6.046J Introduction to Algorithms, Fall 2005
Transcript – Lecture 25

The last lecture of 6.046. We are here today to talk more about cache oblivious algorithms. Last class, we saw several cache oblivious algorithms, although none of them quite too difficult. Today we will see two difficult cache oblivious algorithms, a little bit more advanced. I figure we should do something advanced for the last class just to get to some exciting climax. So without further ado, let's get started. Last time, we looked at the binary search problem.

Or, we looked at binary search, rather. And so, the binary search did not do so well in the cache oblivious context. And, some people asked me after class, is it possible to do binary search while cache obliviously? And, indeed it is with something called static search trees. So, this is really binary search. So, I mean, the abstract problem is I give you N items, say presorted, build some static data structure so that you can search among those N items quickly.

And quickly, I claim, means log base B of N. We know that with B trees, our goal is to get log base B of N. We know that we can achieve that with B trees when we know B. We'd like to do that when we don't know B. And that's what cache oblivious static search trees achieve. So here's what we're going to do. As you might suspect, we're going to use a tree. So, we're going to store our N elements in a complete binary tree. We can't use B trees because we don't know what B is. So, we'll use a binary tree. And the key is how we lay out a binary tree. The binary tree will have N nodes. Or, you can put the data in the leaves. It doesn't really matter.

So, here's our tree. There are the N nodes. And we're storing them, I didn't say, in order, you know, in the usual way, in order in a binary tree, which makes it a binary search tree. So we now had a search in this thing. So, the search will just start at the root and a walk down some root-to-leaf path. OK, and each point you know whether to go left or to go right because things are in order. So we're assuming here that we have an ordered universe of keys.

So that's easy. We know that that will take log N time. The question is how many memory transfers? We'd like a lot of the nodes near the root to be somehow closer and one block. But we don't know what the block size is. So are going to do is carve the tree in the middle level. We're going to use divide and conquer for our layout of the tree, how we order the nodes in memory. And the divide and conquer is based on cutting in the middle, which is a bit weird.

It's not our usual divide and conquer. And we'll see this more than once today. So, when you cut on the middle level, if the height of your original tree is log N, maybe log N plus one or something, it's roughly log N, then the top half will be log N over two. And at the height of the bottom pieces will be log N over two. How many nodes will there be in the top tree? N over two? Not quite. Two to the log N over two, square root of N. OK, so it would be about root N nodes over here. And therefore, there will be about root N subtrees down here, one for each, or a couple for each leaf.

OK, so we have these subtrees of root N, and there are about root N of them. OK, this is how we are carving our tree. Now, we're going to recurse on each of the pieces. I'd like to redraw this slightly, sorry, just to make it a little bit clearer. These triangles are really trees, and they are connected by edges to this tree up here. So what we are really doing is carving in the middle level of edges in the tree. And if N is not exactly a power of two, you have to round your level by taking floors or ceilings. But you cut roughly in the middle level of edges. There is a lot of edges here. You conceptually slice there. That gives you a top tree and the bottom tree, several bottom trees, each of size roughly root N.

OK, and then we are going to recursively layout these root N plus one subtrees, and then concatenate. So, this is the idea of the recursive layout. We sought recursive layouts with matrices last time. This is doing the same thing for a tree. So, I want to recursively layout the top tree. So here's the top tree. And I imagine it being somehow squashed down into a linear array recursively. And then I do the same thing for each of the bottom trees. So here are all the bottom trees. And I squashed each of them down into some linear order. And then I concatenate those linear orders. That's the linear order of the street.

And you need a base case. And the base case, just a single node, is stored in the only order of a single node there is. OK, so that's a recursive layout of a binary search tree. It turns out this works really well. And let's quickly do a little example just so it's completely clear what this layout is because it's a bit bizarre maybe the first time you see it. So let me draw my favorite picture.

So here's a tree of height four or three depending on how you count. We divide in the middle level, and we say, OK, that's the top tree. And then these are the bottom trees. So there's four bottom trees. So there are four children hanging off the root tree. They each have the same size in this case. They should all roughly be the same size. And the first we layout the top thing where we divide on the middle level. We say, OK, this comes first. And then, the bottom subtrees come next, two and three. So, I'm writing down the order in which these nodes are stored in an array. And then, we visit this tree so we get four, five, six. And then we visit this one so we get seven, eight, nine.

And then the subtree, 10, 11, 12, and then the last subtree. So that's the order in which you store these 15 nodes. And you can build that up recursively. OK, so the structure is fairly simple, just a binary structure which we know and love, but store it in this funny order. This is not depth research order or level order, lots of natural things you might try, none of which work in cache oblivious context.

This is pretty much the only thing that works. And the intuition as, well, we are trying to mimic all kinds of B trees. So, if you want a binary tree, well, that's the original tree. It doesn't matter how you store things. If you want a tree where the branching factor is four, well, then here it is. These blocks give you a branching factor of four. If we had more leaves down here, there would be four children hanging off of that node. And these are all clustered together consecutively in memory. So, if your block size happens to be three, then this is a perfect way to store things for a block size of three.

If you're block size happens to be probably 15, right, if we count the number of, right, the number of nodes in here is 15, if you're block size happens to be 15, then

this recursion will give you a perfect blocking in terms of 15. And in general, it's actually mimicking block sizes of 2^K-1. Think powers of two. OK, that's the intuition. Let me give you the formal analysis to make it clearer. So, we claim that there are order, log base B of N memory transfers. That's what we want to prove no matter what B is. So here's what we're going to do. You may recall last time when we analyzed divide and conquer algorithms, we wrote our recurrence, and that the base case was the key.

Here, in fact, we are only going to think about the base case in a certain sense. We don't have, really, recursion in the algorithm. The algorithm is just walking down some root-to-leaf path. We only have a recursion in a definition of the layout. So, we can be a little bit more flexible. We don't have to look at our recurrence. We are just going to think about the base case. I want to imagine, you start with the big triangle. That you cut it in the middle; you get smaller triangles. Imagine the point at which you keep recursively cutting.

So imagine this process. Big triangles halve in height each time. They're getting smaller and smaller, stop cutting at the point where a triangle fits in a block. OK, and look at that time. OK, the recursion actually goes all the way, but in the analysis let's think about the point where the chunk fits in a block in one of these triangles, one of these boxes fits in a block. So, I'm going to call this a recursive level.

So, I'm imagining expanding all of the recursions in parallel. This is some level of detail, some level of refinement of the trees at which the tree you're looking at, the triangle, has size. In other words, there is a number of nodes in that triangle is less than or equal to B. OK, so let me draw a picture. So, I want to draw sort of this picture but where instead of nodes, I have little triangles of size, at most, B. So, the picture looks something like this. We have a little triangle of size, at most, B. It has a bunch of children which are subtrees of size, at most, B, the same size. And then, these are in a chunk, and then we have other chunks that look like that in recursion potentially.

OK, so I haven't drawn everything. There would be a whole bunch of, between B and B^2, in fact, subtrees, other squares of this size. So here, I had to refine the entire tree here. And then I refined each of the subtrees here and here at these levels. And then it turned out after these two recursive levels, everything fits in a block. Everything has the same size, so at some point they will all fit within a block. And they might actually be quite a bit smaller than the block. How small?

So, what I'm doing is cutting the number of levels and half at each point. And I stop when the height of one of these trees is essentially at most log B because that's when the number of nodes at there will be B roughly. So, how small can the height be? I keep dividing at half and stopping when it's, at most, log B. Log B over two. So it's, at most, log B, it's at least half log B. Therefore, the number of nodes it here could be between the square root of B and B. So, this could be a lot smaller and less than a constant factor of a block, a claim that doesn't matter. It's OK. This could be a small square root of B. I'm not even going to write that it could be a small square root of B because that doesn't play a role in the analysis. It's a worry, but it's OK essentially because our bound only involves log B. It doesn't involve B.

So, here's what we do. We know that each of the height of one of these triangles of size, at most, B is at least a half log B. And therefore, if you look at a search path, so, when we do a search in this tree, we're going to start up here. And I'm going to

mess up the diagram now. We're going to follow some path, maybe I should have drawn it going down here. We visit through some of these triangles, but it's a root-to-node path in the tree. So, how many of the triangles could it visit? Well, the height of the tree divided by the height of one of the triangles. So, this visits, at most, log N over half log B triangles, which looks good.

This is log base B of N, mind you off factor of two. Now, what we worry about is how many blocks does a triangle occupy? One of these triangles should fit in a block. We know by the recursive layout, it is stored in a consecutive region in memory. So, how many blocks could occupy? Two, because of alignment, it might fall across the boundary of a block, but at most, one boundary. So, it fits in two blocks. So, each triangle fits in one block, but is in, at most, two blocks, memory blocks, size B depending on alignment. So, the number of memory transfers, in other words, a number of blocks we read, because all we are doing here is reading in a search, is at most two blocks per triangle.

There are this many triangles, so it's at most, 4 log base B of N, OK, which is order log base B of N. And there are papers about decreasing this constant 4 with more sophisticated data structures. You can get it down to a little bit less than two I think. So, there you go. So not quite as good as B trees in terms of the constant, but pretty good. And what's good is that this data structure works for all B at the same time.

This analysis works for all B. So, we have a multilevel memory hierarchy, no problem. Any questions about this data structure? This is already pretty sophisticated, but we are going to get even more sophisticated. Next, OK, good, no questions. This is either perfectly clear, or a little bit difficult, or both. So, now, I debated with myself what exactly I would cover next. There are two natural things I could cover, both of which are complicated. My first result in the cache oblivious world is making this data structure dynamic. So, there is a dynamic B tree that's cache oblivious that works for all values of B. And it gets log base B of N, insert, delete, and search. So, this just gets search in log base B of N. That data structure, our first paper was damn complicated, and then it got simplified.

It's now not too hard, but it takes a couple of lectures in an advanced algorithms class to teach it. So, I'm not going to do that. But there you go. It exists. Instead, we're going to cover our favorite problem sorting in the cache oblivious context. And this is quite complicated, more than you'd expect, OK, much more complicated than it is in a multithreaded setting to get the right answer, anyway.

Maybe to get the best answer in a multithreaded setting is also complicated. The version we got last week was pretty easy. But before we go to cache oblivious sorting, let me talk about cache aware sorting because we need to know what bound we are aiming for. And just to warn you, I may not get to the full analysis of the full cache oblivious sorting. But I want to give you an idea of what goes into it because it's pretty cool, I think, a lot of ideas.

So, how might you sort? So, cache aware, we assume we can do everything. Basically, this means we have B trees. That's the only other structure we know. How would you sort N numbers, given that that's the only data structure you have? Right, just add them into the B tree, and then do an in-order traversal. That's one way to sort, perfectly reasonable. We'll call it repeated insertion into a B tree. OK, we know in the usual setting, and the BST sort, where you use a balanced binary search tree, like red-black trees, that takes N log N time, log N per operation, and that's an

optimal sorting algorithm in the comparison model, only thinking about comparison model here.

So, how many memory transfers does this data structure takes? Sorry, this algorithm for sorting? The number of memory transfers is a function of N, and B_M of N is? This is easy. N insertions, OK, you have to think about N order traversal. You have to remember back your analysis of B trees, but this is not too hard. How long does the insertion take, the N insertions? N log base B of N. How long does the traversal take? Less time. If we think about it, you can get away with N over B memory transfers, so quite a bit less than this. This is bigger than N, which is actually pretty bad. N memory transfers means essentially you're doing random access, visiting every element in some random order. It's even worse. There's even a log factor. Now, the log factor goes down by this log B factor. But, this is actually a really bad sorting bound.

So, unlike normal algorithms, where using a search tree is a good way to sort, in cache oblivious or cache aware sorting it's really, really bad. So, what's another natural algorithm you might try, given what we know for sorting? And, even cache oblivious, all the algorithms we've seen are cache oblivious. So, what's a good one to try? Merge sort. OK, we did merge sort in multithreaded algorithms. Let's try a merge sort, a good divide and conquer thing. So, I'm going to call it binary merge sort because it splits the array into two pieces, and it recurses on the two pieces. So, you get a binary recursion tree. So, let's analyze it. So the number of memory transfers on N elements, so I mean it has a pretty good recursive layout, right? The two subarrays that we get what we partition our array are consecutive. So, we're recursing on this, recursing on this.

So, it's a nice cache oblivious layout. And this is even for cache aware. This is a pretty good algorithm, a lot better than this one, as we'll see. But, what is the recurrence we get? So, here we have to go back to last lecture when we were thinking about recurrences for recursive cache oblivious algorithms. I mean, the first part should be pretty easy. There's an O. Well, OK, let's put the O at the end, the divide and the conquer part at the end. The recursion is 2MT of N over two, good.

All right, that's just like the merge sort recurrence, and that's the additive term that you're thinking about. OK, so normally, we would pay a linear additive term here, order N because merging takes order N time. Now, we are merging, which is three parallel scans, the two inputs and the output. OK, they're not quite parallel interleaved. They're a bit funnily interleaved, but as long as your cache stores at least three blocks, this is also linear time in this setting, which means you visit each block a constant number of times.

OK, that's the recurrence. Now, we also need a base case, of course. We've seen two base cases, one MT of B, and the other, MT of whatever fits in cache. So, let's look at that one because it's better. So, for some constant, C, if I have an array of size M, this fits in cache, actually, probably C is one here, but I'll just be careful. For some constant, this fits in cache. A problem of this size fits in cache, and in that case, the number of memory transfers is, anyone remember? We've used this base case more than once before. Do you remember?

Sorry? CM over B. I've got a big O, so M over B. Order M over B because this is the size of the data. So, I mean, just to read it all in takes M over B. Once it's in cache, it doesn't really matter what I do as long as I use linear space for the right constant

here. As long as I use linear space in that algorithm, I'll stay in cache, and therefore, not have to write anything out until the very end and I spend M over B to write it out.

OK, so I can't really spend more than M over B almost no matter what algorithm I have, so long as it uses linear space. So, this is a base case that's useful pretty much in any algorithm. OK, that's a recurrence. Now we just have to solve it. OK, let's see how good binary merge sort is. OK, and again, I'm going to just give the intuition behind the solution to this recurrence. And I won't use the substitution method to prove it formally. But this one's actually pretty simple. So, we have, at the top, actually I'm going to write it over here. Otherwise I won't be able to see. So, at the top of the recursion, we have N over B costs. I'll ignore the constants. There is probably also on additive one, which I'm ignoring here. Then we split into two problems of half the size. So, we get a half N over B, and a half N over B.

OK, usually this was N, half N, half N. You should regard it as from lecture one. So, the total on this level is N over B. The total on this level is N over B. And, you can prove by induction, that every level is N over B. The question is how many levels are there? Well, at the bottom, so, dot, dot, dot, at the bottom of this recursion tree we should get something of size M, and then we're paying M over B. Actually here we're paying M over B.

So, it's a good thing those match. They should. So here, we have a bunch of leaves, all the size M over B. You can also compute the number of leaves here is N over M. If you want to be extra sure, you should always check the leaf level. It's a good idea. So we have N over M leaves, each costing M over B. This is an M. So, this is N over B also. So, every level here is N over B memory transfers. And the number of levels is one N over B? Log N over B. Yep, that's right. I just didn't hear it right. OK, we are starting at N. We're getting down to M. So, you can think of it as log N, the whole binary tree minus the subtrees log M, and that's the same as log N over M, OK, or however you want to think about it.

The point is that this is a log base two. That's where we are not doing so great. So this is actually a pretty good algorithm. So let me write the solution over here. So, the number of memory transfers on N items is going to be the number of levels times the cost of each level. So, this is N over B times log base two of N over M, which is a lot better than repeated insertion into a B tree. Here, we were getting N times log N over log B, OK, so N log N over log B. We're getting a log B savings over not proving anything, and here we are getting a factor of B savings, N log N over B. In fact, we even made it a little bit smaller by dividing this N by M. That doesn't matter too much. This dividing by B is a big one.

OK, so we're almost there. This is almost an optimal algorithm. It's even cache oblivious, which is pretty cool. And that extra little step, which is that you should be able to get on other log B factor improvement, I want to combine these two ideas. I want to keep this factor B improvement over N log N, and I want to keep this factor log B improvement over N log N, and get them together. So, first, before we do that cache obliviously, let's do it cache aware.

So, this is the third cache aware algorithm. This one was also cache oblivious. So, how should I modify a merge sort in order to do better? I mean, I have this log base two, and I want a log base B, more or less. So, how would I do that with merge sort? Yeah? Split into B subarrays, yeah. Instead of doing binary merge sort, this is what I

was hinting at here, instead of splitting it into two pieces, and recursing on the two pieces, and then merging them, I could split potentially into more pieces. OK, and to do that, I'm going to use my cache. So the idea is B pieces. This is actually not the best thing to do, although B pieces does work. And, it's what I was hinting at because I was saying I want a log B. It's actually not quite log B. It's log M over B. OK, but let's see. So, what is the most pieces I could split into?

Right, well, I could split into N pieces. That would be good, wouldn't it, at only one recursive level? I can't split into N pieces. Why? What happens wrong when I split into N pieces? That would be the ultimate. You can't merge, exactly. So, if I have N pieces, you can't merge in cache. I mean, so in order to merge in cache, what I need is to be able to store an entire block from each of the lists that I'm merging. If I can store an entire block in cache for each of the lists, then it's a bunch of parallel scans. So this is like testing the limit of parallel scanning technology. If you have K parallel scans, and you can fit K blocks in cache, then all is well because you can scan through each of those K arrays, and have one block from each of the K arrays in cache at the same time.

So, that's the idea. Now, how many blocks can I fit in cache? M over B. That's the biggest I could do. So this will give the best running time among these kinds of merge sort algorithms. This is an M over B way merge sort. OK, so now we get somewhat better recurrence. We split into M over B subproblems now, each of size, well, it's N divided by M over B without thinking. And, the claim is that the merge time is still linear because we have barely enough, OK, maybe I should describe this algorithm. So, we divide, because we've never really done non-binary merge sort. We divide into M over B equal size subarrays instead of two. Here, we are clearly doing a cache aware algorithm.

We are assuming we know what M over B is. So, then we recursively sort each subarray, and then we conquer. We merge. And, the reason merge works is because we can afford one block in cache. So, let's call it one cache block per subarray. OK, actually, if you're careful, you also need one block for the output of the merged array before you write it out. So, it should be M over B minus one. But, let's ignore some additive constants. OK, so this is the recurrence we get. The base case is the same.

And, what improves here? I mean, the per level cost doesn't change, I claim, because at the top we get N over B. This does before. Then we split into M over B subproblems, each of which costs a one over M over B factor times N over B. OK, so you add all those up, you still get N over B because we are not decreasing the number of elements. We're just splitting them. There's now M over B subproblems, each of one over M over B the size.

So, just like before, each level will sum to N over B. What changes is the number of levels because now we have bigger branching factor. Instead of log base two, it's now log base the branching factor. So, the height of this tree is log base M over B of N over M, I believe. Let me make sure that agrees with me. Yeah. OK, and if you're careful, this counts not quite the number of levels, but the number of levels minus one. So, I'm going to one plus one here. And the reason why is this is not quite the bound that I want.

So, we have log base M over B. What I really want, actually, is N over B. I claim that these are the same because we have minus, yeah, that's good. OK, this should come as rather mysterious, but it's because I know what the sorting bound should be as

I'm doing this arithmetic. So, I'm taking log base M over B of N over M. I'm not changing the base of the log. I'm just saying, well, N over M, that is N over B divided by M over B because then the B's cancel, and the M goes on the bottom. So, if I do that in the logs, I get log of N over B minus log of M over B minus, because I'm dividing. OK, now, log base M over B of M over B is one.

So, these cancel, and I get log base M over B, N over B, which is what I was aiming for. Why? Because that's the right bound as it's normally written. OK, that's what we will be trying to get cache obliviously. So, that's the height of the search tree, and at each level we are paying N over B memory transfers. So, the overall number of memory transfers for this M over B way merge sort is the sorting bound.

This is, I'll put it in a box. This is the sorting bound, and it's very special because it is the optimal number of memory transfers for sorting N items cache aware. This has been known since, like, 1983. OK, this is the best thing to do. It's a really weird bound, but if you ignore all the divided by B's, it's sort of like N times log base M of N. So, that's little bit more reasonable. But, there's lots of divided by B's. So, the number of the blocks in the input times log base the number of blocks in the cache of the number of blocks in the input. That's a little bit more intuitive. That is the bound.

And that's what we are aiming for. So, this algorithm, crucially, assume that we knew what M over B was. Now, we are going to try and do it without knowing M over B, do it cache obliviously. And that is the result of only a few years ago. Are you ready? Everything clear so far? It's a pretty natural algorithm. We were going to try to mimic it essentially and do a merge sort, but not M over B way merge sort because we don't know how.

We're going to try and do it essentially a square root of N way merge sort. If you play around, that's the natural thing to do. The tricky part is that it's hard to merge square root of N lists at the same time, in a cache efficient way. We know that if the square root of N is bigger than M over B, you're hosed if you just do a straightforward merge. So, we need a fancy merge. We are going to do a divide and conquer merge. It's a lot like the multithreaded algorithms of last week, try and do a divide and conquer merge so that no matter how many lists are merging, as long as it's less than the square root of N, or actually cubed root of N, we can do it cache efficiently, OK? So, we'll do this, we need a bit of setup. But that's where we're going, cache oblivious sorting. So, we want to get the sorting bound, and, yeah.

It turns out, to do cache oblivious sorting, you need an assumption about the cache size. This is kind of annoying, because we said, well, cache oblivious algorithms should work for all values of B and all values of M. But, you can actually prove you need an additional assumption in order to get this bound cache obliviously. That's the result of, like, last year by Garrett Brodel. So, and the assumption is, well, the assumption is fairly weak. That's the good news. OK, we've actually made an assumption several times. We said, well, assuming the cache can store at least three blocks, or assuming the cache can store at least four blocks, yeah, it's reasonable to say the cache can store at least four blocks, or at least any constant number of blocks. This is that the number of blocks that your cache can store is at least B to the epsilon blocks.

This is saying your cache isn't, like, really narrow. It's about as tall as it is wide. This actually gives you a lot of sloth. And, we're going to use a simple version of this

assumption that M is at least B^2. OK, this is pretty natural. It's saying that your cache is at least as tall as it is wide where these are the blocks. OK, the number of blocks is it least the size of a block. That's a pretty reasonable assumption, and if you look at caches these days, they all satisfy this, at least for some epsilon.

Pretty much universally, M is at least B^2 or so. OK, and in fact, if you think from our speed of light arguments from last time, B^2 or B^3 is actually the right thing to do. As you go out, I guess in 3-D, B^2 would be the surface area of the sphere out there. OK, so this is actually the natural thing of how much space you should have at a particular distance. Assuming we live in a constant dimensional space, that assumption would be true. This even allows going up to 42 dimensions or whatever, OK, so a pretty reasonable assumption. Good. Now, we are going to achieve this bound. And what we are going to try to do is use an N to the epsilon way merge sort for some epsilon. And, if we assume that M is at least B^2, the epsilon will be one third, it turns out.

So, we are going to do the cubed root of N way merge sort. I'll start by giving you and analyzing the sorting algorithms, assuming that we know how to do merge in a particular bound. OK, then we'll do the merge. The merge is the hard part. OK, so the merge, I'm going to give you the black box first of all. First of all, what does merge do? The K way merger is called the K funnel just because it looks like a funnel, which you'll see. So, a K funnel is a data structure, or is an algorithm, let's say, that looks like a data structure. And it merges K sorted lists. So, supposing you already have K lists, and they're sorted, and assuming that the lists are relatively long, so we need some additional assumptions for this black box to work, and we'll be able to get them as we sort.

We want the total size of those lists. You add up all the elements, and all the lists should have size at least K^3 is the assumption. Then, it merges these lists using essentially the sorting bound. Actually, I should really say theta K^3. I also don't want to be too much bigger than K^3. Sorry about that. So, the number of memory transfers that this funnel merger uses is the sorting bound on K^3, so K^3 over B, log base M over B of K^3 over B, plus another K memory transfers.

Now, K memory transfers is pretty reasonable. You've got to at least start reading each list, so you got to pay one memory transfer per list. OK, but our challenge in some sense will be getting rid of this plus K. This is how fast we can merge. We'll do that after. Now, assuming we have this, let me tell you how to sort. This is, eventually enough, called funnel sort. But in a certain sense, it's really cubed root of N way merge sort. OK, but we'll analyze it using this. OK, so funnel sort, we are going to define K to be N to the one third, and apply this merger. So, what do we do? It's just like here. We're going to divide our array into N to the one third.

I mean, it they should be consecutive subarrays. I'll call them segments of the array. OK, for cache oblivious, it's really crucial how these things are laid out. We're going to cut and get consecutive chunks of the array, N to the one third of them. Then I'm going to recursively sort them, and then I'm going to merge. OK, and I'm going to merge using the K funnel, the N to the one third funnel because, now, why do I use one third? Well, because of this three. OK, in order to use the N to the one third funnel, I need to guarantee that the total number of elements that I'm merging is at least the cube of this number, K^3. The cube of this number is N. That's exactly how many elements I have in total.

OK, so this is exactly what I can apply the funnel. It's going to require that I have it least K^3 elements, so that I can only use an N to the one third funnel. I mean, if it didn't have this requirement, I could just say, well, I have N lists each of size one. OK, that's clearly not going to work very well for our merger, I mean, intuitively because this plus K will kill you. That will be a plus M which is way too big.

But we can use an N to the one third funnel, and this is how we would sort. So, let's analyze this algorithm. Hopefully, it will give the sorting bound if I did everything correctly. OK, this is pretty easy. The only thing that makes this messy as I have to write the sorting bound over and over. OK, this is the cost of the merge. So that's at the root. But K^3 in this case is N. So at the root of the recursion, let me write the recurrence first. Sorry. So, we have memory transfers on N elements is N to the one third. Let me get this right. Yeah, N to the one third recursions, each of size N to the two thirds, OK, plus this time, except K^3 is N.

So, this is plus N over B, log base M over B of N over B plus cubed root of M. This is additive plus K term. OK, so that's my recurrence. The base case will be the usual. MT is some constant times M is order M over B. So, we sort of know what we should get here. Well, not really. So, in all the previous recurrence is, we have the same costs at every level, and that's where we got our log factor. Now, we already have a log factor, so we better not get another one. Right, this is the bound we want to prove. So, let me cheat here for a second.

All right, indeed. You may already be wondering, this N to the one third seems rather large. If it's bigger than this, we are already in trouble at the very top level of the recursion. So, I claim that that's OK. Let's look at N to the one third. OK, there is a base case here which covers all values of N that are, at most, some constant times N. So, if I'm in this case, I know that N is at least as big as the cache up to some constant.

OK, now the cache is it least B^2, we've assumed. And you can do this with B to the one plus epsilon if you're more careful. So, N is at least B^2, OK? And then, I always have trouble with these. So this means that N divided by B is omega root N. OK, there's many things you could say here, and only one of them is right. So, why? So this says that the square root of N is at least B, and so N divided by B is at most N divided by square root of N. So that's at least the square root of N if you check that all out. I'm going to go through this arithmetic relatively quickly because it's tedious but necessary. OK, the square root of N is strictly bigger than cubed root of N. OK, so that means that N over B is strictly bigger than N to the one third.

Here we have N over B times something that's bigger than one. So this term definitely dominates this term in this case. As long as I'm not in the base case, I know N is at least order M. This term disappears from my recurrence. OK, so, good. That was a bit close. Now, what we want to get is this running time overall. So, the recursive cost better be small, better be less than the constant factor increase over this.

So, let's write the recurrence. So, we get N over B, log base M over B, N over B at the root. Then, we split into a lot of subproblems, N to the one third subproblems here, and each one costs essentially this but with N replaced by N to the two thirds. OK, so N to the two thirds log base M over B, oops I forgot to divide it by B out here, of N to the two thirds divided by B. That's the cost of one of these nodes, N to the one third of them. What should they add up to? Well, there is N to the one third, and

there's an N to the two thirds here that multiplies out to N. So, we get N over B. This looks bad. This looks the same. And we don't want to lose another log factor. But the good news is we have two thirds in here.

OK, this is what we get in total at this level. It looks like the sorting bound, but in the log there's still a two thirds. Now, a power of two thirds and a log comes out as a multiple of two thirds. So, this is in fact two thirds times N over B, log base M over B of N over B, the sorting bound. So, this is two thirds of the sorting bound. And this is the sorting bound, one times the sorting bound.

So, it's going down geometrically, yea! OK, I'm not going to prove it, but it's true. This went down by a factor of two thirds. The next one will also go down by a factor of two thirds by induction. OK, if you prove it at one level, it should be true at all of them. And I'm going to skip the details there. So, we could check the leaf level just to make sure. That's always a good sanity check. At the leaves, we know our cost is M over B.

OK, and how many leaves are there? Just like before, in some sense, we have N/M leaves. OK, so in fact the total cost at the bottom is N over B. And it turns out that that's what you get. So, you essentially, it looks funny, because you'd think that this would actually be smaller than this at some intuitive level. It's not. In fact, what's happening is you have this N over B times this log thing, whatever the log thing is. We don't care too much. Let's just call it log.

What you are taking at the next level is two thirds times that log. And at the next level, it's four ninths times that log and so on. So, it's geometrically decreasing until the log gets down to one. And then you stop the recursion. And that's what you get N over B here with no log. So, what you're doing is decreasing the log, not the N over B stuff. The two thirds should really be over here. In fact, the number of levels here is log, log N.

It's the number of times you have to divide a log by three halves before you get down to one, OK? So, we don't actually need that. We don't care how many levels are because it's geometrically decreasing. It could be infinitely many levels. It's geometrically decreasing, and we get this as our running time. MT of N is the sorting bound for funnel sort. So, this is great. As long as we can get a funnel that merges this quickly, we get a sorting algorithm that sorts as fast as it possibly can. I didn't write that on the board that this is asymptotically optimal. Even if you knew what B and M were, this is the best that you could hope to do. And here, we are doing it no matter what, B and M are.

Good. Get ready for the funnel. The funnel will be another recursion. So, this is a recursive algorithm in a recursive algorithm. It's another divide and conquer, kind of like the static search trees we saw at the beginning of this lecture. So, these all tie together. All right, the K funnel, so, I'm calling it K funnel because I want to think of it at some recursive level, not just N to the one third. OK, we're going to recursively use, in fact, the square root of K funnel. So, here's, and I need to achieve that bound. So, the recursion is like the static search tree, and a little bit hard to draw on one board, but here we go.

So, we have a square root of K funnel. Recursively, we have a buffer up here. This is called the output buffer, and it has size K^3, and just for kicks, let's suppose it that filled up a little bit. And, we have some more buffers. And, let's suppose they've

been filled up by different amounts. And each of these has size K to the three halves, of course. Halves, these are called buffers, let's say, with the intermediate buffers. And, then hanging off of them, we have more funnels, the square root of K funnel here, and a square root of K funnel here, one for each buffer, one for each child of this funnel.

OK, and then hanging off of these funnels are the input arrays. OK, I'm not going to draw all K of them, but there are K input arrays, input lists let's call them down at the bottom. OK, so the idea is we are going to merge bottom-up in this picture. We start with our K input arrays of total size at least K^3. That's what we're assuming we have up here. We are clustering them into groups of size square root of K, so, the square root of K groups, throw each of them into a square root of K funnel that recursively merges those square root of K lists. The output of those funnels we are putting into a buffer to sort of accumulate what the answer should be. These buffers have besides exactly K to the three halves, which might not be perfect because we know that on average, there should be K to the three halves elements in each of these because there's K^3 total, and the square root of K groups.

So, it should be K^3 divided by the square root of K, which is K to the three halves on average. But some of these will be bigger. Some of them will be smaller. I've drawn it here. Some of them had emptied a bit more depending on how you merge things. But on average, these will all fill at the same time. And then, we plug them into a square root of K funnel, and that we get the output of size K^3. So, that is roughly what we should have happen.

OK, but in fact, some of these might fill first, and we have to do some merging in order to empty a buffer, make room for more stuff coming up. That's the picture. Now, before I actually tell you what the algorithm is, or analyze the algorithm, let's first just think about space, a very simple warm-up analysis. So, let's look at the space excluding the inputs and outputs, those buffers. OK, why do I want to exclude input and output buffers? Well, because I want to only count each buffer once, and this buffer is actually the input to this one and the output to this one. So, in order to recursively count all the buffers exactly once, I'm only going to count these middle buffers. And then separately, I'm going to have to think of the overall output and input buffers. But those are sort of given. I mean, I need K^3 for the output. I need K^3 for the input. So ignore those overall. And that if I count the middle buffers recursively, I'll get all the buffers.

So, then we get a very simple recurrence for space. S of K is roughly square root of K plus one times S of square root of K plus order K^2, K^2 because we have the square root of K of these buffers, each of size K to the three halves. Work that out, does that sound right? That sounds an awful lot like K^3, but maybe, all right. Oh, no, that's right. It's K to the three halves times the square root of K, which is K to the three halves plus a half, which is K to the four halves, which is K^2. Phew, OK, good. I'm just bad with my arithmetic here. OK, so K^2 total buffering here. You add them up for each level, each recursion, and the plus one here is to take into account the top guy, the square root of K bottom guys, so the square root of K plus one.

If this were, well, let me just draw the recurrence tree. There's many ways you could solve this recurrence. A natural one is instead of looking at K, you look at log K, because here at log K is getting divided by two. I just going to draw the recursion trees, so you can see the intuition. But if you are going to solve it, you should

probably take the logs, substitute by log. So, we have the square root of K. plus one branching factor.

And then, the problem is size square root of K, so this is going to be K, I believe, for each of these. This is square root of K squared is the cost of these levels. And, you keep going. I don't particularly care what the bottom looks like because at the top we have K^2. That we have K times root K plus one cost at the next level. This is K to the three halves plus K. OK, so we go from K^2 to K to the three halves plus K. This is a super-geometric. It's like an exponential geometric decrease. This is decreasing really fast. So, it's order K^2. That's my hand-waving argument. OK, so the cost is basically the size of the buffers at the top level, the total space. We're going to need this. It's actually theta K^2 because I have a theta K^2 here.

We are going to be this in order to analyze the time. That's why it mentioned it. It's not just a good feeling that the space is not too big. In fact, the funnel is a lot smaller than a total input size. The input size is K^3. But that's not so crucial. What's crucial is that it's K^2, and we'll use that in the analysis. OK, naturally, this thing is laid out recursively. You recursively store the funnel, top funnel. Then, for example, you write out each buffer as a consecutive array, in this case.

There's no recursion there. So just write them all out one by one. Don't interleave them or anything. Store them in order. And that, you write out recursively these funnels, the bottom funnels. OK, any way you do it recursively, as long as each funnel remains a consecutive chunk of memory, each buffer remains a consecutive chuck of memory, the time analysis that we are about to do will work.

OK, let me actually give you the algorithm that we're analyzing. In order to make the funnel go, what we do is say, initially, all the buffers are empty. Everything is at the bottom. And what we are going to do is, say, fill the root buffer. Fill this one. And, that's a recursive algorithm, which I'll define in a second, how to fill a buffer. Once it's filled, that means everything has been pulled up, and then it's merged. OK, so that's how we get started. So, merge means to merge algorithm is fill the topmost buffer, the topmost output buffer.

OK, and now, here's how you fill a buffer. So, in general, if you expand out this recursion all the way, in the base case, I didn't mention you sort of get a little node there. So, if you look at an arbitrary buffer in this picture that you want to fill, so this one's empty and you want to fill it, then immediately below it will be a vertex who has two children, two other buffers. OK, maybe they look like this. You have no idea how big they are, except they are the same size. It could be a lot smaller than this one, a lot bigger, we don't know. But in the end, you do get a binary structure out of this just like we did with the binary search tree at the beginning. So, how do we fill this buffer? Well, we just merge these two child buffers as long as we can.

So, we merge the two children buffers as long as they are both non-empty. So, in general, the invariant will be that this buffer, let me write down a sentence. As long as a buffer is non-empty, or whatever is in that buffer, and hasn't been used already, it's a prefix of the merged output of the entire subtree beneath it. OK, so this is a partially merged subsequence of everything down here. This is a partially merged subsequence of everything down here. I can just merge element by element off the top, and that will give me outputs to put there until one of them gets emptied. And, we have no idea which one will empty first just because it depends on

the order. OK, whenever one of them empties, we recursively fill it, and that's it. That's the algorithm.

Whenever one empties -- -- we recursively fill it. And at the base case at the leaves, there's sort of nothing to do. I believe you just sort of directly read from an input list. So, at the very bottom, if you have some note here that's trying to merge between these two, that's just a straightforward merge between two lists. We know how to do that with two parallel scans. So, in fact, we can merge the entire thing here and just spit it out to the buffer. Well, it depends how big the buffer is. We can only merge it until the buffer fills.

Whenever a buffer is full, we stop and we pop up the recursive layers. OK, so we keep doing this merge until the buffer we are trying to fill fills, and that we stop, pop up. OK, that's the algorithm for merging. Now, we just have to analyze the algorithm. It's actually not too hard, but it's a pretty clever analysis. And, to top it off, it's an amortization, your favorite. OK, so we get one last practice at amortized analysis in the context of cache oblivious algorithms. So, this is going to be a bit sophisticated. We are going to combine all the ideas we've seen. The main analysis idea we've seen is that we are doing this recursion in the construction, and if we imagine, we take our K funnel, we split it in the middle level, make a whole bunch of square root of K funnels, and so on, and then we cut those in the middle level, get fourth root of K funnels, and so on, and so on, at some point the funnel we look at fits in cache.

OK, before we said if it's in a block. Now, we're going to say that at some point, one of these funnels will fit in cache. Each of the funnels at that recursive level of detail will fit in cache. We are going to analyze that level. We'll call that level J. So, consider the first recursive level of detail, and I'll call it J, at which every J funnel we have fits, let's say, not only does it fit in cache, but four of them fit in cache.

It fits in one quarter of the cache. OK, but we need to leave some cache extra for doing other things. But I want to make sure that the J funnel fits. OK, now what does that mean? Well, we've analyzed space. We know that the space of a J funnel is about J^2, some constant times J^2. We'll call it C times J^2. OK, so this is saying that C times J^2 is at most, M over 4, one quarter of the cache.

OK, that means a J funnel that happens at the size sits in the quarter of the cache. OK, at some point in the recursion, we'll have this big tree of J funnels, with all sorts of buffers in between them, and each of the J funnels will fit. So, let's think about one of those J funnels. Suppose J is like the square root of K. So, this is the picture because otherwise I have to draw a bigger one. So, suppose this is a J funnel. It has a bunch of input buffers, has one output buffer.

So, we just want to think about how the J funnel executes. And, for a long time, as long as these buffers are all full, this is just a merger. It's doing something recursively, but we don't really care. As soon as this whole thing swaps in, and actually, I should be drawing this, as soon as the funnel, the output buffer, and the input buffer swap in, in other words, you bring all those blocks in, you can just merge, and you can go on your merry way merging until something empties or you fill the output.

So, let's analyze that. Suppose everything is in memory, because we know it fits. OK, well I have to be a little bit careful. The input buffers are actually pretty big in

total size because the total size is K to the three halves here versus K to the one half. Actually, this is of size K. Let me draw a general picture. We have a J funnel, because otherwise the arithmetic is going to get messy.

We have a J funnel. Its size is C times J^2, we're supposing. The number of inputs is J, and the size of them is pretty big. Where did we define that? We have a K funnel. The total input size is K^3. So, the total input size here would be J^3. We can't afford to put all that in cache. That's an extra factor of J. But, we can afford to one block per input. And for merging, that's all we need. I claim that I can fit the first block of each of these input arrays in cash at the same time along with the J funnel. And so, for that duration, as long as all of that is in cache, this thing can merge at full speed just like we were doing parallel scans. You use up all the blocks down here, and one of them empties. You go to the next block in the input buffer and so on, just like the normal merge analysis of parallel arrays, at this point we assume that everything here is fitting in cache.

So, it's just like before. Of course, in fact, it's recursive but we are analyzing it at this level. OK, I need to prove that you can fit one block per input. It's not hard. It's just computation. And, it's basically the way that these funnels were designed was so that you could fit one block per input buffer. And, here's the argument. So, the claim is you can also fit one memory block in the cache per input buffer. So, this is in addition to one J funnel. You could also fit one block for each of its input buffers. OK, this is of the J funnel.

It's not any funnel because bigger funnels are way too big. OK, so here's how we prove that. J^2 is at most a quarter M. That's what we assumed here, actually CJ2. I'm not going to bother with the C because that's going to make my life even harder. OK, I think this is even a weaker constraint. So, the size of our funnel proves about J^2. That's at most a quarter of the cache. That implies that J, if we take square roots of both sides, is at most a half square root of M. OK, also, we know that B is at most square root of M because M is at least B squared. So, we put these together, and we get J times B is at most a half M.

OK, now I claim that what we are asking for here is J times B because in a J funnel, there are J input arrays. And so, if you want one block each, that costs a space of B each. So, for each input buffer we have one block of size B, and the claim is that that whole thing fits in half the cache. And, we've only used a quarter of the cache. So in total, we use three quarters of the cache and that's all we'll use. OK, so that's good news. We can also fit one more block to the output. Not too big a deal.

So now, as long as this J funnel is running, if it's all in cache, all is well. What does that mean? Let me first analyze how long it takes for us to swap in this funnel. OK, so how long does it take for us to read all the stuff in a J funnel and one block per input buffer? That's what it would take to get started. So, this is swapping in a J funnel, which means reading the J funnel in its entirety, and reading one block per input buffer. OK, the cost of the swap in is pretty natural. The size of the buffer divided by B, because that's just sort of a linear scan to read it in, and we need to read one block per buffer.

These buffers could be all over the place because they're pretty big. So, let's say we pay one memory transfer for each input buffer just to get started to read the first block. OK, the claim is, and here we need to do some more arithmetic. This is, at most, J^3 over B. OK, why is it, at most, J^3 over B? Well, this was the first level at

which things fit in cache. That means the next level bigger, which is J^2, which has size J^4, should be bigger than cache. Otherwise we would have stopped then. OK, so this is just more arithmetic. You can either believe me or follow the arithmetic. We know that J^4 is at least M. So, this means that, and we know that M is at least B^2. Therefore, J^2, instead of J^4, we take the square root of both sides, J^2 is at least B.

OK, so certainly J^2 over B is at most J^3 over B. But also J is at most J^3 over B because J^2 is at least B. Hopefully that should be clear. That's just algebra. OK, so we're not going to use this bound because that's kind of complicated. We're just going to say, well, it causes J^3 over B to get swapped in. Now, why is J^3 over B a good thing? Because we know the total size of inputs to the J funnel is J^3. So, to read all of the inputs to the J funnel takes J^3 over B. So, this is really just a linear extra cost to get the whole thing swapped in. It sounds good. To do the merging would also cost J^3 over B. So, the swap-in causes J^3 over B to merge all these J^3 elements. If they were all there in the inputs, it would take J^3 over B because once everything is there, you're merging at full speed, one per B items per memory transfer on average.

OK, the problem is you're going to swap out, which you may have imagined. As soon as one of your input buffers empties, let's say this one's almost gone, as soon as it empties, you're going to totally obliterate that funnel and swap in this one in order to merge all the stuff there, and fill this buffer back up. This is where the amortization comes in. And this is where the log factor comes in because so far it we've basically paid a linear cost.

We are almost done. So, we charge, sorry, I'm jumping ahead of myself. So, when an input buffer empties, we swap out. And we recursively fill that buffer. OK, I'm going to assume that there is absolutely no reuse, that is recursive filling completely swapped everything out and I have to start from scratch for this funnel. So, when that happens, I feel this buffer, and then I come back and I say, well, I go swap it back in. So when the recursive call finishes, I swap back in. OK, so I recursively fill, and then I swap back in.

And, at the swapping back in costs J^3 over B. I'm going to charge that cost to the elements that just got filled. So this is an amortized charging argument. How many are there? It's the only question. It turns out, things are really good, like here, for the square root of K funnel, we have each buffer has size K to the three halves. OK, so this is a bit complicated. But I claim that the number of elements here that fill the buffer is J^3. So, if you have a J funnel, each of the input buffers has size J^3. It should be correct if you work it out. So, we're charging this J^3 over B cost to J^3 elements, which sounds like you're charging, essentially, one over B to each element.

Sounds great. That means that, so you're thinking overall, I mean, there are N elements, and to each one you charge a one over B cost. That sounds like the total running time is N over B. It's a bit too fast for sorting. We lost the log factor. So, what's going on is that we're actually charging to one element more than once. And, this is something that we don't normally do, never done it in this class, but you can do it as long as you bound that the number of times you charge. OK, and wherever you do a charging argument, you say, well, this doesn't happen too many times because whenever this happens, this happens. You should say, you should prove that the thing that you're charging to, Ito charged to that think very many times. So

here, I have a quantifiable thing that I'm charging to: elements. So, I'm saying that for each element that happened to come into this buffer, I'm going to charge it a one over B cost.

How many times does one element get charged? Well, each time it gets charged to, it's moved into a new buffer. How many buffers could it move through? Well, it's just going up all the time. Merging always goes up. So, we start here and you go to the next buffer, and you go to the next buffer. The number of buffers you visit is the right log, it turns out. I don't know which log that is. So, the number of charges of a one over B cost to each element is the number of buffers it visits, and that's a log factor. That's where we get an extra log factor on the running time. It is, this is the number of levels of J funnels that you can visit. So, it's log K divided by log J, if I got it right.

OK, and we're almost done. Let's wrap up a bit. Just a little bit more arithmetic, unfortunately. So, log K over log J. Now, J^2 is like M, roughly. It might be square root of M. But, log J is basically log M. There's some constants there. So, the number of charges here is theta, log K over log M. So, now this is a bit, we haven't seen this in amortization necessarily, but we just need to count up total amount of charging. All work gets charged to somebody, except we didn't charge the very initial swapping in to everybody. But, every time we do some swapping in, we charge it to someone. So, how many times does everything it charged? Well, there are N elements. Each gets charged to a one over B cost, and the number of times it gets charged is its log K over log M.

So therefore, the total cost is number of elements times a one over B times this log thing. OK, it's actually plus K. We forgot about a plus K, but that's just to get started in the very beginning, and start on all of the input lists. OK, this is an amortization analysis to prove this bound. Sorry, what was N here? I assumed that I started out with K cubed elements at the bottom. The total number of elements in the bottom was K^3 theta. OK, so I should have written K^3 not M. This should be almost the same as this, OK, but not quite. This is log based M of K, and if you do a little bit of arithmetic, this should be K^3 over B times log base M over B of K over B plus K.

That's what I want to prove. Actually there's a K^3 here instead of a K, but that's just a factor of three. And this follows because we assume we are not in the base case. So, K is at least M, which is at least B^2, and therefore K over B is omega square root of K. OK, so K over B is basically the same as K when you put it in a log. So here we have log base M. I turned it into log base M over B. That's even worse. It doesn't matter. And, I have log of K. I replaced it with K over B, but K over B is basically square root of K. So in a log, that's just a factor of a half.

So that concludes the analysis of the funnel. We get this crazy running time, which is basically sorting bound plus a little bit. We plug that into our funnel sort, and we get, magically, optimal cache oblivious sorting just in time. Tuesday is the final. The final is more in the style of quiz one, so not too much creativity, mostly mastery of material. It covers everything. You don't have to worry about the details of funnel sort, but everything else. So it's like quiz one but for the entire class.

It's three hours long, and good luck. It's been a pleasure having you, all the students. I'm sure Charles agrees, so thanks everyone. It was a lot of fun.