could we do to balance this formula?

Change the amounts. As the latency goes up, if we increase the amount, then the amortized cost to access one

element will go down. So, this is amortization in a very simple sense. So, this was to access a whole block, let's say, and this amount was the size of the block. So, the amortized cost, then, to access one element is going to be the latency divided by the size of the block, the amount plus one over the bandwidth. OK, so this is what you should implicitly be thinking in your head.

So, I'm just dividing here by the amounts because the amount is how many elements you get in one access, let's suppose.

OK, so we get this formula for the amortized cost.

The one over bandwidth is going to be good no matter what level we are on, I claim. There's no real fundamental limitation there except it might be expensive.

And the latency week at the amortized out by the amounts, so whatever the latency is, at the latency gets bigger out here, we just get more and more stuff and then we make these two terms equal, let's say. That would be a good way to balance things. So what particular, disc has a really high latency. Not only is there speed of light issues here, but there's actually the speed of the head moving on the tracks of the disk.

That takes a long time. There's a physical motion.

Everything else here doesn't usually have physical motion.

It's just electric. So, this is really, really slow and latency, so when you read something out of disk, you might as well read a lot of data from disc, like a megabyte or so. It's probably even old these days. Maybe you read multiple megabytes when you read anything from disk if you want these to be matched. OK, there's a bit of a problem with doing that. Any suggestions what the problem would be? So, you have this algorithm.

And, whenever it reads something off of desk, it reads an entire megabyte of stuff around the element it asked for. So the amortized cost to access is going to be reasonable, but that's actually sort of assuming something. Yeah?

Right. I'm assuming I'm ever going to use the rest of that data. If I'm going to read 10 MB around the one element that asked for, I access A bracket I, and I get 10 million items from A around I, it would be kind of good if the algorithm actually used that data for something.

It seems reasonable. So, this would be spatial locality. So, we want, I mean the goal of this world in cache oblivious algorithms and cache efficient algorithms in general is you want algorithms that perform well when this is happening.

So, this is the idea of blocking.

And we want the algorithm to use all or at least most of the elements in a block, a consecutive chunk of memory.

So, this is spatial locality.

Ideally, we'd use all of them right then.

But I mean, depending on your algorithm, that's a little bit tricky. There is another issue, though. So, you read in your thing into, read your 10 MB into main memory, let's say, and your memory, let's say, is at least, these days you should have a 4 GB memory or something.

So, you could read and actually a lot of different blocks into main memory. What you'd like is that you can use those blocks for as long as possible.

Maybe you don't even use them. If you have a linear time algorithm, you're probably only going to visit each element a constant number of times. So, this is enough.

But if your algorithm is more than linear time, you're going to be accessing elements more than once.

So, it would be a good idea not only to use all the elements of the blocks, but use them as many times as you can before you have to throw the block out. That's temporal locality.

So ideally, you even reuse blocks as much as possible.

So, I mean, we have all these caches.

So, I didn't write this word. Just in case I don't know how to spell it, it's not the money. We should use those caches for something. I mean, the fact that they store more than one block, each cache can store several blocks. How many?

Well, we'll get to that in a second.

OK, so this is the general motivation, but at this point the model is still pretty damn ugly.

If you wanted to design an algorithm that runs well on this kind of machine directly, it's possible but pretty difficult, and essentially never done, let's say, even though this is what real machines look like.

At least in theory, and pretty much in practice, the main thing to think about is two levels at a time.

So, this is a simplification where we can say a lot more about algorithms, a simplification over this model. So, in this model, each of these levels has different block sizes, and a different total sizes, it's a mess to deal with and

design algorithms for. If you just think about two levels, it's relatively easy. So, we have our CPU which we assume has a constant number of registers only.

So, you know, once it has a couple of data items, you can add them and whatnot.

Then we have this really fast pipe.

So, I draw it thick to some cache.

So this is cache. And, we have a relatively narrow pipe to some really big other storage, which I will call main memory. So, I mean, that's the general picture. Now, this could represent any two of these levels. It could be between L3 cache and make memory. That's maybe, what? The naming corresponds to best.

Or cache could in fact be main memory, what we consider the RAM of the machine, and what's called a memory over there to be the disk. It's whatever you care about.

And usually, if you have a program, that's what usually we assume everything fits in main memory.

Then you care about the caching behavior.

So you probably look between these two levels.

That's probably what matters the most inner program because the cost differential here is really big relative to the cost differential here. If your data doesn't even fit it main memory, and you have to go to disk, then you really care about this level because the cost differential here is huge. It's like six orders of magnitude, let's say. So, in practice you may think of just two memory levels that are the most relevant.

OK, now I'm going to define some parameters.

I'm going to call them cache and make memory just for clarity because I like to think of main memory just the way it used to be. And now all we have to worry about is this extra thing called cache.

It has some bounded size, and there's a block size.

The block size is B. and a number of blocks is M over B. So, the total size of the cache is M. OK, main memory is also blocked into blocks of size B. And we assume that it has essentially infinite size. We don't care about its size in this picture. It's whatever is big enough to hold the size of your algorithm, or data structure, or whatever. OK, so that's the general model. And for strange, historical reasons, which I don't want to get into, these things are called capital M and capital B.

Even though M sounds a lot like memory, it's really for cache, and don't ask. OK, this is to preserve history. OK, now what do we do with this model? It seems nice, but now what do we measure about it?

What I'm going to assume is that the cache is really fast.

So the CPU can access cache essentially instantaneously.

I still have to pay for the computation that the CPU is doing, but I'm assuming cache is close enough that I don't care.

And that may memory is so big that it has to be far away, and therefore, this pipe is a problem.

I mean, what I should really draw is that pipe is still thick, but is really long. So, the latency is high.

The bandwidth is still high. OK, and all transfers here happened as blocks. So, when you don't have something, so the idea is CPU asks for A of I, as for something in memory, if it's in the cache, it gets it. That's free.

Otherwise, it has to grab the entire block containing that element from main memory, brings it into cache, maybe kicks somebody out if the cache was full, and then the CPU can use that data and keep going.

Until it accesses something else that's not in cache, then it has to grab it from main memory.

When you kick something out, you're actually writing back to memory. That's the model.

So, we suppose the accesses to cache are free.

But we can still think about the running time of the algorithm. I'm not going to change the definition of running time. This would be the computation time, or the work if you want to use multithreaded lingo, computation time. OK, so we still have time, and T of N will still mean what it did before.

This is just an extra level of refinement of understanding of what's going on. Essentially, measuring the parallelism that we can exploit out of the memory system, that when you access something you actually get B items. So, this is the old stuff.

Now, what I want to do is count memory transfers.

These are transfers of blocks, so I should say block memory transfers between the two levels, so, between the cache and main memory. So, memory transfers are either reading reads or writes. Maybe I should say that.

These are number of block reads and writes from and to the main memory. OK, so I'm going to introduce some notation. This is new notation, so we'll see how it works out. MT of N I want to represent the number of memory

transfers instead of just normal time of the problem of size N. Really, this is a function that depends not only on N but also on these parameters, B and M, in our model. So, this is what it should be, MT_B,M(N), but that's obviously pretty messy, so I'm going to stick to MT of N.

But this will always, because mainly I care about the growth in terms of N. well, I care about the growth in terms of all things, but the only thing I could change is N. So, most of the time I only think about, like when we are writing recurrences, only N is changing. I can't recurse on the block size. I can't recurse on the size of cache. Those are given to me.

They're fixed. OK, so we'll be changing N mainly. But B and M always matter here.

They're not constants. They're parameters of the model. OK, easy enough.

This is something called the disk access model, if you like DAM models, or the external memory model, or the cache aware model. Maybe I should mention that; this is the cache aware. In general, you have some algorithm that runs on this kind of model, machine model.

That's a cache aware algorithm. OK, we're not too interested in cache aware algorithms. We've seen one, B trees. B trees are cache aware data structure. You assume that there is some block size, B, underlying.

Maybe you didn't see exactly this model.

In particular, it didn't really matter how big the cache was because you just wanted to know.

When I read B items, I can use all of them as much as possible and figure out where I fit among those B items, and that gives me log base B of N memory transfers instead of log N, which would be, if you just threw your favorite balanced binary search tree. So, log base B of N is definitely better than log base 2 of N.

B trees are a cache aware algorithm.

OK, what we would like to do today and next lecture is get cache oblivious algorithms. So, there's essentially only one difference between cache aware algorithms and cache oblivious algorithms. In cache oblivious algorithms, the algorithm doesn't know what B and M are.

So this is a bit of a subtle point, but very cool idea.

You assume that this is the model of the machine, and you care about the number of memory transfers between this cache of size M with blocking B, and main memory with blocking B. But you don't actually know what the model is. You don't know the other parameters of the model. It looks like this, but you don't know the width. You

don't know the height.

Why not? So, the analysis knows what B and M are. We are going to write some algorithms which look just like boring old algorithms that we've seen throughout the lecture. That's one of the nice things about this model. Every algorithm we have seen is a cache oblivious algorithm, all right, because we didn't even know the word cache in this class until today.

So, we already have lots of algorithms to choose from.

The thing is, some of them will perform well in this model, and some of them won't.

So, we would like to design algorithms that just like our old algorithms that happened to perform well in this context, no matter what B and M are. So, another way this is the same algorithm should work well for all values of B and M if you have a good cache oblivious algorithm.

OK, there are a few consequences to this assumption.

In a cache aware algorithm, you can explicitly say, OK, I'm blocking my memory into chunks of size B.

Here they are. I was going to store these B elements here, these B elements here, because you know B, you can do that.

You can say, well, OK, now I want to read these B items into my cache, and then write out these ones over here. You can explicitly maintain your cache. With cache oblivious algorithms, you can't because you don't know what it is.

So, it's got to be all implicit.

And this is pretty much how caches work anyway except for disk. So, it's a pretty reasonable model. In particular, when you access an element that's not in cache, you automatically fetch the block containing that element.

And you pay one memory transfer for that if it wasn't already there. Another bit of a catch here is, what if your cache is full? Then you've got to kick some block out of your cache. And then, so we need some model of which block gets kicked out because we can't control that.

We have no knowledge of what the blocks are in our algorithm.

So what we're going to assume in this model is the ideal thing, that when you fetch a new block, if your cache is full, you evict a block that will be used farthest in the future.

Sorry, the furthest. Farthest is distance.

Furthest is time. Furthest in the future.

OK, this would be the best possible thing to do.

It's a little bit hard to do in practice because you don't know the future generally, unless you're omniscient.

So, this is a bit of an idealized model.

But it's pretty reasonable in the sense that if you've read the reading handout number 20, this paper by Sleator and Tarjan, they introduce the idea of competitive algorithms.

So, we only talked about a small portion of that paper that moved to front heuristic for storing a list.

But, it also proves that there are strategies, and maybe you heard this in recitation.

Some people covered it; some didn't, that these are called paging strategies.

So, you want to maintain some cache of pages or blocks, and you pay whenever you have to access a block that's not in your cache. The best thing to do is to always kick out the block that will be used farthest in the future because that way you'll use all the blocks that are in there. This turns out to be the offline optimal strategy if you knew the future.

But, there are algorithms that are essentially constant competitive against this strategy.

I don't want to get into details because they're not exactly constant competitive. But they are sufficiently constant competitive for the purposes of this lecture that we can assume this, not have to worry about it.

Most of the time, we don't even really use this assumption. But there it is.

That's the cache oblivious model.

It makes things cleaner to think about just anything that should be done, will be done.

And you can simulate that with least recently used or whatever good heuristic that you want to that's competitive against the optimal. OK, that's pretty much the cache oblivious algorithm. Once you have the two level model, you just assume you don't know B and M.

You have this automatic request in writing, and whatnot.

A little bit more to say, I guess, it may be obvious at this point, but I've been drawing everything as tables.

So, it's not really clear what the linear order is.

Linear order is just the reading order.

So, although we don't explicitly say it most of the time, a typical model is that memory is a linear array.

Everything that you ever store in your program is written in this linear array. If you've ever programmed in Assembly or whatever, that's the model.

You have the address space, and any number between here and here, that's where you can actually, this is physical memory. This is all you can write to.

So, it starts at zero and goes out to, let's call it infinity over here. And, if you allocate some array, maybe it occupies some space in the middle.

Who knows? OK, we usually don't think about that much. What I care about now is that memory itself is blocked in this view.

So, however your stuff is stored in memory, it's blocked into clusters of length B.

So, if this is, let me call it one and be a little bit nicer. This is B.

This is position B plus one. This is 2B, and 2B plus one, and so on. These are the indexes into memory. This is how the blocking happens. If you access something here, you get that chunk from U, round it down to the previous multiple of B, round it up to the next multiple of B. That's what you always get.

OK, so if you think about some array that's maybe allocated here, OK, you have to keep in mind that that array may not be perfectly aligned with the blocks.

But more or less it will be so we don't care too much.

But that's a bit of a subtlety there.

OK, so that's pretty much the model.

So every algorithm we've seen, except B trees, is a cache oblivious algorithm. And our question is, now, we know how everything runs in terms of running time.

Now we want to measure the number of memory transfers, MT of N. I want to mention one other fact or theorem.

I'll put it in brackets because I don't want to state it precisely.

But if you have an algorithm that is efficient on two levels, so in other words, what we're looking at, if we just think about the two level world and your algorithm is cache oblivious, then it is efficient on any number of levels in your memory hierarchy, say, L levels. So, I don't want to define what efficient means. But the intuition is, if your machine really looks like this and you have a cache oblivious algorithm, you can apply the cache oblivious analysis for all B and M.

So you can analyze the number of memory transfers here, here, here, here, and here.

And if you have a good cache oblivious algorithm, the performances at all those levels has to be good.

And therefore, the whole performance is good.

Good here means asymptotically optimal up to constant factors, something like that. OK, so I don't want to prove that, and you can read the cache oblivious papers.

That's a nice fact about cache oblivious algorithms.

If you have a cache aware algorithm that tunes to a particular value of B, and a particular value of M, you're not going to have that problem.

So, this is one nice feature of cache obliviousness.

Another nice feature is when you are coding the algorithm, you don't have to put in B and M.

So, that simplifies things a bit.

So, let's do some algorithms. Enough about models.

OK, we're going to start out with some really simple things just to get warmed up on the analysis side.

The most basic thing you can do that's good in a cache oblivious world is scanning. So, scanning is just visiting the items in an array in order. So, visit A_1 up to A_N in order. For some notion of visit, this is presumably some constant time operation.

For example, suppose you want to compute the aggregate of the array. You want to sum all the elements in the array. So, you have one extra variable using, but you can store that in a register or whatever, so that's one simple example. Sum the array.

OK, so here's the picture. We have our memory.

Each of these cells represents one item, one element, log N bits, one word, whatever.

Our array is somewhere in here. Maybe it's there.

And we go from here to here to here to here.

OK, and so on. So, what does this cost?

What is the number of memory transfers?

We know that this is a linear time algorithm.

It takes order N time. What does it cost in terms of memory transfers? N over B, pretty much.

We like to say it's order N over B plus two or one in the big O. This is a bit of worry.

I mean, N could be smaller than B.

We really want to think about all the cases, especially because usually you're not doing this on something of size N. You're doing it on something of size k, where we don't really know much about k.

But in general, it's N over B plus one because we always need at least one memory transfer to look at something, unless N is zero. And in particular, it's plus two if you care about the constants.

If I don't write the big O, then it would be plus two at most because you could essentially waste the first block and that everything is fine for awhile.

And then, if you're unlucky, you essentially waste the last blocked. There is just one element in that block, and you're not getting much out of it.

Everything in the middle, though, every block between the first and last block has to be full.

So, you're using all of those elements.

So out of the N elements, you only have N over B blocks because the block has B elements.

OK, that's pretty trivial. Let me do something slightly more interesting, which is two scans at once.

OK, here we are not assuming anything about M.

we're not assuming anything about the size of the cache, just that I can hold a single block.

The last block that we visited has to be there.

OK, you can also do a constant number of parallel scans.

This is not really parallel in the sense of multithreaded, bit simulated parallelism. I mean, if you have a constant number, do one, do the other, do the other, come back, come back, come back, all right, visit them in turn round robin, whatever. For example, here's a cute piece of code. If you want to reverse an array, OK, then you can do it. This is a good puzzle.

You can do it by essentially two scans where you repeatedly swapped the first and last element.

So I was swapping A_i with N minus i plus one, and just restart at one. So, here's your array.

Suppose this is actually my array.

I swap these two guys, and I saw these two guys, and so on. That will reverse my array, and this should work hopefully the middle as well if it's odd.

It should not do anything. And you can view this as two scans. There is one scan that's coming in this way. There's also a reverse scan, ooh, some more sophisticated, coming back this way.

Of course, reverse scan has the same analysis.

And as long as your cache is big enough to store at least two blocks, which is a pretty reasonable assumption, so let's write it. Assuming the number of blocks in the cache, which is M over B, is at least two in this algorithm, the number of memory transfers is still order N over B plus one.

OK, the constant goes up maybe, but in this case it probably doesn't. But who cares.

OK, as long as you're doing a constant number of scans, and some constant number of arrays, it happens to be one of them's reversed, whatever, it will take, we call this linear time. It's linear in the number of blocks in your input. OK, great.

So now you can reverse an array: exciting.

Let's try another simple algorithm on another board.

Let's try binary search. So just like last week, we're going back to our basics here.

Scanning we didn't even talk about in this class.

Binary search is something we talked about a little bit.

It was a simple divide and conquer algorithm.

I hope you all remember it. And if we look at an array, and I'm not going to draw the cells here because I want to imagine a really big array, binary search, but suppose it always goes to left.

It starts by visiting this element in the middle.

Then ago so the quarter marked. Then it goes to the one eighth mark. OK, this is one hypothetical execution of a binary search. OK, and eventually it finds the element it's looking for. It finds where it fits at least. So x is over here.

So, we know that it takes log N time.

How many memory transfers of the take?

Now, I blocked this array into chunks of size B, blocks of size B. How many blocks do I touch?

This one's a little bit more subtle.

It depends on the relative sizes of N and B, yeah. Log base B of N would be a good guess. We would like it to be, let's say, hope, is that it's log base B of N because we know that B trees can search in what's essentially a sorted list of N items in log base B of N time.

That turns out to be optimal in the cache oblivious model or in the two level model you've got to pay log base B of N.

I won't prove that here. The same reason you need log N comparisons to do a binary search in the normal model.

Alas, it is possible to get log base B of N even without knowing B. But, binary search does not do it. Log of N over B, yes. So the number of memory transfers on N items is log of N over B also known as, let's say, plus one, also known as log N minus log B. OK, whereas log base B of N is log N divided by log B, OK, clearly this is much better than subtracting. So, this would be good, but this is bad. Most of the time, this is log N, which is no better, I mean, you're not using blocks at all essentially.

The idea is, out here, I mean, there's some little, tiny block that contains this thing. I mean, tiny depends on how big B is. But, each of these items will be in a different block until you get essentially within one block worth of x. When you get within one block worth of x, there's only like a constant number of blocks that matter, and so all of

these accesses are indeed within the same block. But, how many are there?

Well, just log B because you're only spending log B within a, if you're within an interval of size k, you're only going to spend log k steps in it. So, you're saving log B in here, but overall you're paying log N, so you only get log N minus log B plus some constant. OK, so this is bad news for binary search. So, not all of the algorithms we've seen are going to work well in this model.

We need a lot more thinking before we can solve what is essentially the binary search problem, finding an element in a sorted list, in log base B of N without knowing B.

OK, we know we could use B trees.

If you knew B, great, that works, and that's optimal. But without knowing B, it's a little bit harder. And this gets us into the world of divide and conquer. Also like last week, and like the first few weeks of this class, divide and conquer is your friend. And, it turns out divide and conquer is not the only tool, but it's a really useful tool in designing cache oblivious algorithms.

And, let me say why.

So, we'll see a bunch of divide and conquer based algorithms, cache oblivious. And, the intuition is that we can take all the favorite algorithms we have, obviously it doesn't always work.

Binary search was a divide and conquer algorithm.

It's not so great. But, in general, the idea is that your algorithm can just do the normal divide and conquer thing, right?

You divide your problem into subproblems of smaller size repeatedly, all the way down to problems of constant size, OK, just like before. But, if you're recursively dividing your problem into smaller things, at some point you can think about it and say, well, wait, I mean, the algorithm divides all the way, but we can think about the point at which the problem fits in a block or fits in cache. OK, and that's the analysis.

OK, we'll think about the time when your problem is small enough that we can analyze it in some other way.

So, usually, we analyze it recursively.

We get a recurrence. What we're changing, essentially, is the base case.

So, in the base case, we don't want to go down to a constant size. That's too far.

I'll show you some examples. We want to consider the point in recursion at which either the problem fits in cache, so it has size less than or equal to M, or it fits in order one blocks. That's another natural time to do it. Order one blocks would be even better than fitting in cache. So, this means a size order B.

OK, this will change the base case of the recurrence, and it will turn out to give us good answers instead of bad ones. So, let's do a simple example.

Our good friend order statistics, in particular, for finding medians. So, I hope you all know this by heart. Remember the worst case linear time, median finding algorithm by Bloom et al.

I'll write this fast. We partition our array.

It turns out, this is a good algorithm as it is. We partition our array conceptually into N over five, five tuples into little groups of five. This may not have been exactly how I wrote it last time. I didn't check.

But, it's the same algorithm. You compute the median of each five tuple. Then you recursively compute the median of the medians of these medians.

Then, you partition around x. So, that gave us some element that was roughly in the middle. It was within the middle half, I think. Partition around x, and then we show that you could always recurse on just one of the sides.

OK, this was our good old friend for computing, order statistics, or medians, or whatnot.

OK, so how much time does this, well, we know how much time this takes. It should be linear time.

But how many memory transfers does this take?

Well, conceptually partitioning that, I can do, in zero. Maybe I have to compute N over five, no big deal here. We're not thinking about computation. I have to find the median of each tuple. So, here it matters how my array is laid out. But, what I'm going to do is take my array, take the first five elements, and then the next five elements and so on.

Those will be my five tuples. So, I can implement this just by scanning, and then computing the median on those five elements, which I stored in the five registers on my CPU.

I'll assume that there are enough registers for that.

And, I compute the median, write it out to some array out here. So, it's going to be one element. So, the median of here goes into there. The median of these guys goes into there, and so on. So, I'm scanning in here, and in parallel, I'm scanning an output in here.

So, it's two parallel scans. So, that takes linear time.

So, this takes order N over B plus one memory transfers.

OK, then we have recursively compute the median of the medians. This step used to be T of N over five. Now it's MT of N over five, OK, with the same values of B and M.

Then we partition around x. Partitioning is also like three parallel scans if you work it out.

So, this is also going to take linear memory transfers, N over B plus one. And then, we recurse on one of the sides, and this is the fun part of the analysis which I won't repeat here. But, we get MT of, like, three quarters N. I think originally it was seven tenths, so we simplified to three quarters, which is hopefully bigger than seven tenths.

Yeah, it is. OK, so this is the new analysis. Now we get a recurrence.

So, let's do that.

So, the analysis is we get this MT of N is MT of N over five plus MT of three quarters N plus, this is just as before.

Before we had linear work here. And now, we have what we call linear number of memory transfers, linear number of blocks. OK, I'll sort of ignore this plus one. It's not too critical.

So, this is our recurrence. Now, it depends what our base case is. And, usually we would use a base case of constant size. So, let's see what happens if we use a base case of constant size just so that it's clear why this base case is so important. OK, this describes a recurrence as one of these hairy recurrences.

And, I don't want to use substitution.

I just want the intuition of why this is going to solve to something rather big. OK, and for me, the best intuition always comes from recursion trees.

If you don't know the solution to recurrence and you need a good guess, use recursion trees. And today, I will only give you good guesses. I don't want to prove anything with substitution because I want to get to the bigger ideas.

So, this is even messy from a recursion tree point of view because you have these unbalanced sizes where you start at the root with some of size N over B.

Then you split it into something size one fifth N over B, and something of size three quarters N over B, which is annoying because now this subtree will be a lot bigger than this one, or this one will terminate faster. So, it's pretty

unbalanced.

But, summing per level doesn't really tell you a lot at this point. But let's just look at the bottom level. Look at all the leaves in this recursion tree. So, that's the base cases.

How many base cases are there? This is an interesting question. We've never thought about it in the context of this recurrence. It gives a somewhat surprising answer. It was surprising to me the first time I worked it out. So, how many leaves does this recursion tree have? Well, we can write a recurrence. The number of leaves in a problem of size N, it's going to be the number of leaves in this problem plus the number of leaves in this problem plus zero. So, that's another recurrence.

We'll call this L of N. OK, now the base case is really relevant. It determines the solution to this recurrence. And let's, again, assume that in a problem of size one, we have one leaf.

That's our only base case. Well, it turns out, and here you need to guess, I think.

This is not particularly obvious.

Any of the TA's have guesses of the form of this solution?

Or anybody, not just TA's. But this is open to everyone.

If Charles were here, I would ask him.

I had to think for a while, and it's not linear, right, because you're somehow decreasing quite a bit.

So, it's smaller than linear, but it's more than a constant.

OK, it's actually more than polylog, so what's your favorite function in the middle? N over log N, that's still too big. Keep going.

You have an oracle here, so you can, N to the k, yeah, close. I mean, k is usually an integer. N to the alpha for some real number between zero and one. Yeah, that's what you meant.

Sorry. It's like the shortest mathematical joke. Let epsilon be less than zero or for a sufficiently large epsilon.

I don't know. So, you've got to use the right letters. So, let's suppose that it's N to the alpha. Then we would get this N over five to the alpha, and we'd get three quarters N to the alpha. When you have a nice recurrence like this, you can just try plugging in a guess and see whether it works, OK, and of course this will work only depending on alpha. So, we should get an equation on alpha here. So, everything has an N to the alpha, in fact, all of these

terms.

So, I can divide through my N to the alpha.

That's assuming that it's not zero or something.

That seems reasonable. So, we have one equals one fifth to the alpha plus three quarters to the alpha.

This is something you won't get on a final because I don't know any good way to solve this except with like Maple or Mathematica. If you're smart I'm sure you could compute it in a nicer way, but alpha is about 0.8, it turns out. So, the number of leaves is this sort of in between constant and linear.

Usually polynomial means you have an integer power.

Let's call it a polynomial. Why not?

There's a lot of leaves, is the point, and if we say that each leaf costs a constant number of memory transfers, we're in trouble because then the number of memory transfers has to be at least this.

If it's at least that, that's potentially bigger than N over B, I mean, bigger than in an asymptotic sense. This is little omega of N over B if B is big. If B is at least N to the 0.2 something, OK, or one seventh something.

But if, in particular, B is at least N to the 0.2, then this should be bigger than that.

So, this is a bad analysis because we're not going to get the answer we want, which is N over B.

The best you can do for median is N over B because you have to read all the element, and you should spend linear time. So, we want to get N over B.

This algorithm is N over B plus one.

So, this is why you need a good base case, all right?

So that makes the point. So, the question is, what base case should I use?

So, we have this recurrence What base case should I use? Constant was too small.

We have a couple of choices listed up here.

Any suggestions? B, OK, MT of B is?

The hard part. So, if my problem, if the size of my array fits in a block and I do all this stuff on it, how many

memory transfers could that take?

One, or a constant, depending on alignment.

OK, maybe it takes two memory transfers, but constant.

Good. That's clearly a lot better than this base case, MT of one equals order one, clearly stronger. So, hopefully, it gives the right answer, and now indeed it does.

I love this analysis. So, I'm going to wave my hands.

OK, but in particular, what this gives us, if we do the previous analysis, what is the number of leaves?

So, in the leaves, now L of B equals one instead of L of one equals one. So, this stops earlier.

When does it stop? Well, instead of getting N to the order of 0.8, whatever, we get N over B to the power of 0.8 whatever. OK, so it turns out the number of leaves is N over B to the alpha, which is little o of N over B. So, we don't care.

It's tiny. And, if you look at the root cost is N over B in the recursion tree, the leaf cost is little o of N over B, and if you wave your hands, and close your eyes, and squint, the cost should be geometrically decreasing as we go down, I hope, more or less. It's a bit messy because of all the things terminating, but let's say cost is roughly geometric. Don't do this in the final, but you won't have any messy recurrences like this.

So, don't worry. Down the tree, so you'd have to prove this formally, but I claim that the root cost dominates. And, the root cost is N over B.

So, we get N over B. OK, so this is a nice, linear time algorithm for order statistics for cache oblivious.

Great. This may turn you off a little bit, but even though this is like the simplest algorithm, it's also probably the most complicated analysis that we will do. In the future, our algorithms will be more complicated, and the analyses will be relatively simple. And usually, it's that way with cache oblivious algorithms.

So, I'm giving you this sort of as the intuition of why this should be enough. Then you have to prove it.

OK, let's go to another problem where divide and conquer is useful, our good friend, matrix multiplication.

I don't know how many times we've seen this in this class, but in particular we saw it last week with a recursive matrix multiply, multithreaded algorithm.

So, I won't give you the algorithm yet again, but we're going to analyze it in a very different way.

So, we have C and we have A, and actually up to you.

So, I could cover standard matrix multiplication, which is when you do it row by row, and column by column.

And, we could see why that's bad.

And then, we could do the recursive one and see why that's good. Or, we could skip the standard algorithm. So, how many people would like to see why the standard algorithm is bad?

Because it's not totally obvious.

One, two, three, four, five, half?

Wow, that's a lot of votes. Now, how many people want to skip to the chase? No one.

One, OK. And, everyone else is asleep.

So, that's pretty good, 50% awake, not bad.

OK, then, so standard matrix multiplication.

I'll do this fast because it is, I mean, you all know the algorithm, right? To compute this value of C; in A, you take this row, and in B you take this column.

Sorry I did a little bit sloppily.

But this is supposed to be aligned.

Right? So I take all of this stuff, I multiply it with all of the stuff, add them up, the dot product. That gives me this element.

And, let's say I do them in this order row by row.

So for every item in C, I loop over this row and this column, B, multiply them together.

That is an access pattern in memory.

So, exactly how much that costs depends how these matrices are laid out in memory. OK, this is a subtlety we haven't had to worry about before because everything was uniform. I'm going to assume to give the standard algorithm the best chances of being good, I'm going to store C in row major order, A in row major order, and B in

column major order.

So, everything is nice and you're scanning.

So then this inner product is a scan.

Cool. Sounds great, doesn't it? It's bad, though.

Assume A is row major, and B is column major.

And C, you could assume is really either way, but if I'm doing it row by row, I'll assume it's row major.

So, this is what I call the layout, the memory layout, of these matrices. OK, it's good for this algorithm, but the algorithm is not good.

So, it won't be that great.

So, how long does this take? How many memory transfers?

We know it takes M^3 time. Not going to try and beat M^3 here. Just going to try and get standard matrix multiplication going faster.

So, well, for each item over here I pay N over B to do the scans and get the inner product. So, N over B per item.

So, it's N over B, or we could go with the plus one here, to compute each c_ij. So that would suggest, as an upper bound at least, it's N^3 over B.

OK, and indeed that is the right bound, so theta.

This is memory transfers, not time, obviously.

That is indeed the case because if you look at consecutive, I do this c_ij, then this one, this one, this one, this one, keep incrementing j and keeping I fixed, right?

So, the row that I use stays fixed for a long time.

I get to reuse that if it happens, say that that fits a block maybe, I get to reuse that row several times if that happens to fit in cache. But the column is changing every single time. OK, so every time I moved here and compute the next c_ij, even if a column could fit in cache, I can't fit all the columns in cache.

And the columns that I'm visiting move, you know, they just scan across.

So, I'm scanning this whole matrix every time.

And unless you're entire matrix fits in cache, in which case you could do anything, I don't care, it will take constant time, or you'll take M over B time, enough to read it into the cache, do your stuff, and write it back out. Except in that boring case, you're going to have to pay N^2 over B for every row here because you have to scan the whole collection of columns.

You have to read this entire matrix for every row over here.

So, you really do need N^3 over B for the whole thing.

So, it's usually a theta. So, you might say, well, that's great. It's the size of my problem, the usual running time, divided by B.

And that was the case when we are thinking about linear time, N versus N over B. It's hard to beat N over B when your problem is of size N. But now we have a cubed.

And, this gets back to, we have good spatial locality.

When we read a block, we use the whole thing.

Great. It seems optimal.

But we don't have good temporal locality.

It could be that maybe if we stored the right things, we kept them around, we could them several times because we're using each element like a cubed number of times.

That's not the right way of saying it, but we're reusing the matrices a lot, reusing those items.

If we are doing N^3 work on N^2 things, we're reusing a lot.

So, we want to do better than this, and that's the recursive algorithm, which we've seen. So, we know the algorithm pretty much. I just have to tell you what the layout is. So, we're going to take C, partition of C_1-1, C_1-2, and so on.

So, I have an N by N matrix, and I'm partitioning into N over 2 by N over 2 submatrices, all three of them times whatever. And, I could write this out yet again but I won't. OK, we can recursively compute this thing with eight matrix multiplies, and a bunch of matrix additions. I don't care how many, but a constant number. We see that at least twice now, so I won't show it again. Now, how do I lay out the matrices? Any suggestions how I lay out the

matrices? I could lay them out in row major order. I'll call it major order.

But that might be less natural now.

We're not doing anything by rows or by columns.

So, what layout should I use? Yeah?

Quartet major order, maybe quadrant major order unless you're musically inclined, yeah.

Good idea. You've never seen this order before, so it's maybe not so natural.

Somehow I want to cluster it by blocks.

OK, I think that's about all. So, I mean, it's a recursive layout. This was not an easy question.

It's OK. Store matrices or lay out the matrices recursively by block. OK, I'm cheating a little bit.

I'm redefining the problem to say, assume that your matrices are laid out in this way. But, it doesn't really matter.

We can cheat, can't we?

In fact, it doesn't matter. You can turn a matrix into this layout without too much linear work, almost linear work.

Log factors, maybe.

OK, so if I want to store my matrix A as a linear thing, I'm going to recursively defined that layout to be recursively store the upper left corner, then store, let's say, the upper right corner.

It doesn't matter which order I do these.

I should have drawn this wider, then store the lower left corner, and then store the lower right corner recursively.

So, how do you store this? Well, you divide it in four, and lay out the top left, and so on.

OK, this is a recursive definition of how the element should be stored in a linear array.

It's a weird one, but this is a very powerful idea in cache oblivious algorithms.

We'll use this multiple times. OK, so now all we have to do is analyze the number of memory transfers.

How hard could it be? So, we're going to store all the matrices in this order, and we want to compute the number of memory transfers on an N by N matrix.

See, I lapsed and I switched to lowercase n.

I should, throughout this week, be using uppercase N because for historical reasons, any external memory kinds of algorithms, to level algorithms, always talk about capital N.

And, don't ask why. You should see what they define little n to be. OK, so, any suggestions on what the recurrence should be now? All his fancy setup with the recurrence is actually pretty easy.

So, definitely it involves multiplying matrices that are N over 2 by N over 2. So, what goes here?

Eight, thank you. That you should know.

And that the tricky part is what goes here.

OK, what goes here is, now, the fact that I can even write this, this is the matrix additions.

Ignore those for now. Suppose there weren't any.

I just have to recursively multiply.

The fact that this actually is eight times memory transfers of N over 2 relies on this layout. Right, I'm assuming that the arrays that I'm given are given as contiguous intervals and memory. If they aren't, I mean, if they're scattered all over memory, I'm screwed. There's nothing I can do.

So, but by assuming that I have this recursive layout, I know that the recursive multiplies will always deal with three consecutive chunks of memory, one for A, one for B, one for C, OK, no matter what I do.

Because these are stored consecutively, recursively I have that invariant.

And I can keep recursing. And I'm always dealing with three consecutive chunks of memory.

That's why I need this layout is to be able to say this.

OK, Now what does addition cost?

I'll just give you two matrices.

They're stored in some linear order, the same linear order among the three of them. Do I care what the linear order is? How should I add two matrices, get the output?

Yeah?

Right, if each of the three arrays I'm dealing with are stored in the same order, I can just scan in parallel through all three of them and just add corresponding elements, and output it to the third. So, I don't care what the order is, as long as it's consistent and I get N^2 over B.

I'll ignore plus one here. That's just looking at the entire matrix. So, there we go: another recurrence. We've seen this with N^2, and we just got N^3. But, it turns out now we get something cooler if we use the right base case.

So now we get to the base case, ah, the tricky part.

So, any suggestions what base case I should use?

The block size, good suggestion.

So, if we have something of size order B, we know that takes a constant number of memory transfers.

It turns out that's not enough. That won't solve it here.

But good guess. In this case, it's not the right answer. I'll give you some intuition why. We are trying to improve on N^3 over B. If you were just trying to get it divided by B, this is a great base case.

But here, we know that just the improvement afforded by the block size is not enough. We have to somehow use the fact that the cache is big. It's M, so however big M is, it's that big. OK, so if we want to get some improvement on this, we've got to have M in the formula somewhere, and there's no M's yet.

So, it's got to involve M. What's that?

MT of M over B? That would work, but MT of M is also OK, I mean, some constant times M, let's say. I want to make this constant small enough so that the entire problem fits in cache.

So, it's like one third. I think it's actually, oh wait, is it the square root of M actually?

Right, this is an N by N matrix.

So, it should be C times the square root of M.

Sorry. So, the square root of M by square root of M matrix has M entries.

If I make C like one third or something, then I can fit all three matrices in memory. Actually, one over square root of three would do, but who cares?

So, for some constant, C, now everything fits in memory. How many memory transfers does it take? One?

It's a bit too small, because I do have to read the problem in. And now, I mean, here was one because there's only one block to read.

Now how many blocks are there to read?

Constants? No.

B? No.

M over B, good. Get it right eventually.

That's the great thing about thinking with an oracle.

You can just keep guessing. M over B because we have cache size M. There are M over B blocks in that cache to read each one, OK?

This is maybe, you forgot what M was because we haven't used it for a long time.

But M is the number of elements in cache.

This is the number of blocks in cache.

OK, some of was saying B, and it's reasonable to assume that M over B is about B. That's like a square cache, but in general, we don't make that assumption.

OK, where are we? We're hopefully done, just about, good, because we have three minutes.

So, that's our base case. I have a square root here; I just forgot it. Now we just have to solve it.

Now, this is an easier recurrence, right?

I don't want to use the master method, because master method is not going to handle these B's and M's, and these crazy base cases. OK, master method would prove N^3. Great.

Master method doesn't really think about these kinds of cases. But with regression trees, if you remember way back to the proof of the master method, just look at the recursion tree as geometric up or down where everything is equal, and then you just add them up, every level. The point is that this is a nice recurrence. All of the sub problems are the same size, and that analysis always works, I say, when everything has the same size, all the children.

So, here's the recursion tree. We have N^2 over B at the top.

We split into eight subproblems where each one, the cost is one half N^2 over B.

I'm not going to write them all.

There they are. You add them up.

How much do you get? Well, there's eight of them.

Eight times a half is two. Four.

[LAUGHTER] Thanks. Four, right?

OK, I'm bad at arithmetic. I probably already said it, but there are three kinds of mathematicians, those who can add, and those who can't.

OK, why am I looking at this? It's obvious.

OK, so we keep going. This looks geometrically increasing. Right?

You just know in your heart that if you work out the first two levels, you can tell whether it's geometrically increasing, decreasing, or they're all equal, or something else.

And then you better think. But I see this as geometrically increasing. It will indeed be like 16 at the next level, I guess.

OK, it should be. So, it's increasing.

That means the leaves matter. So, let's work out the leaves.

And, this is where we use our base case.

So, we have a problem of size square root of M.

And so, yeah, you have a question?

Oh, indeed. I knew there was something.

I knew it was supposed to be two out here.

Thanks. This is why you're here.

It's actually N over two squared over B.

Thanks. I'm substituting N over 2 into this. OK, so this is actually N^2 over 4 B. So, I get two, because there are eight times one over four.

OK, I wasn't that far off then. It's still geometrically increasing, still the case, OK?

But now, it actually doesn't matter.

Whatever the cost is, as long as it's bigger than one, great. Now we look at the leaves.

The leaves are root M by root M.

I substitute root M into this: I get M over B with some constants. Who cares?

So, each leaf is M over B, OK, lots of them.

How many are there? This is the only, deal with recursion trees, counting the number of leaves is always the annoying part. Oh boy, well, we start with an N by N matrix. We stop when we get down to root N by root N matrix. So, that sounds like something.

Oh boy, I'm cheating here. Really?

That many? It sounds plausible.

OK, the claim is, and I'll cheat.

So I'm going to use the oracle here, and we'll figure out why this is the case. N over root N^3 leaves, hey what? I think here, it's hard to see the tree. But it's easy to see in the matrix. Let's enter the matrix.

We have our big matrix. We divided in half.

We recursively divide in half. We recursively divide in half.

You get the idea, OK?

Now, at some point these sectors, let's say one of these sectors, and each of these sectors, fits in cache.

And three of them fit in cache. So, that's when we stop the recursion in the analysis. The algorithm goes all the way.

But in the analysis, let's say we stop at M.

OK, now, how many leaves or problems are there?

Oh man, this is still not obvious.

OK, the number of leaf chunks here is, like, I mean, the number of these things is something like N over root M, right, the number of chunks.

But, it's a little less clear because I have so many of these.

But, all right, so let's just suppose, now, I think of normal, boring, matrix multiplication on chunks of this size. That's essentially what the leaves should tell me. I start with this big problem, I recurse out to all these little, tiny, multiply this by that, OK, this root M by root M chunk. OK, how many operations, how many multiplies do I do on those things?

N^3. But now, N, the size of my matrix in terms of these little sub matrices, is N over root M. So, it should be N over root M^3 subproblems of this size. If you work it out, normally we go down to things of constant size and we get exactly N^3 of them. Now we are stopping at this short point in saying, well, it's however many there are, cubed. OK, this is a bit of hand waving. You could work it out with the recurrence on the number of leaves.

But there it is. So, the total here is N over, let's work it out. N^3 over M to the three halves, that's this number of leaves, times the cost at each leaf, which is M over B. So, some of the N's cancel, and we get N^3 over B root M, which is a root M factor better than N^3 over B. It's actually quite a lot, the square root of the cache size.

That is optimal. The best two level matrix multiplication algorithm is N^3 over B root M memory transfers.

Pretty amazing, and I'm over time.

You can generalize this into all sorts of great things, but the bottom line is this is a great way to do matrix multiplication as a recursion. We'll see more recursion for cache oblivious algorithms on Wednesday.